デベロッパーガイド

# AWS Encryption SDK

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

# AWS Encryption SDK: デベロッパーガイド

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon の商標およびトレードドレスはAmazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

# Table of Contents

AWS Encryption SDKとは ... 1
オープンソースリポジトリで開発 ... 2
暗号化ライブラリやサービスとの互換性 ... 3
サポートとメンテナンス ... 3
詳細情報 ... 4
フィードバックを送る ... 5
概念 ... 6
エンベロープ暗号化 ... 7
データキー ... 8
ラッピングキー ... 8
キーリングおよびマスターキープロバイダー ... 9
暗号化コンテキスト ... 10
暗号化されたメッセージ ... 12
アルゴリズムスイート ... 12
暗号化マテリアルマネージャー ... 13
対称暗号化と非対称暗号化 ... 13
キーコミットメント ... 14
コミットメントポリシー ... 16
デジタル署名 ... 17
SDK のしくみ ... 18
がデータを AWS Encryption SDK 暗号化する方法 ... 18
が暗号化されたメッセージを復 AWS Encryption SDK 号する方法 ... 19
サポートされているアルゴリズムスイート ... 20
推奨: キー取得、署名、キーコミットメントを使用する AES-GCM ... 20
サポートされているその他のアルゴリズムスイート ... 21
の操作 AWS KMS ... 23
ベストプラクティス ... 25
SDK の設定 ... 29
プログラミング言語の選択 ... 29
ラッピングキーの選択 ... 30
マルチリージョンの使用 AWS KMS keys ... 31
アルゴリズムスイートを選択する ... 52
暗号化されたデータキーの制限 ... 64
検出フィルターの作成 ... 71

暗号化コンテキストの要求 ... 74
コミットメントポリシーの設定 ... 81
ストリーミングデータの操作 ... 82
データキーのキャッシュ ... 82
キーストア ... 83
キーストアの用語と概念 ... 83
最小特権のアクセス許可の実装 ... 84
キーストアを作成する ... 85
キーストアアクションを設定する ... 86
キーストアアクションを設定する ... 87
ブランチキーを作成する ... 91
アクティブなブランチキーをローテーションする ... 95
キーリング ... 97
キーリングのしくみ ... 97
キーリングの互換性 ... 98
暗号化キーリングのさまざまな要件 ... 99
互換性があるキーリングおよびマスターキープロバイダー ... 99
AWS KMS キーリング ... 101
AWS KMS キーリングに必要なアクセス許可 ... 103
AWS KMS キーリング AWS KMS keys での の識別 ... 103
AWS KMS キーリングの作成 ... 104
AWS KMS 検出キーリングの使用 ... 119
AWS KMS リージョン検出キーリングの使用 ... 126
AWS KMS 階層キーリング ... 135
仕組み ... 137
前提条件 ... 139
必要なアクセス許可 ... 139
キャッシュを選択する ... 139
階層キーリングを作成する ... 153
AWS KMS ECDH キーリング ... 161
AWS KMS ECDH キーリングに必要なアクセス許可 ... 162
AWS KMS ECDH キーリングの作成 ... 162
AWS KMS ECDH 検出キーリングの作成 ... 170
Raw AES キーリング ... 175
Raw RSA キーリング ... 183
Raw ECDH キーリング ... 192

Raw ECDH キーリングの作成 ..... 193
マルチキーリング ..... 211
プログラミング言語 ..... 221
C ..... 221
インストール ..... 222
C SDK を使用する ..... 223
例 ..... 228
.NET ..... 235
インストールおよび構築 ..... 237
デバッグ ..... 237
例 ..... 238
Go ..... 246
前提条件 ..... 247
インストール ..... 247
Java ..... 248
前提条件 ..... 248
インストール ..... 250
例 ..... 251
JavaScript ..... 264
互換性 ..... 265
インストール ..... 267
モジュール ..... 268
例 ..... 270
Python ..... 279
前提条件 ..... 279
インストール ..... 280
例 ..... 281
Rust ..... 289
前提条件 ..... 290
インストール ..... 290
例 ..... 290
コマンドラインインターフェイス ..... 293
CLI のインストール ..... 294
CLI の使用方法 ..... 298
例 ..... 312
構文およびパラメータのリファレンス ..... 336

バージョン ... 350
データキーキャッシュ ... 353
データキーキャッシュを使用する方法 ... 354
データキーキャッシュを使用する: ステップバイステップ ... 355
データキーキャッシュの例: 文字列を暗号化する ... 362
キャッシュセキュリティのしきい値の設定 ... 379
データキーキャッシュの詳細 ... 380
データキーキャッシュの仕組み ... 381
暗号化マテリアルキャッシュの作成 ... 383
キャッシュ暗号化マテリアルマネージャーの作成 ... 384
データキーキャッシュエントリとは ... 385
暗号化コンテキスト: キャッシュエントリを選択する方法 ... 386
アプリケーションはキャッシュされたデータキーを使用していますか？ ... 386
データキーキャッシュの例 ... 387
ローカルキャッシュの結果 ... 387
サンプルのコード ... 388
AWS CloudFormation テンプレート ... 400
のバージョン AWS Encryption SDK ... 415
C ... 416
C# / .NET ... 416
コマンドラインインターフェイス (CLI) ... 417
Java ... 419
Go ... 421
JavaScript ... 422
Python ... 423
Rust ... 424
バージョンの詳細 ... 425
1.7.x より前のバージョン ... 425
バージョン 1.7.x ... 426
バージョン 2.0.x ... 428
バージョン 2.2.x ... 430
バージョン 2.3.x ... 431
の移行 AWS Encryption SDK ... 432
移行してデプロイする方法 ... 434
ステージ 1: アプリケーションを最新 1.x バージョンに更新 ... 434
ステージ 2: アプリケーションを最新バージョンに更新 ... 435

AWS KMS マスターキープロバイダーの更新 ..... 436
Strict モードへの移行 ..... 437
Discovery モードへの移行 ..... 441
AWS KMS キーリングの更新 ..... 444
コミットメントポリシーの設定 ..... 447
コミットメントポリシーの設定方法 ..... 448
最新バージョンへの移行に関するトラブルシューティング ..... 459
非推奨または削除されたオブジェクト ..... 460
構成の競合: コミットメントポリシーとアルゴリズムスイート ..... 461
構成の競合: コミットメントポリシーと暗号化テキスト ..... 462
キーコミットメントの検証の失敗 ..... 462
その他の暗号化の失敗 ..... 462
その他の復号化の失敗 ..... 463
ロールバックに関する考慮事項 ..... 463
よくある質問 ..... 464
は AWS SDKs とどのように AWS Encryption SDK 異なりますか？ ..... 464
は Amazon S3 暗号化クライアントとどのように AWS Encryption SDK 異なりますか？ ..... 465
でサポートされている暗号化アルゴリズム AWS Encryption SDKとデフォルトはどれですか？ ..... 465
初期化ベクター (IV) はどのように生成され、どこに保存されますか? ..... 466
各データキーはどのように生成、暗号化、および復号されますか? ..... 466
データを暗号化するために使用されたデータキーを追跡するにはどうすればよいですか? ..... 467
では、暗号化されたデータキーを暗号化されたデータと共に AWS Encryption SDK 保存する方法を教えてください。 ..... 467
AWS Encryption SDK メッセージ形式は暗号化されたデータにどのくらいのオーバーヘッドを追加しますか？ ..... 467
独自のマスターキープロバイダーを使用できますか? ..... 468
複数のラッピングキーでデータを暗号化できますか? ..... 468
どのデータ型を で暗号化できますか AWS Encryption SDK？ ..... 468
は入出力 (I/O) ストリームをどのように AWS Encryption SDK 暗号化および復号しますか？ ..... 469
リファレンス ..... 470
メッセージ形式のリファレンス ..... 470
ヘッダーの構造 ..... 471
本文の構造 ..... 479
フッターの構造 ..... 484
メッセージ形式の例 ..... 485

フレーム化されたデータ (メッセージ形式バージョン 1) ..... 485
フレーム化されたデータ (メッセージ形式バージョン 2) ..... 489
フレーム化されていないデータ (メッセージ形式バージョン 1) ..... 491
本文 AAD のリファレンス ..... 495
アルゴリズムのリファレンス ..... 497
初期化ベクトルのリファレンス ..... 502
AWS KMS 階層キーリングの技術的な詳細 ..... 502
ドキュメント履歴 ..... 504
最新の更新 ..... 504
以前の更新 ..... 507
..... dix

# AWS Encryption SDKとは

AWS Encryption SDK は、業界標準とベストプラクティスを使用して、すべてのユーザーがデータの暗号化と復号を簡単に行えるように設計されたクライアント側の暗号化ライブラリです。これにより、データの暗号化と復号の最善の方法ではなく、アプリケーションのコア機能に集中できるようになります。 AWS Encryption SDK は、Apache 2.0 ライセンスの下で無料で提供されます。

AWS Encryption SDK は次のような質問に答えます。

- どの暗号化アルゴリズムを使用するべきですか。
- どのように、またはどのモードで、そのアルゴリズムを使用すべきですか。
- 暗号化キーを生成するにはどうすればよいですか。
- 暗号化キーを保護するにはどうすればよいですか。どこに保存するべきですか。
- 暗号化されたデータをポータブルにするにはどうしたらよいですか。
- 目的の受取人が暗号化されたデータを確実に読めるようにするにはどうすればよいですか。
- 暗号化されたデータが書き込まれてから読み込まれるまでに変更されないようにするにはどうすればよいですか。
- が AWS KMS 返すデータキーの使用方法を教えてください。

では AWS Encryption SDK、データを保護するために使用するラッピングキーを決定するマスターキープロバイダーまたはキーリングを定義します。次に、 が提供する簡単な方法を使用してデータを暗号化および復号します AWS Encryption SDK。が残り AWS Encryption SDK を行います。

を使用しない場合 AWS Encryption SDK、アプリケーションのコア機能よりも暗号化ソリューションの構築により多くの労力を費やす可能性があります。は、以下の情報を提供して AWS Encryption SDK これらの質問に答えます。

暗号化のベストプラクティスに従ったデフォルトの実装

デフォルトでは、 は暗号化するデータオブジェクトごとに一意のデータキー AWS Encryption SDK を生成します。各暗号化操作に一意のデータキーを使用する暗号化のベストプラクティスに従います。

は、安全で認証された対称キーアルゴリズムを使用してデータを AWS Encryption SDK 暗号化します。詳細については、「the section called “サポートされているアルゴリズムスイート”」を参照してください。

ラッピングキーによるデータキーの保護のためのフレームワーク

は、1 つ以上のラッピングキーで暗号化することで、データを暗号化するデータキー AWS Encryption SDK を保護します。複数のラッピングキーを使用してデータキーを暗号化するフレームワークを提供することで、 AWS Encryption SDK は暗号化されたデータを移植可能にします。

たとえば、 AWS KMS key の AWS KMS とオンプレミス HSM のキーでデータを暗号化します。片方が利用できない場合や、呼び出し元に両方のキーを使用する権限がない場合に備えて、いずれかのラッピングキーを使用してデータを復号できます。

暗号化されたデータと暗号化されたデータキーを一緒に保存する形式のメッセージ

は、暗号化されたデータと暗号化されたデータキーを、定義されたデータ形式を使用する暗号化されたメッセージにまとめて AWS Encryption SDK 保存します。つまり、 AWS Encryption SDK がユーザーに代わってデータを暗号化するデータキーを追跡または保護する必要はありません。

の一部の言語実装には AWS SDK AWS Encryption SDK が必要ですが、 AWS Encryption SDK は を必要とせず、どの AWS サービスにも依存 AWS アカウント しません。は、 AWS KMS keysを使用してデータを保護する AWS アカウント 場合にのみ必要です。

## オープンソースリポジトリで開発

AWS 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、課題を読んだり送信したり、言語実装に固有の情報を見つけたりできます。

- AWS Encryption SDK for C — aws-encryption-sdk-c
- AWS Encryption SDK for .NET — `aws-encryption-sdk`リポジトリの .NET ディレクトリ。
- AWS 暗号化 CLI — aws-encryption-sdk-cli
- AWS Encryption SDK for Java — aws-encryption-sdk-java
- AWS Encryption SDK for JavaScript — aws-encryption-sdk-javascript
- AWS Encryption SDK for Python — aws-encryption-sdk-python
- AWS Encryption SDK for Rust — `aws-encryption-sdk`リポジトリの Rust ディレクトリ。
- AWS Encryption SDK for Go — `aws-encryption-sdk`リポジトリの Go ディレクトリ

# 暗号化ライブラリやサービスとの互換性

AWS Encryption SDK は複数のプログラミング言語でサポートされています。言語実装はすべて相互運用可能です。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。また、暗号化および復号を行う場合は、互換性のあるキーリング、またはマスターキーとマスターキープロバイダーを使用する必要があります。詳細については、「the section called “キーリングの互換性”」を参照してください。

ただし、 を他のライブラリと相互運用 AWS Encryption SDK することはできません。各ライブラリは暗号化されたデータを異なる形式で返すため、あるライブラリで暗号化したデータを別のライブラリで復号することはできません。

DynamoDB 暗号化クライアントおよび Amazon S3 クライアント側の暗号化

は、DynamoDB 暗号化クライアントまたは Amazon S3 クライアント側の暗号化によって暗号化されたデータを復号 AWS Encryption SDK できません。これらのライブラリは、 が AWS Encryption SDK 返す暗号化されたメッセージを復号できません。

AWS Key Management Service (AWS KMS)

AWS Encryption SDK は、 AWS KMS keysおよび データキーを使用して、マルチリージョン KMS キーを含むデータを保護できます。たとえば、 AWS KMS keys 内の 1 つ以上の でデータを暗号化 AWS Encryption SDK するように を設定できます AWS アカウント。ただし、 を使用してそのデータを復 AWS Encryption SDK 号する必要があります。

は、 AWS KMS Encrypt または ReEncrypt オペレーションが返す暗号文を復号 AWS Encryption SDK できません。同様に、 AWS KMS Decrypt オペレーションでは、 が AWS Encryption SDK 返す暗号化されたメッセージを復号することはできません。

は、対称暗号化 KMS キーのみ AWS Encryption SDK をサポートします。「 AWS Encryption SDK」では、暗号化または署名に 非対称 KMS キー を使用できません。 AWS Encryption SDK は、メッセージに署名するアルゴリズムスイートに対して、独自の ECDSA 署名キーを生成します。

# サポートとメンテナンス

AWS Encryption SDK は、 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。これには、バージョニングフェーズとライフサイクルフェーズが含まれます。ベストプラ

クティスとして、プログラミング言語 AWS Encryption SDK に利用可能な最新バージョンの 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。バージョン 1.7.x より前の AWS Encryption SDK バージョンからバージョン 2.0.x 以降にアップグレードするなど、バージョンに大きな変更が必要な場合は、詳細な手順が役立ちます。

の各プログラミング言語実装 AWS Encryption SDK は、個別のオープンソース GitHub リポジトリで開発されています。各バージョンのライフサイクルフェーズやサポートフェーズは、リポジトリによって異なる可能性があります。　たとえば、特定のバージョンの は、1 つのプログラミング言語での一般提供 (フルサポート) フェーズにあるものの、別のプログラミング言語でend-of-supportフェーズにある AWS Encryption SDK 場合があります。可能な限り完全にサポートされているバージョンを使用し、サポートが終了したバージョンは避けることをお勧めします。

プログラミング言語 AWS Encryption SDK のバージョンライフサイクルフェーズを確認するには、各 AWS Encryption SDK リポジトリの `SUPPORT_POLICY.rst` ファイルを参照してください。

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

詳細については、のバージョン AWS Encryption SDK「SDK とツールリファレンスガイド」の「」および AWS SDKs」を参照してください。 AWS SDKs

## 詳細情報

AWS Encryption SDK およびクライアント側の暗号化の詳細については、以下のソースを試してください。

- この SDK で使用される用語と概念のヘルプについては、「の概念 AWS Encryption SDK」を参照してください。
- ベストプラクティスのガイドラインについては、「のベストプラクティス AWS Encryption SDK」を参照してください。
- SDK の仕組みについては、「SDK のしくみ」を参照してください。

- でオプションを設定する方法を示す例については AWS Encryption SDK、「」を参照してくださいの設定 AWS Encryption SDK。
- 技術情報の詳細については、「リファレンス」を参照してください。
- の技術仕様については AWS Encryption SDK、GitHub のAWS Encryption SDK 「仕様」を参照してください。
- の使用に関する質問に対する回答については AWS Encryption SDK、AWS 「Crypto Tools Discussion Forum」を読んで投稿してください。

の AWS Encryption SDK さまざまなプログラミング言語での実装については、「」を参照してください。

- C: GitHub の AWS Encryption SDK for C、 AWS Encryption SDK C ドキュメント、および aws-encryption-sdk-c リポジトリを参照してください。
- C#/.NET: GitHub の「AWS Encryption SDK .NET 用」と `aws-encryption-sdk` リポジトリの aws-encryption-sdk-net ディレクトリを参照してください。
- コマンドラインインターフェイス: GitHub のAWS Encryption SDK コマンドラインインターフェイス「」、「Encryption CLI AWS のドキュメントを読む」、およびaws-encryption-sdk-cli」リポジトリを参照してください。 https://aws-encryption-sdk-cli.readthedocs.io/en/latest/
- Java: GitHub のAWS Encryption SDK for Java「」、 AWS Encryption SDK 「Javadoc」、およびaws-encryption-sdk-java」リポジトリを参照してください。

  JavaScript: 「the section called “JavaScript”」、GitHub の aws-encryption-sdk-javascript リポジトリを参照してください。
- Python: GitHub の AWS Encryption SDK for Python、 AWS Encryption SDK Python ドキュメント、および aws-encryption-sdk-python リポジトリ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。

- で潜在的なセキュリティ脆弱性を発見した場合は AWS Encryption SDK、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。
- に関するフィードバックを提供するには AWS Encryption SDK、使用しているプログラミング言語の 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については、このページの [フィードバック] のリンクをご利用ください。また、GitHub のこのドキュメントのオープンソースリポジトリである aws-encryption-sdk-docs で issue の作成やご参加をいただくこともできます。

# の概念 AWS Encryption SDK

このセクションでは、 で使用される概念を紹介し AWS Encryption SDK、用語集とリファレンスを提供します。これは、 の AWS Encryption SDK 仕組みと、その説明に使用する用語を理解するのに役立つように設計されています。

サポートが必要ですか?

- がエンベロープ暗号化 AWS Encryption SDK 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、データ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とマスターキープロバイダーについての説明。
- 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。これはオプションですが、推奨されるベストプラクティスです。
- 暗号化メソッドが返す暗号化されたメッセージについての説明。
- これで、 AWS Encryption SDK 任意のプログラミング言語で を使用する準備が整いました。

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
- キーリングおよびマスターキープロバイダー
- 暗号化コンテキスト
- 暗号化されたメッセージ
- アルゴリズムスイート
- 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

- コミットメントポリシー
- 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

は、各メッセージを一意のデータキーで AWS Encryption SDK 暗号化します。その後、指定したラッピングキーでデータキーを暗号化します。返される暗号化されたメッセージの暗号化されたデータを使用して、暗号化されたデータキーが保存されます。

ラッピングキーを指定するには、キーリングまたはマスターキープロバイダーを使用します。

### 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で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。は、暗号化されたデータを含むすべての暗号化されたデータキーを暗号化されたメッセージに AWS Encryption SDK 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のいずれかを復号できる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、 は AES-GCM 対称暗号化、キー取得関数 (HKDF)、および署名を備えた高度なアルゴリズムスイート AWS Encryption SDK を使用してデータを暗号化します。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。一方、パブリックキーのアルゴリズムはロールを本質的に

分離し、キー管理を簡単にします。それぞれの強みを組み合わせるには、対称キー暗号化でデータを暗号化し、次にデータキーをパブリックキー暗号化で暗号化します。

## データキー

データキーは、データの暗号化に AWS Encryption SDK で使用される暗号化キーです。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。データキーキャッシュを使用している場合を除き、 AWS Encryption SDK は一意のデータキーを使用して各メッセージを暗号化します。

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 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 Encryption SDK はラッピングキーまたはマスターキーと呼ばれる 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。は、プレーンテキストのデータキー AWS Encryption SDK 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、暗号化オペレーションで返る暗号化されたメッセージの暗号化されたデータを使用して、暗号化されたデータキーが保存されます。詳細については、「the section called “SDK のしくみ”」を参照してください。

> **Tip**
>
> では AWS Encryption SDK、データキーとデータ暗号化キーを区別します。デフォルトのスイートを含むサポートされているアルゴリズムスイートのいくつかは、データキーが暗号化の上限に到達することを防ぐ、キー取得関数を使用します。キー取得関数は、データキーを入力として受け取り、データ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 Encryption SDK は復号時に有効なラッピングキーを簡単に識別できます。

## ラッピングキー

ラッピングキー はキー暗号化キーであり、 AWS Encryption SDK ではこれを使用して、データを暗号化する データキー を暗号化します。それぞれのプレーンテキストのデータキーは、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。キーリングまたはマスターキープロバイダーの設定時に、データの保護に使用するラッピングキーを決定します。

> **Note**
>
> ラッピングキーは、キーリングまたはマスターキープロバイダー内のキーを参照します。マスターキーは一般的に、マスターキープロバイダーを使用するときにインスタンス化する `MasterKey` クラスと関連します。

は、 (AWS KMS) 対称 AWS KMS keys (マルチリージョン KMS キーを含む ) 、raw AES-GCM (Advanced Encryption Standard/Galois Counter Mode) キー、raw RSA キーなど AWS Key Management Service 、一般的に使用されるラッピングキーをいくつか AWS Encryption SDK サポートしています。また、独自のラッピングキーを拡張または実装することもできます。

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) などのウェブ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。は AWS Encryption SDK と統合 AWS KMS され、ラッピングキーの保護と使用に役立ちます。ただし、 AWS Encryption SDK には AWS または AWS のサービスは必要ありません。

## キーリングおよびマスターキープロバイダー

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、キーリングまたはマスターキープロバイダーを使用します。が提供するキーリングとマスターキープロバイダーを使用すること AWS Encryption SDK も、独自の実装を設計することもできます。 AWS Encryption SDK では、言語制約の対象となりながらも相互に互換性のあるキーリングとマスターキープロバイダーが提供されます。詳細については、「キーリングの互換性」を参照してください。

キーリングは、データキーの生成、暗号化、復号を行います。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。追加の設定オプションを使用して、ラッピングキーのないキーリングや、より複雑なキーリングを定義することもできます。が AWS Encryption SDK 定義するキーリングの選択と使用については、「」を参照してくださいキーリング。

キーリングは、次のプログラミング言語でサポートされています。

- AWS Encryption SDK for C
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK .NET 用
- のバージョン 3.x AWS Encryption SDK for Java
- オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。
- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

マスターキープロバイダーはキーリングの代替品です。マスターキープロバイダーは、指定したラッピングキー (またはマスターキー) を返します。各マスターキーは 1 つのマスターキープロバイダーに関連付けられていますが、マスターキープロバイダーは通常複数のマスターキーを提供しています。マスターキープロバイダーは、Java、Python、および AWS Encryption CLI でサポートされています。

暗号化には、キーリング (またはマスターキープロバイダー) を指定する必要があります。復号化には、同じキーリング (またはマスターキープロバイダー) を指定することも、別のキーリングを指定することもできます。暗号化時に、 AWS Encryption SDK は指定し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。復号化するとき、 AWS Encryption SDK では、指定したラッピングキーのみを使用して、暗号化されたデータキーを復号します。復号用のラッピングキーの指定はオプションですが、 AWS Encryption SDK ベストプラクティスです。

ラッピングキーの指定の詳細については、「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には、データを暗号化するためのすべてのリクエストに暗号化コンテキストを含めます。暗号化コンテキストの使用はオプションですが、暗号化のベストプラクティスとして使用することをお勧めします。

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。暗号化コンテキストには選択した任意のデータを含むことができますが、一般的には、ファイルの種類、目的、または所有権などの、ログ記録と追跡に有用なデータが含まれます。データを暗号化する場合、暗号化コンテキストは暗号化されたデータに暗号化されてバインドされます。これにより、データ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 Encryption SDK より返る暗号化されたメッセージのヘッダーには、プレーンテキストの暗号化コンテキストが含まれます。

が AWS Encryption SDK 使用する暗号化コンテキストは、指定した暗号化コンテキストと、暗号化マテリアルマネージャー (CMM) が追加するパブリックキーペアで構成されます。具体的には、署名付きの暗号化アルゴリズムを使用する度に、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暗号化コンテキストに名前と値のペアが CMM によって追加されます。暗号化コンテキスト`aws-crypto-public-key`の名前は によって予約 AWS Encryption SDK されており、暗号化コンテキストの他のペアの名前として使用することはできません。詳細については、メッセージ形式リファレンスの「AAD」を参照してください。

以下の暗号化コンテキストの例は、リクエストで指定した 2 つの暗号化コンテキストペアと、CMM によって追加されるパブリックキーのペアで構成されます。

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

データを復号するには、暗号化されたメッセージを渡します。は暗号化されたメッセージヘッダーから暗号化コンテキストを抽出 AWS Encryption SDK できるため、暗号化コンテキストを個別に指定する必要はありません。ただし、暗号化コンテキストは、暗号化された適切なメッセージを復号していることを確認するのに役立ちます。

- AWS Encryption SDK コマンドラインインターフェイス (CKI) において、復号コマンドで暗号化コンテキストを指定した場合、CLI で、プレーンテキストのデータが返る前に、暗号化されたメッセージの暗号化コンテキストにその値が存在することが検証されます。
- 他のプログラミング言語実装では、復号レスポンスに暗号化コンテキストとプレーンテキストデータが含まれます。アプリケーションの復号関数では、プレーンテキストデータを返す前に、復号レスポンスの暗号化コンテキストに暗号化リクスとの暗号化コンテキスト (またはサブセット) が含まれていることを常に確認する必要があります。

> **Note**
>
> 次のバージョンでは、必要な暗号化コンテキスト CMM がサポートされています。これを使用して、すべての暗号化リクエストで暗号化コンテキストを要求できます。

- のバージョン 3.x AWS Encryption SDK for Java
- .NET AWS Encryption SDK 用 のバージョン 4.x
- オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。
- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

暗号化コンテキストを選択する際、シークレットではないことに注意してください。暗号化コンテキストは、 AWS Encryption SDK が返す暗号化されたメッセージのヘッダーのプレーンテキストに表示されます。を使用している場合 AWS Key Management Service、暗号化コンテキストは、 などの監査レコードやログにプレーンテキストで表示される場合があります AWS CloudTrail。

コード内の暗号化コンテキストを送信および検証する例については、使用しているプログラミング言語の例を参照してください。

## 暗号化されたメッセージ

でデータを暗号化すると AWS Encryption SDK、暗号化されたメッセージが返されます。

暗号化されたメッセージは、データキーの暗号化されたコピーと共に暗号化されたデータを含む、小型で書式設定されたデータ構造、アルゴリズム ID、および必要に応じて暗号化コンテキストとデジタル署名です。 AWS Encryption SDK の暗号化オペレーションは暗号化されたメッセージを返し、復号オペレーションは暗号化されたメッセージを入力として受け取ります。

暗号化されたデータとその暗号化されたデータキーを組み合わせることで、復号オペレーションを合理化し、暗号化するデータから暗号化されたデータキーを個別に保存して管理する必要がなくなります。

暗号化されたメッセージに関する技術情報については、「暗号化されたメッセージの形式」を参照してください。

## アルゴリズムスイート

AWS Encryption SDK はアルゴリズムスイートを使用して、暗号化および復号オペレーションが返す暗号化されたメッセージのデータを暗号化および署名します。 AWS Encryption SDK では、いくつかのアルゴリズムスイートがサポートされています。サポートされているすべてのスイート

は、Advanced Encryption Standard (AES) を主なアルゴリズムとして、他のアルゴリズムや値と組み合わせて使用します。

は、すべての暗号化オペレーションのデフォルトとして推奨アルゴリズムスイート AWS Encryption SDK を確立します。標準とベストプラクティスの向上に伴い、デフォルトは変更される可能性があります。データの暗号化リクエスト内、または暗号化マテリアルマネージャー (CMM) の作成時に、代替のアルゴリズムスイートを指定できます。ただし、状況からして代替が必須でない限り、デフォルトを使用することをお勧めします。現在のデフォルトは、HMAC ベースの抽出および展開キー取得関数 (HKDF) を使用する AES-GCM、キーコミットメント、楕円曲線 DSA (ECDSA) 署名、256 ビット暗号化キーです。

アプリケーションで高いパフォーマンスを必要とし、データを暗号化するユーザーとデータを復号化するユーザーが同等に信頼できる場合は、デジタル署名のないアルゴリズムスイートを指定することを検討してください。ただし、キーコミットメントとキー取得関数を含むアルゴリズムスイートを強くお勧めします。これらの機能のないアルゴリズムスイート機能は、下位互換性のためにのみサポートされています。

## 暗号化マテリアルマネージャー

暗号化マテリアルマネージャー (CMM) は、データの暗号化と復号化に使用される暗号化マテリアルを組み立てます。暗号化マテリアルには、プレーンテキストおよび暗号化されたデータキー、オプションのメッセージ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。

AWS Encryption SDK が提供するデフォルトの CMM またはキャッシュ CMM を使用するか、カスタム CMM を記述できます。CMM は指定できますが、必須ではありません。キーリングまたはマスターキープロバイダーを指定すると、 によってデフォルトの CMM AWS Encryption SDK が作成されます。デフォルトの CMM は、指定したキーリングまたはマスターキープロバイダー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。

CMM は AWS Encryption SDK とキーリング (またはマスターキープロバイダー) の間の連絡係として機能するため、ポリシーの適用やキャッシュのサポートなど、カスタマイズや拡張に最適なポイントです。は、データキーキャッシュをサポートするキャッシュ CMM AWS Encryption SDK を提供します。 ???

## 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。

はエンベロープ暗号化 AWS Encryption SDK 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。返される暗号化されたメッセージには、暗号化されたデータおよび少なくとも 1 つの暗号化されたデータキーのコピーが含まれます。

データの暗号化 (対称暗号化)

は、データを暗号化するために、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 AWS Encryption SDK を使用します。データを復号するために、 は同じデータキーと同じアルゴリズムスイート AWS Encryption SDK 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)

暗号化および復号化のオペレーションに指定するキーリングまたはマスターキープロバイダーにより、対称データキーの暗号化および復号化方法が決まります。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マスター AWS KMS キープロバイダー、または raw RSA キーリングや などの非対称暗号化を使用するキーリングまたはマスターキープロバイダーを選択できますJceMasterKey。

## キーコミットメント

は、キーコミットメント (堅牢性とも呼ばれます) AWS Encryption SDK をサポートします。これは、各暗号文を 1 つのプレーンテキストにのみ復号化できることを保証するセキュリティプロパティです。これを行うために、キーコミットメントでは、メッセージを暗号化したデータキーのみが復号化に使用されることが保証されます。キーコミットメントによる暗号化と復号化は、AWS Encryption SDK のベストプラクティスです。

最新の対称暗号 (AES を含む) では、 AWS Encryption SDK が各プレーンテキストメッセージの暗号化に使用する、一意のデータキーなどの単一のシークレットキーでプレーンテキストが暗号化されます。同じデータキーでこのデータを復号すると、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。ただし、2 つの異なるキーで暗号化テキストを復号化することは可能です。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復号化できるキーを見つけることは可能です。

は AWS Encryption SDK 、常に 1 つの一意のデータキーで各プレーンテキストメッセージを暗号化します。複数のラッピングキー (またはマスターキー) 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メッセージには、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メッセージを復号すると 0x0 (false) を返し、同じ暗号化されたメッセージ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、 は暗号化および復号時にキーコミットメント AWS Encryption SDK をサポートします。がキーコミットメントでメッセージを AWS Encryption SDK 暗号化すると、暗号文を生成した一意のデータキーを、シークレット以外のデータキー識別子であるキーコミットメント文字列に暗号でバインドします。その後、キーコミットメント文字列は、暗号化されたメッセージのメタデータに保存されます。キーコミットメントを使用してメッセージを復号すると、 AWS Encryption SDK はデータキーがその暗号化されたメッセージの唯一の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。

キーコミットメントのサポートは、バージョン 1.7.x で導入されました。このバージョンではキーコミットメントでメッセージを復号化できますが、キーコミットメントによる暗号化はできません。このバージョンを使用して、キーコミットメントで暗号化テキストを復号化する機能を完全にデプロイできます。バージョン 2.0.x では、キーコミットメントが完全にサポートされます。デフォルトでは、キーコミットメントでのみ暗号化および復号化が行われます。これは、 の以前のバージョンで暗号化された暗号文を復号する必要がないアプリケーションに最適です AWS Encryption SDK。

キーコミットメントによる暗号化と復号化がベストプラクティスですが、使用時期を決定し、それを採用するペースを調整できます。バージョン 1.7.x 以降、 はデフォルトのアルゴリズムスイートを設定し、使用できるアルゴリズムスイートを制限するコミットメントポリシー AWS Encryption SDK をサポートしています。このポリシーにより、データをキーコミットメントで暗号化および復号化するかどうかが決まります。

キーコミットメントでは、暗号化メッセージがわずかに大きくなり (+ 30 バイト)、処理に時間がかかります。アプリケーションでサイズやパフォーマンスに注意が必要である場合は、キーコミットメントをオプトアウトすることもできます。しかし、必要である場合にのみオプトアウトしてください。

バージョン 1.7.x および 2.0.x への移行、およびキーコミットメント機能に関する詳細については、の移行 AWS Encryption SDK を参照してください。キーコミットメントに関する技術情報については、the section called “アルゴリズムのリファレンス” および the section called “メッセージ形式のリファレンス” を参照してください。

# コミットメントポリシー

コミットメントポリシーは、アプリケーションがキーコミットメントで暗号化および復号化を行うかどうかを決定する構成設定です。キーコミットメントによる暗号化と復号化は、AWS Encryption SDK のベストプラクティスです。

コミットメントポリシーには 3 つの値があります。

> **Note**
>
> テーブル全体を表示するには、水平または垂直にスクロールする必要があります。

コミットメントポリシーの値

| 値 | キーコミットメントで暗号化 | キーコミットメントなしで暗号化 | キーコミットメントで復号化 | キーコミットメントなしで復号化 |
|---|---|---|---|---|
| ForbidEncryptAllowDecrypt | | | | |
| RequireEncryptAllowDecrypt | | | | |
| RequireEncryptRequireDecrypt | | | | |

コミットメントポリシー設定は AWS Encryption SDK 、バージョン 1.7.x で導入されました。すべてのサポート対象プログラミング言語で有効です。

- `ForbidEncryptAllowDecrypt` は、キーコミットメントの有無にかかわらず復号しますが、キーコミットメントでは暗号化しません。バージョン 1.7.x で導入されたこの値は、キーコミットメントで暗号化された暗号文に遭遇する前に、アプリケーションを実行しているすべてのホストがキーコミットメントで復号する準備をするように設計されています。

- `RequireEncryptAllowDecrypt` では常にキーコミットメントで暗号化されます。復号化は、キーコミットメントが使用されているかどうかにかかわらず可能です。バージョン 2.0.x で導入されたこの値では、キーコミットメントによる暗号化を開始できますが、キーコミットメントによらない従来の暗号化テキストを復号化できます。
- `RequireEncryptRequireDecrypt` では、キーコミットメントでのみ暗号化および復号化が行われます。この値がバージョン 2.0.x のデフォルトです。この値は、すべての暗号化テキストがキーコミットメントで暗号化されていることが確実な場合に使用します。

コミットメントポリシー設定により、使用できるアルゴリズムスイートが決まります。バージョン 1.7.x 以降、 では、署名の有無にかかわらず、キーコミットメントのアルゴリズムスイート AWS Encryption SDK がサポートされています。コミットメントポリシーと競合するアルゴリズムスイートを指定した場合、 AWS Encryption SDK はエラーを返します。

コミットメントポリシーの設定については、コミットメントポリシーの設定 を参照してください。

## デジタル署名

は、認証された AWS Encryption SDK 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。例えば、 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、 を呼び出さずに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、 AWS Encryption SDK では、暗号化されたメッセージの末尾に楕円曲線デジタル署名アルゴリズム (ECDSA) 署名を追加することができます。署名アルゴリズムスイートを使用すると、 は暗号化されたメッセージごとに一時的なプライベートキーとパブリックキーのペア AWS Encryption SDK を生成します。は、データキーの暗号化コンテキストにパブリックキー AWS Encryption SDK を保存し、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。アルゴリズムは、メッセージヘッダー内の追加の認証済みデータとしてパブリックキーを暗号化されたデータキーにバインドし、メッセージを復号化できるユーザーのみ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。

署名の検証では、復号化に大きなパフォーマンスコストがかかります。データを暗号化するユーザーとデータを復号化するユーザーが同等に信頼されている場合は、署名を含まないアルゴリズムスイートの使用を検討してください。

> **Note**
>
> キーリングまたはラッピング暗号化マテリアルへのアクセスがエンクリプタと復号器の間で区別されない場合、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子を区別できます。

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

# の AWS Encryption SDK 仕組み

このセクションのワークフローでは、 AWS Encryption SDK がデータを暗号化し、暗号化されたメッセージを復号する方法について説明します。これらのワークフローは、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。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、サポートされているそれぞれの言語実装の GitHub リポジトリを参照してください。

AWS 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。各メッセージは、一意のデータキーで暗号化されます。その後、データキーは指定したラッピングキーにより暗号化されます。暗号化されたメッセージを復号するために、 AWS 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、暗号文を復号化してプレーンテキストのメッセージを返すことができます。

「 AWS Encryption SDK」で使われている用語についてサポートが必要ですか？「the section called “概念”」を参照してください。

## がデータを AWS Encryption SDK 暗号化する方法

AWS Encryption SDK には、文字列、バイト配列、バイトストリームを暗号化するメソッドが用意されています。コードの例については、各 プログラミング言語 セクションの「例」トピックを参照してください。

1. データを保護するラッピングキーを指定する キーリング (または マスターキープロバイダー) を作成します。
2. キーリングとプレーンテキストのデータを暗号化メソッドに渡します。シークレットではないオプションの 暗号化コンテキスト で渡すことをお勧めします。
3. 暗号化メソッドによって、キーリングに暗号化マテリアルが求められます。キーリングは、メッセージの一意のデータ暗号化キーを返します。1 つはプレーンテキストのデータキーで、もう 1 つは指定された各ラッピングキーで暗号化されたデータキーのコピーです。
4. 暗号化方法は、プレーンテキストデータキーを使用してデータを暗号化し、プレーンテキストのデータキーを破棄します。暗号化コンテキスト ( AWS Encryption SDK ベストプラクティス) を指定すると、暗号化メソッドは暗号化コンテキストを暗号化データに暗号的に結合します。
5. 暗号化メソッドによって、暗号化されたデータ、暗号化されたデータキー、使用した場合は暗号化コンテキストなどの他の暗号化されたデータを含む 暗号化されたメッセージ が返ります。

## が暗号化されたメッセージを復 AWS Encryption SDK 号する方法

AWS Encryption SDK は、暗号化されたメッセージを復号し、プレーンテキストを返すメソッドを提供します。コードの例については、各 プログラミング言語 セクションの「例」トピックを参照してください。

暗号化されたメッセージを復号化する キーリング (または マスターキープロバイダー) は、メッセージの暗号化に使用されたキーリング (またはマスターキープロバイダー) と互換性がある必要があります。ラッピングキーのいずれか 1 つが暗号化されたメッセージの暗号化されたデータキーを復号できる必要があります。キーリングとマスターキープロバイダとの互換性については、「the section called “キーリングの互換性”」を参照してください。

1. データを復号できるラッピングキーを使用して、キーリングまたはマスターキープロバイダーを作成します。暗号化メソッドに提供ものと同じキーリングを使用するか、別のキーリングを使用することもできます。
2. 暗号化されたメッセージ とキーリングを復号化メソッドに渡します。
3. 復号の方法では、キーリングまたはマスターキープロバイダーに、暗号化されたメッセージの暗号化されたデータキーのいずれかを復号するように要求します。　次に、暗号化されたメッセージから、暗号化されたデータキーなどの情報を渡します。
4. キーリングは、その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ます。成功した場合、レスポンスにはプレーンテキストのデータキーが含まれます。キーリングまたはマスターキープロバイダーによって指定された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化できない場合、復号の呼び出しは失敗します。

5. 復号方法は、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、プレーンテキストのデータキーを破棄します。

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

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、アルゴリズムの実装を使用して、暗号化テキストメッセージを生成します。

AWS Encryption SDK アルゴリズムスイートは、AES-GCM と呼ばれる Galois/Counter Mode (GCM) の 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。は、256 ビット、192 ビット、および 128 ビットの暗号化キー AWS Encryption SDK をサポートしています。初期化ベクトル (IV) の長さは常に 12 バイトです。認証タグの長さは常に 16 バイトです。

デフォルトでは、 は HMAC ベースのextract-and-expandキー取得関数 (HKDF ) 、署名、および 256 ビット暗号化キーを備えた AES-GCM のアルゴリズムスイート AWS Encryption SDK を使用します。コミットメントポリシーにキーコミットメントが必要な場合、 はキーコミットメントもサポートするアルゴリズムスイート AWS Encryption SDK を選択します。それ以外の場合は、キー取得と署名を含むアルゴリズムスイートを選択しますが、キーコミットメントは選択しません。

## 推奨: キー取得、署名、キーコミットメントを使用する AES-GCM

では、256 ビットのデータ暗号化キーを HMAC ベースのextract-and-expandキー取得関数 (HKDF) に提供して AES-GCM 暗号化キーを取得するアルゴリズムスイート AWS Encryption SDK を推奨しています。は、楕円曲線デジタル署名アルゴリズム (ECDSA) 署名 AWS Encryption SDK を追加します。キーコミットメントをサポートするため、このアルゴリズムスイートは、暗号化されたメッセージのメタデータに保存されているキーコミットメント文字列 (シークレット以外のデータキー識別子) も取得します。このキーコミットメント文字列は、データ暗号化キーの取得と同様の手順を使用して HKDF によっても取得されます。

AWS Encryption SDK アルゴリズムスイート

| 暗号化アルゴリズム | データ暗号化キーの長さ (ビット) | キー導出アルゴリズム | 署名アルゴリズム | キーコミットメント |
|---|---|---|---|---|
| AES-GCM | 256 | SHA-384 を使用する HKDF | P-384 および SHA-384 を使用する ECDSA | SHA-512 を使用する HKDF |

HKDF により、データ暗号化キーの誤った再利用を避けて、データキー乱用のリスクを軽減できます。

署名のために、このアルゴリズムスイートは、暗号化ハッシュ関数アルゴリズム (SHA-384) を含む ECDSA を使用します。基盤となるマスターキーのポリシーによって指定されていない場合でも、ECDSA が、デフォルトで使用されます。メッセージ署名では、メッセージの送信者がメッセージを暗号化する権限があることが検証され、非否認が可能になります。これは、マスターキーの承認ポリシーによって、1 組のユーザーにデータを暗号化させ、別の組のユーザーにデータを復号させる場合に特に便利です。

キーコミットメントを使用するアルゴリズムスイートでは、各暗号化テキストが 1 つのプレーンテキストのみに復号化されるようになります。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の ID を検証することによって行います。暗号化時に、これらのアルゴリズムスイートはキーコミットメント文字列を取得します。復号する前には、データキーがキーコミットメント文字列と一致することが検証されます。一致しない場合、復号呼び出しは失敗します。

## サポートされているその他のアルゴリズムスイート

は、下位互換性のために以下の代替アルゴリズムスイート AWS Encryption SDK をサポートしています。一般的に、これらのアルゴリズムスイートはお勧めしていません。ただし、署名がパフォーマンスを大幅に低下させる可能性があることが分かっているため、このようなケースのためにキー取得を使用するキーコミットスイートが提供されています。より重大なパフォーマンスのトレードオフを行う必要があるアプリケーションのために、署名、キーコミットメント、キー取得がないスイートが引き続き提供されます。

キーコミットメントを使用しない AES-GCM

キーコミットメントを使用しないアルゴリズムスイートでは、復号化前にデータキーが検証されません。その結果、これらのアルゴリズムスイートでは、単一の暗号化テキストがさまざまなプ

レーンテキストメッセージに復号化されることがあります。ただし、キーコミットメントを使用するアルゴリズムスイートでは、暗号化されたメッセージがわずかに大きくなって (+30 バイト) 処理に時間がかかるため、すべてのアプリケーションに最適な選択肢ではない場合があります。

は、キー取得、キーコミットメント、署名、およびキー取得とキーコミットメントを含むアルゴリズムスイート AWS Encryption SDK をサポートしますが、署名はサポートしません。キーコミットメントを使用しないアルゴリズムスイートを使用することはお勧めしません。必要な場合は、キー取得とキーコミットメントを使用するが、署名を使用しないアルゴリズムスイートをお勧めします。ただし、アプリケーションパフォーマンスプロファイルがアルゴリズムスイートの使用をサポートしている場合は、キーコミットメント、キー取得、および署名を使用するアルゴリズムスイートを使用することがベストプラクティスです。

署名を使用しない AES-GCM

署名を使用しないアルゴリズムスイートには、信頼性と非否認を提供する ECDSA 署名がありません。これらのスイートは、データを暗号化するユーザーと復号するユーザーが同じほど信頼できる場合に使用します。

署名を使用しないアルゴリズムスイートを使用するときは、キー取得とキーコミットメントを使用するアルゴリズムスイートの選択をお勧めします。

キー取得を使用しない AES-GCM

キー取得を使用しないアルゴリズムスイートは、キー取得関数ではなく、AES-GCM 暗号化キーとしてデータ暗号化キーを使用して、一意のキーを取得します。このスイートを使用して暗号文を生成することはお勧めしませんが、 AWS Encryption SDK は互換性の理由からサポートしています。

これらのスイートのライブラリ内での表示方法と使用方法の詳細については、「the section called “アルゴリズムのリファレンス”」を参照してください。

# AWS Encryption SDK で を使用する AWS KMS

を使用するには AWS Encryption SDK、ラッピングキーを使用してキーリングまたはマスターキープロバイダーを設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。のコード例の多くには、 AWS Encryption SDK が必要ですAWS KMS key。

を操作するには AWS KMS、 に任意のプログラミング言語の AWS SDK AWS Encryption SDK が必要です。 AWS Encryption SDK クライアントライブラリは AWS SDKs、 に保存されているマスターキーをサポートします AWS KMS。

AWS Encryption SDK で 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、 のキー ID または Amazon リソースネーム (ARN) が必要です AWS KMS key。 AWS KMS keyの ID と 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. AWS SDK for Java、、AWS SDK for Python (Boto)または AWS SDK for C++ (C の場合) の手順とAWS SDK for JavaScript、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 AWS を使用して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 AWS Encryption SDK する 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。

5. をダウンロードしてインストールします AWS Encryption SDK。詳細については、使用するプログラミング言語のインストール方法を参照してください。

# のベストプラクティス AWS Encryption SDK

AWS Encryption SDK は、業界標準とベストプラクティスを使用してデータを簡単に保護できるように設計されています。デフォルト値では多くのベストプラクティスが選択されており、一部のプラクティスはオプションですが、実用的であるときは常に推奨されます。

最新バージョンを使用する

の使用を開始するときは AWS Encryption SDK、任意のプログラミング言語で提供されている最新バージョンを使用してください。を使用している場合は AWS Encryption SDK、できるだけ早く各最新バージョンにアップグレードしてください。そうすると、推奨設定を使用し、新しいセキュリティプロパティを利用してデータを保護できるようになります。移行とデプロイのガイダンスなど、サポート対象バージョンの詳細については、「サポートとメンテナンス」と「のバージョン AWS Encryption SDK」を参照してください。

新しいバージョンでコード内の要素が非推奨になった場合は、できるだけ早く置き換えてください。非推奨の警告とコードコメントにより、通常は適切な代替手段が推奨されます。

大幅なアップグレードを容易にし、エラーの発生を抑えるために、一時的または移行的なリリースが提供されることがあります。これらのリリースとそれに付随するドキュメントを使用すると、本番ワークフローを中断することなくアプリケーションをアップグレードできます。

デフォルト値を使用する

は、ベストプラクティスをデフォルト値に AWS Encryption SDK 設計します。可能な限り、デフォルト値を使用してください。デフォルトが実用的でない場合は、署名なしのアルゴリズムスイートなどの代替手段が提供されます。上級ユーザーは、カスタムキーリング、マスターキープロバイダー、暗号化マテリアルマネージャー (CMM) などのカスタマイズも可能です。これらの上級者向けの代替手段は慎重に使用し、可能な限りセキュリティエンジニアがその代替手段を検証してください。

暗号化コンテキストを使用する

暗号化オペレーションのセキュリティを向上させるには、データを暗号化するためのすべてのリクエストに、意味のある値で暗号化コンテキストを含めます。暗号化コンテキストの使用はオプションですが、暗号化のベストプラクティスとして使用することをお勧めします。暗号化コンテキストでは、 AWS Encryption SDKで認証された暗号化に追加認証データ (AAD) が提供されます。暗号化コンテキストはシークレットではありませんが、暗号化データの整合性と真正性を保護できます。

では AWS Encryption SDK、暗号化時にのみ暗号化コンテキストを指定します。復号時に、 は、が AWS Encryption SDK 返す暗号化されたメッセージのヘッダーで暗号化コンテキスト AWS Encryption SDK を使用します。アプリケーションでプレーンテキストのデータを返す前に、メッセージの暗号化に使用した暗号化コンテキストが、メッセージの復号化に使用した暗号化コンテキストに含まれていることを確認します。詳細については、プログラミング言語の例を参照してください。

コマンドラインインターフェイスを使用すると、 AWS Encryption SDK は暗号化コンテキストを検証します。

ラッピングキーを保護する

は、各プレーンテキストメッセージを暗号化するための一意のデータキー AWS Encryption SDK を生成します。次にデータキーは、指定したラッピングキーで暗号化されます。ラッピングキーが失われたり削除されたりすると、暗号化されたデータは回復できません。キーが保護されていない場合、データが脆弱になる可能性があります。

AWS Key Management Service (AWS KMS) など、安全なキーインフラストラクチャで保護されているラッピングキーを使用してください。Raw AES キーまたは Raw RSA キーを使用する場合は、セキュリティ要件を満たすランダム性と耐久性のあるストレージのソースを使用します。ハードウェアセキュリティモジュール (HSM)、または HSMs を提供するサービスでラッピングキーを生成して保存するのがベストプラクティス AWS CloudHSMです。

キーインフラストラクチャの認可メカニズムを使用して、ラッピングキーへのアクセスを、必要とするユーザーのみに制限してください。最小特権などのベストプラクティスの原則を実装します。を使用する場合は AWS KMS keys、ベストプラクティスの原則を実装するキーポリシーと IAM ポリシーを使用します。

ラッピングキーを指定する

復号化時にも暗号化時にも、明示的に ラッピングキーを指定する ことが常にベストプラクティスです。これを行うと、 は指定したキーのみ AWS Encryption SDK を使用します。この方法では、意図した暗号化キーのみを使用することが保証されます。また AWS KMS 、ラッピングキーの場合、別の AWS アカウント またはリージョンでキーを誤って使用したり、使用するアクセス許可のないキーで復号しようとしたりするのを防ぐことで、パフォーマンスが向上します。

暗号化時に、 AWS Encryption SDK が提供するキーリングとマスターキープロバイダーでは、ラッピングキーを指定する必要があります。使用されるのは、ユーザーが指定したすべてのラッピングキーであり、またそれらのみです。RAW AES キーリング、Raw RSA キーリング、および JCEMasterKey で暗号化および復号化する場合も、ラッピングキーを指定する必要があります。

ただし、 AWS KMS キーリングとマスターキープロバイダーで復号する場合、ラッピングキーを指定する必要はありません。は、暗号化されたデータキーのメタデータからキー識別子を取得 AWS Encryption SDK できます。ただし、ベストプラクティスとして、ラッピングキーを指定することをお勧めします。

AWS KMS ラッピングキーを使用する際にこのベストプラクティスをサポートするには、以下をお勧めします。

- ラッピング AWS KMS キーを指定するキーリングを使用します。暗号化および復号化を行う場合、これらのキーリングでは、指定したラッピングキーのみが使用されます。
- AWS KMS マスターキーとマスターキープロバイダーを使用する場合は、 のバージョン 1.7.xで導入された strict モードコンストラクタを使用します AWS Encryption SDK。指定したラッピングキーでのみ暗号化および復号化するプロバイダーが作成されます。ラッピングキーで常に復号化するマスターキープロバイダーのコンストラクタは、バージョン 1.7.x で非推奨となり、バージョン 2.0.x で削除されました。

復号化のために AWS KMS ラッピングキーを指定するのが実用的でない場合には、検出プロバイダーを使用できます。C および JavaScript AWS Encryption SDK の AWS KMS は検出キーリングをサポートします。検出モードのマスターキープロバイダーは、バージョン 1.7.x 以降の Java および Python で使用できます。これらの検出プロバイダーは、 AWS KMS ラッピングキーによる復号にのみ使用され、データキーを暗号化したラッピングキーを使用する AWS Encryption SDK ように に明示的に指示します。

検出プロバイダーを使用する必要がある場合は、検出フィルター機能を使用して、使用するラッピングキーを制限します。例えば、AWS KMS リージョン検出キーリングでは、特定の AWS リージョンのラッピングキーのみを使用します。特定の のラッピング AWS KMS キーのみを使用するようにキーリングと AWS KMS マスターキープロバイダーを設定することもできます AWS アカウント。 ???また、通常どおり、キーポリシーと IAM ポリシーを使用してラ AWS KMS ッピングキーへのアクセスを制御します。

デジタル署名を使用する

アルゴリズムスイートを署名とともに使用することがベストプラクティスです。デジタル署名では、メッセージ送信者がメッセージを送信する権限があることが確認され、メッセージの整合性が保護されます。のすべてのバージョンでは、デフォルトで署名付きのアルゴリズムスイート AWS Encryption SDK が使用されます。

セキュリティ要件にデジタル署名が含まれていない場合は、デジタル署名なしのアルゴリズムスイートを選択できます。ただし、あるユーザーのグループがデータを暗号化し、別のユーザーグループがそのデータを復号化する場合は特に、デジタル署名の使用をお勧めします。

キーコミットメントを使用する

キーコミットメントセキュリティ機能を使用することがベストプラクティスです。キーコミットメント では、データを暗号化した一意の データキー の ID が確認され、暗号文を復号化して複数のプレーンテキストメッセージが生成されることが防止されます。

AWS Encryption SDK は、バージョン 2.0.x 以降のキーコミットメントによる暗号化と復号化を完全にサポートします。デフォルトでは、すべてのメッセージの暗号化と復号化はキーコミットメントで行われます。のバージョン 1.7.x AWS Encryption SDK では、キーコミットメントを使用して暗号文を復号できます。前バージョンのユーザーでも、バージョン 2.0.x を正常にデプロイできるように設計されています。

キーコミットメントでは新しいアルゴリズムスイートおよび新しいメッセージ形式がサポートされ、キーコミットメントを使用しない暗号化テキストと比較してわずか 30 バイトだけ大きい暗号化テキストを生成できます。この設計により、パフォーマンスへの影響が最小限に抑えられるため、ほとんどのユーザーはキーコミットメントの利点を享受できます。アプリケーションがサイズとパフォーマンスに非常に敏感である場合は、コミットメントポリシー設定を使用してキーコミットメントを無効にするか、コミットメントなしでメッセージを復号 AWS Encryption SDK することを に許可しますが、必要な場合にのみ許可します。

暗号化されたデータキーの数を制限する

復号化するメッセージ、特に信頼できないソースからのメッセージでは、暗号化されたデータキーの数を制限する ことがベストプラクティスです。多数の暗号化されたデータキーでメッセージを復号化すると、復号化できない場合に、遅延の延長、コストの拡大、アプリケーションやアカウントを共有する他のユーザーの制限が発生し、キーインフラストラクチャを使い果たす可能性があります。制限がない場合、暗号化されたメッセージには最大 65,535 (2^16 - 1) の暗号化されたデータキーを使用できます。詳細については、「暗号化されたデータキーの制限」を参照してください。

これらのベストプラクティスの基礎となる AWS Encryption SDK セキュリティ機能の詳細については、 AWS セキュリティブログの「クライアント側の暗号化の改善: 明示的な KeyIds とキーコミットメント」を参照してください。

# の設定 AWS Encryption SDK

AWS Encryption SDK は使いやすいように設計されています。 AWS 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。

実装を設定するときは、 AWS Encryption SDK ベストプラクティスを確認し、できるだけ多くの を実装してください。

トピック

- プログラミング言語の選択
- ラッピングキーの選択
- マルチリージョンの使用 AWS KMS keys
- アルゴリズムスイートを選択する
- 暗号化されたデータキーの制限
- 検出フィルターの作成
- 必要な暗号化コンテキスト CMM の設定
- コミットメントポリシーの設定
- ストリーミングデータの操作
- データキーのキャッシュ

## プログラミング言語の選択

AWS Encryption SDK 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。ただし、特定の実装用にプログラミング言語を選択することもできます。たとえば、キーリングを使用する場合は、 AWS Encryption SDK for C または を選択できます AWS Encryption SDK for JavaScript。

# ラッピングキーの選択

は、各メッセージを暗号化するための一意の対称データキー AWS Encryption SDK を生成します。データキーキャッシュを使用していない場合は、データキーの設定、管理、使用は必要ありません。 AWS Encryption SDK によって自動的に実行されます。

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 Encryption SDK では、さまざまなサイズの AES 対称キーと RSA 非対称キーがサポートされます。AWS Key Management Service (AWS KMS) 対称暗号化 AWS KMS keysもサポートされます。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または 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。

暗号化と復号のラッピングキーを指定するには、キーリング (C、Java、JavaScript、.NET、Python) またはマスターキープロバイダー (Java、Python、 AWS Encryption CLI) を使用します。同じタイプまたは異なるタイプのラッピングキーを 1 つまたは複数指定できます。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、 が AWS Encryption SDK 返す暗号化されたメッセージに暗号化されたデータとともに保存されます。データを復号するには、 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 AWS Encryption SDK する必要があります。

キーリングまたはマスターキープロバイダー 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。キーの AWS KMS 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。

- AWS Encryption SDK for Java、 AWS Encryption SDK for JavaScript AWS Encryption SDK for Python、または AWS Encryption CLI を使用して暗号化する場合、KMS キーに任意の有効なキー識別子 (キー ID、キー ARN、エイリアス名、またはエイリアス ARN) を使用できます。で暗号化する場合 AWS Encryption SDK for C、キー ID またはキー ARN のみを使用できます。

  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 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。
- Strict モード (特定のラッピングキーを指定するモード) で復号する場合は、キー ARN を使用して AWS KMS keysを指定する必要があります。この要件は、 AWS Encryption SDKのすべての言語の実装に適用されます。

AWS KMS キーリングで暗号化すると、 は暗号化されたデータキーのメタデータ AWS KMS key に のキー ARN AWS Encryption SDK を保存します。strict モードで復号する場合、 AWS 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 (またはマスターキープロバイダー) 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 Encryption SDK は を認識または使用しません。

キーリング内のラッピングキーとして Raw AES キー または raw RSA キーペア を指定するには、名前空間と名前を指定する必要があります。マスターキープロバイダーでは、`Provider ID` が名前空間と同等であり、`Key ID` は名前に相当します。復号化する際には、暗号化時に使用したものとまったく同じ名前空間と名前を、各 raw ラッピングキーに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、キーマテリアルが同じであっても、 AWS Encryption SDK はラッピングキーを認識または使用しません。

## マルチリージョンの使用 AWS KMS keys

AWS Key Management Service （ AWS KMS) マルチリージョンキーは、 のラッピングキーとして使用できます AWS Encryption SDK。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。マルチリージョンキーのサポートは、 のバージョン 2.3.x AWS Encryption SDK および AWS Encryption CLI のバージョン 3.0.x で導入されています。

AWS KMS マルチリージョンキーは、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、 には AWS KMS multi-Region-awareキーリングとマスターキープロバイダー AWS Encryption SDK が含まれています。各プログラミング言語の新しいマルチリージョン対応シンボルでは、単一リージョンキーとマルチリージョンキーの両方がサポートされます。

- 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 キーリングとマスターキープロバイダー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。
- マルチリージョンキーの場合、multi-Region-aware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レプリカ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およびマスターキープロバイダーでは、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レプリカキーの各セットから指定できるキーは 1 つだけです。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。

マルチリージョンキーは、標準の単一リージョン AWS KMS キーリングとマスターキープロバイダーでも使用できます。ただし、暗号化と復号には、同じリージョンで同じマルチリージョンキーを使用する必要があります。単一リージョンキーリングとマスターキープロバイダーは、データを暗号化したキーを使用してのみ、暗号化テキストを復号しようとします。

次の例は、マルチリージョンキーおよび新しいマルチリージョン対応キーリングとマスターキープロバイダーを使用して、データを暗号化および復号化する方法を示しています。これらの例では、us-east-1リージョン内のデータを暗号化し、各us-west-2リージョンの関連するマルチリージョンレプリカキーを使用してリージョン内のデータを復号します。これらの例を実行する前に、マルチリージョンキー ARN の例を、自分の AWS アカウントで有効な値に置き換えてください。

C

マルチリージョンキーを使用して暗号化するには、`Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` メソッドを使用してキーリングをインスタンス化します。マルチリージョンキーを指定してください。

この簡単な例には、暗号化コンテキストが含まれていません。C で暗号化コンテキストを使用する例については、「文字列の暗号化と復号」を参照してください。

完全な例については、GitHub の AWS Encryption SDK for C リポジトリの「kms_multi_region_keys.cpp」を参照してください。

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

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

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

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

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

aws_cryptosdk_keyring_release(mrk_keyring);

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

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

C# / .NET

米国東部 (バージニア北部) (us-east-1) リージョンのマルチリージョンキーで暗号化するには、マルチリージョンキーのキー識別子と指定されたリージョンの AWS KMS クライアントを使用してCreateAwsKmsMrkKeyringInputオブジェクトをインスタンス化します。次に、CreateAwsKmsMrkKeyring() メソッドを使用してキーリングを作成します。

この CreateAwsKmsMrkKeyring() メソッドは、正確に 1 つのマルチリージョンキーを持つキーリングを作成します。マルチリージョンキーを含む複数のラッピングキーを使用して暗号化するには、CreateAwsKmsMrkMultiKeyring() メソッドを使用します。

完全な例については、GitHub の for .NET リポジトリのAwsKmsMrkKeyringExample.cs AWS Encryption SDK 」を参照してください。

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

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

```
 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();


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

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

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

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

### AWS Encryption CLI

この例では、us-east-1 リージョンにおいてマルチリージョンキーで hello.txt ファイルを暗号化します。この例ではリージョン要素を持つキー ARN を指定しているため、この例では、--wrapping-keys パラメータの region 属性を使用しません。

ラッピングキーのキー ID がリージョンを指定しない場合は、--wrapping-keys の region 属性を使用して --wrapping-keys key=$keyID region=us-east-1 などのリージョンを指定します。

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

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

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

Java

マルチリージョンキーで暗号化するには、AwsKmsMrkAwareMasterKeyProvider をインスタンス化し、マルチリージョンキーを指定します。

完全な例については、GitHub BasicMultiRegionKeyEncryptionExample.javaの AWS Encryption SDK for Java リポジトリの「」を参照してください。

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

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

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

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

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

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

JavaScript Browser

マルチリージョンキーを使用して暗号化するには、buildAwsKmsMrkAwareStrictMultiKeyringBrowser() メソッドを使用してキーリングを作成し、マルチリージョンキーを指定します。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_multi_region_simple.ts」を参照してください。

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

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


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

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

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

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

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


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

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

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

JavaScript Node.js

マルチリージョンキーを使用して暗号化するには、buildAwsKmsMrkAwareStrictMultiKeyringNode() メソッドを使用してキーリングを作成し、マルチリージョンキーを指定します。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_multi_region_simple.ts」を参照してください。

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

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

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

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

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

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

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

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

Python

AWS KMS マルチリージョンキーで暗号化するには、`MRKAwareStrictAwsKmsMasterKeyProvider()`メソッドを使用してマルチリージョンキーを指定します。

完全な例については、GitHub の AWS Encryption SDK for Python リポジトリの「mrk_aware_kms_provider.py」を参照してください。

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

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

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

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

# Set the encryption context
encryption_context = {
        "purpose": "test"
    }

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

次に、暗号化テキストを us-west-2 リージョンに移動します。暗号化テキストを再暗号化する必要はありません。

us-west-2 リージョンにおいて Strict モードで暗号化テキストを復号するには、us-west-2 リージョンにおいて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 (暗号化した us-east-1 を含む) 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行います。

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。

これらの例を実行する前に、マルチリージョンキー ARN の例を の有効な値に置き換えます AWS アカウント。

C

マルチリージョンキーを使用して Strict モードで暗号化するには、Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder() メソッドを使用してキーリングをインスタンス化します。ローカル (us-west-2) リージョンで、関連するマルチリージョンキーを指定します。

完全な例については、 AWS Encryption SDK for C GitHub のリポジトリの「kms_multi_region_keys.cpp」を参照してください。

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

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

/* Initialize a multi-Region keyring */
const char *mrk_us_west_2 = "arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

struct aws_cryptosdk_keyring *mrk_keyring =
    Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder().Build(mrk_us_west_2);

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

aws_cryptosdk_session_set_commitment_policy(session,
    COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT);

aws_cryptosdk_keyring_release(mrk_keyring);

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

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

C# / .NET

1 つのマルチリージョンキーを使用して Strict モードで復号するには、入力の組立や暗号化用のキーリングの作成に使用したものと同じコンストラクターとメソッドを使用します。関連するマルチリージョンキーのキー ARN と、米国西部 (オレゴン) (us-west-2) リージョンの AWS KMS クライアントを使用してCreateAwsKmsMrkKeyringInputオブジェクトをインスタンス化し

ます。次に、この CreateAwsKmsMrkKeyring() メソッドを使用して 1 つのマルチリージョン KMS キーでマルチリージョンキーリングを作成します。

完全な例については、GitHub の AwsKmsMrkKeyringExample.cs AWS Encryption SDK 」を参照してください。

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

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Specify the key ARN of the multi-Region key in us-west-2
string mrkUSWest2 = "arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

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

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

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

### AWS Encryption CLI

us-west-2 リージョンで関連するマルチリージョンキーを使用して復号するには、--wrapping-keys パラメータの key 属性を使用してキー ARN を指定します。

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

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

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

Java

Strict モードで復号するには、AwsKmsMrkAwareMasterKeyProvider をインスタンス化し、ローカル (us-west-2) リージョンで関連するマルチリージョンキーを指定します。

詳しい例については、GitHub の AWS Encryption SDK for Java リポジトリの「BasicMultiRegionKeyEncryptionExample.java」を参照してください。

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

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

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

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

// Decrypt your ciphertext
```

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

### JavaScript Browser

Strict モードで復号するには、buildAwsKmsMrkAwareStrictMultiKeyringBrowser() メソッドを使用してキーリングを作成し、ローカル (us-west-2) リージョンで関連するマルチリージョンキーを指定します。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_multi_region_simple.ts」を参照してください。

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

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


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

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

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


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

```
    'arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

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


/* Decrypt the data */
const { plaintext, messageHeader } = await decrypt(mrkDecryptKeyring, result)
```

JavaScript Node.js

Strict モードで復号するには、buildAwsKmsMrkAwareStrictMultiKeyringNode( ) メソッドを使用してキーリングを作成し、ローカル (us-west-2) リージョンで関連するマルチリージョンキーを指定します。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_multi_region_simple.ts」を参照してください。

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

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

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

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

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

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

Python

Strict モードで復号化するには、`MRKAwareStrictAwsKmsMasterKeyProvider()` メソッドを使用して、マスターキープロバイダーを作成します。ローカル (us-west-2) リージョンで、関連するマルチリージョンキーを指定します。

完全な例については、GitHub の AWS Encryption SDK for Python リポジトリの「mrk_aware_kms_provider.py」を参照してください。

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

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

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

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

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

AWS KMS 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 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。

> **Note**
>
> 検出モードでマルチリージョン対応シンボルを使用してデータを暗号化すると、暗号化操作に失敗します。

次の例では、検出モードでマルチリージョン対応シンボルを使用して復号する方法を示します。を指定しないため AWS KMS key、 は別のソースからリージョンを取得 AWS Encryption SDK する必要があります。可能であれば、ローカルリージョンを明示的に指定します。それ以外の場合、 はプログラミング言語の AWS SDK で設定されたリージョンからローカルリージョン AWS Encryption SDK を取得します。

これらの例を実行する前に、サンプルアカウント ID とマルチリージョンキー ARN を の有効な値に置き換えます AWS アカウント。

C

マルチリージョンキーを使用して検出モードで復号するには、キーリングを構築する `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` メソッドと検出フィルターを構築する `Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder()` メソッドを使用します。ローカルリージョンを指定するには、`ClientConfiguration` を定義して AWS KMS クライアントにそれを指定します。

完全な例については、 AWS Encryption SDK for C GitHub のリポジトリの「kms_multi_region_keys.cpp」を参照してください。

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

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

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

 Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder(partition).AddAccount(account_id).Buil
```

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

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

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

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

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

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

C# / .NET

for AWS Encryption SDK .NET でmulti-Region-awareの検出キーリングを作成するには、特定の の AWS KMS クライアントを取得するCreateAwsKmsMrkDiscoveryKeyringInputオブジェクトと AWS リージョン、KMS キーを特定の AWS パーティションとアカウントに制限するオプションの検出フィルターをインスタンス化します。次に、入力オブジェクトを使用して CreateAwsKmsMrkDiscoveryKeyring() メソッドを呼び出します。完全な例については、GitHub の for .NET リポジトリのAwsKmsMrkDiscoveryKeyringExample.cs AWS Encryption SDK 」を参照してください。

マルチリージョン対応ディスカバリーキーリングを複数 AWS リージョンで作成するには、CreateAwsKmsMrkDiscoveryMultiKeyring() メソッドを使用してマルチキーリングを

作成するか、CreateAwsKmsMrkDiscoveryKeyring() を使用して複数のマルチリージョン対応の検出キーリングを作成し、その CreateMultiKeyring() メソッドを使用してそれらをマルチキーリングに結合します。

例については、「AwsKmsMrkDiscoveryMultiKeyringExample.cs」を参照してください。

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

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

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

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

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


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

AWS Encryption CLI

検出モードで復号するには、`--wrapping-keys` パラメータの discovery 属性を使用します。discovery-account 属性と discovery-partition 属性により、検出フィルタが作成されます。この検出フィルタはオプションですが、推奨されています。

リージョンを指定するには、このコマンドに `--wrapping-keys` パラメータの region 属性を含めます。

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

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

Java

ローカルリージョンを指定するには、`builder().withDiscoveryMrkRegion` パラメータを使用します。そうでない場合、 AWS Encryption SDK は、AWS SDK for Java で設定されたリージョンからローカルリージョンを取得します。

詳しい例については、「GitHub の AWS Encryption SDK for Java リポジトリ」の「DiscoveryMultiRegionDecryptionExample.java」を参照してください。

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

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

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

AwsKmsMrkAwareMasterKeyProvider mrkDiscoveryProvider =
 AwsKmsMrkAwareMasterKeyProvider
```

```
    .builder()
    .withDiscoveryMrkRegion(Region.US_WEST_2)
    .buildDiscovery(discoveryFilter);

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

### JavaScript Browser

対称マルチリージョンキーを使用して検出モードで復号するには、`AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser()` メソッドを使用します。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_multi_region_discovery.ts」を参照してください。

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

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


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

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

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


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

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


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

JavaScript Node.js

対称マルチリージョンキーを使用して検出モードで復号するには、`AwsKmsMrkAwareSymmetricDiscoveryKeyringNode()` メソッドを使用します。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_multi_region_discovery.ts」を参照してください。

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

import {
  AwsKmsMrkAwareSymmetricDiscoveryKeyringNode,
  buildClient,
  CommitmentPolicy,
  KMS,
} from '@aws-crypto/client-node'

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

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

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

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

/* Decrypt your ciphertext */
```

```
const { plaintext, messageHeader } = await decrypt(mrkDiscoveryKeyring, result)
```

Python

マルチリージョンキーを使用して検出モードで復号するには、`MRKAwareDiscoveryAwsKmsMasterKeyProvider()` メソッドを使用します。

完全な例については、GitHub の AWS Encryption SDK for Python リポジトリの「mrk_aware_kms_provider.py」を参照してください。

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

# Instantiate the client
client = aws_encryption_sdk.EncryptionSDKClient()

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

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

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

# アルゴリズムスイートを選択する

AWS Encryption SDK は、指定したラッピングキーでデータキーを暗号化するための対称暗号化アルゴリズムと非対称暗号化アルゴリズムをサポートしています。ただし、これらのデータキーを使用してデータを暗号化する場合、 AWS Encryption SDK はデフォルトで、キー取得、デジタル署名、およびキーコミットメントで AES-GCM アルゴリズムを使用する推奨アルゴリズムスイートになります。 ???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 Encryption SDK でサポートされるアルゴリズムスイートについては、「でサポートされているアルゴリズムスイート AWS Encryption SDK」を参照してください。

以下の例では、暗号化時に代替アルゴリズムスイートを選択する方法を示します。これらの例では、キー取得とキーコミットメントを含むがデジタル署名を含まない推奨 AES-GCM アルゴリズムスイートを選択します。デジタル署名を含まないアルゴリズムスイートで暗号化する場合は、復号時に署名なし専用の復号化モードを使用します。このモードは、署名付き暗号化テキストを検出すると失敗し、ストリーミング復号化時に最も役立ちます。

C

で代替アルゴリズムスイートを指定するには AWS Encryption SDK for C、CMM を明示的に作成する必要があります。次に、CMM および選択したアルゴリズムスイートで `aws_cryptosdk_default_cmm_set_alg_id` を使用します。

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

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

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

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

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

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

```
/* Encrypt the data
   Use aws_cryptosdk_session_process_full with non-streaming data
 */
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(
                          session,
                          ciphertext,
                          ciphertext_buf_sz,
                          &ciphertext_len,
                          plaintext,
                          plaintext_len)) {
    aws_cryptosdk_session_destroy(session);
    return AWS_OP_ERR;
}
```

デジタル署名なしで暗号化されたデータを復号化する場合は、AWS_CRYPTOSDK_DECRYPT_UNSIGNED を使用します。署名付き暗号化テキストが検出されると、復号は失敗します。

```
/* Decrypt unsigned streaming data */

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

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

/* Create a session for decrypting with the AWS KMS keyring
   Then release the keyring reference
 */
struct aws_cryptosdk_session *session =

 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT_UNSIGNED,
 kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);

if (!session) {
    return AWS_OP_ERR;
}

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

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

C# / .NET

.NET AWS Encryption SDK の で代替アルゴリズムスイートを指定するには、EncryptInput オブジェクトの AlgorithmSuiteIdプロパティを指定します。 AWS Encryption SDK for .NET には、任意のアルゴリズムスイートを識別するために使用できる定数が含まれています。

AWS Encryption SDK for .NET には、ストリーミング復号時に署名付き暗号文を検出するメソッドはありません。このライブラリはストリーミングデータをサポートしていないためです。

```
// Specify an algorithm suite without signing

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

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

// Encrypt your plaintext data
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
```

```
    Keyring = keyring,
    AlgorithmSuiteId = AlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

### AWS Encryption CLI

hello.txt ファイルを暗号化するときには、--algorithm パラメータを使用して、デジタル署名のないアルゴリズムスイートを指定します。

```
# Specify an algorithm suite without signing

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

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

復号するとき、この例では --decrypt-unsigned パラメータを使用します。特に入力と出力を常にストリーミングする CLI で署名なし暗号化テキストを復号化するためには、このパラメータが推奨されます。

```
# Decrypt unsigned streaming data

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

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

Java

代替アルゴリズムスイートを指定するには、AwsCrypto.builder().withEncryptionAlgorithm() メソッドを使用します。この例では、デジタル署名のない代替アルゴリズムスイートを指定します。

```
// Specify an algorithm suite without signing

// Instantiate the client
AwsCrypto crypto = AwsCrypto.builder()
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
    .build();

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

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

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

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

復号化のためにデータをストリーミングする場合は、createUnsignedMessageDecryptingStream() メソッドを使用し、復号しているすべての暗号化テキストが署名なしであることを保証します。

```
// Decrypt unsigned streaming data

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

```
  .build();

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

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

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

### JavaScript Browser

代替アルゴリズムスイートを指定するには、AlgorithmSuiteIdentifier 列挙値を含む suiteId パラメータを使用します。

```
// Specify an algorithm suite without signing

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

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

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

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

復号化するときは、標準の `decrypt` メソッドを使用します。ブラウザの AWS Encryption SDK for JavaScript には `decrypt-unsigned` モードがありません。ブラウザでストリーミングがサポートされないためです。

```
// Decrypt unsigned streaming data

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

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

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

JavaScript Node.js

代替アルゴリズムスイートを指定するには、`AlgorithmSuiteIdentifier` 列挙値を含む `suiteId` パラメータを使用します。

```
// Specify an algorithm suite without signing

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

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

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

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

デジタル署名なしで暗号化されたデータを復号化する場合は、decryptUnsignedMessageStream を使用します。このメソッドは、署名付き暗号化テキストを検出すると失敗します。

```
// Decrypt unsigned streaming data

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

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

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

Python

代替暗号化アルゴリズムを指定するには、Algorithm 列挙値を含む algorithm パラメータを使用します。

```
# Specify an algorithm suite without signing

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

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

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

デジタル署名なしで暗号化されたメッセージを復号化する場合、特にストリーミングしながら復号化する場合は、decrypt-unsigned ストリーミングモードを使用します。

```
# Decrypt unsigned streaming data

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

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

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

# Verify that the encryption context
assert all(
   pair in decryptor.header.encryption_context.items() for pair in
 encryptor.header.encryption_context.items()
)
return ciphertext_filename, cycled_plaintext_filename
```

Rust

Rust AWS Encryption SDK の で代替アルゴリズムスイートを指定するには、暗号化リクエストで `algorithm_suite_id`プロパティを指定します。

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

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

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

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

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

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

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

### Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
```

```
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
 client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
// Instantiate the AWS Encryption SDK client
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

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

# 暗号化されたデータキーの制限

暗号化されたメッセージ内の暗号化されたデータキーの数を制限できます。このベストプラクティス機能は、暗号化時に誤って構成されたキーリングを検出したり、復号時に悪意のある暗号化テキストを検出したりするのに役立ちます。不必要でコストがかかり、潜在的に網羅的な方法によって、キーインフラストラクチャを呼び出すことも防止できます。信頼できない送信元からのメッセージを復号する場合は、暗号化されたデータキーを制限することが最も重要です。

ほとんどの暗号化されたメッセージには、暗号化で使用されるラッピングキーごとに 1 つの暗号化されたデータキーがありますが、暗号化されたメッセージには最大 65,535 個の暗号化されたデータキーを含めることができます。悪意のあるアクターは、何千もの暗号化されたデータキーを使用して暗号化されたメッセージを構築し、いずれも復号できなくする可能性があります。その結果、AWS Encryption SDK は、メッセージ内の暗号化されたデータキーを使い果たすまで、暗号化された各データキーの復号を試みます。

暗号化されたデータキーを制限するには、MaxEncryptedDataKeys パラメータを使用します。このパラメータは、 AWS Encryption SDKのバージョン 1.9.x および 2.2.x 以降のすべてのサポート対象プログラミング言語で使用できます。これはオプションで、暗号化時および復号時に有効です。次の例では、3 つの異なるラッピングキーで暗号化されたデータを復号化します。MaxEncryptedDataKeys の値は 3 に設定します。

C

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

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

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

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

/* Decrypt */
size_t ciphertext_consumed_output;
aws_cryptosdk_session_process(session,
    plaintext_output,
    plaintext_buf_sz_output,
    &plaintext_len_output,
    ciphertext_input,
    ciphertext_len_input,
    &ciphertext_consumed_output);
assert(aws_cryptosdk_session_is_done(session));
assert(ciphertext_consumed == ciphertext_len);
```

C# / .NET

for AWS Encryption SDK .NET で暗号化されたデータキーを制限するには、 AWS Encryption SDK for .NET のクライアントをインスタンス化し、オプションの `MaxEncryptedDataKeys`パラメータを目的の値に設定します。次に、設定した AWS Encryption SDK インスタンスで `Decrypt()` メソッドを呼び出します。

```
// Decrypt with limited data keys

// Instantiate the material providers
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{
    MaxEncryptedDataKeys = 3
```

```
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

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

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

### AWS Encryption CLI

```
# Decrypt with limited encrypted data keys

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

### Java

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

// Create an AWS KMS master key provider
final KmsMasterKeyProvider keyProvider = KmsMasterKeyProvider.builder()
```

```
      .buildStrict(keyArn1, keyArn2, keyArn3);

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

### JavaScript Browser

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

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

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

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

### JavaScript Node.js

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

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

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

### Python

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

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

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

### Rust

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

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

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

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

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

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

    let raw_aes_keyring = mpl
        .create_raw_aes_keyring()
        .key_name(key_name)
        .key_namespace(key_namespace)
```

```
        .wrapping_key(aes_key_bytes)
        .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
        .send()
        .await?;

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

// Create a Multi Keyring with `max_encrypted_data_keys` AES Keyrings
let generator_keyring = raw_aes_keyrings.remove(0);

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

Go

```
import (
    "context"

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

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

// Define the key namespace and key name
```

```
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

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

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

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

# 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` |

このセクションの例では、検出フィルターの作成方法を示します。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。

C

詳しい例については、「 AWS Encryption SDK for C」の「kms_discovery.cpp」を参照してください。

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

const char *account_id = "111122223333";
const char *partition = "aws";
const std::shared_ptr<Aws::Cryptosdk::KmsKeyring::DiscoveryFilter> discovery_filter
 =

 Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder(partition).AddAccount(account_id).Buil
```

C# / .NET

完全な例については、 for .NET のDiscoveryFilterExample.cs AWS Encryption SDK 」を参照してください。

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

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

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

AWS Encryption CLI

```
# Decrypt in discovery mode with a discovery filter

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

Java

詳しい例については、「 AWS Encryption SDK for Java」の「DiscoveryDecryptionExample.java」を参照してください。

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

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

JavaScript (Node and Browser)

詳細な例については、「 AWS Encryption SDK for JavaScript」の「kms_filtered_discovery.ts」(Node.js) と「kms_multi_region_discovery.ts」(ブラウザ) を参照してください。

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

Python

詳しい例については、「 AWS Encryption SDK for Python」の「discovery_kms_provider.py」を参照してください。

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

Rust

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

Go

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

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

# 必要な暗号化コンテキスト CMM の設定

必要な暗号化コンテキスト CMM を使用して、暗号化オペレーションで暗号化コンテキストを要求できます。暗号化コンテキストは、一連の非シークレットのキーと値のペアです。暗号化コンテキストは、暗号化されたデータ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。必要な暗号化コンテキスト CMM を使用する場合、すべての暗号化および復号化の呼び出しに含める必要のある 暗号化コンテキストキー (必須キー) を 1 つ以上指定できます。

> **Note**
>
> 必要な暗号化コンテキスト CMM は、次のバージョンでのみサポートされています。
>
> - のバージョン 3.x AWS Encryption SDK for Java
> - for .NET のバージョン AWS Encryption SDK 4.x
> - オプションの Cryptographic Material Providers Library (MPL) 依存関係とともに使用する場合の AWS Encryption SDK for Pythonのバージョン 4.x。
> - AWS Encryption SDK for Go のバージョン 0.1.x 以降
>
> 必要な暗号化コンテキスト CMM を使用してデータを暗号化する場合、これらのサポートされているバージョンのいずれかでのみ復号できます。

暗号化時に、 AWS Encryption SDK は、必要なすべての暗号化コンテキストキーが、指定した暗号化コンテキストに含まれていることを確認します。は、指定した暗号化コンテキスト AWS Encryption SDK に署名します。必須キーではないキーと値のペアのみがシリアル化され、暗号化操作によって返される暗号化メッセージのヘッダーにプレーンテキストで保存されます。

復号化時には、必要なキーを表すすべてのキーと値のペアを含む暗号化コンテキストを提供する必要があります。 AWS Encryption SDK は、この暗号化コンテキストと、暗号化されたメッセージのヘッダーに保存されているキーと値のペアを使用して、暗号化オペレーションで指定した元の暗号化コンテキストを再構築します。 AWS Encryption SDK が元の暗号化コンテキストを再構築できない場合、復号化操作は失敗します。誤った値を持つ必要なキーを含むキーと値のペアを供給すると、暗

号化されたメッセージは復号化できません。暗号化時に指定したのと同じキーと値のペアを供給する必要があります。

> **Important**
>
> 暗号化のコンテキストで必要なキーにどの値を選択するかを慎重に検討してください。復号化時には、同じキーとそれに対応する値を再度提供できる必要があります。必要なキーを再現できない場合、暗号化されたメッセージは復号化できません。

次の例では、必要な暗号化コンテキスト CMM を使用して AWS KMS キーリングを初期化します。

C# / .NET

```
var encryptionContext = new Dictionary<string, string>()
{
    {"encryption", "context"},
    {"is not", "secret"},
    {"but adds", "useful metadata"},
    {"that can help you", "be confident that"},
    {"the data you are handling", "is what you think it is"}
};

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

// Instantiate the keyring input object
var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = kmsKey
};

// Create the keyring
var kmsKeyring = mpl.CreateAwsKmsKeyring(createKeyringInput);

var createCMMInput = new CreateRequiredEncryptionContextCMMInput
{
    UnderlyingCMM = mpl.CreateDefaultCryptographicMaterialsManager(new
 CreateDefaultCryptographicMaterialsManagerInput{Keyring = kmsKeyring}),
```

```
    // If you pass in a keyring but no underlying cmm, it will result in a failure
 because only cmm is supported.
    RequiredEncryptionContextKeys = new List<string>(encryptionContext.Keys)
};

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

Java

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

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

// Create a list of required encryption contexts
final List<String> requiredEncryptionContextKeys = Arrays.asList("encryption",
 "context");

// Create the keyring
final MaterialProviders materialProviders = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsKeyringInput keyringInput = CreateAwsKmsKeyringInput.builder()
        .kmsKeyId(keyArn)
        .kmsClient(KmsClient.create())
        .build();
IKeyring kmsKeyring = materialProviders.CreateAwsKmsKeyring(keyringInput);

// Create the required encryption context CMM
ICryptographicMaterialsManager cmm =
    materialProviders.CreateDefaultCryptographicMaterialsManager(
        CreateDefaultCryptographicMaterialsManagerInput.builder()
            .keyring(kmsKeyring)
            .build()
```

```
    );
ICryptographicMaterialsManager requiredCMM =
    materialProviders.CreateRequiredEncryptionContextCMM(
        CreateRequiredEncryptionContextCMMInput.builder()
            .requiredEncryptionContextKeys(requiredEncryptionContextKeys)
            .underlyingCMM(cmm)
            .build()
        );
```

Python

必要な暗号化コンテキスト CMM AWS Encryption SDK for Python で を使用するには、マテリアルプロバイダーライブラリ (MPL) も使用する必要があります。

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

# Create your encryption context
encryption_context: Dict[str, str] = {
    "key1": "value1",
    "key2": "value2",
    "requiredKey1": "requiredValue1",
    "requiredKey2": "requiredValue2"
}

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

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

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

# Create the required encryption context CMM
```

```
underlying_cmm: ICryptographicMaterialsManager = \
    mpl.create_default_cryptographic_materials_manager(
        CreateDefaultCryptographicMaterialsManagerInput(
            keyring=kms_keyring
        )
    )

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

### Rust

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

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

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

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

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

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

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

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

let required_ec_cmm = mpl
    .create_required_encryption_context_cmm()
    .underlying_cmm(underlying_cmm.clone())
    .required_encryption_context_keys(required_encryption_context_keys)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

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

// Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

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

```
    panic(err)
}

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

# コミットメントポリシーの設定

コミットメントポリシーは、アプリケーションがキーコミットメントで暗号化および復号化を行うかどうかを決定する構成設定です。キーコミットメントによる暗号化と復号化は、AWS Encryption SDK のベストプラクティスです。

コミットメントポリシーの設定と調整は、 AWS Encryption SDK のバージョン 1.7.x 以前からバージョン 2.0.x 以降に移行する上で重要なステップです。この進行状況については、「移行のトピック」で詳しく説明されています。

AWS Encryption SDK の最新バージョン (バージョン 2.0.x 以降)、`RequireEncryptRequireDecrypt` におけるデフォルトのコミットメントポリシー値は、ほとんどの状況に最適です。ただし、キーコミットなしで暗号化された暗号文を復号する必要がある場合は、コミットメントポリシーを `RequireEncryptAllowDecrypt` に変更する必要がある場合があります。各プログラミング言語でコミットメントポリシーを設定する方法の例については、「コミットメントポリシーの設定」を参照してください。

# ストリーミングデータの操作

復号のためにデータをストリーミングするときは、整合性チェックが完了した後、デジタル署名が検証される前に、 が復号されたプレーンテキストを AWS Encryption SDK 返すことに注意してください。署名が検証されるまでプレーンテキストを返したり使用したりしないようにするには、復号化プロセス全体が完了するまで、ストリーミングされたプレーンテキストをバッファリングすることをお勧めします。

この問題が発生するのは、復号化のために暗号化テキストをストリーミングしているときに、デフォルトのアルゴリズムスイートなど、デジタル署名を含むアルゴリズムスイートを使用している場合のみです。

バッファリングを簡単にするために、Node.js などの一部の AWS Encryption SDK 言語実装 AWS Encryption SDK for JavaScript には、復号メソッドの一部としてバッファリング機能が含まれています。入力と出力を常にストリーミングする AWS Encryption CLI では、バージョン 1.9.x と 2.2.x で `--buffer` パラメータが導入されました。他の言語実装では、既存のバッファリング機能を使用できます。(.NET AWS Encryption SDK 用 はストリーミングをサポートしていません ) 。

デジタル署名のないアルゴリズムスイートを使用している場合は、必ず各言語実装で `decrypt-unsigned` 機能を使用してください。この機能では暗号化テキストが復号されますが、署名付き暗号化テキストを検出すると失敗します。詳細については、「アルゴリズムスイートを選択する」を参照してください。

## データキーのキャッシュ

一般的に、データキーの再利用はお勧めしませんが、 AWS Encryption SDK には、データキーの再利用を制限するデータキーキャッシュオプションが用意されています。データキーキャッシュでは、一部のアプリケーションのパフォーマンスが向上し、キーインフラストラクチャの呼び出しが減ります。本番環境でデータキーキャッシュを使用する前に、セキュリティしきい値を調整してテストし、データキーを再利用することのメリットがデメリットを上回っていることを確認してください。

# のキーストア AWS Encryption SDK

では AWS Encryption SDK、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ために AWS KMS 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。キーストアには、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保存されます。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。

## 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 から派生します。

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。

# 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した場合にのみ実行できます。

`CreateKey` は、キーストアの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、データの暗号化、復号、署名、検証を行うときに、階層キーリングを介してのみキーストアを操作します。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、、、 GetActiveBranchKey GetBranchKeyVersion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GetBeaconKey。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、使用オペレーションを実行できます。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検出用にキーストアアクションを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、キーストアを作成する必要があります。これは、ブランチキーを管理および保護する Amazon DynamoDB テーブルです。

> **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次の必須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、論理キーストア名として DynamoDB テーブル名を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`。

次のステップ

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。では、次のキーストアアクション設定 AWS Encryption SDK がサポートされています。

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、、GetBeaconKey) と管理オペレーション ( GetBranchKeyVersionCreateKey および ) を実行できますVersionKey。 CreateKeyは、キーストアの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に許可リストされている任意の AWS KMS key ARN を使用できます。ただし、マルチリージョン KMS キーが検出され、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、次の前提条件が満たされ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 の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でバインドされます。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、ブランチキーストアの許可リストに KMS キー ARN を追加します。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);
```

### Python

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

### Rust

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let key_store_config = KeyStoreConfig::builder()
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .ddb_client(aws_sdk_dynamodb::Client::new(&sdk_config))
```

```
    .ddb_table_name(key_store_name)
    .logical_key_store_name(logical_key_store_name)
    .kms_configuration(KmsConfiguration::KmsKeyArn(kms_key_arn.to_string()))
    .build()?;

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

Go

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

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

### 検出設定

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);
```

### Python

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

### Rust

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

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

Go

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

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

# 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 が呼び出し回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で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。

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内の許可リストに登録されたキーのいずれかを使用できるように、検出用のキーストアアクションを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`: 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
  });
```

### Python

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

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

### Rust

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

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

### Go

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

branchKey, err := keyStore.CreateKey(context.Background(),
 keystoretypes.CreateKeyInput{
```

```
    BranchKeyIdentifier: &customBranchKeyId,
    EncryptionContext:   additional_encryption_context,
})
if err != nil {
    return "", err
}
```

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))。

次に、CreateKey オペレーションは、以下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"
  }
```

次に、CreateKey 操作は 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も使用できませんが、クエリを実行して、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。

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});
```

Python

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

Rust

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

Go

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

# キーリング

サポートされているプログラミング言語の実装では、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、それぞれのメッセージ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。SDK で提供されるキーリングを使用するか、互換性のある独自のカスタムキーリングを作成できます。

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 ServiceAWS 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。詳細については、AWS Encryption SDK 仕様のトピック「Keyring Interface」を参照してください。

キーリングは、他のプログラミング言語の実装で使用されるマスターキーとマスターキープロバイダーの役割を果たします。 AWS Encryption SDK の異なる言語実装を使用してデータを暗号化および復号する場合は、必ず互換性のあるキーリングとマスターキープロバイダを使用してください。詳細については、「キーリングの互換性」を参照してください。

このトピックでは、 のキーリング機能を使用する方法 AWS Encryption SDK と、キーリングを選択する方法について説明します。

## キーリングのしくみ

データを暗号化すると、 はキーリングに暗号化マテリアル AWS Encryption SDK を要求します。キーリングは、プレーンテキストデータのキーと、キーリング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を返します。 AWS 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、プレーンテキストデータキーを破棄します。次に、 は、暗号化されたデータキーと暗号化されたデータを含む暗号化されたメッセージ AWS Encryption SDK を返します。

データを復号する場合、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、復号化キーリングが暗号化キーリングの少なくとも 1 つのラッピングキーを含んでいる (またはアクセスできる) 必要があります。

は、暗号化されたメッセージから暗号化されたデータキーをキーリングに AWS Encryption SDK 渡し、キーリングにそのいずれかを復号するように要求します。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 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のすべてのラッピングキーで暗号化されたデータキーのコピーを返します。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でキーリングを使用して復号できます。

## キーリングの互換性

の異なる言語実装 AWS Encryption SDK にはアーキテクチャ上の違いがいくつかありますが、言語の制約に応じて完全に互換性があります。ある言語実装によってデータを暗号化し、それを他の言語実装で復号することができます。ただし、データキーの暗号化と復号には、同じまたは対応するラッピングキーを使用する必要があります。言語の制約については、 トピックの など、各言語the section called “互換性”の実装に関する AWS Encryption SDK for JavaScript トピックを参照してください。

キーリングは、次のプログラミング言語でサポートされています。

- AWS Encryption SDK for C
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK .NET 用
- のバージョン 3.x AWS Encryption SDK for Java
- オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。
- AWS Encryption SDK Rust 用
- AWS Encryption SDK Go 用の

## 暗号化キーリングのさまざまな要件

以外の AWS Encryption SDK 言語実装では AWS Encryption SDK for C、暗号化キーリング (またはマルチキーリング) またはマスターキープロバイダーのすべてのラッピングキーが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。

例外は で AWS Encryption SDK for C、暗号化オペレーションでは標準の検出キーリングは無視されますが、マルチリージョンの検出キーリングを単独で指定するか、マルチキーリングで指定すると失敗します。

## 互換性があるキーリングおよびマスターキープロバイダー

次の表は、 AWS Encryption SDK が提供するキーリングと互換性のあるマスターキーとマスターキープロバイダーを示しています。言語の制約によるマイナーな非互換性については、言語実装に関するトピックで説明されています。

| キーリング: | マスターキープロバイダー: |
| --- | --- |
| AWS KMS キーリング | KMSMasterKey (Java)<br><br>KMSMasterKeyProvider (Java)<br><br>KMSMasterKey (Python)<br><br>KMSMasterKeyProvider (Python)<br><br>**Note**<br>AWS Encryption SDK for Python および には、AWS KMS リージョン検出キーリングと同等のマスターキーまたはマスターキープロバイダー AWS Encryption SDK for Java は含まれません。 |
| AWS KMS 階層キーリング | 次のプログラミング言語とバージョンでサポートされています。<br><br>• のバージョン 3.x AWS Encryption SDK for Java |

| キーリング: | マスターキープロバイダー: |
| --- | --- |
| | • .NET AWS Encryption SDK 用 のバージョン 4.x<br>• オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。<br>• AWS Encryption SDK for Rust のバージョン 1.x<br>• AWS Encryption SDK for Go のバージョン 0.1.x 以降 |
| AWS KMS ECDH キーリング | 次のプログラミング言語とバージョンでサポートされています。<br>• のバージョン 3.x AWS Encryption SDK for Java<br>• .NET AWS Encryption SDK 用 のバージョン 4.x<br>• オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する場合の AWS Encryption SDK for Pythonのバージョン 4.x。<br>• AWS Encryption SDK for Rust のバージョン 1.x<br>• AWS Encryption SDK for Go のバージョン 0.1.x 以降 |
| Raw AES キーリング | 対称暗号化キーと一緒に使用する場合:<br>JceMasterKey (Java)<br>RawMasterKey (Python) |
| Raw RSA キーリング | 非対称暗号化キーと一緒に使用する場合:<br>JceMasterKey (Java)<br>RawMasterKey (Python)<br>**Note**<br>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 キーを使用する場合、.NET AWS Encryption SDK 用 のバージョン 4.x では、対称暗号化 (`SYMMETRIC_DEFAULT` ) または非対称 RSA を使用する AWS KMS キーリングがサポートされます AWS KMS keys。 |

| キーリング: | マスターキープロバイダー: |
| --- | --- |
| Raw ECDH キーリング | 次のプログラミング言語とバージョンでサポートされています。<br>• のバージョン 3.x AWS Encryption SDK for Java<br>• .NET AWS Encryption SDK 用 のバージョン 4.x<br>• オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。<br>• AWS Encryption SDK for Rust のバージョン 1.x<br>• AWS Encryption SDK for Go のバージョン 0.1.x 以降 |

# AWS KMS キーリング

AWS KMS キーリングは 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 キーリングか同様のセキュリティ特性を持つキーリングを使用することをお勧めします。

キーリングをサポートするすべてのプログラミング言語の実装は、対称暗号化 KMS AWS KMS キーを使用するキーリングをサポートします。以下のプログラミング言語の実装では、非対称 RSA KMS AWS KMS キーを使用するキーリングもサポートされています。

- のバージョン 3.x AWS Encryption SDK for Java
- for .NET のバージョンAWS Encryption SDK 4.x
- オプションの Cryptographic Material Providers Library (MPL) 依存関係とともに使用する場合の AWS Encryption SDK for Pythonのバージョン 4.x。
- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

他の言語実装の暗号化キーリングに非対称 KMS キーを含めようとすると、暗号化呼び出しは失敗します。復号キーリングに含めても無視されます。

マルチ AWS KMS リージョンキーは、 のバージョン 2.3.x AWS Encryption SDK および AWS Encryption CLI のバージョン 3.0.x 以降、 AWS KMS キーリングまたはマスターキープロバイダー

で使用できます。multi-Region-awareシンボルの使用の詳細と例については、「」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。

> **Note**
>
> の KMS キーリングに関するすべての言及は AWS Encryption SDK AWS KMS 、キーリングを指します。

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。

メッセージを暗号化するには、 にジェネレーターキーが必要です。 AWS KMS キーリングに KMS キーが 1 つだけある場合、そのキーはデータキーの生成と暗号化に使用されます。復号時、ジェネレーターキーはオプションであり、ジェネレーターキーと追加のキーの区別は無視されます。

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使用することもできます。

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
- 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 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、キーリング AWS KMS keys の 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。

- 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。
- 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 AWS KMS 、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「KMS キーのアクセスとアクセス許可」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。

- の暗号化キーリングでは AWS Encryption SDK for C、キー ARN またはエイリアス ARN を使用して KMS キーを識別できます。他のすべての言語実装では、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。

- 復号キーリングでは、キー ARN を使用して AWS KMS keysを指定する必要があります。この要件は、 AWS Encryption SDKのすべての言語の実装に適用されます。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。この要件は、 AWS Encryption SDKのすべての言語の実装に適用されます。

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。は、対称暗号化 KMS キー (SYMMETRIC_DEFAULT) または非対称 RSA KMS キー AWS KMS keys 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。マルチキーリングでは 1 つ以上の AWS KMS 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 定義されている 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。( のない 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 )。

以外の AWS Encryption SDK 言語実装では AWS Encryption SDK for C、暗号化キーリングまたはマルチキーリングのすべてのラッピングキーが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。例外は です。 AWS Encryption SDK for C暗号化オペレーションでは標準検出キーリングは無視されますが、マルチリージョン検出キーリングを単独で指定するか、マルチキーリングで指定すると失敗します。

次の例では、ジェネレーター AWS KMS キーと 1 つの追加キーを使用して キーリングを作成します。ジェネレーターキーと追加キーはどちらも対称暗号化 KMS キーです。これらの例では、キー ARN を使用して KMS キーを識別します。これは、暗号化に使用される AWS KMS キーリングのベストプラクティスであり、復号に使用される AWS KMS キーリングの要件です。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。

C

AWS KMS key の暗号化キーリングで を識別するには AWS Encryption SDK for C、キー ARN またはエイリアス ARN を指定します。復号キーリングでは、キー ARN を使用する必要があります。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。

詳しい例については、string.cpp を参照してください。

```
const char * generator_key = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"

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

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

C# / .NET

for .NET で 1 つ以上の KMS キーを使用してキーリングを作成するには、 AWS Encryption SDK `CreateAwsKmsMultiKeyring()`メソッドを使用します。この例では AWS KMS キーを 2 つ使用しています。1 つの KMS キーを指定するには、`Generator` パラメータのみを使用します。追加の KMS キーを指定する `KmsKeyIds` パラメータはオプションです。

このキーリングの入力は AWS KMS クライアントを取りません。代わりに、 はキーリングの KMS キーで表されるリージョンごとにデフォルトの AWS KMS クライアント AWS Encryption SDK を使用します。たとえば、 `Generator`パラメータの値で識別される KMS キーが米国西部 (オレゴン) リージョン (`us-west-2`) にある場合、 は`us-west-2`リージョンのデフォルト AWS KMS クライアント AWS Encryption SDK を作成します。 AWS KMS クライアントをカスタマイズする必要がある場合は、`CreateAwsKmsKeyring()` メソッドを使用します。

.NET AWS KMS key の で暗号化キーリング AWS Encryption SDK に を指定する場合、キー ID、キー ARN、エイリアス名、エイリアス ARN などの有効なキー識別子を使用できます。 AWS

KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

次の例では、.NET AWS Encryption SDK 用 のバージョン 4.x と CreateAwsKmsKeyring()メソッドを使用して AWS KMS クライアントをカスタマイズします。

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

string generatorKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
List<string> additionalKeys = new List<string> { "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321" };

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

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

JavaScript Browser

で暗号化キーリング AWS KMS key に を指定する場合 AWS Encryption SDK for JavaScript、キー ID、キー ARN、エイリアス名https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html#key-id-alias-name、エイリアス ARN などの有効なキー識別子を使用できます。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_simple.ts」を参照してください。

```
import {
  KmsKeyringNode,
```

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

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

const clientProvider = getClient(KMS, { credentials })
const generatorKeyId = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
const additionalKey = 'alias/exampleAlias'

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

JavaScript Node.js

で暗号化キーリング AWS KMS key に を指定する場合 AWS Encryption SDK for JavaScript、キー ID、キー ARN、エイリアス名https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html#key-id-alias-name、エイリアス ARN などの有効なキー識別子を使用できます。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

完全な例については、GitHub の AWS Encryption SDK for JavaScript リポジトリの「kms_simple.ts」を参照してください。

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

const { encrypt, decrypt } = buildClient(
```

```
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

const additionalKey = 'alias/exampleAlias'

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

Java

1 つ以上のキーを使用して AWS KMS キーリングを作成するには、`CreateAwsKmsMultiKeyring()`メソッドを使用します。この例では、2 つの KMS キーを使用します。1 つの KMS キーを指定するには、`generator` パラメータのみを使用します。追加の KMS キーを指定する `kmsKeyIds` パラメータはオプションです。

このキーリングの入力は AWS KMS クライアントを取りません。代わりに、 はキーリングの KMS キーで表されるリージョンごとにデフォルトの AWS KMS クライアント AWS Encryption SDK を使用します。たとえば、 `Generator`パラメータの値で識別される KMS キーが米国西部 (オレゴン) リージョン (`us-west-2`) にある場合、 は`us-west-2`リージョンのデフォルト AWS KMS クライアント AWS Encryption SDK を作成します。 AWS KMS クライアントをカスタマイズする必要がある場合は、`CreateAwsKmsKeyring()` メソッドを使用します。

で暗号化キーリング AWS KMS key に を指定する場合 AWS Encryption SDK for Java、キー ID、キー ARN、エイリアス名https://docs.aws.amazon.com/kms/latest/developerguide/concepts.html#key-id-alias-name、エイリアス ARN などの有効なキー識別子を使用できます。AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

完全な例については、GitHub の リポジトリの AWS Encryption SDK for Java BasicEncryptionKeyringExample.java」を参照してください。

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

```
String generatorKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
List<String> additionalKey = Collections.singletonList("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");
// Create the keyring
final CreateAwsKmsMultiKeyringInput keyringInput =
 CreateAwsKmsMultiKeyringInput.builder()
        .generator(generatorKey)
        .kmsKeyIds(additionalKey)
        .build();
final IKeyring kmsKeyring =
 materialProviders.CreateAwsKmsMultiKeyring(keyringInput);
```

Python

1 つ以上のキーを使用して AWS KMS キーリングを作成するには、`create_aws_kms_multi_keyring()`メソッドを使用します。この例では、2 つの KMS キーを使用します。1 つの KMS キーを指定するには、`generator` パラメータのみを使用します。追加の KMS キーを指定する `kms_key_ids` パラメータはオプションです。

このキーリングの入力は AWS KMS クライアントを取りません。代わりに、 はキーリングの KMS キーで表されるリージョンごとにデフォルトの AWS KMS クライアント AWS Encryption SDK を使用します。たとえば、 `generator`パラメータの値で識別される KMS キーが米国西部 (オレゴン) リージョン (`us-west-2`) にある場合、 は`us-west-2`リージョンのデフォルト AWS KMS クライアント AWS Encryption SDK を作成します。 AWS KMS クライアントをカスタマイズする必要がある場合は、`create_aws_kms_keyring()` メソッドを使用します。

で暗号化キーリング AWS KMS key に を指定する場合 AWS Encryption SDK for Python、キー ID、キー ARN、エイリアス名、エイリアス ARN などの有効なキー識別子を使用できます。AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

次の例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。
完全な例については、GitHub の AWS Encryption SDK for Python リポジトリの「aws_kms_multi_keyring_example.py」を参照してください。

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

```
)

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

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

Rust

1 つ以上のキーを使用して AWS KMS キーリングを作成するには、`create_aws_kms_multi_keyring()`メソッドを使用します。この例では、2 つの KMS キーを使用します。1 つの KMS キーを指定するには、`generator` パラメータのみを使用します。追加の KMS キーを指定する `kms_key_ids` パラメータはオプションです。

このキーリングの入力は AWS KMS クライアントを取りません。代わりに、 はキーリングの KMS キーで表されるリージョンごとにデフォルトの AWS KMS クライアント AWS Encryption SDK を使用します。たとえば、 `generator`パラメータの値で識別される KMS キーが米国西部 (オレゴン) リージョン (`us-west-2`) にある場合、 は`us-west-2`リージョンのデフォルト AWS KMS クライアント AWS Encryption SDK を作成します。 AWS KMS クライアントをカスタマイズする必要がある場合は、`create_aws_kms_keyring()` メソッドを使用します。

for Rust で AWS KMS key 暗号化キーリング AWS Encryption SDK に を指定する場合、キー ID、キー ARN、エイリアス名、エイリアス ARN などの有効なキー識別子を使用できます。AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

次の例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。完全な例については、GitHub の aws-encryption-sdk リポジトリの Rust ディレクトリの aws_kms_keyring_example.rs を参照してください。 aws-encryption-sdk

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

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

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

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

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

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

```
)
```

Go

1 つ以上のキーを使用して AWS KMS キーリングを作成するには、`create_aws_kms_multi_keyring()`メソッドを使用します。この例では、2 つの KMS キーを使用します。1 つの KMS キーを指定するには、`generator` パラメータのみを使用します。追加の KMS キーを指定する `kms_key_ids` パラメータはオプションです。

このキーリングの入力は AWS KMS クライアントを取りません。代わりに、 はキーリングの KMS キーで表されるリージョンごとにデフォルトの AWS KMS クライアント AWS Encryption SDK を使用します。たとえば、 `generator`パラメータの値で識別される KMS キーが米国西部 (オレゴン) リージョン (`us-west-2`) にある場合、 は`us-west-2`リージョンのデフォルト AWS KMS クライアント AWS Encryption SDK を作成します。 AWS KMS クライアントをカスタマイズする必要がある場合は、`create_aws_kms_keyring()` メソッドを使用します。

for Go で AWS KMS key 暗号化キーリング AWS Encryption SDK に を指定する場合、キー ID、キー ARN、エイリアス名、エイリアス ARN などの有効なキー識別子を使用できます。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

次の例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。

```
import (
    "context"

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

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

```
// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

は、非対称 RSA KMS AWS KMS キーを使用するキーリング AWS Encryption SDK もサポートしています。非対称 RSA AWS KMS キーリングに含めることができるキーペアは 1 つだけです。

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。 AWS KMS このキーリングで暗号化する場合、 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。

> **Note**
>
> 非対称 RSA KMS AWS KMS キーを使用する キーリングを作成するには、次のいずれかのプログラミング言語実装を使用する必要があります。
>
> - のバージョン 3.x AWS Encryption SDK for Java

- for .NET のバージョンAWS Encryption SDK 4.x
- オプションの Cryptographic Material Providers Library (MPL) 依存関係とともに使用する場合の AWS Encryption SDK for Pythonのバージョン 4.x。
- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

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`

C# / .NET

非対称 RSA AWS KMS キーリングを作成するには、非対称 RSA KMS キーからパブリックキーとプライベートキー ARN を指定する必要があります。パブリックキーは PEM でエンコードされている必要があります。次の例では、非対称 RSA AWS KMS キーペアを使用して キーリングを作成します。

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

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

// Instantiate the keyring input object
var createKeyringInput = new CreateAwsKmsRsaKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = AWS KMS RSA private key ARN,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
```

```
};

// Create the keyring
var kmsRsaKeyring = mpl.CreateAwsKmsRsaKeyring(createKeyringInput);
```

Java

非対称 RSA AWS KMS キーリングを作成するには、非対称 RSA KMS キーからパブリックキーとプライベートキー ARN を指定する必要があります。パブリックキーは PEM でエンコードされている必要があります。次の例では、非対称 RSA AWS KMS キーペアを使用して キーリングを作成します。

```
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder()
        // Specify algorithmSuite without asymmetric signing here
        //
        // ALG_AES_128_GCM_IV12_TAG16_NO_KDF("0x0014"),
        // ALG_AES_192_GCM_IV12_TAG16_NO_KDF("0x0046"),
        // ALG_AES_256_GCM_IV12_TAG16_NO_KDF("0x0078"),
        // ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256("0x0114"),
        // ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256("0x0146"),
        // ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256("0x0178")

 .withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256)
        .build();

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

// Create a KMS RSA keyring.
//    This keyring takes in:
//     - kmsClient
//     - kmsKeyId: Must be an ARN representing an asymmetric RSA KMS key
//     - publicKey: A ByteBuffer of a UTF-8 encoded PEM file representing the public
//                  key for the key passed into kmsKeyId
//     - encryptionAlgorithm: Must be either RSAES_OAEP_SHA_256 or RSAES_OAEP_SHA_1
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
        CreateAwsKmsRsaKeyringInput.builder()
                .kmsClient(KmsClient.create())
                .kmsKeyId(rsaKeyArn)
                .publicKey(publicKey)
                .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
```

```
                .build();
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

Python

非対称 RSA AWS KMS キーリングを作成するには、非対称 RSA KMS キーからパブリックキーとプライベートキー ARN を指定する必要があります。パブリックキーは PEM でエンコードされている必要があります。次の例では、非対称 RSA AWS KMS キーペアを使用して キーリングを作成します。

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

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

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

Rust

非対称 RSA AWS KMS キーリングを作成するには、非対称 RSA KMS キーからパブリックキーとプライベートキー ARN を指定する必要があります。パブリックキーは PEM でエンコードされている必要があります。次の例では、非対称 RSA AWS KMS キーペアを使用して キーリングを作成します。

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

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

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

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

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

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

Go

```
import (
    "context"

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

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

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

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

```
    panic(err)
}

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

## AWS KMS 検出キーリングの使用

復号するときは、 で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 AWS Encryption SDK 。このベストプラクティスに従うには、ラ AWS KMS ッピングキーを指定したキーに制限する復 AWS KMS 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 Encryption SDK には、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が用意されています。 AWS Encryption SDKでのマルチリージョンキーの使用の詳細については、「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、検出キーリングではデータを暗号化することはできません。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。例外は です。 AWS Encryption SDK for C暗号化オペレーションでは標準検出キーリングは無視されますが、マルチリージョン検出キーリングを単独で指定するか、マルチキーリングで指定すると失敗します。

復号時に、検出キーリングを使用すると、 AWS 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 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。 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 Encryption SDK には、便利な AWS KMS 検出キーリングが用意されています。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、暗号化 AWS KMS key されたメッセージ内のデータキーを暗号化するために使用された任意の を使用できるため、呼び出し元はそれ AWS KMS key を使用して復号するアクセス許可を持ちます。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 KMS 検出キーリングは、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 AWS Encryption SDK 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、他のキーリングよりも明らかに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。検出フィルターは、 AWS Encryption SDKのバージョン 1.7.x 以降でサポートされています。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。

次のコードは、 が AWS Encryption SDK 使用できる KMS キーをawsパーティションおよび 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、 AWS KMS 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。それ以外の AWS リージョンの場合は、aws パーティション値を使用してください。

C

詳しい例については、kms_discovery.cpp を参照してください。

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

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

C# / .NET

次の例では、.NET 用 AWS Encryption SDK のバージョン 4.x を使用します。

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

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

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

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

JavaScript Browser

JavaScript では、discovery プロパティを明示的に指定する必要があります。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

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

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

const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

JavaScript では、discovery プロパティを明示的に指定する必要があります。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

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

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

```
const discovery = true

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

### Java

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

### Python

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

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

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

# Instantiate the material providers
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
```

```
    config=MaterialProvidersConfig()
)

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

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

### Rust

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

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


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

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

// Create the AWS KMS discovery keyring
let discovery_keyring = mpl
    .create_aws_kms_discovery_keyring()
```

```
    .kms_client(kms_client.clone())
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
```

```
    "the data you are handling": "is what you think it is",
}

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

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

## AWS KMS 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 Encryption SDK は特に KMS キーのみを使用して復号できます 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、 は指定された AWS Encryption SDK AWS KMS key の で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、呼び出し元が、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 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に、リージョン検出キーリングは暗号化には影響しません。暗号化されたメッセージ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検出キーリングを使用する場合、そのキーリングは復号化時にのみ有効です。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。

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 AWS Encryption SDK for C を試みます。 AWS Encryption SDK for JavaScript および .NET AWS Encryption SDK で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 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。検出フィルターは、 AWS Encryption SDKのバージョン 1.7.x 以降でサポートされています。

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 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。

C

実例でこのキーリングや `create_kms_client` メソッドを表示する方法については、「kms_discovery.cpp」を参照してください。

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

struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()

 .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery(discovery_filter))
```

C# / .NET

AWS Encryption SDK for .NET には、専用のリージョン検出キーリングはありません。ただし、復号時に使用する KMS キーを特定のリージョンに制限する手法はいくつかあります。

検出キーリング内のリージョンを制限する最も効率的な方法は、単一リージョンキーのみを使用してデータを暗号化した場合でも、マルチリージョン対応の検出キーリングを使用することです。単一リージョンキーが見つかった場合、マルチリージョン対応キーリングはマルチリージョン機能を使用しません。

`CreateAwsKmsMrkDiscoveryKeyring()` メソッドによって返されるキーリングは、 AWS KMSを呼び出す前に KMS キーをリージョン別にフィルタリングします。暗号化されたデータキーが、 `CreateAwsKmsMrkDiscoveryKeyringInput` オブジェクトの `Region`パラメータで指定されたリージョンの KMS キーによって暗号化された AWS KMS 場合にのみ、復号リクエストを に送信します。

次の例では、.NET 用 AWS Encryption SDK のバージョン 4.x を使用します。

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

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

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

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

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

AWS KMS クライアントのインスタンス (AmazonKeyManagementServiceClient) でリージョンを指定 AWS リージョン することで、KMS キーを特定の に制限することもできます。ただし、この構成は、マルチリージョン対応の検出キーリングを使用するよりも効率が悪く、コストもかかる可能性があります。を呼び出す前に KMS キーをリージョン別にフィルタリングする代わりに AWS KMS、 AWS Encryption SDK for .NET は暗号化されたデータキー AWS KMS ごとに (復号されるまで) を呼び出し、 AWS KMS を使用して使用する KMS キーを指定されたリージョンに制限します。

次の例では、.NET 用 AWS Encryption SDK のバージョン 4.x を使用します。

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

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

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

var kmsRegionalDiscoveryKeyring =
 mlp.CreateAwsKmsDiscoveryKeyring(createRegionalDiscoveryKeyringInput);
```

JavaScript Browser

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

```
import {
  KmsKeyringNode,
  buildClient,
  CommitmentPolicy,
```

```
} from '@aws-crypto/client-node'

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

const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

このキーリングと limitRegions関数を実際の例で表示するには、「kms_regional_discovery.ts」を参照してください。

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

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

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

### Java

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

### Python

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

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

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

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

# Create the AWS KMS regional discovery keyring
regional_discovery_keyring_input: CreateAwsKmsMrkDiscoveryKeyringInput = \
    CreateAwsKmsMrkDiscoveryKeyringInput(
```

```
        kms_client=kms_client,
        region=mrk_replica_decrypt_region,
        discovery_filter=DiscoveryFilter(
            account_ids=[111122223333],
            partition="aws"
        )
)

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

### Rust

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

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

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

// Create an AWS KMS client
let decrypt_kms_config = aws_sdk_kms::config::Builder::from(&sdk_config)
    .region(Region::new(mrk_replica_decrypt_region.clone()))
    .build();
let decrypt_kms_client = aws_sdk_kms::Client::from_conf(decrypt_kms_config);


// Create discovery filter
```

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

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

### Go

```
import (
    "context"

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

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

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

```
    o.Region = KmsKeyRegion
})

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

は、Node.js とブラウザの `excludeRegions`関数 AWS Encryption SDK for JavaScript もエクスポートします。この関数は、特定の AWS KMS リージョン AWS KMS keys で を省略するリージョン検出キーリングを作成します。次の例では、米国東部 (バージニア北部) (us-east-1) AWS リージョンを除くすべての でアカウント 111122223333 AWS KMS keys で使用できる AWS KMS リージョン検出キーリングを作成します。

には同様のメソッド AWS Encryption SDK for C はありませんが、カスタム ClientSupplier を作成することで実装できます。

この例は、Node.js 用のコードを示しています。

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

# AWS KMS 階層キーリング

AWS KMS 階層キーリングを使用すると、データを暗号化または復号 AWS KMS するたびに 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、一意のデータキーを使用して各メッセージを暗号化し、暗号化リクエストごとに各データ暗号化キーを暗号化し、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 呼び出しを行い、 オペレーションで が初めてbranch-key-id指定されたときにブランチキーを復号し、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**
>
> 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 AWS Encryption SDK を参照します。

### プログラミング言語の互換性

階層キーリングは、次のプログラミング言語とバージョンでサポートされています。

- のバージョン 3.x AWS Encryption SDK for Java
- for .NET のバージョン AWS Encryption SDK 4.x
- オプションの MPL 依存関係とともに AWS Encryption SDK for Python使用する場合のバージョン 4.x。
- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

### 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

## 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データキーを生成し、ラッピングキーを生成するために、ローカルキャッシュに有効なブランチマテリアルがあるかどうかを確認します。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。生成されたラッピングキーを使用して、プレーンテキストデータキーのコピーを暗号化します。

暗号化方法では、暗号化マテリアルを使用してデータを暗号化します。詳細については、AWS Encryption SDK「 がデータを暗号化する方法」を参照してください。

復号と検証

次のチュートリアルでは、階層型キーリングが復号マテリアルを組み立て、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. 復号方法では、暗号化されたメッセージから暗号化されたデータキーを識別し、階層型キーリングに渡します。
2. 階層型キーリングは、ブランチキーバージョン、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その他の情報を含む、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と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. 階層型キーリングは、再生し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。

復号の方法では、復号マテリアルとプレーンテキストのデータキーを使用して、暗号化されたメッセージを復号化します。詳細については、「 が暗号化されたメッセージを復 AWS Encryption SDK 号する方法」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、次の前提条件が満たされ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。

- キーストアとブランチキーにアクセスして使用するために必要なアクセス AWS KMS 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 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、 AWS アカウント と、キーストアの対称暗号化 AWS KMS key ( 複数可) に対する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、 への呼び出し回数を減らします。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”

> **Important**
>
> サポートされているすべてのキャッシュタイプは、マルチスレッド環境をサポートするように設計されています。
> ただし、 と共に使用する場合 AWS Encryption SDK for Python、階層キーリングはマルチスレッド環境をサポートしていません。詳細については、GitHub の aws-cryptographic-material-providers-libraryaws-cryptographic-material-providers-library リポジトリにある Python README.rst ファイルを参照してください。

## 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の有効期限が 10 秒前であることを 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 を呼び出すのを防ぎます。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済みます。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}
};
```

Python

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

Rust

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

Go

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

```
}
```

## MultiThreaded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
    }
};
```

Python

```
multithreaded_cache = CacheTypeMultiThreaded(
    value=MultiThreadedCache(
```

```
        entry_capacity=100,
        entry_pruning_tail_size=1
    )
)
```

Rust

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

Go

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

## StormTracking 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 を呼び出すのを防ぎます。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ミリ秒数を定義します。

  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
    }
};
```

### Python

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

### Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

### Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberStormTracking{
  Value: mpltypes.StormTrackingCache{
```

```
    EntryCapacity:        100,
    EntryPruningTailSize: &entryPruningTailSize,
    GraceInterval:        1,
    GracePeriod:          10,
    FanOut:               20,
    InFlightTTL:          10,
    SleepMilli:           20,
  },
}
```

### 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。これは、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できるため、全体的なメモリフットプリントが削減されます。

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。the section called “StormTracking キャッシュ” キャッシュタイプとして 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 を指定するか、互換性のあるカスタムキャッシュを置き換えることができます。

#### 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場合、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的な分割として機能し、各階層キーリングが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るようにします。

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 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);
```

Python

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

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

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

shared_cryptographic_materials_cache =
 mat_prov.create_cryptographic_materials_cache(
    cryptographic_materials_cache_input
)
```

Rust

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

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

```
);

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

Go

```
import (
    "context"

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

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

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

// Create a CMC using the default cache
cmcCacheInput := mpltypes.CreateCryptographicMaterialsCacheInput{
    Cache: &cache,
}
sharedCryptographicMaterialsCache, err :=
 matProv.CreateCryptographicMaterialsCache(context.Background(), cmcCacheInput)
if err != nil {
    panic(err)
}
```

2. 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 };
   ```

   Python

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

   Rust

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

   Go

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

3. 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、 と 600 秒のbranch key ID supplierキャッシュ制限を持つ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);
```

### Python

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

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

### Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
 clone it to
    // pass it to different Hierarchical Keyrings, it will still point to the
 same
    // underlying cache, and increment the reference count accordingly.
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

Go

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

## 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 を呼び出す頻度を決定します。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 サプライヤーでは、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。

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,
   BranchKeyId = branch-key-id,
   TtlSeconds = 600
};
```

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

### Python

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

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

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

### Rust

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

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

### Go

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

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

### 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 の例では、2 つのブランチキーのフレンドリ名と、ブランチキー ID サプライヤーを作成`CreateDynamoDbEncryptionBranchKeyIdSupplier`するための呼び出しを作成します。

   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;
```

### Python

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

### Rust

```
// Create branch key ID supplier that maps the branch key ID to a friendly name
let branch_key_id_supplier = ExampleBranchKeyIdSupplier::new(
    &branch_key_id_a,
    &branch_key_id_b
);
```

### Go

```
// Create branch key ID supplier that maps the branch key ID to a friendly name
keySupplier := branchKeySupplier{branchKeyA: branchKeyA, branchKeyB: branchKeyB}
```

2. 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、600 秒のキャッシュ制限 TLL、最大キャッシュサイズ 1000 を使用して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);
```

Python

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

```
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=CacheTypeDefault(
        value=DefaultCache(
            entry_capacity=100
        )
    ),
)

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

Rust

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

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

Go

```
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:            keyStore,
    BranchKeyIdSupplier: &keySupplier,
    TtlSeconds:          600,
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
```

```
}
```

# AWS KMS ECDH キーリング

AWS KMS ECDH キーリングは、非対称キー契約AWS KMS keysを使用して、2 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 キーペアのプライベートキーと受信者のパブリック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。が (KDF_CTR_HMAC_SHA384) AWS Encryption SDK 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 の推奨事項に準拠しています。

キー取得関数は、64 バイトのキーマテリアルを返します。両者が正しいキーマテリアルを使用するように、 AWS 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、キーリングがメッセージヘッダー暗号文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。例えば、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データを暗号化する場合、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、データを復号化できます。Bob のパブリックキーが KMS キーペアからではない場合、Bob は Raw ECDH キーリングを作成してデータを復号できます。

AWS KMS ECDH キーリングは、AES-GCM を使用して対称キーでデータを暗号化します。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 キーリングには共有ラッピングキーを 1 つだけ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。

プログラミング言語の互換性

AWS KMS ECDH キーリングは Cryptographic Material Providers Library (MPL) のバージョン 1.5.0 で導入され、次のプログラミング言語とバージョンでサポートされています。

- のバージョン 3.x AWS Encryption SDK for Java
- .NET AWS Encryption SDK 用 のバージョン 4.x
- オプションの MPL 依存関係とともに AWS Encryption SDK for Python使用する場合のバージョン 4.x。

- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

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

には AWS アカウント AWS Encryption SDK は必要なく、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データ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データを復号するには、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 キーペアを特定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

  グラントを使用して AWS KMS ECDH キーリングの KMS キーへのアクセスを制御する場合は、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提供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、 で 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、 で 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();
```

Python

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、 で 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。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey,
    KmsPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Retrieve public keys
# Must be DER-encoded X.509 public keys
bob_public_key = get_public_key_bytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
alice_public_key = get_public_key_bytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321")
```

```
# Create the AWS KMS ECDH static keyring
sender_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey(
        KmsPrivateKeyToStaticPublicKeyInput(
            sender_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            sender_public_key = bob_public_key,
            recipient_public_key = alice_public_key,

        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(sender_keyring_input)
```

Rust

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、 で 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。

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

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

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

```
]);

// Retrieve public keys
// Must be DER-encoded X.509 keys
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();

let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

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

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}

// Retrieve public keys
// Must be DER-encoded X.509 keys
publicKeySender, err := utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameSender)
```

```
if err != nil {
    panic(err)
}
publicKeyRecipient, err :=
 utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Create KmsPrivateKeyToStaticPublicKeyInput
kmsEcdhStaticConfigurationInput := mpltypes.KmsPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey:  publicKeyRecipient,
    SenderKmsIdentifier: arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    SenderPublicKey:     publicKeySender,
}
kmsEcdhStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPrivateKeyToStaticPublicKey{
    Value: kmsEcdhStaticConfigurationInput,
}

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

// Create AWS KMS ECDH keyring
awsKmsEcdhKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhKeyring, err := matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

# AWS KMS ECDH 検出キーリングの作成

復号するときは、 が AWS Encryption SDK 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、キーアグリーメントスキーマで AWS KMS ECDH `KmsPrivateKeyToStaticPublicKey` 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メッセージを復号する場合、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 キーペアを特定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

  グラントを使用して AWS KMS ECDH キーリングの KMS キーへのアクセスを制御する場合は、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提供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 キーペアに対して 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、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。

```
// 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 キーペアに対して 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、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 the AWS KMS ECDH discovery keyring
final CreateAwsKmsEcdhKeyringInput recipientKeyringInput =
  CreateAwsKmsEcdhKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
    .KeyAgreementScheme(
      KmsEcdhStaticConfigurations.builder()
        .KmsPublicKeyDiscovery(
          KmsPublicKeyDiscoveryInput.builder()
            .recipientKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321").build()
        ).build())
    .build();
```

### Python

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 キーペアに対して 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、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。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery,
    KmsPublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Create the AWS KMS ECDH discovery keyring
create_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery(
        KmsPublicKeyDiscoveryInput(
            recipient_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321",
        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(create_keyring_input)
```

### Rust

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

// Create the AWS KMS client
```

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

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

// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;

let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

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

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
    .await?;
```

### Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
```

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

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

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}

// Create KmsPublicKeyDiscoveryInput
kmsEcdhDiscoveryStaticConfigurationInput := mpltypes.KmsPublicKeyDiscoveryInput{
    RecipientKmsIdentifier: eccRecipientKeyArn,
}
kmsEcdhDiscoveryStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPublicKeyDiscovery{
    Value: kmsEcdhDiscoveryStaticConfigurationInput,
}

// Instantiate the material providers library
```

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

// Create AWS KMS ECDH discovery keyring
awsKmsEcdhDiscoveryKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhDiscoveryStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhDiscoveryKeyring, err :=
 matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhDiscoveryKeyringInput)
if err != nil {
    panic(err)
}
```

# Raw AES キーリング

AWS 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。

Raw AES キーリングは、AES 暗号化キーとともに使用する場合、 の JceMasterKey クラス AWS Encryption SDK for Java と の RawMasterKey クラスと同等で相互運用 AWS Encryption SDK for Python されます。ある実装でデータを暗号化し、それを他の実装で、同じラッピングキーを使用して復号することができます。詳細については、「キーリングの互換性」を参照してください。

キーの名前空間と名前

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。これらは、暗号化オペレーション

が返す 暗号化されたメッセージ のヘッダー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名前空間と、そのシステムの 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` と `RawMasterKey` のプロバイダー ID (またはプロバイダー) とキー ID フィールドに相当します。
> .NET AWS Encryption SDK の AWS Encryption SDK for C および は、KMS `aws-kms`キーのキー名前空間値を予約します。これらのライブラリの Raw AES キーリングや Raw RSA キーリングでは、この名前空間値を使用しないでください。

特定のメッセージを暗号化および復号化するために異なるキーリングを作成する場合、名前空間と名前の値は重要です。復号キーリング内のキー名前空間とキー名が、暗号化キーリング内のキー名前空間とキー名と大文字と小文字を区別して完全に一致しない場合、キーマテリアルのバイトが同じであっても、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。

C

で Raw AES キーリングをインスタンス化するには AWS Encryption SDK for C、 を使用しますaws_cryptosdk_raw_aes_keyring_new()。完全な例については、「raw_aes_keyring.c」を参照してください。

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

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

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

C# / .NET

for .NET で Raw AES AWS Encryption SDK キーリングを作成するには、`materialProviders.CreateRawAesKeyring()`メソッドを使用します。完全な例については、「RawAESKeyringExample.cs」を参照してください。

次の例では、.NET 用 AWS Encryption SDK のバージョン 4.x を使用します。

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

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

// This example uses the key generator in Bouncy Castle to generate the key
 material.
// In production, use key material from a secure source.
var aesWrappingKey = new
 MemoryStream(GeneratorUtilities.GetKeyGenerator("AES256").GenerateKey());

// Create the keyring that determines how your data keys are protected.
var createKeyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = aesWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var keyring = materialProviders.CreateRawAesKeyring(createKeyringInput);
```

JavaScript Browser

ブラウザ AWS Encryption SDK for JavaScript の はWebCrypto API から暗号化プリミティブを取得します。キーリングを作成する前に、`RawAesKeyringWebCrypto.importCryptoKey()` を使用して Raw キーマテリアルを WebCrypto バックエンドにインポートする必要があります。これにより、WebCrypto へのすべての呼び出しが非同期であっても、キーリングが完成することが保証されます。

次に、Raw AES キーリングをインスタンス化するには、`RawAesKeyringWebCrypto()` メソッドを使用します。キーマテリアルの長さに基づいて AES ラッピングアルゴリズム (「ラッピング

スイート」) を指定する必要があります。完全な例については、「aes_simple.ts (JavaScript ブラウザー)」を参照してください。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

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

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

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

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

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

const rawAesKeyring = new RawAesKeyringWebCrypto({
  keyName,
  keyNamespace,
  wrappingSuite,
  aesWrappingKey
})
```

JavaScript Node.js

Node.js AWS Encryption SDK for JavaScript の で Raw AES キーリングをインスタンス化するには、 RawAesKeyringNodeクラスのインスタンスを作成します。キーマテリアルの長さに基づ

いて AES ラッピングアルゴリズム (「ラッピングスイート」) を指定する必要があります。完全な例については、「aes_simple.ts (JavaScript Node.js)」を参照してください。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

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

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

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

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

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

Java

で Raw AES キーリングをインスタンス化するには AWS Encryption SDK for Java、 を使用しますmatProv.CreateRawAesKeyring()。

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

```
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

Python

次の例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。完全な例については、GitHub の AWS Encryption SDK for Python リポジトリの「raw_aes_keyring_example.py」を参照してください。

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

# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "AES_256_012"

# Optional: Create an encryption context
encryption_context: Dict[str, str] = {
    "encryption": "context",
    "is not": "secret",
    "but adds": "useful metadata",
    "that can help you": "be confident that",
    "the data you are handling": "is what you think it is",
}

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

# Create Raw AES keyring
keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)
```

```
raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=keyring_input
)
```

### Rust

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

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

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

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

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

### Go

```
import (
```

```
    mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
    client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
    esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
//Instantiate the AWS Encryption SDK client.
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
// Define the key namespace and key name
var keyNamespace = "A managed aes keys"
var keyName = "My 256-bit AES wrapping key"

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}
// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
// Create Raw AES keyring
aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  aesWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
if err != nil {
    panic(err)
}
```

# Raw RSA キーリング

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行います。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。の一部の言語実装 AWS Encryption SDK では、異なるペアのキーを持つ Raw RSA キーリングは構築されません。他の人は、キーがが同じキーペアであることを確認することを頼っています。

Raw RSA キーリングは、RSA 非対称暗号化キーとともに使用する場合、 の JceMasterKey AWS Encryption SDK for Java および の RawMasterKey と同等であり、相互運用 AWS Encryption SDK for Python されます。ある実装でデータを暗号化し、それを他の実装で、同じラッピングキーを使用して復号することができます。詳細については、「キーリングの互換性」を参照してください。

> **Note**
>
> Raw RSA キーリングは非対称 KMS キーをサポートしていません。非対称 RSA KMS キーを使用する場合、次のプログラミング言語は非対称 RSA を使用する AWS KMS キーリングをサポートします AWS KMS keys。
>
> - のバージョン 3.x AWS Encryption SDK for Java
> - .NET AWS Encryption SDK 用 のバージョン 4.x
> - オプションの Cryptographic Material Providers Library (MPL) 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。
> - AWS Encryption SDK for Go のバージョン 0.1.x 以降

RSA KMS キーのパブリックキーを含む Raw RSA キーリングでデータを暗号化する場合、AWS Encryption SDK も もデータを復号 AWS KMS できません。 AWS KMS 非対称 KMS キーのプライベートキーを Raw RSA キーリングにエクスポートすることはできません。AWS KMS Decrypt オペレーションでは、 が AWS Encryption SDK 返す暗号化されたメッセージを復号できません。

で Raw RSA キーリングを構築するときは AWS Encryption SDK for C、パスやファイル名ではなく、null で終了された C 文字列として各キーを含む PEM ファイルの内容を必ず指定してください。JavaScript で Raw RSA キーリングを作成する場合は、他の言語の実装との互換性が失われる可能性があることに注意してください。

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。これらは、暗号化オペレーションが返す 暗号化されたメッセージ のヘッダー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 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` と `RawMasterKey` のプロバイダー ID (またはプロバイダー) とキー ID フィールドに相当します。
> は、KMS `aws-kms`キーのキー名前空間値 AWS Encryption SDK for C を予約します。 AWS Encryption SDK for Cの Raw AES キーリングや Raw RSA キーリングには使用しないでください。

特定のメッセージを暗号化および復号化するために異なるキーリングを作成する場合、名前空間と名前の値は重要です。復号キーリング内のキー名前空間とキー名が、暗号化キーリング内のキー名前空間とキー名と大文字と小文字を区別して完全に一致しない場合、キーが同じキーペアであっても、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て Raw RSA キーリングを作成し、SHA-256 パディングモードで OAEP を SHA-256 パディングモードで MGF1 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。

C

で Raw RSA キーリングを作成するには AWS Encryption SDK for C、 を使用しますaws_cryptosdk_raw_rsa_keyring_new。

で Raw RSA キーリングを構築するときは AWS Encryption SDK for C、パスやファイル名ではなく、null で終了された C 文字列として各キーを含む PEM ファイルの内容を必ず指定してください。完全な例については、「raw_rsa_keyring.c」を参照してください。

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

AWS_STATIC_STRING_FROM_LITERAL(key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(key_name, "RSA_2048_06");
```

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

C# / .NET

for .NET で Raw RSA AWS Encryption SDK キーリングをインスタンス化するには、`materialProviders.CreateRawRsaKeyring()`メソッドを使用します。完全な例については、「RawRSAKeyringExample.cs」を参照してください。

次の例では、.NET 用 AWS Encryption SDK のバージョン 4.x を使用します。

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

var keyNamespace = "HSM_01";
var keyName = "RSA_2048_06";

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

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

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

JavaScript Browser

ブラウザ AWS Encryption SDK for JavaScript の は、WebCrypto ライブラリから暗号化プリミティブを取得します。キーリングを作成する前に、`importPublicKey()` および / または `importPrivateKey()` を使用して RAW キーマテリアルを WebCrypto バックエンドにインポートする必要があります。これにより、WebCrypto へのすべての呼び出しが非同期であっても、キーリングが完成することが保証されます。インポートメソッドが受け取るオブジェクトには、ラッピングアルゴリズムとそのパディングモードが含まれます。

キーマテリアルをインポートしたら、`RawRsaKeyringWebCrypto()` メソッドを使用してキーリングをインスタンス化します。JavaScript で Raw RSA キーリングを作成する場合は、他の言語の実装との互換性が失われる可能性があることに注意してください。

次の例では、 `buildClient`関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

完全な例については、「rsa_simple.ts (JavaScript ブラウザ)」を参照してください。

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

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

const privateKey = await RawRsaKeyringWebCrypto.importPrivateKey(
  privateRsaJwKKey
)

const publicKey = await RawRsaKeyringWebCrypto.importPublicKey(
  publicRsaJwKKey
)

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'
```

```
const keyring = new RawRsaKeyringWebCrypto({
  keyName,
  keyNamespace,
  publicKey,
  privateKey,
})
```

JavaScript Node.js

Node.js AWS Encryption SDK for JavaScript の で Raw RSA キーリングをインスタンス化するには、 RawRsaKeyringNodeクラスの新しいインスタンスを作成します。wrapKey パラメータはパブリックキーを保持します。unwrapKey パラメータはプライベートキーを保持します。RawRsaKeyringNode コンストラクターはデフォルトのパディングモードを自動的に計算しますが、好みのパディングモードを指定することもできます。

JavaScript で Raw RSA キーリングを作成する場合は、他の言語の実装との 非互換性の可能性 があることに注意してください。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

完全な例については、「rsa_simple.ts (JavaScript Node.js)」を参照してください。

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

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

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringNode({ keyName, keyNamespace, rsaPublicKey,
 rsaPrivateKey})
```

Java

```
final CreateRawRsaKeyringInput keyringInput = CreateRawRsaKeyringInput.builder()
        .keyName("RSA_2048_06")
        .keyNamespace("HSM_01")
        .paddingScheme(PaddingScheme.OAEP_SHA256_MGF1)
        .publicKey(RSAPublicKey)
        .privateKey(RSAPrivateKey)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Python

次の例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。完全な例については、GitHub の AWS Encryption SDK for Python リポジトリの「raw_rsa_keyring_example.py」を参照してください。

```
# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "RSA_2048_06"

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

# Create Raw RSA keyring
keyring_input: CreateRawRsaKeyringInput = CreateRawRsaKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    padding_scheme=PaddingScheme.OAEP_SHA256_MGF1,
    public_key=RSAPublicKey,
    private_key=RSAPrivateKey
)

raw_rsa_keyring: IKeyring = mat_prov.create_raw_rsa_keyring(
    input=keyring_input
)
```

### Rust

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

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

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "RSA_2048_06";

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

// Create Raw RSA keyring
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(aws_smithy_types::Blob::new(RSAPublicKey))
    .private_key(aws_smithy_types::Blob::new(RSAPrivateKey))
    .send()
    .await?;
```

### Go

```
// Instantiate the material providers library
matProv, err :=
 awscryptographymaterialproviderssmithygenerated.NewClient(awscryptographymaterialproviderss

// Create Raw RSA keyring
```

```
rsaKeyRingInput :=
 awscryptographymaterialproviderssmithygeneratedtypes.CreateRawRsaKeyringInput{
 KeyName:       "rsa",
 KeyNamespace:  "rsa-keyring",
 PaddingScheme:
 awscryptographymaterialproviderssmithygeneratedtypes.PaddingSchemePkcs1,
 PublicKey:     pem.EncodeToMemory(publicKeyBlock),
 PrivateKey:    pem.EncodeToMemory(privateKeyBlock),
}

rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
```

Go

```
import (
    "context"

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

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":                "context",
    "is not":                    "secret",
    "but adds":                  "useful metadata",
    "that can help you":         "be confident that",
    "the data you are handling": "is what you think it is",
}
```

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

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

// Create Raw RSA keyring
rsaKeyRingInput := mpltypes.CreateRawRsaKeyringInput{
    KeyName:       keyName,
    KeyNamespace:  keyNamespace,
    PaddingScheme: mpltypes.PaddingSchemeOaepSha512Mgf1,
    PublicKey:     (RSAPublicKey),
    PrivateKey:    (RSAPrivateKey),
}
rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
if err != nil {
    panic(err)
}
```

## Raw ECDH キーリング

Raw ECDH キーリングは、指定した楕円曲線のパブリックキーとプライベートキーのペアを使用して、2 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。が (KDF_CTR_HMAC_SHA384) AWS Encryption SDK 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 の推奨事項に準拠しています。

キー取得関数は、64 バイトのキーマテリアルを返します。両者が正しいキーマテリアルを使用するように、 AWS 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、キーリングがメッセージヘッダー暗号文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。例えば、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

リングを使用してデータを暗号化する場合、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、データを復号化できます。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データを復号できます。

Raw ECDH キーリングは、AES-GCM を使用して対称キーでデータを暗号化します。その後、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 キーリングには共有ラッピングキーを 1 つだけ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、できれば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責任があります。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。は、次の楕円曲線の仕様 AWS Encryption SDK をサポートしています。

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

プログラミング言語の互換性

Raw ECDH キーリングは、 暗号化マテリアルプロバイダーライブラリ (MPL) のバージョン 1.5.0 で導入され、次のプログラミング言語とバージョンでサポートされています。

- のバージョン 3.x AWS Encryption SDK for Java
- .NET AWS Encryption SDK 用 のバージョン 4.x
- オプションの MPL 依存関係で使用する AWS Encryption SDK for Python場合のバージョン 4.x。
- AWS Encryption SDK for Rust のバージョン 1.x
- AWS Encryption SDK for Go のバージョン 0.1.x 以降

## 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、データを暗号化および復号できます。

キーアグリーメントスキーマを使用して Raw ECDH `RawPrivateKeyToStaticPublicKey` 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、 (`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

  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`

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);
}
```

Python

次の Python の例で
は、RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey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。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey,
    RawPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Must be a PEM-encoded private key
bob_private_key = get_private_key_bytes()
# Must be a DER-encoded X.509 public key
alice_public_key = get_public_key_bytes()

# Create the raw ECDH static keyring
```

```
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey(
        RawPrivateKeyToStaticPublicKeyInput(
            sender_static_private_key = bob_private_key,
            recipient_public_key = alice_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

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。

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

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

// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;
```

```
let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

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

// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

// Optional: Create your encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
```

```
}

// Create keyring input
rawEcdhStaticConfigurationInput := mpltypes.RawPrivateKeyToStaticPublicKeyInput{
    SenderStaticPrivateKey: privateKeySender,
    RecipientPublicKey:     publicKeyRecipient,
}
rawECDHStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberRawPrivateKeyToStaticPublicKey{
    Value: rawEcdhStaticConfigurationInput,
}
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: rawECDHStaticConfiguration,
}

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

// Create raw ECDH static keyring
rawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

## EphemeralPrivateKeyToStaticPublicKey

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、ローカルに新しいキーペアを作成し、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、メッセージ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メッセージを復号するには、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。

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曲線上にローカルに新しいキーペアを作成します。

```
// 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曲線上にローカルに新しいキーペアを作成します。

```
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);
}
```

### Python

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 RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey 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曲線上にローカルに新しいキーペアを作成します。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey,
    EphemeralPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Your get_public_key_bytes must return a DER-encoded X.509 public key
recipient_public_key = get_public_key_bytes()

# Create the raw ECDH ephemeral private key keyring
ephemeral_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey(
        EphemeralPrivateKeyToStaticPublicKeyInput(
            recipient_public_key = recipient_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(ephemeral_input)
```

### Rust

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上にローカルに新しいキーペアを作成します。

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

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

// Load public key from UTF-8 encoded PEM files into a DER encoded public key.
let public_key_file_content =
 std::fs::read_to_string(Path::new(EXAMPLE_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content = parse(public_key_file_content)?;
let public_key_recipient_utf8_bytes = parsed_public_key_file_content.contents();

// Create EphemeralPrivateKeyToStaticPublicKeyInput
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

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

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

// Optional: Create your encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}

// Load public key from UTF-8 encoded PEM files into a DER encoded public key
publicKeyRecipient, err := LoadPublicKeyFromPEM(eccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Create EphemeralPrivateKeyToStaticPublicKeyInput
ephemeralRawEcdhStaticConfigurationInput :=
 mpltypes.EphemeralPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey: publicKeyRecipient,
}
ephemeralRawECDHStaticConfiguration :=
    mpltypes.RawEcdhStaticConfigurationsMemberEphemeralPrivateKeyToStaticPublicKey{
        Value: ephemeralRawEcdhStaticConfigurationInput,
```

```
    }

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

// Create raw ECDH ephemeral private key keyring
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: &ephemeralRawECDHStaticConfiguration,
}
ecdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

### PublicKeyDiscovery

復号するときは、 が AWS Encryption SDK 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、メッセージ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メッセージを復号する場合、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 Raw ECDH `PublicKeyDiscovery` 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

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、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。

```
// 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、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。

```
private static void RawEcdhDiscovery() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
              PublicKeyDiscoveryInput.builder()
                // Must be a PEM-encoded private key

 .recipientStaticPrivateKey(ByteBuffer.wrap(sender.getPrivate().getEncoded()))
                .build()
            )
            .build()
        ).build();

    final IKeyring publicKeyDiscovery  =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

### Python

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 RawEcdhStaticConfigurationsPublicKeyDiscovery 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。

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsPublicKeyDiscovery,
    PublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Your get_private_key_bytes must return a PEM-encoded private key
recipient_private_key = get_private_key_bytes()

# Create the raw ECDH discovery keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = RawEcdhStaticConfigurationsPublicKeyDiscovery(
        PublicKeyDiscoveryInput(
            recipient_static_private_key = recipient_private_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

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 `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。

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

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

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

// Load keys from UTF-8 encoded PEM files.
let mut file = File::open(Path::new(EXAMPLE_ECC_PRIVATE_KEY_FILENAME_RECIPIENT))?;
let mut private_key_recipient_utf8_bytes = Vec::new();
file.read_to_end(&mut private_key_recipient_utf8_bytes)?;

// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
```

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

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

// Optional: Create your encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}

// Load keys from UTF-8 encoded PEM files.
privateKeyRecipient, err := os.ReadFile(eccPrivateKeyFileNameRecipient)
if err != nil {
    panic(err)
}

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

// Create PublicKeyDiscoveryInput
discoveryRawEcdhStaticConfigurationInput := mpltypes.PublicKeyDiscoveryInput{
    RecipientStaticPrivateKey: privateKeyRecipient,
}

discoveryRawEcdhStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberPublicKeyDiscovery{
    Value: discoveryRawEcdhStaticConfigurationInput,
}
```

```
// Create raw ECDH discovery private key keyring
discoveryRawEcdhKeyringInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: discoveryRawEcdhStaticConfiguration,
}

discoveryRawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 discoveryRawEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

# マルチキーリング

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のキーを生成して暗号化します。次に、AWS KMS キーリングのすべての追加 AWS KMS keys 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。

ジェネレーターキーリングなしでマルチキーリングを作成する場合は、それ自体を使用してデータを復号できますが、暗号化することはできません。または、暗号化オペレーションで生成キーリングなしでマルチキーリングを使用するには、別のマルチキーリングで子キーリングとして指定できます。ジェネレーターキーリングのないマルチキーリングは、別のマルチキーリングのジェネレーターキーリングとして指定することはできません。

復号時に、 AWS 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。

バージョン 1.7.x 以降、暗号化されたデータキーが AWS Key Management Service（ AWS KMS) キーリング (またはマスターキープロバイダー) で暗号化されている場合、 AWS Encryption SDK は常に のキー ARN を AWS KMS Decrypt オペレーションの `KeyId`パラメータ AWS KMS key に渡します。これは、使用する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することを保証する AWS KMS ベストプラクティスです。

マルチキーリングの実際の例については、以下を参照してください。

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

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、マルチキーリングでサポートされている任意のキーリングを組み合わせることができます。

C

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

// Define a Raw AES keyring. For details, see raw_aes_keyring.c */
struct aws_cryptosdk_keyring *aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
        alloc, wrapping_key_namespace, wrapping_key_name, wrapping_key,
 AWS_CRYPTOSDK_AES256);
```

C# / .NET

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

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

JavaScript Browser

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

```
import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  RawAesKeyringWebCrypto,
  RawAesWrappingSuiteIdentifier,
  MultiKeyringWebCrypto,
  buildClient,
  CommitmentPolicy,
  synchronousRandomValues,
} from '@aws-crypto/client-browser'

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

const clientProvider = getClient(KMS, { credentials })

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

// Define a Raw AES keyring. For details, see aes_simple.ts.
const aesKeyring = new RawAesKeyringWebCrypto({ keyName, keyNamespace,
 wrappingSuite, masterKey })
```

JavaScript Node.js

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

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

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

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

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

### Java

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

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

### Python

次の例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。

```
# Create the AWS KMS keyring
kms_client = boto3.client('kms', region_name="us-west-2")

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

kms_keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
    generator=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    kms_client=kms_client
)

kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
    input=kms_keyring_input
)

# Create Raw AES keyring
key_name_space = "HSM_01"
key_name = "AES_256_012"

raw_aes_keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)

raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=raw_aes_keyring_input
)
```

### Rust

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

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

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

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

// Create a Raw AES keyring
let key_namespace: &str = "my-key-namespace";
let key_name: &str = "my-aes-key-name";

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

### Go

```
import (
    "context"

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

```
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

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

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

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

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

// Create a Raw AES keyring
var keyNamespace = "my-key-namespace"
var keyName = "my-aes-key-name"

aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  AESWrappingKey,
```

```
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
```

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、キーリングがジェネレータ AWS KMS 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。

C

C のマルチキーリングのコンストラクタでは、ジェネレーターキーリングのみを指定します。

```
struct aws_cryptosdk_keyring *multi_keyring = aws_cryptosdk_multi_keyring_new(alloc,
 kms_keyring);
```

マルチキーリングに子キーリングを追加するには、aws_cryptosdk_multi_keyring_add_child メソッドを使用します。このメソッドは、追加する子キーリングごとに呼び出す必要があります。

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

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 = kmsKeyring,
    ChildKeyrings = new List<IKeyring>() {aesKeyring}
};

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

JavaScript Browser

JavaScript のマルチキーリングはイミュータブルです。JavaScript のマルチキーリングのコンストラクタでは、ジェネレーターキーリングと複数の子キーリングを指定できます。

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

const multiKeyring = new MultiKeyringWebCrypto(generator: kmsKeyring, children:
 [aesKeyring]);
```

JavaScript Node.js

JavaScript のマルチキーリングはイミュータブルです。JavaScript のマルチキーリングのコンストラクタでは、ジェネレーターキーリングと複数の子キーリングを指定できます。

```
const multiKeyring = new MultiKeyringNode(generator: kmsKeyring, children:
 [aesKeyring]);
```

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);
```

Python

```
multi_keyring_input: CreateMultiKeyringInput = CreateMultiKeyringInput(
    generator=kms_keyring,
    child_keyrings=[raw_aes_keyring]
)

multi_keyring: IKeyring = mat_prov.create_multi_keyring(
    input=multi_keyring_input
)
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(kms_keyring.clone())
```

```
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

Go

```
createMultiKeyringInput := mpltypes.CreateMultiKeyringInput{
  Generator:     awsKmsKeyring,
  ChildKeyrings: []mpltypes.IKeyring{rawAESKeyring},
 }
 multiKeyring, err := matProv.CreateMultiKeyring(context.Background(),
 createMultiKeyringInput)
 if err != nil {
  panic(err)
 }
```

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。

# AWS Encryption SDK プログラミング言語

AWS Encryption SDK は、次のプログラミング言語で使用できます。言語実装はすべて相互運用可能です。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。また、暗号化および復号を行う場合は、互換性のあるキーリング、またはマスターキーとマスターキープロバイダーを使用する必要があります。詳細については、「the section called “キーリングの互換性”」を参照してください。

トピック

- AWS Encryption SDK for C
- AWS Encryption SDK .NET 用
- AWS Encryption SDK Go 用の
- AWS Encryption SDK for Java
- AWS Encryption SDK for JavaScript
- AWS Encryption SDK for Python
- AWS Encryption SDK Rust 用
- AWS Encryption SDK コマンドラインインターフェイス

## AWS Encryption SDK for C

AWS Encryption SDK for C は、C でアプリケーションを作成するデベロッパー向けのクライアント側の暗号化ライブラリを提供します。また、 AWS Encryption SDK 高レベルのプログラミング言語で を実装するための基盤としても機能します。

のすべての実装と同様に AWS Encryption SDK、 は高度なデータ保護機能 AWS Encryption SDK for C を提供します。これにはエンベロープ暗号化、追加の認証データ (AAD)、キー取得および署名で使用する 256 ビット AES-GCM などのセキュアで認証済みの対称キーアルゴリズムスイートなどが含まれます。

の言語固有の実装 AWS Encryption SDK はすべて完全に相互運用可能です。例えば、 でデータを暗号化 AWS Encryption SDK for C し、AWS Encryption CLI を含むサポートされている任意の言語実装で復号できます。

AWS Encryption SDK for C では、 が AWS Key Management Service () とやり取り AWS SDK for C ++ する必要がありますAWS KMS。オプションの AWS KMS キーリング を使用する場合にのみ、使

用する必要があります。ただし、 AWS Encryption SDK には AWS KMS や他の AWS サービスは必要ありません。

詳細はこちら

- を使用したプログラミングの詳細については AWS Encryption SDK for C、C の例、GitHub の aws-encryption-sdk-c リポジトリの例、および AWS Encryption SDK for C API ドキュメントを参照してください。
- を使用して複数の で復号できるようにデータを AWS Encryption SDK for C 暗号化する方法については AWS リージョン、 AWS セキュリティブログのAWS Encryption SDK 「C の を使用して複数のリージョンの暗号文を復号する方法」を参照してください。

トピック

- のインストール AWS Encryption SDK for C
- の使用 AWS Encryption SDK for C
- AWS Encryption SDK for C の例

## のインストール AWS Encryption SDK for C

AWS Encryption SDK for Cの最新バージョンをインストールします。

> **Note**
>
> 2.0.0 より AWS Encryption SDK for C 前の のすべてのバージョンはend-of-supportフェーズにあります。
> バージョン 2.0.x 以降から AWS Encryption SDK for C の最新バージョンにコードやデータを変更せずに安全に更新できます。ただし、バージョン 2.0.x で導入された新しいセキュリティ機能には下位互換性がありません。1.7.x より前のバージョンから 2.0.x 以降のバージョンに更新するには、まず AWS Encryption SDK for Cの最新の 1.x バージョンに更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。

のインストールと構築の詳細な手順は AWS Encryption SDK for C 、aws-encryption-sdk-c リポジトリの README ファイルにあります。Amazon Linux、Ubuntu、macOS、およびWindowsプラットフォームで構築するための手順が含まれています。

開始前に、 AWS Encryption SDKで AWS KMS キーリングを使用するかどうかを決定してください。 AWS KMS キーリングを使用する場合は、 をインストールする必要があります AWS SDK for C++。AWS Key Management Service () を操作するには、 AWS SDK が必要ですAWS KMS。 AWS KMS キーリングを使用すると、 AWS Encryption SDK は AWS KMS を使用して、データを保護する暗号化キーを生成および保護します。

raw AES キーリング、raw RSA キーリング、または キーリングを含まないマルチキーリングなど、別の AWS KMS キーリングタイプ AWS SDK for C++ を使用している場合は、 をインストールする必要はありません。ただし、未加工のキーリングタイプを使用する場合は、独自の未加工のラッピングキーを生成して保護する必要があります。

インストールに問題がある場合は、`aws-encryption-sdk-c` リポジトリで問題を提起するか、このページのフィードバックリンクのいずれかを使用してください。

## の使用 AWS Encryption SDK for C

このトピックでは、他のプログラミング言語の実装ではサポート AWS Encryption SDK for C されていない の機能の一部について説明します。

このセクションの例では、 AWS Encryption SDK for Cのバージョン 2.0.x 以降の使用方法について説明します。前バージョンを使用する例については、GitHub の aws-encryption-sdk-c リポジトリのリリースリストで使用中のリリースを検索してください。

を使用したプログラミングの詳細については AWS Encryption SDK for C、C の例、GitHub の aws-encryption-sdk-c リポジトリの例、および AWS Encryption SDK for C API ドキュメントを参照してください。

「キーリング」も参照してください。

トピック

- データの暗号化と復号の流れ
- 参照カウント

### データの暗号化と復号の流れ

を使用する場合は AWS Encryption SDK for C、次のようなパターンに従います。キーリングの作成、キーリングを使用する CMM の作成、CMM (およびキーリング) を使用するセッションの作成、セッションの処理です。

1. エラー文字列をロード

   C コードまたは C++ コードで aws_cryptosdk_load_error_strings() メソッドを呼び出します。デバッグに非常に役立つエラー情報をロードします。

   main メソッド内でなど、1 回だけ呼び出す必要があります。

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

2. キーリングを作成します。

   データキーの暗号化に使用するラッピングキーでキーリングを設定します。この例では、1 つので AWS KMS キーリングを使用しますが AWS KMS key、代わりに任意のタイプのキーリングを使用できます。

   AWS KMS key の暗号化キーリングで を識別するには AWS Encryption SDK for C、キー ARN またはエイリアス ARN を指定します。復号キーリングでは、キー ARN を使用する必要があります。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。

```
const char * KEY_ARN = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
struct aws_cryptosdk_keyring *kms_keyring =
       Aws::Cryptosdk::KmsKeyring::Builder().Build(KEY_ARN);
```

3. セッションを作成します。

   では AWS Encryption SDK for C、セッションを使用して、サイズに関係なく、単一のプレーンテキストメッセージを暗号化するか、単一の暗号テキストメッセージを復号します。セッションでは、そのプロセスを通じてメッセージの状態が維持されます。

   アロケーター、キーリング、モード (AWS_CRYPTOSDK_ENCRYPT または AWS_CRYPTOSDK_DECRYPT) を使用してセッションを設定します。セッションのモードを変更する必要がある場合は、aws_cryptosdk_session_reset メソッドを使用します。

   キーリングを使用してセッションを作成すると、 によってデフォルトの暗号化マテリアルマネージャー (CMM) AWS Encryption SDK for C が自動的に作成されます。このオブジェクトの作成、管理、破棄を行う必要はありません。

   例えば、次のセッションでは、ステップ 1 で定義したアロケーターとキーリングを使用します。データを暗号化する場合は、モードは AWS_CRYPTOSDK_ENCRYPT です。

```
struct aws_cryptosdk_session * session =
 aws_cryptosdk_session_new_from_keyring_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);
```

4. データを暗号化または復号します。

セッションでデータを処理するには、aws_cryptosdk_session_process メソッドを使用します。入力バッファがプレーンテキスト全体を保持するのに十分なサイズであり、出力バッファが暗号化テキスト全体を保持するのに十分なサイズである場合は、aws_cryptosdk_session_process_full を呼び出すことができます。ただし、ストリーミングデータを処理する必要がある場合は、aws_cryptosdk_session_process をループで呼び出すことができます。例については、file_streaming.cpp の例を参照してください。aws_cryptosdk_session_process_full は AWS Encryption SDK 、バージョン 1.9.x および 2.2.x で導入されています。

セッションでデータを暗号化するように設定されている場合、プレーンテキストフィールドは入力、暗号化テキストフィールドは出力を表します。plaintext フィールドは、暗号化するメッセージを保持しており、ciphertext フィールドは、暗号化メソッドによって返る暗号化されたメッセージを取得します。

```
/* Encrypting data */
aws_cryptosdk_session_process_full(session,
                                   ciphertext,
                                   ciphertext_buffer_size,
                                   &ciphertext_length,
                                   plaintext,
                                   plaintext_length)
```

セッションでデータを復号するように設定されている場合、暗号化テキストフィールドは入力、プレーンテキストフィールドは出力を表します。ciphertext フィールドは、暗号化メソッドより返る暗号化されたメッセージを保持しており、plaintext フィールドは、復号メソッドより返るプレーンテキストメッセージを取得します。

データを復号するには、aws_cryptosdk_session_process_full メソッドを呼び出します。

```
/* Decrypting data */
aws_cryptosdk_session_process_full(session,
                                   plaintext,
```

```
                                    plaintext_buffer_size,
                                    &plaintext_length,
                                    ciphertext,
                                    ciphertext_length)
```

## 参照カウント

メモリリークを防ぐために、作成したすべてのオブジェクトへの参照はオブジェクトの使用が完了したら解放するようにします。それ以外の場合は、最終的にメモリリークが発生します。SDK には、この作業を簡略化するメソッドがあります。

次のいずれかの子オブジェクトを使用して親オブジェクトを作成すると、親オブジェクトは子オブジェクトへの参照を作成して管理します。

- キーリング (キーリングを使用したセッションの作成など)
- デフォルトの暗号化マテリアルマネージャー (CMM) (デフォルトの CMM を使用したセッションやカスタム CMM の作成など)
- データキーキャッシュ (キーリングとキャッシュを使用したキャッシュ CMM の作成など)

子オブジェクトへの独立参照が必要でない限り、親オブジェクトを作成したら子オブジェクトへの参照はすぐに解放できます。親オブジェクトが破棄されると、残っている子オブジェクトへの参照は解放されます。このパターンでは、各オブジェクトへの参照を必要な期間だけ保持することができます。また、参照の未開放が原因でメモリリークが発生することもありません。

作成した子オブジェクトへの参照の明示的な解放は、お客様が行う必要があります。SDK が作成したオブジェクトへの参照の管理は、お客様が行う必要はありません。SDK がオブジェクト (`aws_cryptosdk_caching_cmm_new_from_keyring` メソッドがセッションに追加するデフォルトの CMM など) を作成する場合は、SDK がオブジェクトとその参照の作成と破棄を管理します。

次の例では、キーリングを使用してセッションを作成すると、セッションはキーリングへの参照を作成し、セッションが破棄されるまでその参照を管理します。キーリングへの追加の参照を管理する必要がない場合は、セッションを作成したら `aws_cryptosdk_keyring_release` メソッドを使用してキーリングオブジェクトを解放できます。このメソッドでは、キーリングの参照カウントは減少します。`aws_cryptosdk_session_destroy` を呼び出してセッションを破棄すると、セッションのキーリングへの参照は解放されます。

```
// The session gets a reference to the keyring.
```

```
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_ENCRYPT, keyring);

// After you create a session with a keyring, release the reference to the keyring
 object.
aws_cryptosdk_keyring_release(keyring);
```

複数のセッションでキーリングを再利用したり、CMM でアルゴリズムスイートを指定したりするなど、より複雑なタスクの場合は、オブジェクトへの独立参照を管理する必要があります。その場合は、解放のメソッドをすぐに呼び出さないでください。代わりに、セッションの破棄だけでなく、オブジェクトを使用しなくなったときにも参照を解放します。

この参照カウント手法は、データキーキャッシュ用の CMM のキャッシングなど、代替 CMM を使用している場合にも機能します。キャッシュとキーリングからキャッシュ CMM を作成すると、キャッシュ CMM はその両方のオブジェクトへの参照を作成します。別のタスクでそれらが必要でない限り、キャッシュ CMM を作成したらキャッシュとキーリングへの独立参照はすぐに解放できます。その後、キャッシュ CMM を使用してセッションを作成するときに、キャッシュ CMM への参照を解放できます。

作成したオブジェクトへの参照の明示的な解放は、お客様が行う必要があります。メソッドが作成するオブジェクト (キャッシュ CMM の元になるデフォルトの CMM など) は、メソッドが管理します。

```
/ Create the caching CMM from a cache and a keyring.
struct aws_cryptosdk_cmm *caching_cmm =
 aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL, 60,
 AWS_TIMESTAMP_SECS);

// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);

// Create a session with the caching CMM.
struct aws_cryptosdk_session *session = aws_cryptosdk_session_new_from_cmm_2(allocator,
 AWS_CRYPTOSDK_ENCRYPT, caching_cmm);

// Release your references to the caching CMM.
aws_cryptosdk_cmm_release(caching_cmm);

// ...
```

```
aws_cryptosdk_session_destroy(session);
```

# AWS Encryption SDK for C の例

次の例は、 を使用してデータを AWS Encryption SDK for C 暗号化および復号する方法を示しています。

このセクションの例では、 AWS Encryption SDK for Cのバージョン 2.0.x 以降の使用方法について説明します。　前バージョンを使用する例については、GitHub の aws-encryption-sdk-c リポジトリのリリースリストで使用中のリリースを検索してください。

をインストールして構築すると AWS Encryption SDK for C、これらの例やその他の例のソースコードが `examples`サブディレクトリに含まれ、コンパイルされて `build` ディレクトリに構築されます。GitHub の aws-encryption-sdk-c リポジトリの examples サブディレクトリで検索することもできます。

トピック

- 文字列の暗号化と復号

## 文字列の暗号化と復号

次の例は、 を使用して文字列を AWS Encryption SDK for C 暗号化および復号する方法を示しています。

この例では、 AWS KMS key AWS Key Management Service (AWS KMS ) の を使用してデータAWS KMS キーを生成および暗号化するキーリングの一種である キーリングを使用しています。この例には C++ で記述されたコードが含まれています。　AWS Encryption SDK for C では AWS SDK for C++ 、 AWS KMS キーリングを使用する AWS KMS ときに を呼び出す必要があります。raw AES キーリング AWS KMS、raw RSA キーリング、または キーリングを含まないマルチキーリングなど、 とやり取りしない AWS KMS キーリングを使用している場合、 AWS SDK for C++ は必要ありません。

の作成については AWS KMS key、「 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。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

完全なコードサンプルの参照: string.cpp

トピック

- 文字列の暗号化
- 文字列の復号

### 文字列の暗号化

この例の最初の部分では、 AWS KMS キーリングと キーリング AWS KMS key を使用してプレーンテキスト文字列を暗号化します。

ステップ 1. エラー文字列をロード

C コードまたは C++ コードで `aws_cryptosdk_load_error_strings()` メソッドを呼び出します。デバッグに非常に役立つエラー情報をロードします。

`main` メソッド内でなど、1 回だけ呼び出す必要があります。

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

ステップ 2: キーリングを作成します。

暗号化用の AWS KMS キーリングを作成します。この例のキーリングは 1 つで設定されていますが AWS KMS key、異なるアカウント AWS リージョン や異なるアカウントを含む AWS KMS keys複数の AWS KMS keys で AWS KMS キーリングを設定できます。

AWS KMS key の暗号化キーリングで を識別するには AWS Encryption SDK for C、キー ARN またはエイリアス ARN を指定します。復号キーリングでは、キー ARN を使用する必要があります。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。

AWS KMS キーリング AWS KMS keys での の識別

複数の を使用してキーリングを作成するときは AWS KMS keys、プレーンテキストデータキーの生成と暗号化 AWS KMS key に使用される と、同じプレーンテキストデータキーを暗号化 AWS KMS keys する追加のオプションの配列を指定します。この場合、ジェネレーターのみを指定します AWS KMS key。

このコードを実行する前に、キー ARN を有効なキー ARN に置き換えます。

```
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
```

```
struct aws_cryptosdk_keyring *kms_keyring =
       Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

ステップ 3: セッションを作成します。

アロケーター、モードの列挙子、キーリングを使用してセッションを作成します。

各セッションは、AWS_CRYPTOSDK_ENCRYPT モード (暗号化) または AWS_CRYPTOSDK_DECRYPT モード (復号) にする必要があります。既存のセッションのモードを変更するには、aws_cryptosdk_session_reset メソッドを使用します。

キーリングを使用してセッションを作成したら、SDK が提供する機能を使用してキーリングへの参照を解放できます。セッションは、その有効期間中、キーリングオブジェクトへの参照を保持します。セッションを破棄すると、キーリングオブジェクトやセッションオブジェクトへの参照が解放されます。この参照カウント方式は、メモリリークを防ぎ、オブジェクトが使用中に解放されないようにするために役立ちます。

```
struct aws_cryptosdk_session *session =
       aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);

/* When you add the keyring to the session, release the keyring object */
aws_cryptosdk_keyring_release(kms_keyring);
```

ステップ 4: 暗号化コンテキストを設定します。

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です。暗号化時に暗号化コンテキストを指定すると、 は暗号化コンテキストを暗号化テキストに AWS Encryption SDK 暗号的にバインドし、データの復号に同じ暗号化コンテキストが必要になります。暗号化コンテキストの使用はオプションですが、ベストプラクティスとして推奨します。

まず、暗号化コンテキスト文字列を含むハッシュテーブルを作成します。

```
/* Allocate a hash table for the encryption context */
int set_up_enc_ctx(struct aws_allocator *alloc, struct aws_hash_table *my_enc_ctx)

// Create encryption context strings
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key1, "Example");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value1, "String");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key2, "Company");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value2, "MyCryptoCorp");
```

```
// Put the key-value pairs in the hash table
aws_hash_table_put(my_enc_ctx, enc_ctx_key1, (void *)enc_ctx_value1, &was_created)
aws_hash_table_put(my_enc_ctx, enc_ctx_key2, (void *)enc_ctx_value2, &was_created)
```

セッション内の暗号化コンテキストへの変更可能なポインタを取得します。次に、aws_cryptosdk_enc_ctx_clone 関数を使用して、暗号化コンテキストをセッションにコピーします。コピーを my_enc_ctx に保持しているため、データの復号後に値を検証することができます。

暗号化コンテキストはセッションの一部であり、セッション処理関数に渡されるパラメータではありません。これにより、メッセージ全体を暗号化するためにセッション処理関数が複数回呼び出された場合でも、必ずメッセージのすべてのセグメントに同じ暗号化コンテキストが使用されるようになります。

```
struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);

aws_cryptosdk_enc_ctx_clone(alloc, session_enc_ctx, my_enc_ctx)
```

ステップ 5: 文字列を暗号化します。

プレーンテキストの文字列を暗号化するには、暗号化モードのセッションで aws_cryptosdk_session_process_full メソッドを使用します。このメソッドは、AWS Encryption SDK バージョン 1.9.x および 2.2.x で導入され、非ストリーミング暗号化と復号化用に設計されています。ストリーミングデータを処理するには、ループで aws_cryptosdk_session_process を呼び出します。

暗号化する場合、プレーンテキストフィールドは入力フィールド、暗号化テキストフィールドは出力フィールドです。処理が完了すると、ciphertext_output フィールドには、暗号化されたメッセージ (例: 実際の暗号化テキスト、暗号化されたデータキー、暗号化コンテキスト) が含まれます。この暗号化されたメッセージを復号するには、サポートされているプログラミング言語 AWS Encryption SDK の を使用します。

```
/* Gets the length of the plaintext that the session processed */
size_t ciphertext_len_output;
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
                                    ciphertext_output,
                                    ciphertext_buf_sz_output,
                                    &ciphertext_len_output,
```

```
                                    plaintext_input,
                                    plaintext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 8;
}
```

ステップ 6: セッションをクリーンアップします。

最終ステップでは、CMM とキーリングへの参照を含むセッションを破棄します。

必要に応じて、セッションを破棄せずに、同じキーリングと CMM でセッションを再利用し、文字列の復号や、他のメッセージの暗号化または復号を行うことができます。復号のセッションを使用するには、`aws_cryptosdk_session_reset` メソッドを使用して、モードを `AWS_CRYPTOSDK_DECRYPT` に変更します。

文字列の復号

この例の 2 番目の部分では、元の文字列の暗号化テキストを含む暗号化されたメッセージを復号します。

ステップ 1: エラー文字列をロード

C コードまたは C++ コードで `aws_cryptosdk_load_error_strings()` メソッドを呼び出します。デバッグに非常に役立つエラー情報をロードします。

`main` メソッド内でなど、1 回だけ呼び出す必要があります。

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

ステップ 2: キーリングを作成します。

でデータを復号するとき AWS KMSは、暗号化 API が返した暗号化されたメッセージを渡します。Decrypt API は を入力 AWS KMS key として受け取りません。代わりに、 は同じ AWS KMS を使用して AWS KMS key 、暗号化に使用した暗号文を復号します。ただし、 AWS Encryption SDK では、暗号化と復号 AWS KMS keys 時に で AWS KMS キーリングを指定できます。

復号では、暗号化 AWS KMS keys されたメッセージの復号に使用する のみを使用してキーリングを設定できます。例えば、組織内の特定のロール AWS KMS key によって使用される のみを使用してキーリングを作成できます。復号キーリングに表示され AWS KMS key ない限り、 AWS Encryption SDK は を使用しません。指定したキーリング AWS KMS keys で を使用して暗号化さ

れたデータキーを SDK が復号できない場合、キーリング AWS KMS keys の のいずれもデータキーの暗号化に使用されていないか、呼び出し元にキーリング AWS KMS keys の を使用して復号するアクセス許可がないため、復号呼び出しは失敗します。

復号キーリング AWS KMS key に を指定する場合は、そのキー ARN を使用する必要があります。エイリアス ARN は、暗号化キーリングでのみ許可されます。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してくださいAWS KMS キーリング AWS KMS keys での の識別。

この例では、文字列の暗号化 AWS KMS key に使用したものと同じ で設定されたキーリングを指定します。このコードを実行する前に、キー ARN を有効なキー ARN に置き換えます。

```
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"

struct aws_cryptosdk_keyring *kms_keyring =
        Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

ステップ 3: セッションを作成します。

アロケーターとキーリングを使用してセッションを作成します。セッションを復号用に設定するには、セッションを AWS_CRYPTOSDK_DECRYPT モードに設定します。

キーリングを使用してセッションを作成したら、SDK が提供する機能を使用してキーリングへの参照を解放できます。セッションは、その有効期間中、キーリングオブジェクトへの参照を保持します。セッションを破棄すると、セッションとキーリングの両方が解放されます。この参照カウント方式は、メモリリークを防ぎ、オブジェクトが使用中に解放されないようにするために役立ちます。

```
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT,
 kms_keyring);

/* When you add the keyring to the session, release the keyring object */
aws_cryptosdk_keyring_release(kms_keyring);
```

ステップ 4: 文字列を復号します。

文字列を復号するには、復号用に設定されているセッションで aws_cryptosdk_session_process_full メソッドを使用します。 AWS Encryption SDK バージョン 1.9.x および 2.2.x で導入されたこのメソッドは、非ストリーミングの暗号化お

よび復号化のために設計されています。ストリーミングデータを処理するには、ループで aws_cryptosdk_session_process を呼び出します。

復号する際、暗号化テキストフィールドは入力フィールド、プレーンテキストフィールドは出力フィールドです。ciphertext_input フィールドには、返るメソッドを暗号化する暗号化されたメッセージが含まれます。処理が完了すると、plaintext_output フィールドには、プレーンテキスト (復号された) 文字列が含まれます。

```
size_t plaintext_len_output;

if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
                                    plaintext_output,
                                    plaintext_buf_sz_output,
                                    &plaintext_len_output,
                                    ciphertext_input,
                                    ciphertext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 13;
}
```

ステップ 5: 暗号化コンテキストを確認します。

実際の暗号化コンテキスト (メッセージの復号に使用されたもの) に、メッセージの暗号化時に指定した暗号化コンテキストが含まれていることを確認します。暗号化マテリアルマネージャー (CMM) によって、メッセージの暗号化前に指定した暗号化コンテキストにペアが追加される場合があるため、実際の暗号化コンテキストには、追加のペアが含まれる場合があります。

では AWS Encryption SDK for C、SDK が返す暗号化されたメッセージに暗号化コンテキストが含まれているため、復号時に暗号化コンテキストを指定する必要はありません。ただし、プレーンテキストのメッセージが返る前に、復号関数を使用して、返った暗号化コンテキスト内のすべてのペアが、メッセージの復号に使用された暗号化コンテキスト内にあることを確認する必要があります。

まず、セッション内のハッシュテーブルへの読み取り専用ポインタを取得します。このハッシュテーブルには、メッセージの復号に使用された暗号化コンテキストが含まれています。

```
const struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr(session);
```

次に、暗号化時にコピーした my_enc_ctx ハッシュテーブル内の暗号化コンテキストをループします。暗号化に使用された my_enc_ctx ハッシュテーブルの各ペアが、復号に使用された

session_enc_ctx ハッシュテーブルに表示されていることを確認します。キーが見つからない場合や、キーが別の値の場合は、処理を停止し、エラーメッセージを書き込みます。

```
for (struct aws_hash_iter iter = aws_hash_iter_begin(my_enc_ctx); !
aws_hash_iter_done(&iter);
      aws_hash_iter_next(&iter)) {
     struct aws_hash_element *session_enc_ctx_kv_pair;
     aws_hash_table_find(session_enc_ctx, iter.element.key,
 &session_enc_ctx_kv_pair)

    if (!session_enc_ctx_kv_pair ||
        !aws_string_eq(
            (struct aws_string *)iter.element.value, (struct aws_string
 *)session_enc_ctx_kv_pair->value)) {
        fprintf(stderr, "Wrong encryption context!\n");
        abort();
    }
}
```

ステップ 6: セッションをクリーンアップします。

暗号化コンテキストを確認したら、セッションは破棄、または再利用できます。セッションを再設定する必要がある場合は、aws_cryptosdk_session_reset メソッドを使用します。

```
aws_cryptosdk_session_destroy(session);
```

# AWS Encryption SDK .NET 用

AWS Encryption SDK for .NET 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向けのクライアント側の暗号化ライブラリです。Windows、macOS、Linux でサポートされています。

> **Note**
>
> AWS Encryption SDK for .NET のバージョン 4.0.0 がメッセージ仕様から逸脱しています AWS Encryption SDK 。その結果、バージョン 4.0.0 で暗号化されたメッセージは、.NET AWS Encryption SDK 用 のバージョン 4.0.0 以降でのみ復号できます。その他のプログラミング言語実装では復号化できません。
> AWS Encryption SDK for .NET のバージョン 4.0.1 は、 AWS Encryption SDK メッセージ仕様に従ってメッセージを書き込み、他のプログラミング言語の実装と相互運用できます。

デフォルトでは、バージョン 4.0.1 はバージョン 4.0.0 で暗号化されたメッセージを読み取ることができます。ただし、バージョン 4.0.0 で暗号化されたメッセージを復号化したくない場合は、`NetV4_0_0_RetryPolicy` プロパティを指定してクライアントがこれらのメッセージを読み取らないようにすることができます。詳細については、GitHub の aws-encryption-sdk リポジトリにある v4.0.1 リリースノートを参照してください。

AWS Encryption SDK for .NET は、以下の点 AWS Encryption SDK で の他のプログラミング言語実装とは異なります。

- データキーキャッシュ はサポートされていません

  > **Note**
  >
  > AWS Encryption SDK for .NET のバージョン 4.x は、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングをサポートしています。

- ストリーミングデータはサポートしていません
- .NET 用 AWS Encryption SDK から ログ記録やスタックトレースはありません
- が必要です AWS SDK for .NET

AWS Encryption SDK for .NET には、 の他の言語実装のバージョン 2.0.x 以降で導入されたすべてのセキュリティ機能が含まれています AWS Encryption SDK。ただし、 AWS Encryption SDK for .NET を使用して、 の 2.0.x より前のバージョンの別の言語実装によって暗号化されたデータを復号する場合は AWS Encryption SDK、コミットメントポリシーを調整する必要がある場合があります。詳細については、「コミットメントポリシーの設定方法」を参照してください。

AWS Encryption SDK for .NET は、仕様を記述する正式な検証言語である Dafny AWS Encryption SDK の の製品であり、実装するコード、およびテストするための証明です。その結果、機能の正確性を保証するフレームワークに、 AWS Encryption SDK の機能を実装するライブラリができあがりました。

詳細はこちら

- 代替アルゴリズムスイートの指定 AWS Encryption SDK、暗号化されたデータキーの制限、 AWS KMS マルチリージョンキーの使用など、 でオプションを設定する方法の例については、「」を参照してくださいの設定 AWS Encryption SDK。

- AWS Encryption SDK for .NET を使用したプログラミングの詳細については、GitHub の aws-encryption-sdk リポジトリの aws-encryption-sdk-net ディレクトリを参照してください。

トピック


- AWS Encryption SDK for .NET のインストール
- for .NET AWS Encryption SDK のデバッグ
- AWS Encryption SDK for .NET の例


## AWS Encryption SDK for .NET のインストール

AWS Encryption SDK for .NET は NuGet の AWS.Cryptography.EncryptionSDKパッケージとして利用できます。for .NET AWS Encryption SDK のインストールと構築の詳細については、 aws-encryption-sdk-netリポジトリの README.md ファイルを参照してください。

バージョン 3.x

AWS Encryption SDK for .NET のバージョン 3.x は、Windows でのみ .NET Framework 4.5.2 – 4.8 をサポートしています。サポートされているすべてのオペレーティングシステムで、.NET Core 3.0 以降と.NET 5.0 以降をサポートします。

バージョン 4.x

AWS Encryption SDK for .NET のバージョン 4.x は、.NET 6.0 および .NET Framework net48 以降をサポートしています。

AWS Encryption SDK for .NET では、 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。NuGet パッケージと共にインストールされます。ただし、 AWS KMS キーを使用している場合を除き、 AWS Encryption SDK for .NET では AWS アカウント、、 AWS 認証情報、または AWS サービスとの対話は必要ありません。必要に応じて AWS アカウントを設定する方法については、「」を参照してくださいAWS Encryption SDK で を使用する AWS KMS。

## for .NET AWS Encryption SDK のデバッグ

AWS Encryption SDK for .NET はログを生成しません。for .NET AWS Encryption SDK 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 for .NET AWS Encryption SDK 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)。

## AWS Encryption SDK for .NET の例

次の例は、 AWS Encryption SDK for .NET でプログラミングするときに使用する基本的なコーディングパターンを示しています。具体的には、 AWS Encryption SDK とマテリアルプロバイダーライブラリをインスタンス化します。次に、各メソッドを呼び出す前に、メソッドの入力を定義するオブジェクトをインスタンス化します。これは、 SDK for .NETで使用するコーディングパターンとよく似ています。

代替アルゴリズムスイートの指定 AWS Encryption SDK、暗号化されたデータキーの制限、 AWS KMS マルチリージョンキーの使用など、 でオプションを設定する方法の例については、「」を参照してくださいの設定 AWS Encryption SDK。

for .NET を使用したプログラミングのその他の例については、GitHub AWS Encryption SDK のaws-encryption-sdkリポジトリの aws-encryption-sdk-net ディレクトリにある例を参照してください。

### 「.NET 用 AWS Encryption SDK 」でのデータの暗号化

この例では、データを暗号化するための基本的なパターンを示しています。　1 つのラ AWS KMS ッピングキーで保護されたデータキーを持つ小さなファイルを暗号化します。

ステップ 1: AWS Encryption SDK とマテリアルプロバイダーライブラリをインスタンス化します。

まず、 AWS Encryption SDK とマテリアルプロバイダーライブラリをインスタンス化します。のメソッドを使用して、データを AWS Encryption SDK 暗号化および復号します。マテリアルプロバイダライブラリのメソッドを使用して、データを保護するキーを指定するキーリングを作成します。

AWS Encryption SDK とマテリアルプロバイダーライブラリをインスタンス化する方法は、.NET AWS Encryption SDK 用 のバージョン 3.x と 4.x で異なります。次の手順はすべて、.NET AWS Encryption SDK 用 のバージョン 3.x と 4.x の両方で同じです。

Version 3.x

```
// Instantiate the AWS Encryption SDK and material providers
```

```
var encryptionSdk = AwsEncryptionSdkFactory.CreateDefaultAwsEncryptionSdk();
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders(
```

Version 4.x

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

ステップ 2: キーリング用の入力オブジェクトを作成します。

キーリングを作成する各メソッドには、対応する入力オブジェクトクラスがあります。たとえば、`CreateAwsKmsKeyring()` メソッドの入力オブジェクトを作成するには、`CreateAwsKmsKeyringInput` クラスのインスタンスを作成します。

このキーリングの入力では ジェネレータキー は指定されていませんが、`KmsKeyId` パラメータで指定される単一の KMS キーはジェネレータキーとなります。データを暗号化するデータキーを生成し、暗号化します。

この入力オブジェクトには、KMS キー AWS リージョン の 用の AWS KMS クライアントが必要です。 AWS KMS クライアントを作成するには、 で `AmazonKeyManagementServiceClient` クラスをインスタンス化します SDK for .NET。パラメータなしで `AmazonKeyManagementServiceClient()` コンストラクタを呼び出すと、デフォルト値でクライアントが作成されます。

.NET AWS Encryption SDK 用 で暗号化するために使用される AWS KMS キーリングでは、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 KMS キーを識別できます。復号に使用される AWS KMS キーリングでは、キー ARN を使用して各 KMS キーを識別する必要があります。復号に暗号化キーリングを再利用する場合は、すべての KMS キーにキー ARN ID を使用します。

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

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
```

```
    KmsKeyId = keyArn
};
```

ステップ 3: キーリングを作成します。

キーリングを作成するには、キーリング入力オブジェクトを使用してキーリングメソッドを呼び出します。この例では、KMS キーを 1 つだけ取得する、この `CreateAwsKmsKeyring()` メソッドを使用しています。

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

ステップ 4: 暗号化コンテキストを定義します。

暗号化コンテキストはオプションですが、 の暗号化オペレーションでは強く推奨されます AWS Encryption SDK。1 つ以上の非シークレットキーと値のペアを定義できます。

> **Note**
>
> .NET AWS Encryption SDK 用 のバージョン 4.x では、必要な暗号化コンテキスト CMM を使用して、すべての暗号化リクエストで暗号化コンテキストを要求できます。

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

ステップ 5: 暗号化用の入力オブジェクトを作成します。

`Encrypt()` メソッドを呼び出す前に、`EncryptInput` クラスのインスタンスを作成します。

```
string plaintext = File.ReadAllText("C:\\Documents\\CryptoTest\\TestFile.txt");

// Define the encrypt input
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
```

```
};
```

ステップ 6: プレーンテキストを暗号化します。

の Encrypt()メソッド AWS Encryption SDK を使用して、定義したキーリングを使用してプレーンテキストを暗号化します。

この Encrypt() メソッドが返す EncryptOutput には、暗号化されたメッセージ (Ciphertext)、暗号化コンテキスト、アルゴリズムスイートを取得するメソッドがあります。

```
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

ステップ 7: 暗号化されたメッセージを取得します。

for .NET AWS Encryption SDK の Decrypt()メソッドは、EncryptOutputインスタンスCiphertextのメンバーを取得します。

EncryptOutput オブジェクトの Ciphertext メンバーは 暗号化されたメッセージ であり、暗号化されたデータ、暗号化されたデータキー、メタデータ (暗号化コンテキストを含む) を含むポータブルオブジェクトです。暗号化されたメッセージを長期間安全に保管したり、Decrypt()メソッドに送信してプレーンテキストを復元することもできます。

```
var encryptedMessage = encryptOutput.Ciphertext;
```

## 「.NET 用 AWS Encryption SDK 」では、Strict モードで復号化

ベストプラクティスでは、データを復号する際に使用するキーを指定することを推奨していますが、これは Strict モードと呼ばれるオプションです。 AWS Encryption SDK は、キーリングで指定した KMS キーのみを使用して暗号文を復号します。復号化キーリング内のキーには、データを暗号化したキーが少なくとも 1 つ含まれている必要があります。

この例は、.NET 用 AWS Encryption SDK による Strict モードでの復号化の基本パターンを示しています。

ステップ 1: AWS Encryption SDK およびマテリアルプロバイダーライブラリをインスタンス化します。

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

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

ステップ 2: キーリング用の入力オブジェクトを作成します。

キーリングメソッドのパラメータを指定するには、入力オブジェクトを作成します。 AWS Encryption SDK for .NET の各キーリングメソッドには、対応する入力オブジェクトがあります。この例では、CreateAwsKmsKeyring() メソッドを使用してキーリングを作成しているため、入力用の CreateAwsKmsKeyringInput クラスをインスタンス化しています。

復号キーリングでは、キー ARN を使用して KMS キー を指定する必要があります。

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

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
```

ステップ 3: キーリングを作成します。

復号化キーリングを作成するために、この例では CreateAwsKmsKeyring() メソッドとキーリング入力オブジェクトを使用します。

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

ステップ 4: 復号化用の入力オブジェクトを作成します。

Decrypt() メソッドの入力オブジェクトを作成するには、DecryptInput クラスをインスタンス化します。

DecryptInput() コンストラクタの Ciphertext パラメータは、Encrypt() メソッドが返した EncryptOutput オブジェクトの Ciphertext メンバーを受け取ります。Ciphertext プロパティは 暗号化されたメッセージ を表します。これには、 AWS Encryption SDK がメッセージの復号化に必要な暗号化されたデータ、暗号化されたデータキー、メタデータが含まれます。

.NET AWS Encryption SDK 用 のバージョン 4.x では、オプションの EncryptionContextパラメータを使用して、 Decrypt()メソッドで暗号化コンテキストを指定できます。

この EncryptionContext パラメータを使用して、暗号化時に使用された暗号化コンテキストが、暗号文の復号化に使用された暗号化コンテキストに含まれていることを確認します。は、デフォルトのアルゴリズムスイートなどの署名付きアルゴリズムスイートを使用している場合、デジタル署名を含むペアを暗号化コンテキスト AWS Encryption SDK に追加します。

```
var encryptedMessage = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = encryptedMessage,
    Keyring = keyring,
    EncryptionContext = encryptionContext // OPTIONAL
};
```

ステップ 5: 暗号文を復号化します。

```
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

ステップ 6: 暗号化コンテキストを確認します — バージョン 3.x

for .NET のバージョン 3.x AWS Encryption SDK の Decrypt()メソッドは、暗号化コンテキストを使用しません。暗号化されたメッセージのメタデータから暗号化コンテキストの値を取得します。ただし、プレーンテキストを返したり使用したりする前に、暗号文の復号に使用した暗号化コンテキストに、暗号化時に指定した暗号化コンテキストが含まれていることを確認することがベストプラクティスです。

暗号化に使用した暗号化コンテキストが、暗号文の復号に使用された暗号化コンテキストに含まれていることを確認します。は、デフォルトのアルゴリズムスイートなどの署名付きアルゴリズムスイートを使用している場合、デジタル署名を含むペアを暗号化コンテキスト AWS Encryption SDK に追加します。

```
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var
 decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

## AWS Encryption SDK for .NET の検出キーリングを使用した復号

復号化に KMS キーを指定するのではなく、KMS キーを指定しないキーリングである AWS KMS ディスカバリーキーリングを提供できます。検出キーリングを使用すると、発信者がキーに対する AWS Encryption SDK 復号アクセス許可を持っている限り、 は暗号化された KMS キーを使用してデータを復号できます。ベストプラクティスとして、 AWS アカウント 特定のパーティションで使用できる KMS キーを制限する検出フィルターを追加します。

AWS Encryption SDK for .NET には、 AWS KMS クライアントを必要とする基本的な検出キーリングと、1 つ以上の を指定する必要がある検出マルチキーリングが用意されています AWS リージョン。クライアントとリージョンはどちらも、暗号化されたメッセージの復号化に使用できる KMS キーを制限します。　どちらのキーリングの入力オブジェクトにも、推奨ディスカバリーフィルターが適用されます。

次の例は、 AWS KMS ディスカバリーキーリングとディスカバリーフィルターを使用してデータを復号化するパターンを示しています。

ステップ 1: AWS Encryption SDK とマテリアルプロバイダーライブラリをインスタンス化します。

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

ステップ 2: キーリング用の入力オブジェクトを作成します。

キーリングメソッドのパラメータを指定するには、入力オブジェクトを作成します。　AWS Encryption SDK for .NET の各キーリングメソッドには、対応する入力オブジェクトがあります。この例では、`CreateAwsKmsDiscoveryKeyring()` メソッドを使用してキーリングを作成しているため、入力用の `CreateAwsKmsDiscoveryKeyringInput` クラスをインスタンス化しています。

```
List<string> accounts = new List<string> { "111122223333" };

var discoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = accounts,
        Partition = "aws"
```

```
    }
};
```

ステップ 3: キーリングを作成します。

復号化キーリングを作成するために、この例では CreateAwsKmsDiscoveryKeyring() メソッドとキーリング入力オブジェクトを使用します。

```
var discoveryKeyring =
 materialProviders.CreateAwsKmsDiscoveryKeyring(discoveryKeyringInput);
```

ステップ 4: 復号化用の入力オブジェクトを作成します。

Decrypt() メソッドの入力オブジェクトを作成するには、DecryptInput クラスをインスタンス化します。Ciphertext パラメータの値は、Encrypt() メソッドが返す EncryptOutput オブジェクトの Ciphertext メンバーです。

.NET AWS Encryption SDK 用 のバージョン 4.x では、オプションの EncryptionContextパラメータを使用して、 Decrypt()メソッドで暗号化コンテキストを指定できます。

この EncryptionContext パラメータを使用して、暗号化時に使用された暗号化コンテキストが、暗号文の復号化に使用された暗号化コンテキストに含まれていることを確認します。は、デフォルトのアルゴリズムスイートなどの署名付きアルゴリズムスイートを使用している場合、デジタル署名を含むペアを暗号化コンテキスト AWS Encryption SDK に追加します。

```
var ciphertext = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = discoveryKeyring,
    EncryptionContext = encryptionContext // OPTIONAL

};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

ステップ 5: 暗号化コンテキストを確認します — バージョン 3.x

for .NET のバージョン 3.x AWS Encryption SDK の Decrypt()メソッドは、 で暗号化コンテキストを使用しませんDecrypt()。暗号化されたメッセージのメタデータから暗号化コンテキストの値を取得します。ただし、プレーンテキストを返したり使用したりする前に、暗号文の復号に

使用した暗号化コンテキストに、暗号化時に指定した暗号化コンテキストが含まれていることを確認することがベストプラクティスです。

暗号化で使用された暗号化コンテキストが、暗号文の復号に使用された暗号化コンテキストに含まれていることを確認します。は、デフォルトのアルゴリズムスイートなどの署名付きアルゴリズムスイートを使用している場合、デジタル署名を含むペアを暗号化コンテキスト AWS Encryption SDK に追加します。

```
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var
 decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

# AWS Encryption SDK Go 用の

このトピックでは、 AWS Encryption SDK for Go 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 Encryption SDK for Go を使用したプログラミングの詳細については、GitHub の aws-encryption-sdk リポジトリの go ディレクトリを参照してください。

AWS Encryption SDK for Go は、以下の点 AWS Encryption SDK で の他のプログラミング言語実装とは異なります。

- データキーキャッシュはサポートされていません。ただし、 AWS Encryption SDK for Go は、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングをサポートしています。
- ストリーミングデータはサポートしていません

AWS Encryption SDK for Go には、 の他の言語実装のバージョン 2.0.x 以降で導入されたすべてのセキュリティ機能が含まれています AWS Encryption SDK。ただし、 AWS Encryption SDK for Go を使用して、 の 2.0.x より前のバージョンの別の言語実装で暗号化されたデータを復号する場合は AWS Encryption SDK、コミットメントポリシーを調整する必要がある場合があります。詳細については、「コミットメントポリシーの設定方法」を参照してください。

AWS Encryption SDK for Go は、Dafny AWS Encryption SDK の 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正式な検証言語です。 https://github.com/dafny-lang/dafny/blob/master/README.mdその結果、機能の正確性を保証するフレームワークに、AWS Encryption SDK の機能を実装するライブラリができあがりました。

詳細はこちら

- 代替アルゴリズムスイートの指定 AWS Encryption SDK、暗号化されたデータキーの制限、 AWS KMS マルチリージョンキーの使用など、 でオプションを設定する方法の例については、「」を参照してくださいの設定 AWS Encryption SDK。
- AWS Encryption SDK for Go を設定して使用する方法の例については、GitHub の aws-encryption-sdk リポジトリにある Go の例を参照してください。

トピック


- 前提条件
- インストール


## 前提条件

AWS Encryption SDK for Go 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。

Go のサポートされているバージョン

Go 1.23 以降は、 が Go AWS Encryption SDK に必要です。

Go のダウンロードとインストールの詳細については、「Go installation」を参照してください。

## インストール

AWS Encryption SDK for Go の最新バージョンをインストールします。 AWS Encryption SDK for Go のインストールと構築の詳細については、GitHub の aws-encryption-sdk リポジトリの go ディレクトリにある README.md を参照してください。

最新バージョンをインストールするには

- AWS Encryption SDK for Go をインストールする

```
go get github.com/aws/aws-encryption-sdk/releases/go/encryption-sdk@latest
```

- 暗号化マテリアルプロバイダーライブラリ (MPL) をインストールする

```
go get github.com/aws/aws-cryptographic-material-providers-library/releases/go/mpl
```

# AWS Encryption SDK for Java

このトピックでは、 AWS Encryption SDK 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。を使用したプログラミングの詳細については AWS Encryption SDK for Java、GitHub の aws-encryption-sdk-java リポジトリを参照してください。API のドキュメントについては、 AWS Encryption SDK for Javaの Javadoc を参照してください。

トピック

- 前提条件
- インストール
- AWS Encryption SDK for Java の例

## 前提条件

をインストールする前に AWS Encryption SDK for Java、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。

Bouncy Castle

には Bouncy Castle AWS Encryption SDK for Java が必要です。

- AWS Encryption SDK for Java バージョン 1.6.1 以降では、Bouncy Castle を使用して暗号化オブジェクトをシリアル化および逆シリアル化します。この要件を満たすには、Bouncy Castle または Bouncy Castle FIPS を使用できます。Bouncy Castle FIPS のインストールおよび設定

については、Bouncy Castle FIPS のドキュメントの特にユーザーガイドとセキュリティポリシーの PDF を参照してください。

- の以前のバージョンでは、Bouncy Castle の Java 用暗号化 API AWS Encryption SDK for Java を使用しています。この要件は、FIPS 非対応の Bouncy Castle によってのみ満たされます。

Bouncy Castle がない場合は、Java 用 Bouncy Castle のダウンロードに移動して、JDK に対応するプロバイダーファイルをダウンロードします。また、Apache Maven を使用して、標準の Bouncy Castle プロバイダー (bcprov-ext-jdk15on) のアーティファクトや Bouncy Castle FIPS (bc-fips) のアーティファクトを取得することもできます。

AWS SDK for Java

のバージョン 3.x では AWS SDK for Java 2.x、 AWS KMS キーリングを使用していない場合でも、 AWS Encryption SDK for Java が必要です。

のバージョン 2.x 以前 AWS Encryption SDK for Java では、 は必要ありません AWS SDK for Java。ただし、マスターキープロバイダーとして AWS Key Management Service (AWS KMS) を使用するには、 AWS SDK for Java が必要です。 AWS Encryption SDK for Java バージョン 2.4.0 以降、 AWS Encryption SDK for Java は、バージョン 1.x と 2.x の両方の AWS SDK for Java 1.x と 2.x の . AWS SDK for Java AWS Encryption SDK code をサポートしています。たとえば、AWS SDK for Java 1.x をサポートする AWS Encryption SDK コードでデータを暗号化し、 をサポートするコードを使用して復号できます AWS SDK for Java 2.x （またはその逆）。2.4.0 より AWS Encryption SDK for Java 前のバージョンの は、 AWS SDK for Java 1.x のみをサポートしています。のバージョンの更新については AWS Encryption SDK、「」を参照してくださいの移行 AWS Encryption SDK。

AWS Encryption SDK for Java コードを AWS SDK for Java 1.x から に更新するときは AWS SDK for Java 2.x、`AWSKMS`インターフェイス in AWS SDK for Java 1.x への参照を の`KmsClient`インターフェイスへの参照に置き換えます AWS SDK for Java 2.x。 AWS Encryption SDK for Java は `KmsAsyncClient`インターフェイスをサポートしていません。また、`kms` 名前空間の代わりに、`kmssdkv2` 名前空間の AWS KMS関連オブジェクトを使用するようにコードを更新してください。

をインストールするには AWS SDK for Java、Apache Maven を使用します。

- 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。
- モジュール AWS KMS in AWS SDK for Java 1.x にのみ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、 `artifactId`を に設定しますaws-java-sdk-kms。

- AWS KMS モジュール in AWS SDK for Java 2.x にのみ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 `kms` に設定します。

その他の変更点については、「 デベロッパーガイド」の AWS SDK for Java 「1.x と 2.x の違い」を参照してください。 AWS SDK for Java 2.x

「 AWS Encryption SDK デベロッパーガイド」の Java の例は、 を使用します AWS SDK for Java 2.x。

## インストール

AWS Encryption SDK for Javaの最新バージョンをインストールします。

> **Note**
>
> 2.0.0 より AWS Encryption SDK for Java 前の のすべてのバージョンはend-of-supportフェーズにあります。
> バージョン 2.0.x 以降から AWS Encryption SDK for Java の最新バージョンにコードやデータを変更せずに安全に更新できます。ただし、バージョン 2.0.x で導入された新しいセキュリティ機能には下位互換性がありません。1.7.x より前のバージョンから 2.0.x 以降のバージョンに更新するには、まず AWS Encryption SDKの最新の 1.x バージョンに更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。

は以下の AWS Encryption SDK for Java 方法でインストールできます。

手動

をインストールするには AWS Encryption SDK for Java、aws-encryption-sdk-java 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。

Apache Maven の使用

AWS Encryption SDK for Java は、次の依存関係定義を使用して Apache Maven から利用できます。

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>3.0.0</version>
```

```
</dependency>
```

SDK をインストールしたら、このガイドの Java コード例と GitHub の Javadoc を見て開始します。

## AWS Encryption SDK for Java の例

次の例は、 AWS Encryption SDK for Java を使用してデータを暗号化および復号する方法を示しています。これらの例は、 のバージョン 3.x 以降を使用する方法を示しています AWS Encryption SDK for Java。のバージョン 3.x には AWS Encryption SDK for Java が必要です AWS SDK for Java 2.x。のバージョン 3.x は AWS Encryption SDK for Java 、マスターキープロバイダーをキーリングに置き換えます。前バージョンを使用する例については、GitHub の aws-encryption-sdk-java リポジトリのリリースリストで使用中のリリースを検索してください。

トピック

- 文字列の暗号化と復号
- バイトストリームの暗号化と復号
- マルチキーリングを使用したバイトストリームの暗号化と復号

### 文字列の暗号化と復号

次の例は、 のバージョン 3.x を使用して文字列 AWS Encryption SDK for Java を暗号化および復号する方法を示しています。文字列を使用する前にバイト配列に変換します。

この例では、 AWS KMS キーリングを使用します。 AWS KMS キーリングで暗号化する場合、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 KMS キーを識別できます。復号するときは、キー ARN を使用して KMS キーを識別する必要があります。

`encryptData()` メソッドを呼び出すと、暗号化テキスト、暗号化されたデータキー、暗号化コンテキストを含む暗号化されたメッセージ (`CryptoResult`) が返されます。`CryptoResult` オブジェクトで `getResult` を呼び出すと、暗号化されたメッセージの Base-64 でエンコードされた文字列バージョンが返され、`decryptData()` メソッドに渡すことができるようになります。

同様に、 を呼び出すと`decryptData()`、返される`CryptoResult`オブジェクトにはプレーンテキストメッセージと AWS KMS key ID が含まれます。アプリケーションがプレーンテキストを返す前に、暗号化されたメッセージの AWS KMS key ID と暗号化コンテキストが想定どおりであることを確認します。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import
 software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;

import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.Collections;
import java.util.Map;

/**
 * Encrypts and then decrypts data using an AWS KMS Keyring.
 *
 * <p>Arguments:
 *
 * <ol>
 *   <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS
 customer master
 *       key (CMK), see 'Viewing Keys' at
 *       http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html
 * </ol>
 */
public class BasicEncryptionKeyringExample {

  private static final byte[] EXAMPLE_DATA = "Hello
 World".getBytes(StandardCharsets.UTF_8);

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

    encryptAndDecryptWithKeyring(keyArn);
  }

  public static void encryptAndDecryptWithKeyring(final String keyArn) {
    // 1. Instantiate the SDK
```

```
    // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
 commitment policy,
    // which means this client only encrypts using committing algorithm suites and
 enforces
    // that the client will only decrypt encrypted messages that were created with a
 committing
    // algorithm suite.
    // This is the default commitment policy if you build the client with
    // `AwsCrypto.builder().build()`
    // or `AwsCrypto.standard()`.
    final AwsCrypto crypto =
        AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();

    // 2. Create the AWS KMS keyring.
    // This example creates a multi keyring, which automatically creates the KMS
 client.
    final MaterialProviders materialProviders =
        MaterialProviders.builder()
            .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
            .build();
    final CreateAwsKmsMultiKeyringInput keyringInput =
        CreateAwsKmsMultiKeyringInput.builder().generator(keyArn).build();
    final IKeyring kmsKeyring =
 materialProviders.CreateAwsKmsMultiKeyring(keyringInput);

    // 3. Create an encryption context
    // We recommend using an encryption context whenever possible
    // to protect integrity. This sample uses placeholder values.
    // For more information see:
    // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-Integrity-
of-Your-Encrypted-Data-by-Using-AWS-Key-Management
    final Map<String, String> encryptionContext =
        Collections.singletonMap("ExampleContextKey", "ExampleContextValue");

    // 4. Encrypt the data
    final CryptoResult<byte[], ?> encryptResult =
        crypto.encryptData(kmsKeyring, EXAMPLE_DATA, encryptionContext);
    final byte[] ciphertext = encryptResult.getResult();

    // 5. Decrypt the data
    final CryptoResult<byte[], ?> decryptResult =
        crypto.decryptData(
```

```
            kmsKeyring,
            ciphertext,
            // Verify that the encryption context in the result contains the
            // encryption context supplied to the encryptData method
            encryptionContext);

    // 6. Verify that the decrypted plaintext matches the original plaintext
    assert Arrays.equals(decryptResult.getResult(), EXAMPLE_DATA);
  }
}
```

## バイトストリームの暗号化と復号

次の例は、 を使用してバイトストリーム AWS Encryption SDK を暗号化および復号する方法を示しています。

この例では、Raw AES キーリングを使用します。

暗号化するときには、AwsCrypto.builder() .withEncryptionAlgorithm() メソッドを使用して、デジタル署名のないアルゴリズムスイートを指定します。復号化時に、暗号化テキストが署名なしであることを確認するために、この例では createUnsignedMessageDecryptingStream() メソッドを使用します。createUnsignedMessageDecryptingStream() メソッドは、デジタル署名を持つ暗号文を検出すると失敗します。

デジタル署名を含むデフォルトのアルゴリズムスイートで暗号化する場合は、次の例に示すように、代わりに createDecryptingStream() メソッドを使用します。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoAlgorithm;
import com.amazonaws.encryptionsdk.CryptoInputStream;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import software.amazon.cryptography.materialproviders.model.AesWrappingAlg;
import software.amazon.cryptography.materialproviders.model.CreateRawAesKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.security.SecureRandom;
import java.util.Collections;
import java.util.Map;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;


/**
 * <p>
 * Encrypts and then decrypts a file under a random key.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 *
 * <p>
 * This program demonstrates using a standard Java {@link SecretKey} object as a {@link
 IKeyring} to
 * encrypt and decrypt streaming data.
 */
public class FileStreamingKeyringExample {
    private static String srcFile;

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

        // In this example, we generate a random key. In practice,
        // you would get a key from an existing store
        SecretKey cryptoKey = retrieveEncryptionKey();

        // Create a Raw Aes Keyring using the random key and an AES-GCM encryption
 algorithm
        final MaterialProviders materialProviders = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateRawAesKeyringInput keyringInput =
 CreateRawAesKeyringInput.builder()
```

```
                .wrappingKey(ByteBuffer.wrap(cryptoKey.getEncoded()))
                .keyNamespace("Example")
                .keyName("RandomKey")
                .wrappingAlg(AesWrappingAlg.ALG_AES128_GCM_IV12_TAG16)
                .build();
        IKeyring keyring = materialProviders.CreateRawAesKeyring(keyringInput);

        // Instantiate the SDK.
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
        // which means this client only encrypts using committing algorithm suites and
enforces
        // that the client will only decrypt encrypted messages that were created with
a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        // This example encrypts with an algorithm suite that doesn't include signing
for faster decryption,
        // since this use case assumes that the contexts that encrypt and decrypt are
equally trusted.
        final AwsCrypto crypto = AwsCrypto.builder()
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)

.withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
                .build();

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

        // Because the file might be too large to load into memory, we stream the data,
instead of
        //loading it all at once.
        FileInputStream in = new FileInputStream(srcFile);
        CryptoInputStream<JceMasterKey> encryptingStream =
crypto.createEncryptingStream(keyring, in, context);

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

```
        // Decrypt the file. Verify the encryption context before returning the
 plaintext.
        // Since the data was encrypted using an unsigned algorithm suite, use the
 recommended
        // createUnsignedMessageDecryptingStream method, which only accepts unsigned
 messages.
        in = new FileInputStream(srcFile + ".encrypted");
        CryptoInputStream<JceMasterKey> decryptingStream =
 crypto.createUnsignedMessageDecryptingStream(keyring, in);
        // Does it contain the expected encryption context?
        if
 (!"FileStreaming".equals(decryptingStream.getCryptoResult().getEncryptionContext().get("Exampl
 {
            throw new IllegalStateException("Bad encryption context");
        }

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

    /**
     * In practice, this key would be saved in a secure location.
     * For this demo, we generate a new random key for each operation.
     */
    private static SecretKey retrieveEncryptionKey() {
        SecureRandom rnd = new SecureRandom();
        byte[] rawKey = new byte[16]; // 128 bits
        rnd.nextBytes(rawKey);
        return new SecretKeySpec(rawKey, "AES");
    }
}
```

## マルチキーリングを使用したバイトストリームの暗号化と復号

次の例は、マルチキーリング AWS Encryption SDK で を使用する方法を示しています。 ???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。この例では、 AWS KMS キーリングと Raw RSA キーリングを子キーリングとして使用します。

この例では、デジタル署名を含むデフォルトのアルゴリズムスイートで暗号化します。ストリーミング時、 は、整合性チェックの後、デジタル署名を検証する前にプレーンテキストを AWS Encryption SDK リリースします。署名が検証されるまでプレーンテキストを使用しないようにするため、この例ではプレーンテキストをバッファリングし、復号化および検証が完了したときにのみディスクに書き込みます。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoOutputStream;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import
 software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateRawRsaKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;
import software.amazon.cryptography.materialproviders.model.PaddingScheme;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.security.GeneralSecurityException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Collections;

/**
 * <p>
 * Encrypts a file using both AWS KMS Key and an asymmetric key pair.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS key,
```

```
 *   see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/developerguide/
viewing-keys.html
 *
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 * <p>
 * You might use AWS Key Management Service (AWS KMS) for most encryption and
 decryption operations, but
 * still want the option of decrypting your data offline independently of AWS KMS. This
 sample
 * demonstrates one way to do this.
 * <p>
 * The sample encrypts data under both an AWS KMS key and an "escrowed" RSA key pair
 * so that either key alone can decrypt it. You might commonly use the AWS KMS key for
 decryption. However,
 * at any time, you can use the private RSA key to decrypt the ciphertext independent
 of AWS KMS.
 * <p>
 * This sample uses the RawRsaKeyring to generate a RSA public-private key pair
 * and saves the key pair in memory. In practice, you would store the private key in a
 secure offline
 * location, such as an offline HSM, and distribute the public key to your development
 team.
 */
public class EscrowedEncryptKeyringExample {
    private static ByteBuffer publicEscrowKey;
    private static ByteBuffer privateEscrowKey;

    public static void main(final String[] args) throws Exception {
        // This sample generates a new random key for each operation.
        // In practice, you would distribute the public key and save the private key in
 secure
        // storage.
        generateEscrowKeyPair();

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

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

        escrowDecrypt(fileName);
    }
```

```
   private static void standardEncrypt(final String kmsArn, final String fileName)
throws Exception {
       // Encrypt with the KMS key and the escrowed public key
       // 1. Instantiate the SDK
       // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
       // which means this client only encrypts using committing algorithm suites and
enforces
       // that the client will only decrypt encrypted messages that were created with
a committing
       // algorithm suite.
       // This is the default commitment policy if you build the client with
       // `AwsCrypto.builder().build()`
       // or `AwsCrypto.standard()`.
       final AwsCrypto crypto = AwsCrypto.builder()
               .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
               .build();

       // 2. Create the AWS KMS keyring.
       // This example creates a multi keyring, which automatically creates the KMS
client.
       final MaterialProviders matProv = MaterialProviders.builder()
               .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
               .build();
       final CreateAwsKmsMultiKeyringInput keyringInput =
CreateAwsKmsMultiKeyringInput.builder()
               .generator(kmsArn)
               .build();
       IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

       // 3. Create the Raw Rsa Keyring with Public Key.
       final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
               .keyName("Escrow")
               .keyNamespace("Escrow")
               .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
               .publicKey(publicEscrowKey)
               .build();
       IKeyring rsaPublicKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);

       // 4. Create the multi-keyring.
       final CreateMultiKeyringInput createMultiKeyringInput =
CreateMultiKeyringInput.builder()
```

```
                .generator(kmsKeyring)
                .childKeyrings(Collections.singletonList(rsaPublicKeyring))
                .build();
        IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

        // 5. Encrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName);
        final FileOutputStream out = new FileOutputStream(fileName + ".encrypted");
        final CryptoOutputStream<?> encryptingStream =
crypto.createEncryptingStream(multiKeyring, out);

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

    private static void standardDecrypt(final String kmsArn, final String fileName)
throws Exception {
        // Decrypt with the AWS KMS key and the escrow public key.

        // 1. Instantiate the SDK.
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
        // which means this client only encrypts using committing algorithm suites and
enforces
        // that the client will only decrypt encrypted messages that were created with
a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        final AwsCrypto crypto = AwsCrypto.builder()
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
                .build();

        // 2. Create the AWS KMS keyring.
        // This example creates a multi keyring, which automatically creates the KMS
client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
```

```
        final CreateAwsKmsMultiKeyringInput keyringInput =
CreateAwsKmsMultiKeyringInput.builder()
                .generator(kmsArn)
                .build();
        IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

        // 3. Create the Raw Rsa Keyring with Public Key.
        final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .build();
        IKeyring rsaPublicKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);

        // 4. Create the multi-keyring.
        final CreateMultiKeyringInput createMultiKeyringInput =
CreateMultiKeyringInput.builder()
                .generator(kmsKeyring)
                .childKeyrings(Collections.singletonList(rsaPublicKeyring))
                .build();
        IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

        // 5. Decrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".decrypted");
        // Since we are using a signing algorithm suite, we avoid streaming decryption
directly to the output file,
        // to ensure that the trailing signature is verified before writing any
untrusted plaintext to disk.
        final ByteArrayOutputStream plaintextBuffer = new ByteArrayOutputStream();
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(multiKeyring, plaintextBuffer);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();
        final ByteArrayInputStream plaintextReader = new
ByteArrayInputStream(plaintextBuffer.toByteArray());
        IOUtils.copy(plaintextReader, out);
```

```
        out.close();
    }

    private static void escrowDecrypt(final String fileName) throws Exception {
        // You can decrypt the stream using only the private key.
        // This method does not call AWS KMS.

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

        // 2. Create the Raw Rsa Keyring with Private Key.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .privateKey(privateEscrowKey)
                .build();
        IKeyring escrowPrivateKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);


        // 3. Decrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".deescrowed");
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(escrowPrivateKeyring, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();

    }

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

```
        publicEscrowKey = RawRsaKeyringExample.getPEMPublicKey(keyPair.getPublic());
        privateEscrowKey = RawRsaKeyringExample.getPEMPrivateKey(keyPair.getPrivate());

    }
}
```

# AWS Encryption SDK for JavaScript

AWS Encryption SDK for JavaScript は、JavaScript でウェブブラウザアプリケーションまたは Node.js でウェブサーバーアプリケーションを記述しているデベロッパー向けに、クライアント側の暗号化ライブラリを提供するように設計されています。

のすべての実装と同様に AWS Encryption SDK、 は高度なデータ保護機能 AWS Encryption SDK for JavaScript を提供します。これにはエンベロープ暗号化、追加の認証データ (AAD)、キー取得および署名で使用する 256 ビット AES-GCM などのセキュアで認証済みの対称キーアルゴリズムスイートなどが含まれます。

のすべての言語固有の実装 AWS Encryption SDK は、言語の制約に従って相互運用できるように設計されています。JavaScript の言語による制約の詳細については、「the section called “互換性”」を参照してください。

詳細はこちら

- を使用したプログラミングの詳細については AWS Encryption SDK for JavaScript、GitHub の aws-encryption-sdk-javascript リポジトリを参照してください。
- プログラミング例については、aws-encryption-sdk-javascript リポジトリの「the section called “例”」および example-browser と example-node モジュールを参照してください。
- を使用してウェブアプリケーションのデータを AWS Encryption SDK for JavaScript 暗号化する実際の例については、 AWS セキュリティブログの「 AWS Encryption SDK for JavaScript と Node.js を使用してブラウザで暗号化を有効にする方法」を参照してください。

トピック

- の互換性 AWS Encryption SDK for JavaScript
- のインストール AWS Encryption SDK for JavaScript
- のモジュール AWS Encryption SDK for JavaScript
- AWS Encryption SDK for JavaScript の例

# の互換性 AWS Encryption SDK for JavaScript

AWS Encryption SDK for JavaScript は、 の他の言語実装と相互運用できるように設計されています AWS Encryption SDK。ほとんどの場合、 でデータを暗号化 AWS Encryption SDK for JavaScript し、 AWS Encryption SDK コマンドラインインターフェイスを含む他の言語実装で復号できます。また、 を使用して AWS Encryption SDK for JavaScript 、 の他の言語実装によって生成された暗号化されたメッセージを復号できます AWS Encryption SDK。

ただし、 を使用する場合は AWS Encryption SDK for JavaScript、JavaScript 言語の実装とウェブブラウザでの互換性の問題に注意する必要があります。

また、他の言語の実装を使用する場合は、必ず互換性のあるマスターキープロバイダー、マスターキー、キーリングを設定してください。詳細については、「キーリングの互換性」を参照してください。

## AWS Encryption SDK for JavaScript 互換性

の JavaScript 実装は、次の点で他の言語実装 AWS Encryption SDK とは異なります。

- の暗号化オペレーション AWS Encryption SDK for JavaScript は、フレーム化されていない暗号文を返しません。ただし、 AWS Encryption SDK for JavaScript は、 の他の言語実装によって返されるフレーム化された暗号文とフレーム化されていない暗号文を復号します AWS Encryption SDK。
- Node.js のバージョン 12.9.0 以降で、以下の RSA キーのラッピングオプションをサポートしています。
  - OAEP と SHA1、SHA256、SHA384、SHA512
  - OAEP と SHA1 および MGF1 と SHA1
  - PKCS1v15
- バージョン 12.9.0 より前の Node.js では、以下の RSA キーのラッピングオプションのみをサポートしています。
  - OAEP と SHA1 および MGF1 と SHA1
  - PKCS1v15

## ブラウザの互換性

ウェブブラウザによっては、 AWS Encryption SDK for JavaScript が必要とする基本的な暗号化オペレーションがサポートされていません。ブラウザが実装している WebCrypto API のフォールバックを設定することで、不足しているオペレーションの一部を補うことができます。

### ウェブブラウザの制限事項

以下の制限は、すべてのウェブブラウザに共通です。

- WebCrypto API では、PKCS1v15 のキーのラッピングはサポートされていません。
- ブラウザでは、192 ビットキーはサポートされていません。

### 必要な暗号化オペレーション

では、ウェブブラウザで以下のオペレーション AWS Encryption SDK for JavaScript が必要です。ブラウザでこれらのオペレーションがサポートされていない場合は、 AWS Encryption SDK for JavaScriptとの互換性がありません。

- ブラウザには、暗号化の乱数を生成するメソッドである `crypto.getRandomValues()` が含まれている必要があります。`crypto.getRandomValues()` をサポートしているウェブブラウザのバージョンについては、「Can I Use crypto.getRandomValues()?」を参照してください。

### 必要なフォールバック

では、ウェブブラウザで以下のライブラリとオペレーション AWS Encryption SDK for JavaScript が必要です。これらの要件を満たしていないウェブブラウザをサポートする場合は、フォールバックを設定する必要があります。そうしないと、ブラウザ AWS Encryption SDK for JavaScript で を使用しようとすると失敗します。

- ウェブアプリケーションで基本的な暗号化オペレーションを行う WebCrypto API は、すべてのブラウザで使用できるわけではありません。ウェブでの暗号化をサポートしているウェブブラウザのバージョンについては、「Can I Use Web Cryptography?」を参照してください。
- Safari ウェブブラウザの最新バージョンでは、 が AWS Encryption SDK 必要とする 0 バイトの AES-GCM 暗号化はサポートされていません。ブラウザが WebCrypto API を実装しているが、AES-GCM を使用してゼロバイトを暗号化できない場合、 はフォールバックライブラリをゼロバイト暗号化にのみ AWS Encryption SDK for JavaScript 使用します。他のすべてのオペレーションには、WebCrypto API を使用します。

いずれかの制限のフォールバックを設定するには、コードに次のステートメントを追加します。configureFallback 関数に不足している機能をサポートするライブラリを指定します。この例では、マイクロソフトリサーチの JavaScript 暗号化ライブラリ (`msrcrypto`) を使用していますが、互

換性のあるライブラリに置き換えることができます。詳しい例については、fallback.ts を参照してください。

```
import { configureFallback } from '@aws-crypto/client-browser'
configureFallback(msrCrypto)
```

## のインストール AWS Encryption SDK for JavaScript

AWS Encryption SDK for JavaScript は、相互に依存するモジュールのコレクションで構成されています。このモジュールのいくつかは、一緒に動作するように設計されたモジュールの集まりです。一部のモジュールは、単独で動作するように設計されています。すべての実装に必要なモジュールはほんの少しです。また、特殊な場合にのみ必要なモジュールもほんの少しです。for JavaScript のモジュールの詳細については、 AWS Encryption SDK のモジュール AWS Encryption SDK for JavaScript「」および GitHub の aws-encryption-sdk-javascript リポジトリの各モジュールの `README.md` ファイルを参照してください。

> **Note**
>
> 2.0.0 より AWS Encryption SDK for JavaScript 前の のすべてのバージョンはend-of-support フェーズにあります。
> バージョン 2.0.x 以降から AWS Encryption SDK for JavaScript の最新バージョンにコードやデータを変更せずに安全に更新できます。ただし、バージョン 2.0.x で導入された新しいセキュリティ機能には下位互換性がありません。1.7.x より前のバージョンから 2.0.x 以降のバージョンに更新するには、まず AWS Encryption SDK for JavaScriptの最新の 1.x バージョンに更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。

モジュールをインストールするには、npm パッケージマネージャーを使用します。

例えば、 AWS Encryption SDK for JavaScript Node.js の でプログラムするために必要なすべてのモジュールを含む `client-node`モジュールをインストールするには、次のコマンドを使用します。

```
npm install @aws-crypto/client-node
```

AWS Encryption SDK for JavaScript ブラウザで を使用してプログラムするために必要なすべてのモジュールを含む `client-browser`モジュールをインストールするには、次のコマンドを使用します。

```
npm install @aws-crypto/client-browser
```

の使用方法の実例については AWS Encryption SDK for JavaScript、GitHub の aws-encryption-sdk-javascript リポジトリの `example-node`および `example-browser`モジュールの例を参照してください。

## のモジュール AWS Encryption SDK for JavaScript

のモジュール AWS Encryption SDK for JavaScript を使用すると、プロジェクトに必要なコードを簡単にインストールできます。

### JavaScript Node.js 用のモジュール

client-node

AWS Encryption SDK for JavaScript Node.js の でプログラミングするために必要なすべてのモジュールが含まれています。

caching-materials-manager-node

Node.js の でデータキーキャッシュ機能をサポートする関数 AWS Encryption SDK for JavaScript をエクスポートします。

decrypt-node

データとデータストリームを表す暗号化されたメッセージを復号および検証する関数をエクスポートします。これは、`client-node` モジュールに含まれています。

encrypt-node

さまざまなタイプのデータを暗号化して署名する関数をエクスポートします。これは、`client-node` モジュールに含まれています。

example-node

Node.js の を使用したプログラミングの実例 AWS Encryption SDK for JavaScript をエクスポートします。さまざまなタイプのキーリングやさまざまなタイプのデータの例が含まれています。

hkdf-node

AWS Encryption SDK for JavaScript Node.js の が特定のアルゴリズムスイートで使用する HMAC ベースのキー導出関数 (HKDF) をエクスポートします。ブラウザ AWS Encryption SDK for JavaScript の は、WebCrypto API のネイティブ HKDF 関数を使用します。

integration-node

Node.js AWS Encryption SDK for JavaScript の が の他の言語実装と互換性があることを確認するテストを定義します AWS Encryption SDK。

kms-keyring-node

Node.js の AWS KMS キーリングをサポートする関数をエクスポートします。

raw-aes-keyring-node

Node.js で Raw AES キーリングをサポートする関数をエクスポートします。

raw-rsa-keyring-node

Node.js で Raw RSA キーリングをサポートする関数をエクスポートします。

## JavaScript ブラウザ用のモジュール

client-browser

ブラウザで を使用してプログラムするために必要なすべてのモジュール AWS Encryption SDK for JavaScript が含まれています。

caching-materials-manager-browser

ブラウザで JavaScript のデータキーキャッシュ機能をサポートする関数をエクスポートします。

decrypt-browser

データとデータストリームを表す暗号化されたメッセージを復号および検証する関数をエクスポートします。

encrypt-browser

さまざまなタイプのデータを暗号化して署名する関数をエクスポートします。

example-browser

ブラウザ AWS Encryption SDK for JavaScript での を使用したプログラミングの実例。さまざまなタイプのキーリングやさまざまなタイプのデータの例が含まれています。

integration-browser

ブラウザのスクリプトが AWS Encryption SDK for Javaの他の言語実装と互換性があることを確認するテストを定義します AWS Encryption SDK。

kms-keyring-browser

ブラウザで AWS KMS キーリングをサポートする関数をエクスポートします。

raw-aes-keyring-browser

ブラウザで Raw AES キーリングをサポートする関数をエクスポートします。

raw-rsa-keyring-browser

ブラウザで Raw RSA キーリングをサポートする関数をエクスポートします。

### すべての実装用のモジュール

cache-material

データキーキャッシュ機能をサポートします。各データキーでキャッシュされる暗号化マテリアルを収集するためのコードを提供します。

kms-keyring

KMS キーリングをサポートする関数をエクスポートします。

material-management

暗号化マテリアルマネージャー (CMM) を実装します。

raw-keyring

Raw AES キーリングと Raw RSA キーリングに必要な関数をエクスポートします。

serialize

SDK が出力をシリアル化するために使用する関数をエクスポートします。

web-crypto-backend

ブラウザの で WebCrypto API を使用する関数をエクスポート AWS Encryption SDK for JavaScript します。

## AWS Encryption SDK for JavaScript の例

以下の例では、 AWS Encryption SDK for JavaScript を使用してデータの暗号化と復号を行う方法を示します。

を使用するその他の例については、GitHub の aws-encryption-sdk-javascript リポジトリ AWS Encryption SDK for JavaScript の example-node モジュールと example-browser モジュールを参照してください。 https://github.com/aws/aws-encryption-sdk-javascript/tree/master/modules/example-nodeこれらのサンプルモジュールは、`client-browser` モジュールや `client-node` モジュールのインストール時にはインストールされません。

詳しいサンプルコードについては、ノードの場合は kms_simple.ts、ブラウザの場合は kms_simple.ts を参照してください。

トピック

- AWS KMS キーリングによるデータの暗号化
- AWS KMS キーリングを使用したデータの復号

## AWS KMS キーリングによるデータの暗号化

次の例は、 を使用して短い文字列またはバイト配列を AWS Encryption SDK for JavaScript 暗号化および復号する方法を示しています。

この例では、 AWS KMS キーリングを使用しています。これは、 を使用してデータキー AWS KMS key を生成および暗号化するキーリングの一種です。の作成については AWS KMS key、「 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。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してください。 AWS KMS キーリング AWS KMS keys での の識別

ステップ 1: コミットメントポリシーを設定します。

> のバージョン 1.7.x 以降では AWS Encryption SDK for JavaScript、 AWS Encryption SDK クライアントをインスタンス化する新しい `buildClient`関数を呼び出すときに、コミットメントポリシーを設定できます。`buildClient` 関数は、コミットメントポリシーを表す列挙値を取ります。更新された `encrypt` 関数と `decrypt` 関数が返されて、暗号化および復号化時にコミットメントポリシーが適用されます。
>
> 次の例では、 `buildClient`関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。
>
> JavaScript Browser

```
import {
```

```
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

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

JavaScript Node.js

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

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

ステップ 2: キーリングを作成します。

暗号化用の AWS KMS キーリングを作成します。

AWS KMS キーリングで暗号化する場合は、ジェネレーターキー、つまりプレーンテキストのデータキーを生成して暗号化する AWS KMS key ために使用される を指定する必要があります。また、同じプレーンテキストのデータキーを暗号化する追加のキーを必要な数だけ指定することもできます。キーリングは、プレーンテキストのデータキーと、ジェネレーターキーを含むキーリング AWS KMS key 内の各 の暗号化されたデータキーの 1 つのコピーを返します。データを復号するには、この暗号化されたデータキーのいずれかを復号する必要があります。

で暗号化キーリング AWS KMS keys の を指定するには AWS Encryption SDK for JavaScript、サポートされている任意の AWS KMS キー識別子を使用できます。この例では、エイリアス ARN で指定するジェネレーターキーとキー ARN で指定する 1 つの追加のキーを使用します。

> **Note**
>
> AWS KMS キーリングを復号に再利用する場合は、キー ARNs を使用してキーリング AWS KMS keys 内の を識別する必要があります。

このコードを実行する前に、サンプル AWS KMS key 識別子を有効な識別子に置き換えます。キーリングの AWS KMS keysを使用するために必要なアクセス許可を持っている必要があります。

JavaScript Browser

まず、ブラウザの認証情報を指定します。この AWS Encryption SDK for JavaScript の例では、認証情報の定数を実際の認証情報に置き換える webpack.DefinePlugin を使用しています。ただし、認証情報の指定には任意の方法を使用することができます。次に、認証情報を使用して AWS KMS クライアントを作成します。

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

const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
})
```

次に、ジェネレーターキーと追加キー AWS KMS keys の を指定します。次に、 クライアントと を使用して AWS KMS AWS KMS キーリングを作成します AWS KMS keys。

```
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

const keyring = new KmsKeyringBrowser({ clientProvider, generatorKeyId, keyIds })
```

JavaScript Node.js

```
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'
```

```
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })
```

ステップ 3: 暗号化コンテキストを設定します。

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です。暗号化時に暗号化コンテキストを指定すると、 は暗号化コンテキストを暗号化テキストに AWS Encryption SDK 暗号的にバインドし、データの復号に同じ暗号化コンテキストが必要になります。暗号化コンテキストの使用はオプションですが、ベストプラクティスとして推奨します。

暗号化コンテキストのペアを含むシンプルなオブジェクトを作成します。各ペアのキーと値は、文字列である必要があります。

JavaScript Browser

```
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

JavaScript Node.js

```
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

ステップ 4: データを暗号化します。

プレーンテキストのデータを暗号化するには、encrypt 関数を呼び出します。 AWS KMS キーリング、プレーンテキストデータ、および暗号化コンテキストを渡します。

encrypt 関数は、暗号化されたデータ、暗号化されたデータキー、重要なメタデータ (暗号化コンテキストや署名など) を含む暗号化されたメッセージ (result) を返します。

この暗号化されたメッセージを復号するには、サポートされている任意のプログラミング言語 AWS Encryption SDK の を使用します。

JavaScript Browser

```
const plaintext = new Uint8Array([1, 2, 3, 4, 5])

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

JavaScript Node.js

```
const plaintext = 'asdf'

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

## AWS KMS キーリングを使用したデータの復号

を使用して AWS Encryption SDK for JavaScript 、暗号化されたメッセージを復号し、元のデータを復元できます。

この例では、「the section called “AWS KMS キーリングによるデータの暗号化”」の例で暗号化したデータを復号します。

ステップ 1: コミットメントポリシーを設定します。

のバージョン 1.7.x 以降では AWS Encryption SDK for JavaScript、 AWS Encryption SDK クライアントをインスタンス化する新しい buildClient関数を呼び出すときに、コミットメントポリシーを設定できます。buildClient 関数は、コミットメントポリシーを表す列挙値を取ります。更新された encrypt 関数と decrypt 関数が返されて、暗号化および復号化時にコミットメントポリシーが適用されます。

次の例では、 buildClient関数を使用してデフォルトのコミットメントポリシー を指定しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。を使用してbuildClient、暗号化されたメッセージ内の暗号化されたデータキーの数を制限することもできます。詳細については、「the section called “暗号化されたデータキーの制限”」を参照してください。

JavaScript Browser

```
import {
  KmsKeyringBrowser,
```

```
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

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

JavaScript Node.js

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

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

ステップ 2: キーリングを作成します。

データを復号するには、encrypt 関数が返す暗号化されたメッセージ (result) を渡します。暗号化されたメッセージには、暗号化されたデータ、暗号化されたデータキー、重要なメタデータ (暗号化コンテキストや署名など) が含まれています。

AWS KMS キーリングは、復号時にも指定する必要があります。データの暗号化に使用したも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。成功するには、復号キーリングの少なくとも AWS KMS key 1 つの が、暗号化されたメッセージ内の暗号化されたデータキーの 1 つを復号できる必要があります。データキーは生成されないため、復号キーリングでジェネレーターキーを指定する必要はありません。指定しても、ジェネレーターキーと追加のキーは同じように扱われます。

で復号キーリング AWS KMS key の を指定するには AWS Encryption SDK for JavaScript、キー ARN を使用する必要があります。それ以外の場合、 AWS KMS key は認識されません。 AWS KMS キーリング AWS KMS keys で を識別する方法については、「」を参照してください。 AWS KMS キーリング AWS KMS keys での の識別

> **Note**
>
> 暗号化と復号に同じキーリングを使用する場合は、キー ARNs を使用してキーリング AWS KMS keys 内の を識別します。

この例では、暗号化キーリング AWS KMS keys に の 1 つだけを含むキーリングを作成します。このコードを実行する前に、キー ARN を有効なキー ARN に置き換えます。 AWS KMS keyに対する `kms:Decrypt` アクセス許可が必要です。

JavaScript Browser

まず、ブラウザの認証情報を指定します。この AWS Encryption SDK for JavaScript の例では、認証情報の定数を実際の認証情報に置き換える webpack.DefinePlugin を使用しています。ただし、認証情報の指定には任意の方法を使用することができます。次に、認証情報を使用して AWS KMS クライアントを作成します。

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

const clientProvider = getClient(KMS, {
  credentials: {
    accessKeyId,
    secretAccessKey,
    sessionToken
  }
})
```

次に、 AWS KMS クライアントを使用して AWS KMS キーリングを作成します。この例では、暗号化キーリングから の 1 AWS KMS keys つだけを使用します。

```
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']

const keyring = new KmsKeyringBrowser({ clientProvider, keyIds })
```

JavaScript Node.js

```
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
```

```
const keyring = new KmsKeyringNode({ keyIds })
```

ステップ 3: データを復号します。

次に、decrypt 関数を呼び出します。先ほど作成した復号化キーリング (keyring) および encrypt 関数が返す暗号化されたメッセージ (result) を渡します。 AWS 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 つを復号します。次に、そのプレーンテキストのデータキーを使用してデータを復号します。

呼び出しが成功すると、plaintext フィールドにはプレーンテキストの (復号された) データが含まれます。messageHeader フィールドには、データの復号に使用した暗号化コンテキストなどの復号プロセスに関するメタデータが含まれます。

JavaScript Browser

```
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

JavaScript Node.js

```
const { plaintext, messageHeader } = await decrypt(keyring, result)
```

ステップ 4: 暗号化コンテキストを確認します。

データの復号に使用した暗号化コンテキストは、decrypt 関数から返されるメッセージヘッダー (messageHeader) に含まれます。アプリケーションでプレーンテキストのデータを返す前に、暗号化時に指定した暗号化コンテキストが復号時に使用した暗号化コンテキストに含まれていることを確認します。一致しない場合は、データが改ざんされたか、復号する暗号化テキストを間違ったことを示している可能性があります。

暗号化コンテキストを確認する際は、完全に一致している必要ありません。署名付きの暗号化アルゴリズムを使用する場合、暗号化マテリアルマネージャー (CMM) は、メッセージを暗号化する前にパブリック署名キーを暗号化コンテキストに追加します。ただし、送信したすべての暗号化コンテキストのペアが返された暗号化コンテキストに含まれている必要があります。

まず、メッセージヘッダーから暗号化コンテキストを取得します。次に、元の暗号化コンテキスト (context) のキーと値の各ペアが、返された暗号化コンテキスト (encryptionContext) のキーと値のペアと一致することを確認します。

JavaScript Browser

```
const { encryptionContext } = messageHeader
```

```
Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context
 does not match expected values')
})
```

JavaScript Node.js

```
const { encryptionContext } = messageHeader

Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context
 does not match expected values')
})
```

暗号化コンテキストを確認して問題がなければ、プレーンテキストのデータを返すことができます。

# AWS Encryption SDK for Python

このトピックでは、 AWS Encryption SDK for Pythonをインストールして使用する方法について説明します。を使用したプログラミングの詳細については AWS Encryption SDK for Python、GitHub の aws-encryption-sdk-python リポジトリを参照してください。API のドキュメントについては、Read the Docs を参照してください。

トピック

- 前提条件
- インストール
- AWS Encryption SDK for Python サンプルコード

## 前提条件

をインストールする前に AWS Encryption SDK for Python、次の前提条件を満たし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

AWS Encryption SDK for Python バージョン 3.2.0 以降では、Python 3.8 以降が必要です。

> **Note**
>
> AWS 暗号化マテリアルプロバイダーライブラリ (MPL) は、バージョン 4.x で AWS Encryption SDK for Python 導入された のオプション依存関係です。MPL をインストールする場合は、Python 3.11 以降を使用する必要があります。

以前のバージョンの は Python 2.7 および Python 3.4 以降 AWS Encryption SDK をサポートしていますが、最新バージョンの を使用することをお勧めします AWS Encryption SDK。

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。

Python 用 pip インストールツール

`pip` は、Python 3.6 以降のバージョンには含まれていますが、アップグレードすることをお勧めします。`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」の「インストール」を参照してください。

## インストール

AWS Encryption SDK for Pythonの最新バージョンをインストールします。

> **Note**
>
> 3.0.0 より AWS Encryption SDK for Python 前の のすべてのバージョンはend-of-support フェーズにあります。
> バージョン 2.0.x 以降から AWS Encryption SDK の最新バージョンにコードやデータを変更せずに安全に更新できます。ただし、バージョン 2.0.x で導入された新しいセキュリティ機能には下位互換性がありません。1.7.x より前のバージョンから 2.0.x 以降のバージョンに更新するには、まず AWS Encryption SDKの最新の 1.x バージョンに更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。

次の例に示すように AWS Encryption SDK for Python、 `pip`を使用して 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 "aws-encryption-sdk[MPL]"
```

`[MPL]` サフィックスは AWS Cryptographic Material Providers Library (MPL) をインストールします。MPL には、データを暗号化および復号するためのコンストラクトが含まれています。MPL は、バージョン 4.x で AWS Encryption SDK for Python 導入された のオプション依存関係です。MPL のインストールを強くお勧めします。ただし、MPL を使用しない場合は、`[MPL]`サフィックスを省略できます。

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。

では、すべてのプラットフォームで暗号化ライブラリ (pyca/cryptography) AWS Encryption SDK for Python 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` が自動的に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、`cryptography` 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。

バージョン 1.10.0 および 2.5.0 は、2.5.0 と 3.3.2 の間の暗号化の依存関係 AWS Encryption SDK for Python を固定します。の他のバージョンでは、最新バージョンの暗号化 AWS Encryption SDK for Python がインストールされます。3.3.2 以降の暗号化のバージョンが必要な場合は、 AWS Encryption SDK for Pythonの最新のメジャーバージョンを使用することを推奨します。

の最新バージョンについては AWS Encryption SDK for Python、GitHub の aws-encryption-sdk-python リポジトリを参照してください。

をインストールしたら AWS Encryption SDK for Python、このガイドの Python サンプルコードを確認して開始します。

## AWS Encryption SDK for Python サンプルコード

次の例は、 を使用してデータを AWS Encryption SDK for Python 暗号化および復号する方法を示しています。

このセクションの例では、オプションの暗号化マテリアルプロバイダーライブラリの依存関係 () AWS Encryption SDK for Python で のバージョン 4.x を使用する方法を示しますaws-cryptographic-material-providers。以前のバージョンを使用する例、またはマテリアルプ

ロバイダーライブラリ (MPL) を使用しないインストールを表示するには、GitHub の aws-encryption-sdk-python リポジトリのリリースリストでリリースを見つけます。

MPL AWS Encryption SDK for Python で のバージョン 4.x を使用する場合、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。 AWS Encryption SDK には、以前のバージョンで使用していたマスターキープロバイダーと互換性のあるキーリングが用意されています。詳細については、「the section called “キーリングの互換性”」を参照してください。マスターキープロバイダーからキーリングへの移行の例については、GitHub の `aws-encryption-sdk-python`リポジトリの「移行例」を参照してください。

トピック

- 文字列の暗号化と復号
- バイトストリームの暗号化と復号

## 文字列の暗号化と復号

次の例は、 を使用して文字列 AWS Encryption SDK を暗号化および復号する方法を示しています。この例では、対称暗号化 KMS キーを持つ キーAWS KMS リングを使用します。

この例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。詳細については、「the section called “コミットメントポリシーの設定”」を参照してください。

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example sets up the KMS Keyring

The AWS KMS keyring uses symmetric encryption KMS keys to generate, encrypt and
decrypt data keys. This example creates a KMS Keyring and then encrypts a custom input
 EXAMPLE_DATA
with an encryption context. This example also includes some sanity checks for
 demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

AWS KMS keyrings can be used independently or in a multi-keyring with other keyrings
```

```
of the same or a different type.

"""

import boto3
from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import CreateAwsKmsKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy

EXAMPLE_DATA: bytes = b"Hello World"


def encrypt_and_decrypt_with_keyring(
    kms_key_id: str
):
    """Demonstrate an encrypt/decrypt cycle using an AWS KMS keyring.

    Usage: encrypt_and_decrypt_with_keyring(kms_key_id)
    :param kms_key_id: KMS Key identifier for the KMS key you want to use for
 encryption and
    decryption of your data keys.
    :type kms_key_id: string

    """
    # 1. Instantiate the encryption SDK client.
    # This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
    # which enforces that this client only encrypts using committing algorithm suites
 and enforces
    # that this client will only decrypt encrypted messages that were created with a
 committing
    # algorithm suite.
    # This is the default commitment policy if you were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. Create a boto3 client for KMS.
```

```
    kms_client = boto3.client('kms', region_name="us-west-2")

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "encryption": "context",
        "is not": "secret",
        "but adds": "useful metadata",
        "that can help you": "be confident that",
        "the data you are handling": "is what you think it is",
    }

    # 4. Create your keyring
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
        kms_key_id=kms_key_id,
        kms_client=kms_client
    )

    kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
        input=keyring_input
    )

    # 5. Encrypt the data with the encryptionContext.
    ciphertext, _ = client.encrypt(
        source=EXAMPLE_DATA,
        keyring=kms_keyring,
        encryption_context=encryption_context
    )

    # 6. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert ciphertext != EXAMPLE_DATA, \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 7. Decrypt your encrypted data using the same keyring you used on encrypt.
    plaintext_bytes, _ = client.decrypt(
        source=ciphertext,
        keyring=kms_keyring,
        # Provide the encryption context that was supplied to the encrypt method
```

```
        encryption_context=encryption_context,
    )

    # 8. Demonstrate that the decrypted plaintext is identical to the original
 plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
 code.)
    assert plaintext_bytes == EXAMPLE_DATA, \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
 decryption"
```

## バイトストリームの暗号化と復号

次の例は、 を使用してバイトストリーム AWS Encryption SDK を暗号化および復号する方法を示しています。この例では、Raw AES キーリングを使用します。

この例では、デフォルトのコミットメントポリシー を使用して AWS Encryption SDK クライアントをインスタンス化しますREQUIRE_ENCRYPT_REQUIRE_DECRYPT。詳細については、「the section called “コミットメントポリシーの設定”」を参照してください。

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example demonstrates file streaming for encryption and decryption.

File streaming is useful when the plaintext or ciphertext file/data is too large to
 load into
memory. Therefore, the AWS Encryption SDK allows users to stream the data, instead of
 loading it
all at once in memory. In this example, we demonstrate file streaming for encryption
 and decryption
using a Raw AES keyring. However, you can use any keyring with streaming.

This example creates a Raw AES Keyring and then encrypts an input stream from the file
`plaintext_filename` with an encryption context to an output (encrypted) file
 `ciphertext_filename`.
It then decrypts the ciphertext from `ciphertext_filename` to a new file
 `decrypted_filename`.
This example also includes some sanity checks for demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
```

```
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

See raw_aes_keyring_example.py in the same directory for another raw AES keyring
 example
in the AWS Encryption SDK for Python.
"""
import filecmp
import secrets

from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import AesWrappingAlg,
 CreateRawAesKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_and_decrypt_with_keyring(
    plaintext_filename: str,
    ciphertext_filename: str,
    decrypted_filename: str
):
    """Demonstrate a streaming encrypt/decrypt cycle.

    Usage: encrypt_and_decrypt_with_keyring(plaintext_filename
                                            ciphertext_filename
                                            decrypted_filename)
    :param plaintext_filename: filename of the plaintext data
    :type plaintext_filename: string
    :param ciphertext_filename: filename of the ciphertext data
    :type ciphertext_filename: string
    :param decrypted_filename: filename of the decrypted data
    :type decrypted_filename: string
    """
    # 1. Instantiate the encryption SDK client.
    # This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
    # which enforces that this client only encrypts using committing algorithm suites
 and enforces
```

```
    # that this client will only decrypt encrypted messages that were created with a
committing
    # algorithm suite.
    # This is the default commitment policy if you were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. The key namespace and key name are defined by you.
    # and are used by the Raw AES keyring to determine
    # whether it should attempt to decrypt an encrypted data key.
    key_name_space = "Some managed raw keys"
    key_name = "My 256-bit AES wrapping key"

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "encryption": "context",
        "is not": "secret",
        "but adds": "useful metadata",
        "that can help you": "be confident that",
        "the data you are handling": "is what you think it is",
    }

    # 4. Generate a 256-bit AES key to use with your keyring.
    # In practice, you should get this key from a secure key management system such as
an HSM.

    # Here, the input to secrets.token_bytes() = 32 bytes = 256 bits
    static_key = secrets.token_bytes(32)

    # 5. Create a Raw AES keyring
    # We choose to use a raw AES keyring, but any keyring can be used with streaming.
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
        key_namespace=key_name_space,
        key_name=key_name,
        wrapping_key=static_key,
        wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
    )
```

```
    raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
        input=keyring_input
    )

    # 6. Encrypt the data stream with the encryptionContext
    with open(plaintext_filename, 'rb') as pt_file, open(ciphertext_filename, 'wb') as
ct_file:
        with client.stream(
            mode='e',
            source=pt_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as encryptor:
            for chunk in encryptor:
                ct_file.write(chunk)

    # 7. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert not filecmp.cmp(plaintext_filename, ciphertext_filename), \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 8. Decrypt your encrypted data stream using the same keyring you used on
encrypt.
    with open(ciphertext_filename, 'rb') as ct_file, open(decrypted_filename, 'wb') as
pt_file:
        with client.stream(
            mode='d',
            source=ct_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as decryptor:
            for chunk in decryptor:
                pt_file.write(chunk)

    # 10. Demonstrate that the decrypted plaintext is identical to the original
plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert filecmp.cmp(plaintext_filename, decrypted_filename), \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
decryption"
```

# AWS Encryption SDK Rust 用

このトピックでは、 AWS Encryption SDK for Rust 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 Encryption SDK for Rust を使用したプログラミングの詳細については、GitHub の aws-encryption-sdk リポジトリの Rust ディレクトリを参照してください。

AWS Encryption SDK for Rust は、以下の点 AWS Encryption SDK で の他のプログラミング言語実装とは異なります。

- データキーキャッシュはサポートされていません。ただし、 AWS Encryption SDK for Rust は、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングをサポートしています。
- ストリーミングデータはサポートしていません

AWS Encryption SDK for Rust には、 の他の言語実装のバージョン 2.0.x 以降で導入されたすべてのセキュリティ機能が含まれています AWS Encryption SDK。ただし、 AWS Encryption SDK for Rust を使用して、 の 2.0.x より前のバージョンの別の言語実装で暗号化されたデータを復号する場合は AWS Encryption SDK、コミットメントポリシーを調整する必要がある場合があります。詳細については、「コミットメントポリシーの設定方法」を参照してください。

AWS Encryption SDK for Rust は、Dafny AWS Encryption SDK の 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正式な検証言語です。その結果、機能の正確性を保証するフレームワークに、 AWS Encryption SDK の機能を実装するライブラリができあがりました。

詳細はこちら

- 代替アルゴリズムスイートの指定 AWS Encryption SDK、暗号化されたデータキーの制限、 AWS KMS マルチリージョンキーの使用など、 でオプションを設定する方法の例については、「」を参照してくださいの設定 AWS Encryption SDK。
- AWS Encryption SDK for Rust を設定して使用する方法の例については、GitHub の aws-encryption-sdk リポジトリにある Rust の例を参照してください。

トピック


- 前提条件
- インストール
- AWS Encryption SDK Rust のサンプルコード用

## 前提条件

AWS Encryption SDK for Rust 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。

Rust と Cargo をインストールする

rustup を使用して Rust の現在の安定版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、「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

AWS Encryption SDK for Rust は`aws-esdk`、Crates.io で木箱として利用できます。 AWS Encryption SDK for Rust のインストールと構築の詳細については、GitHub の aws-encryption-sdk リポジトリの README.md を参照してください。

AWS Encryption SDK for Rust は、次の方法でインストールできます。

手動

AWS Encryption SDK for Rust をインストールするには、aws-encryption-sdk 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。

Crates.io の使用

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-esdk
```

または、Cargo.toml に次の行を追加します。

```
aws-esdk = "<version>"
```

## AWS Encryption SDK Rust のサンプルコード用

次の例は、 AWS Encryption SDK for Rust でプログラミングするときに使用する基本的なコーディングパターンを示しています。具体的には、 AWS Encryption SDK とマテリアルプロバイダーライブ

ラリをインスタンス化します。次に、各メソッドを呼び出す前に、メソッドの入力を定義する オブジェクトをインスタンス化します。

代替アルゴリズムスイートの指定や暗号化されたデータキーの制限など AWS Encryption SDK、 でオプションを設定する方法の例については、GitHub の aws-encryption-sdk リポジトリの Rust の例を参照してください。

## AWS Encryption SDK for Rust でのデータの暗号化と復号

この例は、データを暗号化および復号するための基本的なパターンを示しています。1 つの AWS KMS ラッピングキーで保護されたデータキーを使用して小さなファイルを暗号化します。

ステップ 1: をインスタンス化します AWS Encryption SDK。

のメソッドを使用して AWS Encryption SDK 、データを暗号化および復号します。

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

ステップ 2: AWS KMS クライアントを作成する。

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

オプション: 暗号化コンテキストを作成します。

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

ステップ 3: マテリアルプロバイダーライブラリをインスタンス化します。

マテリアルプロバイダライブラリのメソッドを使用して、データを保護するキーを指定するキーリングを作成します。

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
```

```
let mpl = mpl_client::Client::from_conf(mpl_config)?;
```

ステップ 4: AWS KMS キーリングを作成します。

キーリングを作成するには、キーリング入力オブジェクトを使用してキーリングメソッドを呼び出します。この例では、 `create_aws_kms_keyring()`メソッドを使用し、1 つの KMS キーを指定します。

```
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;
```

ステップ 5: プレーンテキストを暗号化します。

```
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

let ciphertext = encryption_response
                    .ciphertext
                    .expect("Unable to unwrap ciphertext from encryption response");
```

ステップ 6: 暗号化に使用したのと同じキーリングを使用して、暗号化されたデータを復号します。

```
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
    .send()
    .await?;

let decrypted_plaintext = decryption_response
                            .plaintext
```

```
                        .expect("Unable to unwrap plaintext from decryption
response");
```

# AWS Encryption SDK コマンドラインインターフェイス

AWS Encryption SDK コマンドラインインターフェイス (AWS Encryption CLI) を使用すると、 AWS Encryption SDK を使用して、コマンドラインとスクリプトでデータをインタラクティブに暗号化および復号化できます。暗号化やプログラミングの専門知識は必要ありません。

> **Note**
>
> 4.0.0 より前のバージョンの AWS Encryption end-of-supportフェーズにあります。
> バージョン 2.1.x 以降から、コードやデータを変更せずに最新バージョンの AWS Encryption CLI に安全に更新できます。ただし、バージョン 2.1.x で導入された新しいセキュリティ機能には下位互換性がありません。バージョン 1.7.x 以前から更新するには、まず Encryption CLI の最新バージョンの AWS 1.x に更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

のすべての実装と同様に AWS Encryption SDK、 Encryption AWS CLI は高度なデータ保護機能を提供します。これにはエンベロープ暗号化、追加の認証データ (AAD)、キー取得、キーコミットメント、署名で使用する 256 ビット AES-GCM などのセキュアで認証済みの対称キーアルゴリズムスイートなどが含まれます。

AWS Encryption CLI は 上に構築AWS Encryption SDK for Pythonされており、Linux、macOS、および Windows でサポートされています。コマンドやスクリプトを実行して、Linux や macOS の任意のシェル、Windows のコマンドプロンプトウィンドウ (cmd.exe)、任意のシステムの PowerShell コンソールでデータを暗号化および復号することができます。

Encryption CLI AWS AWS Encryption SDKを含む の言語固有の実装はすべて相互運用可能です。たとえば、 でデータを暗号化AWS Encryption SDK for Javaし、 Encryption CLI AWS で復号できます。

このトピックでは、 Encryption AWS CLI について紹介し、インストールして使用する方法と、使用開始に役立つくつかの例を示します。クイックスタートについては、 AWS セキュリティブログの「暗号化 CLI AWS を使用してデータを暗号化および復号する方法」を参照してください。詳細については、「 Docs を読み、GitHub AWS の aws-encryption-sdk-cli リポジトリで Encryption CLI の開発に参加してください」を参照してください。

パフォーマンス

Encryption AWS CLI は 上に構築されています AWS Encryption SDK for Python。CLI を実行するたびに、Python ランタイムの新しいインスタンスが起動されます。パフォーマンスを向上させるには、個別のコマンドを何回も使用するのではなく、できるだけ 1 つのコマンドを使用します。例えば、ファイルごとに個別のコマンドを実行するのではなく、ディレクトリ内のファイルを再帰的に処理する 1 つのコマンドを実行します。

トピック

- AWS Encryption SDK コマンドラインインターフェイスのインストール
- Encryption AWS CLI の使用方法
- Encryption AWS CLI の例
- AWS Encryption SDK CLI 構文とパラメータリファレンス
- Encryption AWS CLI のバージョン

## AWS Encryption SDK コマンドラインインターフェイスのインストール

このトピックでは、 Encryption AWS CLI をインストールする方法について説明します。詳細については、GitHub の aws-encryption-sdk-cli リポジトリおよび「ドキュメントを読む」を参照してください。

トピック

- 前提条件のインストール
- Encryption AWS CLI のインストールと更新

### 前提条件のインストール

Encryption AWS CLI は 上に構築されています AWS Encryption SDK for Python。 AWS Encryption CLI をインストールするには、Python と pipPython パッケージ管理ツールである が必要です。Python と pip は、サポートされているすべてのプラットフォームで使用できます。

Encryption CLI AWS をインストールする前に、次の前提条件をインストールします。

Python

Encryption CLI バージョン AWS 4.2.0 以降では、Python 3.8 以降が必要です。

以前のバージョンの AWS Encryption CLI は Python 2.7 および 3.4 以降をサポートしていますが、最新バージョンの AWS Encryption CLI を使用することをお勧めします。

Python は、ほとんどの Linux と macOS のインストールに含まれていますが、Python 3.6 以降にアップグレードする必要があります。最新バージョンの Python の使用をお勧めします。Windows では Python をインストールする必要があります。デフォルトではインストールされていません。Python をダウンロードしてインストールするには、「Python のダウンロード」を参照してください。

Python がインストールされているかどうかを確認するには、コマンドラインで次のように入力します。

```
python
```

Python のバージョンを確認するには `-V` (大文字 V) パラメータを使用します。

```
python -V
```

Windows では、Python をインストールしたら、`Python.exe` ファイルのパスを Path 環境変数の値に追加します。

デフォルトでは、`$home` サブディレクトリの Python がインストールされているすべてのユーザーまたはユーザープロファイルディレクトリ内 (`%userprofile%` または `AppData\Local\Programs\Python`) にあります。システム内の `Python.exe` ファイルの場所を確認するには、次のいずれかのレジストリキーを確認します。PowerShell を使用してレジストリを検索できます。

```
PS C:\> dir HKLM:\Software\Python\PythonCore\version\InstallPath
# -or-
PS C:\> dir HKCU:\Software\Python\PythonCore\version\InstallPath
```

pip

pip は Python パッケージマネージャーです。Encryption AWS CLI とその依存関係をインストールするには、8.1 pip 以降が必要です。pip のインストールまたはアップグレードのヘルプについては、pip ドキュメントの「インストール」を参照してください。

Linux インストールでは、 8.1 よりpip前のバージョンの では、Encryption CLI AWS に必要な暗号化ライブラリを構築できません。pip バージョンを更新しないことを選択した場合は、ビルドツールを個別にインストールできます。詳細については、「Building cryptography on Linux」を参照してください。

AWS Command Line Interface

AWS Command Line Interface（ AWS CLI) は、 Encryption CLI で (AWS KMS) AWS KMS keys で AWS Key Management Service AWS を使用している場合にのみ必要です。別のマスターキープロバイダーを使用している場合、 AWS CLI は必要ありません。

Encryption CLI AWS KMS keys で AWS を使用するには、 をインストールして設定する必要があります AWS CLI。この設定により、認証に使用する認証情報が AWS Encryption CLI AWS KMS で使用できるようになります。

## Encryption AWS CLI のインストールと更新

Encryption AWS CLI の最新バージョンをインストールします。pip を使用して Encryption AWS CLI をインストールすると、、Python 暗号化ライブラリAWS Encryption SDK for Python、 など、 CLI に必要なライブラリが自動的にインストールされますAWS SDK for Python (Boto3)。

> **Note**
>
> 4.0.0 より前のバージョンの AWS Encryption end-of-supportフェーズにあります。
> バージョン 2.1.x 以降から、コードやデータを変更せずに最新バージョンの AWS Encryption CLI に安全に更新できます。ただし、バージョン 2.1.x で導入された新しいセキュリティ機能には下位互換性がありません。バージョン 1.7.x 以前から更新するには、まず Encryption CLI の最新バージョンの AWS 1.x に更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細について

は、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

Encryption AWS CLI の最新バージョンをインストールするには

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

Encryption AWS CLI の最新バージョンにアップグレードするには

```
pip install --upgrade aws-encryption-sdk-cli
```

Encryption CLI と AWS のバージョン番号を確認するには AWS Encryption SDK

```
aws-encryption-cli --version
```

出力には、両方のライブラリのバージョン番号が表示されます。

```
aws-encryption-sdk-cli/2.1.0 aws-encryption-sdk/2.0.0
```

Encryption AWS CLI の最新バージョンにアップグレードするには

```
pip install --upgrade aws-encryption-sdk-cli
```

AWS Encryption CLI をインストールすると AWS SDK for Python (Boto3)、最新バージョンの もインストールされます。Boto3 がインストールされている場合、インストーラは Boto3 のバージョンを確認し、必要に応じて更新します。

インストールされている Boto3 のバージョンを確認するには

```
pip show boto3
```

Boto3 の最新バージョンに更新するには

```
pip install --upgrade boto3
```

現在開発中の Encryption CLI AWS のバージョンをインストールするには、GitHub の aws-encryption-sdk-cli リポジトリを参照してください。

pip を使用した Python パッケージのインストールおよびアップグレードの詳細については、pip のドキュメントを参照してください。

## Encryption AWS CLI の使用方法

このトピックでは、 Encryption AWS CLI でパラメータを使用する方法について説明します。例については「Encryption AWS CLI の例」を参照してください。完全なドキュメントについては、「ドキュメントを読む」を参照してください。これらの例に示す構文は、Encryption CLI AWS バージョン 2.1.x 以降用です。

> **Note**
>
> 4.0.0 より前のバージョンの AWS Encryption end-of-supportフェーズにあります。
> バージョン 2.1.x 以降から、コードやデータを変更せずに最新バージョンの AWS Encryption CLI に安全に更新できます。ただし、バージョン 2.1.x で導入された新しいセキュリティ機能には下位互換性がありません。バージョン 1.7.x 以前から更新するには、まず Encryption CLI の最新バージョンの AWS 1.x に更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

暗号化されたデータキーを制限するセキュリティ機能の使用方法の例については、「暗号化されたデータキーの制限」を参照してください。

AWS KMS マルチリージョンキーの使用方法を示す例については、「」を参照してくださいマルチリージョンの使用 AWS KMS keys。

トピック

- データを暗号化および復号する方法
- ラッピングキーの指定方法
- 入力を指定する方法
- 出力の場所を指定する方法
- 暗号化コンテキストを使用する方法

- コミットメントポリシーの指定方法
- 設定ファイルにパラメータを保存する方法

## データを暗号化および復号する方法

AWS Encryption CLI は の機能を使用して AWS Encryption SDK 、データの安全な暗号化と復号を容易にします。

> **Note**
>
> `--master-keys` パラメータは AWS Encryption CLI のバージョン 1.8.x で非推奨となり、バージョン 2.1.x で削除されます。代わりに、`--wrapping-keys` パラメータを使用します。バージョン 2.1.x 以降は、`--wrapping-keys` パラメータが暗号化および復号化時に必要となります。詳細については、「AWS Encryption SDK CLI 構文とパラメータリファレンス」を参照してください。

- AWS Encryption CLI でデータを暗号化するときは、プレーンテキストデータと in AWS Key Management Service () などのラッピングキー (またはマスターキー) AWS KMS key を指定しますAWS KMS。カスタムのマスターキープロバイダーを使用する場合は、プロバイダーを指定する必要もあります。また、暗号化されたメッセージおよび暗号化オペレーションに関するメタデータの出力場所を指定します。暗号化コンテキストはオプションですが、推奨されています。

  バージョン 1.8.x では、`--wrapping-keys` パラメータを使用するときに `--commitment-policy` パラメータが必要です。これがない場合は無効です。バージョン 2.1.x 以降では、`--commitment-policy` パラメータはオプションですが推奨されます。

```
aws-encryption-cli --encrypt --input myPlaintextData \
                   --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                   --output myEncryptedMessage \
                   --metadata-output ~/metadata \
                   --encryption-context purpose=test \
                   --commitment-policy require-encrypt-require-decrypt
```

  AWS Encryption CLI は、一意のデータキーでデータを暗号化します。その後、指定したラッピングキーでデータキーを暗号化します。暗号化されたメッセージとオペレーションに関するメタデータが返されます。暗号化されたメッセージには、暗号化されたデータ (暗号化テキスト) およ

びデータキーの暗号化されたコピーが含まれます。データキーの保存、管理、または紛失について心配する必要はありません。

- データを復号する際、暗号化されたメッセージ、オプションの暗号化コンテキスト、プレーンテキスト出力およびメタデータの場所を渡します。また、 AWS Encryption CLI がメッセージを復号するために使用できるラッピングキーを指定したり、メッセージを暗号化したラッピングキーを使用できることを AWS Encryption CLI に伝えたりします。

  バージョン 1.8.x 以降では、復号時の `--wrapping-keys` パラメータはオプションですが推奨されます。バージョン 2.1.x 以降は、`--wrapping-keys` パラメータが暗号化および復号化時に必要となります。

  復号するときには、`--wrapping-keys` パラメータの key 属性を使用して、データを復号化するラッピングキーを指定します。復号時に AWS KMS ラッピングキーを指定することはオプションですが、使用する予定のないキーを使用できないようにするベストプラクティスです。カスタムのマスターキープロバイダーを使用する場合は、プロバイダーおよびラッピングキーを指定する必要があります。

  key 属性を使用しない場合は、 `--wrapping-keys`パラメータの discovery 属性を に設定する必要があります。これにより`true`、 Encryption AWS CLI はメッセージを暗号化したラッピングキーを使用して復号化できます。

  ベストプラクティスとして、`--max-encrypted-data-keys` パラメータを使用して、暗号化されたデータキーの数が多すぎる不正な形式のメッセージの復号化を回避してください。暗号化されたデータキーの予想数 (暗号化で使用されるラッピングキーごとに 1 つ)、または適切な最大値 (5 など) を指定します。詳細については、「暗号化されたデータキーの制限」を参照してください。

  `--buffer` パラメータでは、デジタル署名が存在する場合の検証も含めて、すべての入力が処理された後にのみプレーンテキストが返されます。

  `--decrypt-unsigned` パラメータでは、暗号化テキストを復号し、復号化前にメッセージが署名なしであることを確認します。このパラメータは、`--algorithm` パラメータを使用し、データを暗号化するためのデジタル署名なしのアルゴリズムスイートを選択した場合に使用します。暗号化テキストが署名されている場合、復号化は失敗します。

  `--decrypt` または `--decrypt-unsigned` を復号化に使用できますが、両方とも使用することはできません。

```
aws-encryption-cli --decrypt --input myEncryptedMessage \
                   --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                   --output myPlaintextData \
                   --metadata-output ~/metadata \
                   --max-encrypted-data-keys 1 \
                   --buffer \
                   --encryption-context purpose=test \
                   --commitment-policy require-encrypt-require-decrypt
```

AWS Encryption CLI はラッピングキーを使用して、暗号化されたメッセージのデータキーを復号します。次に、データキーを使ってデータを復号します。プレーンテキストのデータとオペレーションに関するメタデータが返されます。

## ラッピングキーの指定方法

Encryption CLI でデータを暗号化する場合は、少なくとも AWS 1 つのラッピングキー (またはマスターキー) を指定する必要があります。 AWS KMS keys in AWS Key Management Service （ AWS KMS ）、カスタムマスターキープロバイダーからのラッピングキー、またはその両方を使用できます。カスタムのマスターキープロバイダは、互換性がある Python マスターキープロバイダのいずれかです。

バージョン 1.8.x 以降でラッピングキーを指定するには、`--wrapping-keys` パラメータ (`-w`) を使用します。このパラメータの値は、`attribute=value` 形式を使用する属性の集合です。使用する属性は、マスターキープロバイダやコマンドによって異なります。

- AWS KMS。暗号化コマンドでは、key 属性を使用して `--wrapping-keys` パラメータを指定する必要があります。バージョン 2.1.x 以降は、`--wrapping-keys` パラメータが復号化コマンドにも必要となります。復号化するとき、`--wrapping-keys` パラメータでは、key 属性を指定するか、discovery 属性を `true` にする必要があります (両方ではない)。その他の属性はオプションです。
- カスタムマスターキープロバイダー。どのコマンドでも `--wrapping-keys` パラメータを指定する必要があります。パラメータ値に key および provider 属性を含める必要があります。

同じコマンドで複数の `--wrapping-keys` パラメータおよび複数の key 属性を含めることができます。

### ラッピングキーパラメータの属性

`--wrapping-keys` パラメータの値は、次の属性と値で構成されます。`--wrapping-keys` パラメータ (または `--master-keys` パラメータ) は、すべての暗号化コマンドで必要です。バージョン 2.1.x 以降は、`--wrapping-keys` パラメータが復号化時にも必要となります。

属性名や値にスペースや特殊文字が含まれている場合、名前と値の両方を引用符で囲みます。例えば、`--wrapping-keys key=12345 "provider=my cool provider"` と指定します。

Key: ラッピングキーを指定します。

key 属性を使用してラッピングキーを識別します。暗号化時に、この値は、マスターキープロバイダーが認識する任意のキー識別子を使用できます。

```
--wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab
```

暗号化コマンドでは、少なくとも 1 つの key 属性と値が含まれている必要があります。複数のラッピングキーでデータキーを暗号化するには、複数の key 属性を使用します。

```
aws-encryption-cli --encrypt --wrapping-keys
 key=1234abcd-12ab-34cd-56ef-1234567890ab key=1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d
```

が使用する暗号化コマンドでは AWS KMS keys、キーの値は、キー ID、キー ARN、エイリアス名、またはエイリアス ARN です。たとえば、この暗号化コマンドでは、key 属性の値のエイリアス ARN を使用しています。のキー識別子の詳細については AWS KMS key、「 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-encryption-cli --encrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:alias/ExampleAlias
```

カスタムのマスターキープロバイダーを使用する復号コマンドでは、key および provider 属性が必須です。

```
\\ Custom master key provider
aws-encryption-cli --decrypt --wrapping-keys provider='myProvider' key='100101'
```

を使用する復号コマンドでは AWS KMS、key 属性を使用して復号 AWS KMS keys に使用するを指定するか、discovery 属性を の値で指定してtrue、Encryption AWS CLI AWS KMS key がメッセージの暗号化に使用された を使用できるようにします。を指定する場合 AWS KMS key、メッセージの暗号化に使用されるラッピングキーの 1 つである必要があります。

ラッピングキーの指定は、AWS Encryption SDK のベストプラクティスです。これにより、使用する AWS KMS key 予定の が確実に使用されます。

復号コマンドでは、key 属性の値はキー ARN にする必要があります。

```
\\ AWS KMS key
aws-encryption-cli --decrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

検出: 復号 AWS KMS key 時に任意の を使用する

復号時に AWS KMS keys 使用する を制限する必要がない場合は、discovery 属性を の値で使用できますtrue。の値trueを指定すると、 Encryption AWS CLI はメッセージを暗号化 AWS KMS key した を使用して復号できます。discovery 属性は、指定しない場合、false です (デフォルト)。discovery 属性は、復号コマンドで、メッセージが暗号化された場合にのみ有効です AWS KMS keys。

discovery 属性を true にするのは、key 属性を使用して AWS KMS keysを指定することに代わる方法です。で暗号化されたメッセージを復号する場合 AWS KMS keys、各--wrapping-keysパラメータにはキー属性または の値を持つ discovery 属性が必要ですがtrue、両方を持つことはできません。

discovery が true の場合、discovery-partition 属性と discovery-account 属性を使用して、 AWS KMS keys 使用する を AWS アカウント 指定した 内のものに制限するのがベストプラクティスです。次の例では、検出属性により、 Encryption AWS CLI は指定された AWS KMS key で を使用できます AWS アカウント。

```
aws-encryption-cli --decrypt --wrapping-keys \
    discovery=true \
    discovery-partition=aws \
    discovery-account=111122223333 \
    discovery-account=444455556666
```

Provider: マスターキープロバイダーを指定します。

provider 属性は、マスターキープロバイダーを識別します。デフォルト値は aws-kms であり、AWS KMSを表します。別のマスターキープロバイダーを使用している場合、provider 属性が必要です。

```
--wrapping-keys key=12345 provider=my_custom_provider
```

カスタム (AWS KMSではない) マスターキープロバイダーを使用する方法の詳細については、AWS Encryption CLI レポジトリの README ファイルにあるトピック「高度な設定」を参照してください。

リージョン: を指定する AWS リージョン

region 属性を使用して、 AWS リージョン の を指定します AWS KMS key。この属性は、暗号化コマンドで、マスターキープロバイダが AWS KMSのときにのみ有効です。

```
--encrypt --wrapping-keys key=alias/primary-key region=us-east-2
```

AWS Encryption CLI コマンドは AWS リージョン 、ARN などのリージョンが含まれている場合、キー属性値で指定された を使用します。キー値が を指定すると AWS リージョン、リージョン属性は無視されます。

region 属性は、他のリージョンの仕様よりも優先されます。region 属性を使用しない場合、AWS Encryption CLI コマンドは、 AWS CLI 名前付きプロファイルで AWS リージョン 指定された 、存在する場合は 、またはデフォルトのプロファイルを使用します。

profile: 名前付きプロファイルを指定

profile 属性を使用して AWS CLI の名前付きプロファイルを指定します。名前付きプロファイルには、認証情報と AWS リージョンを含めることができます。この属性は、マスターキープロバイダが AWS KMSでない場合にのみ有効です。

```
--wrapping-keys key=alias/primary-key profile=admin-1
```

profile 属性を使用して、暗号化と復号コマンドで別の認証情報を指定できます。暗号化コマンドでは、 AWS Encryption CLI は、キー値にリージョンが含まれておらず、リージョン属性がない場合にのみ、名前付きプロファイル AWS リージョン の を使用します。復号コマンドでは、名前プロファイル AWS リージョン の は無視されます。

### 複数のラッピングキーを指定する方法

複数のラッピングキー (マスターキー) を各コマンドで指定できます。

複数のラッピングキーを指定した場合、最初のラッピングキーはデータの暗号化に使用するデータキーを生成および暗号化します。その他のラッピングキーは、同じデータキーを暗号化します。結果として得られる暗号化されたメッセージには、暗号化されたデータ (暗号化テキスト) と各ラッピングキーで 1 つずつ暗号化された一組のデータキーが含まれます。どのラッピングも、1 つの暗号化されたデータキーを復号してデータを復号することができます。

複数のラッピングキーを指定するには、2 つの方法があります。

- `--wrapping-keys` パラメータの値に複数の key 属性を含めます。

```
$key_oregon=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$key_ohio=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef

--wrapping-keys key=$key_oregon key=$key_ohio
```

- 同じコマンドに複数の `--wrapping-keys` パラメータを含めます。この構文は、指定する属性値をコマンドのラッピングキーに一括で適用しない場合に使用します。

```
--wrapping-keys region=us-east-2 key=alias/test_key \
--wrapping-keys region=us-west-1 key=alias/test_key
```

discovery 属性の値が `true`の場合、Encryption AWS CLI はメッセージを暗号化 AWS KMS key した を使用できます。複数の `--wrapping-keys` パラメータを同じコマンドで使用する場合、`--wrapping-keys` パラメータで `discovery=true` を使用すると、その他の `--wrapping-keys` パラメータで key 属性の制限が事実上無効になります。

例えば、次のコマンドでは、最初の`--wrapping-keys`パラメータの key 属性は、Encryption AWS CLI を指定された に制限します AWS KMS key。ただし、2 番目の`--wrapping-keys`パラメータの discovery 属性により、Encryption AWS CLI AWS KMS key は指定されたアカウントの を使用してメッセージを復号できます。

```
aws-encryption-cli --decrypt \
    --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab \
    --wrapping-keys discovery=true \
                    discovery-partition=aws \
                    discovery-account=111122223333 \
                    discovery-account=444455556666
```

## 入力を指定する方法

AWS Encryption CLI の暗号化オペレーションは、プレーンテキストデータを入力として受け取り、暗号化されたメッセージを返します。復号オペレーションは、暗号化されたメッセージを入力として受け取り、プレーンテキストのデータを返します。

すべての Encryption AWS CLI コマンドでは、入力の場所を Encryption CLI AWS に指示する `--input`パラメータ (`-i`) が必要です。

次のいずれかの方法で入力を指定できます。

- ファイルを使用します。

```
--input myData.txt
```

- ファイル名のパターンを使用します。

```
--input testdir/*.xml
```

- ディレクトリまたはディレクトリ名のパターンを使用します。入力がディレクトリの場合、`--recursive` パラメータ (`-r`、`-R`) が必要です。

```
--input testdir --recursive
```

- 入力をコマンド (stdin) へパイプします。`-` パラメータに `--input` の値を使用します。(`--input` パラメータは常に必須です。)

```
echo 'Hello World' | aws-encryption-cli --encrypt --input -
```

## 出力の場所を指定する方法

`--output` パラメータは AWS 、暗号化または復号オペレーションの結果を書き込む場所を Encryption CLI に指示します。これは、すべての Encryption AWS CLI コマンドで必要です。 AWS Encryption CLI は、オペレーションの入力ファイルごとに新しい出力ファイルを作成します。

出力ファイルが既に存在する場合、デフォルトでは Encryption AWS CLI は警告を出力し、ファイルを上書きします。上書きされないようにするには、上書きする前に確認のメッセージが表示する `--interactive` パラメータを使用するか、または、出力が上書きしようとすると入力をスキップする `--no-overwrite` を使用します。上書きの警告を表示しないようにするには、`--quiet` を使用します。Encryption CLI AWS からエラーと警告をキャプチャするには、`2>&1`リダイレクト演算子を使用して出力ストリームに書き込みます。

> **Note**
>
> 出力ファイルを上書きするコマンドは、出力ファイルを削除することで開始します。コマンドが失敗した場合は、出力ファイルが既に削除されている場合があります。

さまざまな方法で出力場所を変更できます。

- ファイル名を指定します。ファイルにパスを指定する場合、コマンドの実行前にパス内のすべてのディレクトリが存在している必要があります。

  ```
  --output myEncryptedData.txt
  ```

- ディレクトリを指定します。コマンドの実行前に出力ディレクトリが存在している必要があります。

  入力にサブディレクトリが含まれている場合、コマンドは指定されたディレクトリの下にサブディレクトリを再現します。

  ```
  --output Test
  ```

  出力場所がディレクトリ (ファイル名なし) AWS の場合、Encryption CLI は入力ファイル名とサフィックスに基づいて出力ファイル名を作成します。暗号化オペレーションは、入力ファイル名に `.encrypted` を追加します。復号オペレーションは `.decrypted` を追加します。サフィックスを変更するには、`--suffix` パラメータを使用します。

  たとえば、`file.txt` を暗号化する場合、暗号化コマンドは `file.txt.encrypted` を作成します。`file.txt.encrypted` を復号する場合、復号コマンドは `file.txt.encrypted.decrypted` を作成します。

- コマンドライン (stdout) に書き込みます。- パラメータに `--output` の値を入力します。`--output -` を使用して、出力を他のコマンドやプログラムにパイプできます。

  ```
  --output -
  ```

## 暗号化コンテキストを使用する方法

AWS Encryption CLI では、暗号化コマンドと復号コマンドで暗号化コンテキストを指定できます。これは必須ではありませんが、推奨される暗号化のベストプラクティスです。

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です。 AWS Encryption CLI では、暗号化コンテキストは `name=value` のペアの集合で構成されます。ペアの内容はどれでも使用できます。これには、権限やポリシーに必要とされるログ、またはデータ内の暗号化オペレーションを探すのに役立つファイルやデータに関する情報が含まれます。

暗号化コマンドの場合

暗号化コンポーネントによって追加された追加の暗号化コンテキストと共に、CMM によって追加されたペアは、暗号化されたデータに暗号化されてバインドされます。これは、コマンドが返す暗号化されたメッセージにも含まれています (プレーンテキスト)。を使用している場合 AWS KMS key、暗号化コンテキストは、 などの監査レコードやログにプレーンテキストで表示される場合があります AWS CloudTrail。

次の例は、`name=value` の 3 つのペアを持つ暗号化コンテキストを示しています。

```
--encryption-context purpose=test dept=IT class=confidential
```

復号コマンドの場合

復号コマンドにおいて、暗号化コンテキストは、暗号化された適切なメッセージを復号しているかどうか確認するのに役立ちます。

暗号化コンテキストが暗号化で使用されていないとしても、復号コマンドで暗号化コンテキストを指定する必要はありません。ただし、その場合、Encryption AWS CLI は、復号コマンドの暗号化コンテキストのすべての要素が、暗号化されたメッセージの暗号化コンテキストの要素と一致することを確認します。いずれかの要素が一致しない場合、復号コマンドは失敗します。

たとえば、次のコマンドは、暗号化コンテキストに `dept=IT` が含まれている場合にのみ暗号化メッセージを復号します。

```
aws-encryption-cli --decrypt --encryption-context dept=IT ...
```

暗号化コンテキストは、セキュリティ戦略の重要な部分です。ただし、暗号化コンテキストを選択する際、その値がシークレットではないことに注意してください。暗号化コンテキストに機密データを含めないでください。

暗号化コンテキストを指定するには

- 暗号化コマンドでは、--encryption-context パラメータを 1 つ以上の name=value ペアで使用します。各ペアを区切るためにスペースを使用します。

```
--encryption-context name=value [name=value] ...
```

- 復号コマンドでは、--encryption-context パラメータ値に name=value ペア、name 要素 (値なし)、または両方の組み合わせを含めることができます。

```
--encryption-context name[=value] [name] [name=value] ...
```

name ペアの value や name=value にスペースや特殊文字が含まれている場合、ペア全体を引用符で囲みます。

```
--encryption-context "department=software engineering" "AWS #####=us-west-2"
```

たとえば、この暗号化コマンドには、purpose=test と dept=23 という 2 つのペアを持つ暗号化コンテキストが含まれています。

```
aws-encryption-cli --encrypt --encryption-context purpose=test dept=23 ...
```

これらの復号コマンドは成功します。各コマンドの暗号化コンテキストは、元の暗号化コンテキストのサブセットです。

```
\\ Any one or both of the encryption context pairs
aws-encryption-cli --decrypt --encryption-context dept=23 ...

\\ Any one or both of the encryption context names
aws-encryption-cli --decrypt --encryption-context purpose ...

\\ Any combination of names and pairs
aws-encryption-cli --decrypt --encryption-context dept purpose=test ...
```

ただし、これらの復号コマンドは失敗します。暗号化されたメッセージの暗号化コンテキストには、指定された要素は含まれていません。

```
aws-encryption-cli --decrypt --encryption-context dept=Finance ...
```

```
aws-encryption-cli --decrypt --encryption-context scope ...
```

## コミットメントポリシーの指定方法

コマンドにコミットメントポリシーを設定するには、`--commitment-policy` パラメータを使用します。このパラメータはバージョン 1.8.x で導入されました。暗号化コマンドと復号コマンドで有効です。設定するコミットメントポリシーは、表示されるコマンドに対してのみ有効です。コマンドのコミットメントポリシーを設定しない場合、Encryption AWS CLI はデフォルト値を使用します。

例えば、次のパラメータ値ではコミットメントポリシーが `require-encrypt-allow-decrypt` に設定され、常にキーコミットメントで暗号化されますが、暗号化された暗号化テキストはキーコミットメントの有無にかかわらず復号化されます。

```
--commitment-policy require-encrypt-allow-decrypt
```

## 設定ファイルにパラメータを保存する方法

頻繁に使用する Encryption CLI AWS パラメータと値を設定ファイルに保存することで、時間を節約し、入力エラーを回避できます。

設定ファイルは、Encryption AWS CLI コマンドのパラメータと値を含むテキストファイルです。AWS Encryption CLI コマンドで設定ファイルを参照すると、リファレンスは設定ファイルのパラメータと値で置き換えられます。ファイルの内容をコマンドラインで入力した場合にも同じ効果が得られます。設定ファイルは任意の名前を使用でき、現在のユーザーがアクセス可能な任意のディレクトリに配置できます。

次の設定ファイル (`key.conf`) の例では、2 つの AWS KMS keys を異なるリージョンで指定しています。

```
--wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
--wrapping-keys key=arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
```

コマンドで設定ファイルを使用するには、ファイル名の先頭にアットマーク (@) を使用します。PowerShell コンソールでは、バックティック文字を使用してアットマーク (`@) をエスケープする必要があります。

このコマンド例では、暗号化コマンドで `key.conf` ファイルを使用します。

Bash

```
$ aws-encryption-cli -e @key.conf -i hello.txt -o testdir
```

PowerShell

```
PS C:\> aws-encryption-cli -e `@key.conf -i .\Hello.txt -o .\TestDir
```

### 設定ファイルのルール

設定ファイルを使用するためのルールは次のとおりです。

- 各設定ファイルで複数のパラメータを含めることができ、任意の順序で表示できます。各パラメータとその値 (あれば) を個別の行で表示します。
- # を使用して行の全体または一部にコメントを追加します。
- 他の設定ファイルへの参照を含めることができます。PowerShell コンソールでも、バックティック文字を使用して @ 文字を エスケープすることはしないでください。
- 設定ファイルで引用符を使用する場合、引用されたテキストが複数の行にまたがることはできません。

たとえば、これはサンプル `encrypt.conf` ファイルの内容です。

```
# Archive Files
--encrypt
--output /archive/logs
--recursive
--interactive
--encryption-context class=unclassified dept=IT
--suffix  # No suffix
--metadata-output ~/metadata
@caching.conf  # Use limited caching
```

コマンドには複数の設定ファイルを含めることもできます。このコマンド例では、`encrypt.conf` との両方の `master-keys.conf` 設定ファイルが使用されます。

Bash

```
$  aws-encryption-cli -i /usr/logs @encrypt.conf @master-keys.conf
```

PowerShell

```
PS C:\> aws-encryption-cli -i $home\Test\*.log `@encrypt.conf `@master-keys.conf
```

Next: AWS Encryption CLI の例を試します。

## Encryption AWS CLI の例

次の例を使用して、希望するプラットフォームで AWS Encryption CLI を試します。マスターキーおよびその他のパラメータのヘルプについては、「Encryption AWS CLI の使用方法」を参照してください。クイックリファレンスについては、「AWS Encryption SDK CLI 構文とパラメータリファレンス」を参照してください。

> **Note**
>
> 次の例では、Encryption CLI AWS バージョン 2.1.x の構文を使用します。
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

暗号化されたデータキーを制限するセキュリティ機能の使用方法の例については、「暗号化されたデータキーの制限」を参照してください。

AWS KMS マルチリージョンキーの使用方法を示す例については、「」を参照してくださいマルチリージョンの使用 AWS KMS keys。

トピック

- ファイルの暗号化
- ファイルの復号
- ディレクトリ内のすべてのファイルの暗号化
- ディレクトリ内のすべてのファイルの復号
- コマンドラインでの暗号化と復号
- 複数のマスターキーの使用

- スクリプトでの暗号化と復号
- データキーキャッシュの使用

## ファイルの暗号化

この例では、Encryption AWS CLI を使用して、「Hello World」文字列を含む hello.txt ファイルの内容が暗号化されます。

ファイルで暗号化コマンドを実行すると、 AWS Encryption CLI はファイルの内容を取得し、一意のデータキーを生成して、データキーでファイル内容を暗号化してから、暗号化されたメッセージを新しいファイルに書き込みます。

最初のコマンドは、 のキー ARN を $keyArn変数 AWS KMS key に保存します。で暗号化する場合 AWS KMS key、キー ID、キー ARN、エイリアス名、またはエイリアス ARN を使用して識別できます。のキー識別子の詳細については AWS KMS key、「 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。

2 番目のコマンドは、ファイルの内容を暗号化します。コマンドは --encrypt パラメータを使用してオペレーションを指定し、--input パラメータを使用して暗号化するファイルを指定します。--wrapping-keys パラメータとその必須の key 属性は、キー ARN で AWS KMS key 表される を使用するように コマンドに指示します。

このコマンドは --metadata-output パラメータを使用して、暗号化オペレーションに関するメタデータのテキストファイルを指定します。ベストプラクティスとして、このコマンドは --encryption-context パラメータを使用して暗号化コンテキストを指定します。

このコマンドでは、--commitment-policy パラメータも使用してコミットメントポリシーを明示的に設定します。バージョン 1.8.x では、--wrapping-keys パラメータを使用するときにこのパラメータが必要です。バージョン 2.1.x 以降では、--commitment-policy パラメータはオプションですが推奨されます。

--output パラメータの値のドット (.) は、出力ファイルを現在のディレクトリに書き込むようコマンドに指示します。

Bash

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

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

PowerShell

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

PS C:\> aws-encryption-cli --encrypt `
                           --input Hello.txt `
                           --wrapping-keys key=$keyArn `
                           --metadata-output $home\Metadata.txt `
                           --commitment-policy require-encrypt-require-decrypt `
                           --encryption-context purpose=test `
                           --output .
```

暗号化コマンドが成功した場合、出力を返しません。コマンドが成功したかどうかを確認するには、$? 変数のブール値をチェックします。コマンドが成功した場合、$? の値は 0 (Bash) または True (PowerShell) です。コマンドが失敗した場合、$? の値は 0 以外 (Bash) または False (PowerShell) です。

Bash

```
$ echo $?
0
```

PowerShell

```
PS C:\> $?
True
```

また、ディレクトリー覧コマンドを使用して、暗号化コマンドが新しいファイル hello.txt.encrypted を作成したかどうかを確認することもできます。Encryption コマン

ドは出力のファイル名を指定しなかったため、 AWS Encryption CLI は入力ファイルと同じ名前と.encryptedサフィックスを持つファイルに出力を書き込みました。別のサフィックスを使用するか、サフィックスを抑制するには、--suffix パラメータを使用します。

hello.txt.encrypted ファイルには、暗号化メッセージが含まれています。それには、hello.txt ファイルの暗号テキスト、データキーの暗号化コピー、および暗号化コンテキストを含む追加のメタデータが含まれています。

Bash

```
$ ls
hello.txt  hello.txt.encrypted
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
```

## ファイルの復号

この例では、 Encryption AWS CLI を使用して、前の例で暗号化されたHello.txt.encryptedファイルの内容を復号します。

復号コマンドは --decrypt パラメータを使用してオペレーションを指定し、--input パラメータを使用して復号するファイルを指定します。--output パラメータの値は、現在のディレクトリを表すドットです。

key 属性を含む --wrapping-keys パラメータでは、暗号化されたメッセージの復号に使用するラッピングキーを指定します。を使用した復号コマンドでは AWS KMS keys、key 属性の値はキー ARN である必要があります。復号コマンドには --wrapping-keys パラメータが必要です。AWS KMS keysを使用している場合は、key 属性を使用して復号用の AWS KMS keys を指定するか、discovery 属性を true にします (両方ではない)。カスタムマスターキープロバイダーを使用している場合、key 属性と provider 属性は必須です。

バージョン 2.1.x 以降では、--commitment-policy パラメータはオプションですが推奨されます。明示的に使用すると、デフォルト値 require-encrypt-require-decrypt を指定した場合でも、意図が明確になります。

--encryption-context パラメータは、暗号化コマンドで暗号化コンテキストが指定されている場合でも、復号コマンドではオプションです。この場合、復号コマンドは暗号化コマンドで提供されたものと同じ暗号化コンテキストを使用します。暗号化 CLI AWS は、復号する前に、暗号化されたメッセージの暗号化コンテキストにpurpose=testペアが含まれていることを確認します。そうでない場合、復号コマンドは失敗します。

--metadata-output パラメータは、復号オペレーションに関するメタデータのファイルを指定します。--output パラメータの値のドット (.) は、出力ファイルを現在のディレクトリに書き込みます。

ベストプラクティスとして、--max-encrypted-data-keys パラメータを使用して、暗号化されたデータキーの数が多すぎる不正な形式のメッセージの復号化を回避してください。暗号化されたデータキーの予想数 (暗号化で使用されるラッピングキーごとに 1 つ)、または適切な最大値 (5 など) を指定します。詳細については、「暗号化されたデータキーの制限」を参照してください。

--buffer では、デジタル署名が存在する場合の検証も含めて、すべての入力が処理された後にのみプレーンテキストが返されます。

Bash

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

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

PowerShell

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

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

PS C:\> aws-encryption-cli --decrypt `
                           --input Hello.txt.encrypted `
                           --wrapping-keys key=$keyArn `
                           --commitment-policy require-encrypt-require-decrypt `
                           --encryption-context purpose=test `
                           --metadata-output $home\Metadata.txt `
                           --max-encrypted-data-keys 1 `
                           --buffer `
                           --output .
```

復号コマンドが成功した場合、出力を返しません。コマンドが成功したかどうかを確認するには、$? 変数の値を取得します。また、ディレクトリー覧コマンドを使用して、コマンドが .decrypted サフィックスが付加された新しいファイルを作成したかどうかを確認することもできます。プレーンテキストコンテンツを表示するには、ファイルコンテンツを取得するコマンドの cat や Get-Content などを使用します。

Bash

```
$  ls
hello.txt  hello.txt.encrypted  hello.txt.encrypted.decrypted

$  cat hello.txt.encrypted.decrypted
Hello World
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/17/2017   1:01 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
-a----        9/17/2017   1:08 PM             11 Hello.txt.encrypted.decrypted


PS C:\> Get-Content Hello.txt.encrypted.decrypted
```

```
Hello World
```

## ディレクトリ内のすべてのファイルの暗号化

この例では、Encryption AWS CLI を使用してディレクトリ内のすべてのファイルの内容を暗号化します。

コマンドが複数のファイルに影響する場合、 Encryption AWS CLI は各ファイルを個別に処理します。コマンドはファイルの内容を取得し、マスターキーからファイルの一意のデータキーを取得し、データキーでファイルの内容を暗号化して、結果を出力ディレクトリの新しいファイルに書き込みます。その結果、出力ファイルを個別に復号することができます。

TestDir ディレクトリのこのリストには、暗号化するプレーンテキストファイルが表示されています。

Bash

```
$  ls testdir
cool-new-thing.py  hello.txt  employees.csv
```

PowerShell

```
PS C:\> dir C:\TestDir

    Directory: C:\TestDir

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/12/2017   3:11 PM           2139 cool-new-thing.py
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:44 PM             46 Employees.csv
```

最初のコマンドは、 の Amazon リソースネーム (ARN) を $keyArn変数 AWS KMS key に保存します。

2 番目のコマンドは、TestDir ディレクトリ内のファイルのコンテンツを暗号化し、暗号化されたコンテンツのファイルを TestEnc ディレクトリに書き込みます。TestEnc ディレクトリが存在しない場合、コマンドは失敗します。入力場所はディレクトリのため、--recursive パラメータが必要です。

--wrapping-keys パラメータとその必須の key 属性は、使用するラッピングキーを指定します。暗号化コマンドには暗号化コンテキスト、dept=IT が含まれています。複数のファイルを暗号化するコマンドに暗号化コンテキストを指定すると、すべてのファイルで同じ暗号化コンテキストが使用されます。

コマンドには、暗号化オペレーションに関するメタデータを書き込む場所を Encryption AWS CLI に指示する --metadata-outputパラメータもあります。 AWS Encryption CLI は、暗号化されたファイルごとに 1 つのメタデータレコードを書き込みます。

バージョン 2.1.x.以降では、--commitment-policy parameter はオプションですが推奨されます。コマンドまたはスクリプトが暗号化テキストを復号化できないために失敗した場合は、明示的なコミットメントポリシー設定により、問題を迅速に検出できます。

コマンドが完了すると、Encryption AWS CLI は暗号化されたファイルを TestEnc ディレクトリに書き込みますが、出力は返しません。

最後のコマンドは TestEnc ディレクトリ内のファイルを一覧表示します。プレーンテキストの入力ファイルごとに、暗号化されたコンテンツの出力ファイルが 1 つあります。コマンドは代替サフィックスを指定していないため、暗号化コマンドは各入力ファイル名に .encrypted を付加しました。

Bash

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

$ aws-encryption-cli --encrypt \
                     --input testdir --recursive\
                     --wrapping-keys key=$keyArn \
                     --encryption-context dept=IT \
                     --commitment-policy require-encrypt-require-decrypt \
                     --metadata-output ~/metadata \
                     --output testenc

$ ls testenc
cool-new-thing.py.encrypted  employees.csv.encrypted  hello.txt.encrypted
```

PowerShell

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

PS C:\> aws-encryption-cli --encrypt `
                           --input .\TestDir --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output .\Metadata\Metadata.txt `
                           --output .\TestEnc

PS C:\> dir .\TestEnc

    Directory: C:\TestEnc

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

## ディレクトリ内のすべてのファイルの復号

この例では、ディレクトリ内のすべてのファイルを復号します。これは、前の例で暗号化された TestEnc ディレクトリ内のファイルから始まります。

Bash

```
$  ls testenc
cool-new-thing.py.encrypted  hello.txt.encrypted  employees.csv.encrypted
```

PowerShell

```
PS C:\> dir C:\TestEnc

    Directory: C:\TestEnc

Mode                 LastWriteTime         Length Name
```

```
----                 -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

この復号コマンドは、TestEnc ディレクトリ内のすべてのファイルを復号し、プレーンテキストファイルを TestDec ディレクトリに書き込みます。key 属性と key ARN 値を持つ --wrapping-keysパラメータは、ファイルの復号 AWS KMS keys に使用する を AWS Encryption CLI に伝えます。コマンドは、 --interactiveパラメータを使用して AWS 、同じ名前のファイルを上書きする前に Encryption CLI にプロンプトを表示します。

このコマンドは、ファイルが暗号化されたときに指定された暗号化コンテキストも使用します。複数のファイルを復号する場合、Encryption AWS CLI はすべてのファイルの暗号化コンテキストをチェックします。ファイルの暗号化コンテキストチェックが失敗した場合、Encryption AWS CLI はファイルを拒否し、警告を書き込み、メタデータに失敗を記録してから、残りのファイルのチェックを続行します。 AWS Encryption CLI が他の理由でファイルの復号に失敗した場合、復号コマンド全体がすぐに失敗します。

この例では、すべての入力ファイルで暗号化されたメッセージに dept=IT 暗号化コンテキスト要素が含まれています。ただし、異なる暗号化コンテキストを使用してメッセージを復号している場合でも、暗号化コンテキストの一部を確認できる可能性があります。たとえば、あるメッセージの暗号化コンテキストが dept=finance で、他のメッセージが dept=IT だった場合、その値を指定しないことで暗号化コンテキストに常に dept 名が含まれていることを確認できます。より具体的にするには、別々のコマンドでファイルを復号します。

復号コマンドから返される出力はありませんが、ディレクトリー覧コマンドを使用して .decrypted サフィックスが付いた新しいファイルを作成したことを確認できます。プレーンテキストコンテンツを表示するには、ファイルコンテンツを取得するコマンドを使用します。

Bash

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

$ aws-encryption-cli --decrypt \
                     --input testenc --recursive \
                     --wrapping-keys key=$keyArn \
                     --encryption-context dept=IT \
                     --commitment-policy require-encrypt-require-decrypt \
```

```
                    --metadata-output ~/metadata \
                    --max-encrypted-data-keys 1 \
                    --buffer \
                    --output testdec --interactive

$ ls testdec
cool-new-thing.py.encrypted.decrypted  hello.txt.encrypted.decrypted
 employees.csv.encrypted.decrypted
```

PowerShell

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

PS C:\> aws-encryption-cli --decrypt `
                           --input C:\TestEnc --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output $home\Metadata.txt `
                           --max-encrypted-data-keys 1 `
                           --buffer `
                           --output C:\TestDec --interactive

PS C:\> dir .\TestDec


    Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        10/8/2017   4:57 PM           2139 cool-new-
thing.py.encrypted.decrypted
-a----        10/8/2017   4:57 PM             46 Employees.csv.encrypted.decrypted
-a----        10/8/2017   4:57 PM             11 Hello.txt.encrypted.decrypted
```

## コマンドラインでの暗号化と復号

これらの例は、入力をコマンドにパイプし (stdin)、出力をコマンドライン (stdout) に書き込む方法を示しています。これらは、コマンドで stdin と stdout を表す方法と、組み込みの Base64 エンコードツールを使用してシェルが ASCII 以外の文字を誤って解釈するのを防ぐ方法について説明します。

この例では、パイププレーンテキストの文字列を暗号化コマンドにパイプし、暗号化されたメッセージを変数に保存します。次に、変数に格納された暗号化されたメッセージを復号コマンドにパイプし、その出力をパイプライン (stdout) に書き込みます。

この例では、3 つのコマンドで構成されています。

- 最初のコマンドは、 のキー ARN を $keyArn変数 AWS KMS key に保存します。

  Bash

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

  PowerShell

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

- 2 番目のコマンドは、Hello World 文字列を暗号化コマンドにパイプし、その結果を $encrypted 変数に保存します。

  --input および --output パラメータは、すべての AWS Encryption CLI コマンドで必須です。入力がコマンドにパイプされている (stdin) ことを示すには、(-) を --input パラメータの値に使用します。出力をコマンドラインに送信する (stdout) には、--output パラメータの値にハイフンを使用します。

  --encode パラメータは、出力を返す前に Base64 エンコードします。これにより、暗号化されたメッセージの ASCII 以外の文字をシェルが誤って解釈することを防止します。

  このコマンドは PoC (概念実証) に過ぎないため、暗号化コンテキストを省略し、メタデータを抑制します (-S)。

  Bash

  ```
  $ encrypted=$(echo 'Hello World' | aws-encryption-cli --encrypt -S \
                                                       --input - --output - --
  encode \
                                                       --wrapping-keys key=
  $keyArn )
  ```

PowerShell

```
PS C:\> $encrypted = 'Hello World' | aws-encryption-cli --encrypt -S `
                                                       --input - --output - --
encode `
                                                       --wrapping-keys key=
$keyArn
```

- 3 番目のコマンドは、$encrypted 変数に格納された暗号化されたメッセージを復号コマンドにパイプします。

  この復号コマンドでは、入力がパイプラインから来ている (stdin) ことを示すために --input - を使い、出力をパイプラインに送る (stdout) ために --output - を使います。(入力パラメータは実際の入力バイトではなく、入力の場所を取るため、$encrypted 変数を --input パラメータの値として使用することはできません。)

  この例では、 --wrapping-keysパラメータの discovery 属性を使用して、Encryption AWS CLI が任意の AWS KMS key を使用してデータを復号できるようにします。コミットメントポリシーは指定しないため、バージョン 2.1.x 以降のデフォルト値 require-encrypt-require-decrypt が使用されます。

  出力が暗号化されてエンコードされたため、復号コマンドは --decode パラメータを使用して Base64 でエンコードされた入力を復号する前にデコードします。また、--decode パラメータを使用して、Base64 でエンコードされた入力を暗号化する前にデコードすることもできます。

  ここでも、コマンドは暗号化コンテキストを省略し、メタデータを抑制します (-S)。

  Bash

  ```
  $  echo $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=true
   --input - --output - --decode --buffer -S
  Hello World
  ```

  PowerShell

  ```
  PS C:\> $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=$true
   --input - --output - --decode --buffer -S
  Hello World
  ```

暗号化および復号オペレーションは、介在する変数なしで 1 つのコマンドで実行することもできます。

前の例と同様に、--input および --output パラメータには - 値があり、このコマンドは --encode パラメータを使用して出力をエンコードし、--decode パラメータを使用して入力をデコードします。

Bash

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

$  echo 'Hello World' |
          aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --
output - --encode -S |
          aws-encryption-cli --decrypt --wrapping-keys discovery=true --input - --
output - --decode -S
Hello World
```

PowerShell

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

PS C:\> 'Hello World' |
               aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --
output - --encode -S |
               aws-encryption-cli --decrypt --wrapping-keys discovery=$true --input
 - --output - --decode -S
Hello World
```

## 複数のマスターキーの使用

この例では、 AWS Encryption CLI でデータを暗号化および復号するときに複数のマスターキーを使用する方法を示します。

複数のマスターキーを使用してデータを暗号化すると、いずれかのマスターキーを使用してデータを復号できます。この戦略では、マスターキーの 1 つが使用できなくてもデータを復号できます。暗号化されたデータを複数の に保存する場合 AWS リージョン、この戦略では、同じリージョンのマスターキーを使用してデータを復号できます。

複数のマスターキーで暗号化する場合、最初のマスターキーが特別な役割を果たします。それは、データの暗号化に使用されるデータキーを生成します。残りのマスターキーは、プレーンテキストのデータキーを暗号化します。結果として得られる暗号化されたメッセージには、暗号化されたデータと、各マスターキーに対して 1 つずつの暗号化されたデータキーの集合が含まれます。最初のマスターキーがデータキーを生成したにもかかわらず、いずれのマスターキーもデータキーの 1 つを復号でき、それをデータキーの復号に使用することができます。

3 つのマスターキーによる暗号化

このコマンド例では、3 つの AWS リージョンの 3 つのラッピングキーを使用して `Finance.log` ファイルを暗号化しています。

これは、暗号化されたメッセージを `Archive` ディレクトリに書き込みます。このコマンドでは、サフィックスを抑制する値を指定せずに `--suffix` パラメータを使用しているため、入力ファイル名と出力ファイル名は変わりません。

このコマンドは、`--wrapping-keys` パラメータとその 3 つの key 属性を使用します。同じコマンドで複数の `--wrapping-keys` パラメータを使用することもできます。

ログファイルを暗号化するために、 Encryption AWS CLI はリストの最初のラッピングキーに$key1、データの暗号化に使用するデータキーを生成するように要求します。次に、他の各ラッピングキーを使用して同じデータキーのプレーンテキストコピーを暗号化します。出力ファイルの暗号化されたメッセージには、暗号化された 3 つのデータキーがすべて含まれています。

Bash

```
$ key1=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$ key2=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
$ key3=arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d

$ aws-encryption-cli --encrypt --input /logs/finance.log \
                               --output /archive --suffix \
                               --encryption-context class=log \
                               --metadata-output ~/metadata \
                               --wrapping-keys key=$key1 key=$key2 key=$key3
```

PowerShell

```
PS C:\> $key1 = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
```

```
PS C:\> $key2 = 'arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef'
PS C:\> $key3 = 'arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d'

PS C:\> aws-encryption-cli --encrypt --input D:\Logs\Finance.log `
                           --output D:\Archive --suffix `
                           --encryption-context class=log `
                           --metadata-output $home\Metadata.txt `
                           --wrapping-keys key=$key1 key=$key2 key=$key3
```

このコマンドは、Finance.log ファイルの暗号化されたコピーを復号し、それを Finance.log.clear ディレクトリの Finance ファイルに書き込みます。3 つで暗号化されたデータを復号するには AWS KMS keys、同じ 3 AWS KMS keys つ、またはそのサブセットを指定できます。この例では AWS KMS keysを 1 つだけ指定します。

データの復号 AWS KMS keys に使用する AWS Encryption CLI に指示するには、 --wrapping-keysパラメータの key 属性を使用します。で復号する場合 AWS KMS keys、key 属性の値はキー ARN である必要があります。

AWS KMS keys 指定した で Decrypt API を呼び出すアクセス許可が必要です。詳細については、「AWS KMSに対する認証とアクセスコントロール」を参照してください。

ベストプラクティスとして、この例では --max-encrypted-data-keys パラメータを使用して、暗号化されたデータキーの数が多すぎる不正なメッセージの復号化を回避します。この例では、復号化にラッピングキーを 1 つだけ使用しますが、暗号化されたメッセージには、暗号化時に使用される 3 つのラッピングキーごとに 1 つずつ、合計 3 つの暗号化されたデータキーがあります。暗号化されたデータキーの予想数または適切な最大値 (5 など) を指定します。3 より小さい最大値を指定すると、コマンドは失敗します。詳細については、「暗号化されたデータキーの制限」を参照してください。

Bash

```
$ aws-encryption-cli --decrypt --input /archive/finance.log \
                     --wrapping-keys key=$key1 \
                     --output /finance --suffix '.clear' \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 3 \
                     --buffer \
                     --encryption-context class=log
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt `
                           --input D:\Archive\Finance.log `
                           --wrapping-keys key=$key1 `
                           --output D:\Finance --suffix '.clear' `
                           --metadata-output .\Metadata\Metadata.txt `
                           --max-encrypted-data-keys 3 `
                           --buffer `
                           --encryption-context class=log
```

## スクリプトでの暗号化と復号

この例では、スクリプトで AWS Encryption CLI を使用する方法を示します。データを暗号化または復号するだけのスクリプト、またはデータ管理プロセスの一部として暗号化または復号するスクリプトを作成できます。

この例では、スクリプトはログファイルのコレクションを取得、圧縮してから暗号化し、暗号化されたファイルを Amazon S3 バケットにコピーします。このスクリプトは各ファイルを別々に処理するため、それらを個別に復号して展開できます。

ファイルを圧縮して暗号化するときは、暗号化する前に圧縮する必要があります。適切に暗号化されたデータは圧縮できません。

> **Warning**
>
> シークレットデータと悪意のあるユーザーによって制御される可能性のあるデータの両方を含むデータを圧縮するときは注意が必要です。圧縮されたデータの最終サイズは、誤ってそのコンテンツに関する機密情報を明らかにする可能性があります。

Bash

```
# Continue running even if an operation fails.
set +e

dir=$1
encryptionContext=$2
s3bucket=$3
s3folder=$4
```

```
masterKeyProvider="aws-kms"
metadataOutput="/tmp/metadata-$(date +%s)"

compress(){
    gzip -qf $1
}

encrypt(){
    # -e encrypt
    # -i input
    # -o output
    # --metadata-output unique file for metadata
    # -m masterKey read from environment variable
    # -c encryption context read from the second argument.
    # -v be verbose
    aws-encryption-cli -e -i ${1} -o $(dirname ${1}) --metadata-output
 ${metadataOutput} -m key="${masterKey}" provider="${masterKeyProvider}" -c
 "${encryptionContext}" -v
}


s3put (){
    # copy file argument 1 to s3 location passed into the script.
    aws s3 cp ${1} ${s3bucket}/${s3folder}
}

# Validate all required arguments are present.
if [ "${dir}" ] && [ "${encryptionContext}" ] && [ "${s3bucket}" ] &&
 [ "${s3folder}" ] && [ "${masterKey}" ]; then

# Is $dir a valid directory?
test -d "${dir}"
if [ $? -ne 0 ]; then
    echo "Input is not a directory; exiting"
    exit 1
fi

# Iterate over all the files in the directory, except *gz and *encrypted (in case of
 a re-run).
for f in $(find ${dir} -type f \( -name "*" ! -name \*.gz ! -name \*encrypted \) );
 do
    echo "Working on $f"
    compress ${f}
    encrypt ${f}.gz
```

```
    rm -f ${f}.gz
    s3put ${f}.gz.encrypted
done;
else
    echo "Arguments: <Directory> <encryption context> <s3://bucketname> <s3 folder>"
    echo " and ENV var \$masterKey must be set"
    exit 255
fi
```

### PowerShell

```
#Requires -Modules AWSPowerShell, Microsoft.PowerShell.Archive
Param
(
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String[]]
    $FilePath,

    [Parameter()]
    [Switch]
    $Recurse,

    [Parameter(Mandatory=$true)]
    [String]
    $wrappingKeyID,

    [Parameter()]
    [String]
    $masterKeyProvider = 'aws-kms',

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $ZipDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $EncryptDirectory,

    [Parameter()]
    [String]
```

```
    $EncryptionContext,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $MetadataDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-S3Bucket -BucketName $_})]
    [String]
    $S3Bucket,

    [Parameter()]
    [String]
    $S3BucketFolder
)

BEGIN {}
PROCESS {
    if ($files = dir $FilePath -Recurse:$Recurse)
    {

        # Step 1: Compress
        foreach ($file in $files)
        {
            $fileName = $file.Name
            try
            {
                Microsoft.PowerShell.Archive\Compress-Archive -Path $file.FullName -
DestinationPath $ZipDirectory\$filename.zip
            }
            catch
            {
                Write-Error "Zip failed on $file.FullName"
            }

            # Step 2: Encrypt
            if (-not (Test-Path "$ZipDirectory\$filename.zip"))
            {
                Write-Error "Cannot find zipped file: $ZipDirectory\$filename.zip"
            }
            else
            {
                # 2>&1 captures command output
```

```
                $err = (aws-encryption-cli -e -i "$ZipDirectory\$filename.zip" `
                                           -o $EncryptDirectory `
                                           -m key=$wrappingKeyID provider=
$masterKeyProvider `
                                           -c $EncryptionContext `
                                           --metadata-output $MetadataDirectory `
                                           -v) 2>&1

                # Check error status
                if ($? -eq $false)
                {
                    # Write the error
                    $err
                }
                elseif (Test-Path "$EncryptDirectory\$fileName.zip.encrypted")
                {
                    # Step 3: Write to S3 bucket
                    if ($S3BucketFolder)
                    {
                        Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted" -Key "$S3BucketFolder/
$fileName.zip.encrypted"

                    }
                    else
                    {
                        Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted"
                    }
                }
            }
        }
    }
}
```

## データキーキャッシュの使用

この例では、多数のファイルを暗号化するコマンドでデータキーキャッシュを使用します。

デフォルトでは、Encryption AWS CLI (および の他のバージョン AWS Encryption SDK) は、暗号化するファイルごとに一意のデータキーを生成します。各オペレーションに一意のデータキーを使用するのは暗号化のベストプラクティスですが、一部の状況では限定的なデータキーの再利用が許容さ

れます。データキーキャッシュを検討している場合は、セキュリティエンジニアに相談して、アプリケーションのセキュリティ要件を理解し、適切なセキュリティしきい値を判断してください。

この例では、マスターキープロバイダへのリクエストの頻度を減らすことによって、データキーキャッシュは暗号化オペレーションを高速化します。

この例のコマンドは、合計約 800 の小さなログファイルを含む複数のサブディレクトリを持つ大きなディレクトリを暗号化します。最初のコマンドは、 AWS KMS key の ARN を `keyARN` 変数に保存します。2 番目のコマンドは、入力ディレクトリ内のすべてのファイルを (再帰的に) 暗号化し、アーカイブディレクトリに書き込みます。このコマンドでは、`--suffix` パラメータを使用して `.archive` サフィックスを指定します。

`--caching` パラメータはデータキーキャッシュを有効にします。シリアルのファイル処理では一度に複数のデータキーを使用することはないため、キャッシュ内のデータキーの数を制限する capacity 属性は 1 に設定されます。キャッシュされたデータキーの使用可能時間を決定する max_age 属性は 10 秒に設定します。

オプションの max_messages_encrypted 属性は 10 個のメッセージに設定されているため、1 つのデータキーが 10 個以上のファイルを暗号化するために使用されることはありません。各データキーで暗号化するファイル数を制限することで、万一データキーが侵害された場合に影響を受けるファイルの数を減らすことができます。

オペレーティングシステムが生成するログファイルに対してこのコマンドを実行するには、管理者アクセス許可 (Linux では `sudo`、Windows では [管理者として実行]) が必要な場合があります。

Bash

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

$  aws-encryption-cli --encrypt \
                     --input /var/log/httpd --recursive \
                     --output ~/archive --suffix .archive \
                     --wrapping-keys key=$keyArn \
                     --encryption-context class=log \
                     --suppress-metadata \
                     --caching capacity=1 max_age=10 max_messages_encrypted=10
```

PowerShell

```
PS C:\> $keyARN = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --encrypt `
                           --input C:\Windows\Logs --recursive `
                           --output $home\Archive --suffix '.archive' `
                           --wrapping-keys key=$keyARN `
                           --encryption-context class=log `
                           --suppress-metadata `
                           --caching capacity=1 max_age=10
 max_messages_encrypted=10
```

データキーキャッシュの効果をテストするために、この例では PowerShell の Measure-Command コマンドレットを使用します。データキーキャッシュなしでこの例を実行すると、完了に約 25 秒かかります。このプロセスは、ディレクトリ内のファイルごとに新しいデータキーを生成します。

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 25
Milliseconds      : 453
Ticks             : 254531202
TotalDays         : 0.000294596298611111
TotalHours        : 0.00707031116666667
TotalMinutes      : 0.42421867
TotalSeconds      : 25.4531202
TotalMilliseconds : 25453.1202
```

データキーキャッシュを使用すると、各データキーを最大 10 個のファイルに制限しても、処理が迅速になります。このコマンドは完了するまでに 12 秒もかからず、マスターキープロバイダへの呼び出し回数を元の値の 1/10 に減らしました。

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10
 max_messages_encrypted=10}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 11
Milliseconds      : 813
Ticks             : 118132640
TotalDays         : 0.000136727592592593
TotalHours        : 0.00328146222222222
TotalMinutes      : 0.196887733333333
TotalSeconds      : 11.813264
TotalMilliseconds : 11813.264
```

max_messages_encrypted の制限を解除すると、すべてのファイルが同じデータキーで暗号化されます。この変更では、プロセスは大幅に高速化されず、データキーの再利用に伴うリスクが高まります。ただし、マスターキープロバイダの呼び出し数は 1 回に減ります。

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10}


Days              : 0
```

```
Hours             : 0
Minutes           : 0
Seconds           : 10
Milliseconds      : 252
Ticks             : 102523367
TotalDays         : 0.000118661304398148
TotalHours        : 0.00284787130555556
TotalMinutes      : 0.170872278333333
TotalSeconds      : 10.2523367
TotalMilliseconds : 10252.3367
```

# AWS Encryption SDK CLI 構文とパラメータリファレンス

このトピックでは、 AWS Encryption SDK コマンドラインインターフェイス (CLI) の使用に役立つ構文の例を示し、パラメータについて簡単に説明します。ラッピングキーおよびその他のパラメータのヘルプについては、「Encryption AWS CLI の使用方法」を参照してください。例については「Encryption AWS CLI の例」を参照してください。完全なドキュメントについては、「ドキュメントを読む」を参照してください。

トピック

- AWS 暗号化 CLI 構文
- AWS Encryption CLI コマンドラインパラメータ
- 高度なパラメータ

## AWS 暗号化 CLI 構文

これらの AWS Encryption CLI 構文図は、 AWS Encryption CLI で実行する各タスクの構文を示しています。これらは、Encryption CLI AWS バージョン 2.1.x 以降の推奨構文を表します。

新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

> **Note**
>
> パラメータの説明に記載されている場合を除き、各パラメータまたは属性は、各コマンドで 1 回のみ使用できます。

パラメータがサポートしていない属性を使用する場合、Encryption AWS CLI は警告やエラーなしでサポートされていない属性を無視します。

### ヘルプの表示

パラメータの説明を含む完全な AWS Encryption CLI 構文を取得するには、 `--help`または を使用します`-h`。

```
aws-encryption-cli (--help | -h)
```

### バージョンの取得

Encryption CLI AWS インストールのバージョン番号を取得するには、 を使用します`--version`。Encryption CLI AWS の使用に関する質問、問題の報告、ヒントの共有を行う場合は、必ず バージョンを含めてください。

```
aws-encryption-cli --version
```

### データを暗号化する

次の構文の例は、encrypt コマンドで使用するパラメータを示しています。

```
aws-encryption-cli --encrypt
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite] [--suffix
 [<suffix>]] [--encode]
                   --wrapping-keys  [--wrapping-keys] ...
                       key=<keyID> [key=<keyID>] ...
                       [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                   --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
                   [--commitment-policy <commitment-policy>]
                   [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                   [--max-encrypted-data-keys <integer>]
                   [--algorithm <algorithm_suite>]
                   [--caching <attributes>]
                   [--frame-length <length>]
                   [-v | -vv | -vvv | -vvvv]
```

```
                    [--quiet]
```

### データを復号化する

次の構文の例は、decrypt コマンドで使用するパラメータを示しています。

バージョン 1.8.x では、復号時の `--wrapping-keys` パラメータはオプションですが推奨されます。バージョン 2.1.x 以降は、`--wrapping-keys` パラメータが暗号化および復号化時に必要となります。 AWS KMS keysの場合、key 属性を使用してラッピングキーを指定するか (ベストプラクティス)、discovery 属性を `true` に設定して、 AWS Encryption CLI が使用できるラッピングキーを制限しません。

```
aws-encryption-cli --decrypt (or [--decrypt-unsigned])
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite]  [--suffix
 [<suffix>]] [--encode]
                   --wrapping-keys  [--wrapping-keys] ...
                       [key=<keyID>] [key=<keyID>] ...
                       [discovery={true|false}] [discovery-partition=<aws-partition-
name> discovery-account=<aws-account-ID> [discovery-account=<aws-account-ID>] ...]
                       [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                   --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
                   [--commitment-policy <commitment-policy>]
                   [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                   [--buffer]
                   [--max-encrypted-data-keys <integer>]
                   [--caching <attributes>]
                   [--max-length <length>]
                   [-v | -vv | -vvv | -vvvv]
                   [--quiet]
```

### 設定ファイルの使用

パラメータとその値が格納されている設定ファイルを参照できます。これは、コマンドでパラメータと値を入力するのに相当します。例については、設定ファイルにパラメータを保存する方法を参照してください。

```
aws-encryption-cli @<configuration_file>
```

```
# In a PowerShell console, use a backtick to escape the @.
aws-encryption-cli `@<configuration_file>
```

## AWS Encryption CLI コマンドラインパラメータ

このリストでは、Encryption AWS CLI コマンドパラメータの基本的な説明を示します。詳細な説明については、aws-encryption-sdk-cli のドキュメントを参照してください。

--encrypt (-e)

入力データを暗号化します。すべてのコマンドに、--encrypt、--decrypt、--decrypt-unsigned パラメータのいずれかが必要です。

--decrypt (-d)

入力データを復号します。すべてのコマンドに、--encrypt、--decrypt、--decrypt-unsigned パラメータのいずれかが必要です。

—decrypt-unsigned [バージョン 1.9.x および 2.2.x で導入]

--decrypt-unsigned パラメータでは、暗号化テキストを復号し、復号化前にメッセージが署名なしであることを確認します。このパラメータは、--algorithm パラメータを使用し、データを暗号化するためのデジタル署名なしのアルゴリズムスイートを選択した場合に使用します。暗号化テキストが署名されている場合、復号化は失敗します。

--decrypt または --decrypt-unsigned を復号化に使用できますが、両方とも使用することはできません。

—wrapping-keys (-w) [バージョン 1.8.x で導入]

暗号化と復号のオペレーションで使用されるラッピングキー (マスターキー) を指定します。各コマンドで複数の --wrapping-keys パラメータを使用できます。

バージョン 2.1.x 以降は、--wrapping-keys パラメータが暗号化コマンドおよび復号化コマンド時に必要となります。バージョン 1.8.x では、暗号化コマンドには --wrapping-keys または --master-keys パラメーターが必要です。バージョン 1.8.x の復号化のコマンドでは、--wrapping-keys パラメータはオプションですが推奨されます。

カスタムのマスターキープロバイダーを使用するとき、暗号化と復号のコマンドでは、key および provider 属性が必須です。を使用する場合 AWS KMS keys、暗号化コマンドには key 属性が必要です。復号コマンドでは、true 値の key 属性、または discovery 属性が必要です ( 両方で

はない)。復号時に key 属性を使用することが、AWS Encryption SDK のベストプラクティスです。Amazon S3 バケットや Amazon SQS キュー内のメッセージなど、なじみのないメッセージのバッチを復号化する場合は、これが特に重要です。

AWS KMS マルチリージョンキーをラッピングキーとして使用する方法の例については、「」を参照してくださいマルチリージョンの使用 AWS KMS keys。

属性: `--wrapping-keys` パラメータの値は、以下の属性で構成されます。形式は `attribute_name=value` です。

キー

オペレーションで使用するラッピングキーを識別します。形式は、key=ID のペアです。各 `--wrapping-keys` パラメータ値に、複数の key 属性を指定できます。

- 暗号化コマンド: すべての暗号化コマンドには key 属性が必要です。暗号化コマンド AWS KMS key で を使用する場合、key 属性の値は、キー ID、キー ARN、エイリアス名、またはエイリアス ARN です。 AWS KMS 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。
- 復号コマンド: AWS KMS keysで復号する場合、`--wrapping-keys` パラメータでは key 属性をキー ARN にするか、discovery 属性を `true` にする必要があります (両方ではない)。key 属性を使用することが、AWS Encryption SDK のベストプラクティスです。カスタムのマスターキープロバイダーで復号化する場合、key 属性は必須です。

> **Note**
>
> 復号コマンドで AWS KMS ラッピングキーを指定するには、key 属性の値がキー ARN である必要があります。キー ID、エイリアス名、またはエイリアス ARN を使用する場合、Encryption AWS CLI はラッピングキーを認識しません。

各 `--wrapping-keys` パラメータ値に、複数の key 属性を指定できます。ただし、`--wrapping-keys` パラメータの provider、region、profile 属性は、そのパラメータ値のすべてのラッピングキーに適用されます。異なる属性値を持つラッピングキーを指定するには、コマンドで複数の `--wrapping-keys` パラメータを使用します。

discovery

AWS Encryption CLI が任意の AWS KMS key を使用してメッセージを復号できるようにします。discovery の値は、`true` または `false` にすることができます。デフォルト値は `false` です。discovery 属性は、復号コマンドで、マスターキープロバイダーが AWS KMSのときにのみ有効です。

で復号する場合 AWS KMS keys、 `--wrapping-keys`パラメータには key 属性または discovery 属性が必要です `true` (両方ではありません ) 。key 属性を使用する場合は、discovery 属性を `false` にして、検出を明示的に拒否できます。

- `False` (デフォルト) — discovery 属性が指定されていないか、その値が の場合`false`、AWS Encryption CLI は `--wrapping-keys`パラメータの key 属性で AWS KMS keys 指定された のみを使用してメッセージを復号します。discovery を `false` にして key 属性を指定しないと、復号コマンドは失敗します。この値は Encryption AWS CLI のベストプラクティスをサポートしています。
- `True` — discovery 属性の値が の場合`true`、Encryption AWS CLI は暗号化されたメッセージのメタデータ AWS KMS keys から AWS KMS keys を取得し、それらを使用してメッセージを復号します。の値を持つ discovery 属性`true`は、復号時にラッピングキーを指定できなかったバージョン 1.8.x より前のバージョンの AWS Encryption CLI のように動作します。ただし、 を使用する意図 AWS KMS key は明示的です。discovery を `true` にして key 属性を指定すると、復号コマンドは失敗します。

  この`true`値を指定する AWS アカウント と、 Encryption AWS CLI AWS KMS keys がさまざまな およびリージョン AWS KMS keys で を使用するか、ユーザーが使用を認可されていない を使用しようとする可能性があります。

discovery が の場合`true`、discovery-partition 属性と discovery-account 属性を使用して、AWS KMS keys 使用する を AWS アカウント 指定した 内のものに制限するのがベストプラクティスです。

discovery-account

復号 AWS KMS keys に使用する を、指定された 内のものに制限します AWS アカウント。この属性で有効な値は AWS アカウント ID のみです。

この属性はオプションであり、discovery 属性 AWS KMS keys が に設定`true`され、discovery-partition 属性が指定されている 復号コマンドでのみ有効です。

各 discovery-account 属性は 1 つの AWS アカウント ID のみを取りますが、同じ`--wrapping-keys`パラメータで複数の discovery-account 属性を指定できます。特定の `--wrapping-keys` パラメータで指定するすべてのアカウントは、指定した AWS パーティション内に存在する必要があります。

discovery-partition

discovery-account 属性でアカウントの AWS パーティションを指定します。その値は、、`aws`、 `aws-cn`などの AWS パーティションである必要がありま`aws-gov-cloud`す。

詳細については、「AWS 全般のリファレンス」の「Amazon リソースネーム」を参照してください。

この属性は、discovery-account 属性を使用するとき必須です。各 `--wrapping keys` パラメータに指定できる discovery-partition パーティションは 1 つだけです。複数のパーティション AWS アカウント で を指定するには、追加の `--wrapping-keys` パラメータを使用します。

provider

マスターキープロバイダーを指定します。形式は、provider=ID のペアです。デフォルト値の aws-kms は を表します AWS KMS。この属性は、マスターキープロバイダーがそうでない場合にのみ必要です AWS KMS。

region

AWS リージョン の を識別します AWS KMS key。この属性は に対してのみ有効です AWS KMS keys。key の識別子が特定のリージョンを示していない場合にのみ使用され、それ以外の場合は無視されます。これを使用すると、 AWS CLI 名前付きプロファイルのデフォルトリージョンが上書きされます。

profile

AWS CLI 名前付きプロファイルを識別します。この属性は に対してのみ有効です AWS KMS keys。プロファイルのリージョンは、key の識別子が特定のリージョンを示しておらず、コマンドに region 属性がない場合にのみ使用されます。

--input (-i)

暗号化または復号するデータの場所を指定します。このパラメータは必須です。指定できる値は、ファイルかディレクトリへのパス、またはファイル名のパターンです。コマンド (stdin) にパイピング入力する場合は、- を使用します。

入力が存在しない場合、エラーまたは警告なしでコマンドが正常に完了します。

--recursive (-r, -R)

入力ディレクトリとそのサブディレクトリにあるファイルでオペレーションを実行します。このパラメータは、`--input` の値がディレクトリの場合に必要です。

--decode

Base64-encoded 入力をデコードします。

暗号化されエンコードされたメッセージを復号する場合は、復号する前にメッセージをデコードする必要があります。これはパラメータによって実行されます。

たとえば、暗号化コマンドで `--encode` パラメータを使用した場合、対応する復号コマンドで `--decode` パラメータを使用します。また、このパラメータを使用して、Base64 でエンコードされた入力を暗号化する前にデコードすることもできます。

--output (-o)

出力先を指定します。このパラメータは必須です。値には、既存のディレクトリ、ファイル名、またはコマンドライン (stdout) に出力を書き込む - を使用できます。

指定した出力ディレクトリが存在しない場合、コマンドは失敗します。入力にサブディレクトリが含まれている場合、Encryption AWS CLI は指定した出力ディレクトリの下にサブディレクトリを再現します。

デフォルトでは、Encryption AWS CLI は同じ名前のファイルを上書きします。この動作を変更するには、`--interactive` または `--no-overwrite` パラメータを使用します。上書きの警告を表示しないようにするには、`--quiet` パラメータを使用します。

> **Note**
>
> 出力ファイルを上書きするコマンドが失敗した場合、出力ファイルは削除されます。

--インタラクティブ

ファイルを上書きする前にプロンプトが表示されます。

--no-overwrite

ファイルは上書きされません。代わりに、出力ファイルが存在する場合、Encryption AWS CLI は対応する入力をスキップします。

--サフィックス

Encryption AWS CLI が作成するファイルのカスタムファイル名サフィックスを指定します。サフィックスがないことを示すには、値のないパラメータ (`--suffix`) を使用します。

デフォルトでは、`--output` パラメータでファイル名が指定されない場合、出力ファイル名は同じ名前の入力ファイル名にサフィックスを加えたものになります。暗号化コマンドのサフィックスは `.encrypted` です。復号コマンドのサフィックスは `.decrypted` です。

--encode

Base64 (バイナリからテキスト) エンコーディングを出力に適用します。エンコーディングによりシェルホストプログラムが、出力テキストの非 ASCII 文字を誤って解釈するのを防ぎます。

出力を別のコマンドにパイピング、または変数に保存する場合でも、暗号化された出力を stdout (`--output -`) 特に PowerShell コンソールに書き込むときはこのパラメータを使用します。

--metadata-output

暗号化オペレーションに関するメタデータの場所を指定します。パスとファイル名を入力します。ディレクトリが存在しない場合、コマンドは失敗します。コマンドライン (stdout) にメタデータを書き込むには、- を使用します。

同じコマンドでコマンド出力 (`--output`) とメタデータ出力 (`--metadata-output`) を stdout に記述することはできません また、`--input` や `--output` の値がディレクトリ (ファイル名なし) の場合は、メタデータ出力を同じディレクトリまたはそのディレクトリのサブディレクトリに書き込むことはできません。

既存のファイルを指定すると、デフォルトでは、Encryption AWS CLI はファイル内の任意のコンテンツに新しいメタデータレコードを追加します。この機能を使用すると、すべての暗号化オペレーションのメタデータが格納された 1 つのファイルを作成できます。既存のファイルのコンテンツを上書きするには、`--overwrite-metadata` パラメータを使用します。

AWS Encryption CLI は、コマンドが実行する暗号化または復号オペレーションごとに JSON 形式のメタデータレコードを返します。各メタデータレコードには、入力ファイルと出力ファイル、暗号化コンテキスト、アルゴリズムスイート、セキュリティ基準を満たしているかどうかを検証しオペレーションを確認するために使用できるその他の有益な情報への完全なパスが含まれます。

--overwrite-metadata

メタデータの出力ファイルでコンテンツが上書きされます。デフォルトでは、`--metadata-output` パラメータはファイル内の既存のコンテンツにメタデータを追加します。

--suppress-metadata (-S)

暗号化または復号オペレーションに関するメタデータを抑制します。

--commitment-policy

暗号化および復号コマンドのコミットメントポリシーを指定します。コミットメントポリシーは、キーコミットメントセキュリティ機能を使用してメッセージを暗号化および復号化するかどうかを決定します。

`--commitment-policy` パラメータはバージョン 1.8.x で導入されました。暗号化コマンドと復号コマンドで有効です。

バージョン 1.8.x AWS では、Encryption CLI はすべての暗号化および復号オペレーションに`forbid-encrypt-allow-decrypt`コミットメントポリシーを使用します。`--wrapping-keys` パラメータを暗号化コマンドまたは復号コマンドで使用するときには、値 `forbid-encrypt-allow-decrypt` を指定した `--commitment-policy` パラメータが必要です。`--wrapping-keys` パラメータを使用しない場合、`--commitment-policy` パラメータは無効です。コミットメントポリシーを明示的に設定すると、バージョン 2.1.x へのアップグレード時にコミットメントポリシーが自動的に `require-encrypt-require-decrypt` に変更されなくなります。

バージョン 2.1.x 以降は、すべてのコミットメントポリシーの値がサポートされます。`--commitment-policy` パラメータはオプションであり、デフォルト値は `require-encrypt-require-decrypt` です。

このパラメータには次の値があります。

- `forbid-encrypt-allow-decrypt` — キーコミットメントで暗号化することはできません。キーコミットメントが使用されているかどうかにかかわらず、暗号化された暗号化テキストを復号化できます。

  バージョン 1.8.x では、これが唯一の有効な値です。 AWS Encryption CLI は、すべての暗号化および復号オペレーションに`forbid-encrypt-allow-decrypt`コミットメントポリシーを使用します。
- `require-encrypt-allow-decrypt` — キーコミットメントで暗号化します。復号化はキーコミットメントの有無に関係なく行われます。この値はバージョン 2.1.x で導入されました。
- `require-encrypt-require-decrypt` (デフォルト) — キーコミットメントでのみ暗号化および復号化が行われます。この値はバージョン 2.1.x で導入されました。バージョン 2.1.x 以降では、これがデフォルト値です。この値では、 AWS Encryption CLI は、以前のバージョンので暗号化された暗号文を復号しません AWS Encryption SDK。

コミットメントポリシーの設定の詳細については、「の移行 AWS Encryption SDK」を参照してください。

--encryption-context (-c)

オペレーションの暗号化コンテキストを指定します。このパラメータは必須ではありませんが、推奨されています。

- `--encrypt` コマンドでは、1 つまたは複数の `name=value` ペアを入力します。ペアを区切るには、スペースを使用します。
- `--decrypt` コマンドでは、`name=value` ペア、値のない `name` 要素、またはその両方を入力します。

`name` ペアの `value` や `name=value` にスペースや特殊文字が含まれている場合、ペア全体を引用符で囲みます。例えば、`--encryption-context "department=software development"` と指定します。

—buffer (-b) [バージョン 1.9.x および 2.2.x で導入]

デジタル署名が存在する場合の検証も含めて、すべての入力が処理された後にのみプレーンテキストが返されます。

--max-encrypted-data-keys [バージョン 1.9.x および 2.2.x で導入]

暗号化されたメッセージ内の暗号化されたデータキーの最大数を指定します。このパラメータはオプションです。

有効な値は 1～65,535 です。このパラメータを省略すると、Encryption AWS CLI は最大値を適用しません。暗号化されたメッセージには、最大 65,535 (2^16 - 1) の暗号化されたデータキーを使用できます。

このパラメータを暗号化コマンドで使用して、不正な形式のメッセージを防ぐことができます。これを復号コマンドで使用して、悪意のあるメッセージを検出し、復号できない多数の暗号化されたデータキーを含むメッセージの復号化を回避できます。詳細と例については、「暗号化されたデータキーの制限」を参照してください。

--help (-h)

コマンドラインで使用量と構文を表示します。

--version

Encryption AWS CLI のバージョンを取得します。

-v | -vv | -vvv | -vvvv

詳細な情報、警告、およびデバッグメッセージを表示します。出力の詳細は、パラメータ内の vs 数とともに増加します。最も詳細な設定 (-vvvv) は、 Encryption AWS CLI とそれが使用するすべてのコンポーネントからデバッグレベルのデータを返します。

--quiet (-q)

出力ファイルを上書きしたときに表示されるメッセージなど、警告メッセージを抑制します。

--master-keys (-m) [非推奨]

> **Note**
>
> —master-keys パラメータは 1.8.x で非推奨となり、バージョン 2.1.x で削除されました。代わりに、--wrapping-keys パラメータを使用してください。

暗号化と復号のオペレーションで使用されるマスターキーを指定します。複数のマスターキーパラメータを各コマンドで使用できます。

この暗号化コマンドには、`--master-keys` パラメータが必要です。これはカスタム (AWS KMS 以外の) マスターキープロバイダーを使用しているときにのみ、復号コマンドで必要です。

属性: `--master-keys` パラメータの値は、以下の属性で構成されます。形式は `attribute_name=value` です。

キー

オペレーションで使用するラッピングキーを識別します。形式は、key=ID のペアです。すべての暗号化コマンドには、key 属性が必要です。

暗号化コマンド AWS KMS key で を使用する場合、key 属性の値は、キー ID、キー ARN、エイリアス名、またはエイリアス ARN です。 AWS KMS 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。

マスターキープロバイダーが AWS KMSでない場合、復号コマンドには key 属性が必須です。 AWS KMS keyで暗号化されたデータを復号するコマンドでは、key 属性は許可されていません。

各 `--master-keys` パラメータ値に、複数の key 属性を指定できます。ただし、provider、region、および profile 属性は、パラメータ値のマスターキーすべてに適用さ

れます。異なる属性値を持つマスターキーを指定するには、コマンドで複数の `--master-keys` パラメータを使用します。

provider

マスターキープロバイダーを指定します。形式は、provider=ID のペアです。デフォルト値 aws-kms は を表します AWS KMS。この属性は、マスターキープロバイダーがそうでない場合にのみ必要です AWS KMS。

region

AWS リージョン の を識別します AWS KMS key。この属性は に対してのみ有効です AWS KMS keys。key の識別子が特定のリージョンを示していない場合にのみ使用され、それ以外の場合は無視されます。これを使用すると、 AWS CLI 名前付きプロファイルのデフォルトリージョンが上書きされます。

profile

AWS CLI 名前付きプロファイルを識別します。この属性は に対してのみ有効です AWS KMS keys。プロファイルのリージョンは、key の識別子が特定のリージョンを示しておらず、コマンドに region 属性がない場合にのみ使用されます。

## 高度なパラメータ

--algorithm

アルゴリズムスイートの代替を指定します。このパラメータはオプションであり、暗号化コマンドでのみ有効です。

このパラメータを省略すると、Encryption CLI はバージョン AWS 1.8.x で AWS Encryption SDK 導入された のデフォルトのアルゴリズムスイートのいずれかを使用します。どちらのデフォルトアルゴリズムも、HKDF、ECDSA 署名、および 256 ビットの暗号化キーを含む AES-GCM を使用します。キーコミットメントは、使用される場合と使用されない場合があります。デフォルトのアルゴリズムスイートは、コマンドのコミットメントポリシーによって選択されます。

デフォルトアルゴリズムスイートは、ほとんどの暗号化オペレーションで推奨されます。有効な値のリストについては、「ドキュメントを読む」の `algorithm` パラメータの値を参照してください。

--frame-length

指定されたフレームの長さで出力を作成します。このパラメータはオプションであり、暗号化コマンドでのみ有効です。

値をバイトで入力します。有効な値は、0 および 1 ~ 2^31 - 1 です。値 0 は、フレーム化されていないデータを示します。デフォルト値は 4096 (バイト) です。

> **Note**
>
> 可能な限り、フレーム化されたデータを使用してください。は、レガシー使用のためにのみフレーム化されていないデータ AWS Encryption SDK をサポートします。の一部の言語実装では、フレーム化されていない暗号文を生成 AWS Encryption SDK できます。サポートされているすべての言語実装では、フレーム化された暗号化テキストとフレーム化されていない暗号化文書を復号化できます。

--max-length

暗号化されたメッセージから読み取る最大フレームサイズ (またはフレーム化されていないメッセージの最大コンテンツ長) をバイト数で指定します。このパラメータはオプションであり、復号コマンドでのみ有効です。これは、悪意のある膨大な量の暗号化テキストを復号する際に保護できるように設計されています。

値をバイトで入力します。このパラメータを省略した場合、 AWS Encryption SDK は復号時にフレームサイズを制限しません。

--caching

入力ファイルごとに新しいデータキーを生成する代わりに、データキーを再利用するデータキーキャッシュ機能を有効にします。このパラメータでは、高度なシナリオがサポートされています。この機能を使用する前に、データキーキャッシュのドキュメントを参照してください。

`--caching` パラメータには以下のような属性があります。

capacity (必須)

キャッシュのエントリの最大数を決定します。

最小値は 1 です。最大値は存在しません。

max_age (必須)

キャッシュエントリがキャッシュに追加された時点から、どのくらいの期間使用されるかを決定します (秒単位)。

0 より大きい値を入力します。最大値は存在しません。

max_messages_encrypted (オプション)

キャッシュされたエントリが暗号化できるメッセージの最大数を決定します。

有効な値は 1～2^32 です。デフォルト値は 2^32 (メッセージ) です。

max_bytes_encrypted (オプション)

キャッシュされたエントリが暗号化できるバイトの最大数を決定します。

有効な値は、0 および 1 ～ 2^63 - 1 です。デフォルト値は 2^63 - 1 (メッセージ) です。値を 0 に指定することで、空のメッセージ文字列を暗号化している場合にのみデータキーキャッシュを使用できます。

## Encryption AWS CLI のバージョン

最新バージョンの Encryption AWS CLI を使用することをお勧めします。

> **Note**
>
> 4.0.0 より前のバージョンの AWS Encryption end-of-supportフェーズにあります。
> バージョン 2.1.x 以降から、コードやデータを変更せずに最新バージョンの AWS Encryption CLI に安全に更新できます。ただし、バージョン 2.1.x で導入された新しいセキュリティ機能には下位互換性がありません。バージョン 1.7.x 以前から更新するには、まず Encryption CLI の最新バージョンの AWS 1.x に更新する必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

の重要バージョンについては AWS Encryption SDK、「」を参照してくださいのバージョン AWS Encryption SDK。

使用すべきバージョン

AWS Encryption CLI を初めて使用する場合は、最新バージョンを使用してください。

バージョン 1.7.x より AWS Encryption SDK 前の のバージョンで暗号化されたデータを復号するには、まず Encryption AWS CLI の最新バージョンに移行します。推奨される変更をすべて行ってから、バージョン 2.1.x 以降にアップデートしてください。詳細については、「の移行 AWS Encryption SDK」を参照してください。

詳細はこちら

- これらの新しいバージョンへの移行に関する変更とガイダンスについては、「の移行 AWS Encryption SDK」を参照してください。
- 新しい Encryption AWS CLI パラメータと属性の詳細については、「」を参照してくださいAWS Encryption SDK CLI 構文とパラメータリファレンス。

次のリストでは、バージョン AWS 1.8.x および 2.1.x での Encryption CLI の変更について説明します。

## Encryption CLI のバージョン AWS 1.8.x の変更

- `--master-keys` パラメータは非推奨となります。代わりに、`--wrapping-keys` パラメータを使用します。
- `--wrapping-keys` (`-w`) パラメータが追加されます。`--master-keys` パラメータのすべての属性がサポートされます。また、次のオプションの属性も追加されます。これは、 AWS KMS keys で復号化する場合にのみ有効です。
  - discovery
  - discovery-partition
  - discovery-account

  カスタムマスターキープロバイダーの場合、`--encrypt` と `--decrypt` コマンドには、`--wrapping-keys` パラメータまたは `--master-keys` パラメータが必要です (両方ではない)。また、 を使用する `--encrypt` コマンド AWS KMS keys には、 `--wrapping-keys`パラメータまたは `--master-keys`パラメータ (両方ではない) が必要です。

  を使用する`--decrypt`コマンドでは AWS KMS keys、 `--wrapping-keys`パラメータはオプションですが、バージョン 2.1.x で必須であるため、推奨されます。使用する場合は、key 属性を指定するか、discovery 属性を `true` にする必要があります (両方ではない)。
- `--commitment-policy` パラメータが追加されます。唯一の有効な値は `forbid-encrypt-allow-decrypt` です。`forbid-encrypt-allow-decrypt` コミットポリシーは、すべての暗号化と復号コマンドで使用されます。

バージョン 1.8.x では、`--wrapping-keys` パラメータを使用するときに、値が `forbid-encrypt-allow-decrypt` の `--commitment-policy` パラメータが必要です。値を明示的に設定すると、バージョン 2.1.x へのアップグレード時にコミットメントポリシーが自動的に `require-encrypt-require-decrypt` に変更されなくなります。

## Encryption CLI のバージョン AWS 2.1.x の変更

- `--master-keys` パラメータは削除されます。代わりに、`--wrapping-keys` パラメータを使用します。
- すべての暗号化コマンドと復号コマンドには、`--wrapping-keys` パラメータが必要です。key 属性を指定するか、discovery 属性を `true` にする必要があります (両方ではない)。
- `--commitment-policy` パラメータでは次の値がサポートされます。詳細については、「コミットメントポリシーの設定」を参照してください。
  - `forbid-encrypt-allow-decrypt`
  - `require-encrypt-allow-decrypt`
  - `require-encrypt-require decrypt` (デフォルト)
- バージョン 2.1.x では、`--commitment-policy` パラメータはオプションです。デフォルト値は `require-encrypt-require-decrypt` です。

## バージョン 1.9.x および 2.2.x での AWS Encryption CLI の変更

- `--decrypt-unsigned` パラメータが追加されます。詳細については、「バージョン 2.2.x」を参照してください。
- `--buffer` パラメータが追加されます。詳細については、「バージョン 2.2.x」を参照してください。
- `--max-encrypted-data-keys` パラメータが追加されます。詳細については、「暗号化されたデータキーの制限」を参照してください。

## Encryption CLI のバージョン AWS 3.0.x の変更

- AWS KMS マルチリージョンキーのサポートを追加しました。詳細については、「マルチリージョンの使用 AWS KMS keys」を参照してください。

# データキーキャッシュ

データキーキャッシュにより、キャッシュにデータキーおよび関連する暗号化マテリアルが保存されます。データを暗号化または復号すると、 はキャッシュ内の一致するデータキー AWS Encryption SDK を検索します。一致が見つかった場合、新しいデータキーを生成するのではなく、キャッシュされたデータキーを使用します。データキーキャッシュによりパフォーマンスが向上し、コストを削減します。また、アプリケーションの拡張の際、サービス制限内に収まるよう役立ちます。

以下の場合、アプリケーションはデータキーキャッシュからメリットを得られます。

- データキーを再利用できる場合。
- 多数のデータキーを生成する場合。
- 暗号化オペレーションが許容できないほど時間とコストがかかり、制限があり、リソースを多く使用する場合。

キャッシュにより、 AWS Key Management Service () などの暗号化サービスの使用を減らすことができますAWS KMS。AWS KMS の 1 秒あたりのリクエスト数の制限に達している場合、キャッシュが役に立ちます。アプリケーションは、キャッシュされたキーを使用して、 を呼び出す代わりにデータキーリクエストの一部を処理できます AWS KMS。(また、AWS サポートセンターにケースを作成してアカウントの制限を引き上げることができます。)

AWS Encryption SDK は、データキーキャッシュの作成と管理に役立ちます。キャッシュとやり取りがあり、設定したセキュリティのしきい値を適用するローカルキャッシュおよびキャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) を提供します。連携によって、これらのコンポーネントはシステムのセキュリティを維持しながら、データキーの再利用による効率の恩恵を受けるのに役立ちます。

データキーキャッシュは、 のオプション機能 AWS Encryption SDK であり、慎重に使用する必要があります。デフォルトでは、 は暗号化オペレーションごとに新しいデータキー AWS Encryption SDK を生成します。この手法では、暗号化のベストプラクティスをサポートしており、過剰なデータキーの再利用を防ぎます。一般的に、パフォーマンスの目標を満たすために必要な場合のみ、データキーキャッシュを使用します。次に、データキーキャッシュのセキュリティしきい値を使用して、コストとパフォーマンスの目標を満たすために必要なキャッシュの最小量を使用していることを確認します。

のバージョン 3.x は、キーリングインターフェイスではなく、レガシーマスターキープロバイダーインターフェイスを使用したキャッシュ CMM AWS Encryption SDK for Java のみをサポートしま

す。ただし、.NET AWS Encryption SDK 用 のバージョン 4.x、 のバージョン 3.x AWS Encryption SDK for Java、 のバージョン 4.x AWS Encryption SDK for Python、Rust AWS Encryption SDK 用 のバージョン 1.x、Go AWS Encryption SDK 用 のバージョン 0.1.x 以降では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングがサポートされています。 AWS KMS 階層キーリングで暗号化されたコンテンツは、 AWS KMS 階層キーリングでのみ復号できます。

これらセキュリティトレードオフの詳細については、 AWS セキュリティブログの「AWS Encryption SDK: How to Decide if Data Key Caching is Right for Your Application」を参照してください。

トピック


- データキーキャッシュを使用する方法
- キャッシュセキュリティのしきい値の設定
- データキーキャッシュの詳細
- データキーキャッシュの例


# データキーキャッシュを使用する方法

このトピックでは、アプリケーションのデータキーキャッシュを使用する方法について説明します。ここでは、プロセスを手順ごとに説明します。次に、文字列を暗号化するオペレーションでデータキーキャッシュを使用する簡単な例の中ですべての手順を結びつけます。

このセクションの例では、 AWS Encryption SDKのバージョン 2.0.x 以降の使用方法について説明します。前バージョンを使用する例については、使用しているプログラミング言語の GitHub リポジトリのリリースリストで、使用中のリリースを検索してください。

でデータキーキャッシュを使用するための完全でテスト済みの例については AWS Encryption SDK、以下を参照してください。

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript Browser: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

.NET 用AWS Encryption SDK はデータキーキャッシュをサポートしていません。

トピック


- データキーキャッシュを使用する: ステップバイステップ
- データキーキャッシュの例: 文字列を暗号化する


## データキーキャッシュを使用する: ステップバイステップ

これらの段階的な手順では、データキーキャッシュの実装に必要なコンポーネントを作成する方法を示しています。

- データキーキャッシュを作成します。これらの例では、 AWS Encryption SDK が提供するローカルキャッシュを使用します。キャッシュは、10 個のデータキーに制限されています。

C

```
// Cache capacity (maximum number of entries) is required
size_t cache_capacity = 10;
struct aws_allocator *allocator = aws_default_allocator();

struct aws_cryptosdk_materials_cache *cache =
 aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);
```

Java

次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java は、データキーキャッシュ CMM を廃止します。バージョン 3.x では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングを使用することもできます。

```
// Cache capacity (maximum number of entries) is required
int MAX_CACHE_SIZE = 10;

CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(MAX_CACHE_SIZE);
```

JavaScript Browser

```
const capacity = 10
```

```
const cache = getLocalCryptographicMaterialsCache(capacity)
```

JavaScript Node.js

```
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

Python

```
# Cache capacity (maximum number of entries) is required
MAX_CACHE_SIZE = 10

cache = aws_encryption_sdk.LocalCryptoMaterialsCache(MAX_CACHE_SIZE)
```

- マスターキープロバイダー (Java および Python) またはキーリング (C および JavaScript) を作成します。これらの例では、 AWS Key Management Service （ AWS KMS) マスターキープロバイダーまたは互換性のあるAWS KMS キーリングを使用します。

  C

  ```
  // Create an AWS KMS keyring
  //   The input is the Amazon Resource Name (ARN)
  //   of an AWS KMS key
  struct aws_cryptosdk_keyring *kms_keyring =
   Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_arn);
  ```

  Java

  次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java は、データキーキャッシュ CMM を廃止します。バージョン 3.x では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングを使用することもできます。

  ```
  // Create an AWS KMS master key provider
  //   The input is the Amazon Resource Name (ARN)
  //   of an AWS KMS key
  ```

```
MasterKeyProvider<KmsMasterKey> keyProvider =
 KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn);
```

### JavaScript Browser

ブラウザでは、認証情報を安全に挿入する必要があります。この例では、ランタイムに認証情報を解決する Webpack (kms.webpack.config) の認証情報を定義します。 AWS KMS クライアントと認証情報から AWS KMS クライアントプロバイダーインスタンスを作成します。次に、キーリングを作成するときに、 AWS KMS key () とともにクライアントプロバイダーをコンストラクタに渡しますgeneratorKeyId)。

```
const { accessKeyId, secretAccessKey, sessionToken } = credentials

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

/*  Create an AWS KMS keyring
 *  You must configure the AWS KMS keyring with at least one AWS KMS key
*  The input is the Amazon Resource Name (ARN)
 */ of an AWS KMS key
const keyring = new KmsKeyringBrowser({
    clientProvider,
    generatorKeyId,
    keyIds,
  })
```

### JavaScript Node.js

```
/* Create an AWS KMS keyring
 *   The input is the Amazon Resource Name (ARN)
*/   of an AWS KMS key
const keyring = new KmsKeyringNode({ generatorKeyId })
```

### Python

```
# Create an AWS KMS master key provider
```

```
#  The input is the Amazon Resource Name (ARN)
#  of an AWS KMS key
key_provider =
 aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])
```

- キャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) を作成します。

  キャッシュ CMM をキャッシュとマスターキープロバイダーまたはキーリングに関連付けます。次に、キャッシュ CMM のキャッシュセキュリティのしきい値を設定します。

  C

  では AWS Encryption SDK for C、デフォルトの CMM などの基盤となる CMM またはキーリングからキャッシュ CMM を作成できます。この例では、キーリングからキャッシュ CMM を作成します。

  キャッシュ CMM を作成したら、キーリングおよびキャッシュへのリファレンスを解放できます。詳細については、「the section called “参照カウント”」を参照してください。

  ```
  // Create the caching CMM
  //   Set the partition ID to NULL.
  //   Set the required maximum age value to 60 seconds.
  struct aws_cryptosdk_cmm *caching_cmm =
   aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL,
   60, AWS_TIMESTAMP_SECS);

  // Add an optional message threshold
  //   The cached data key will not be used for more than 10 messages.
  aws_status = aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, 10);

  // Release your references to the cache and the keyring.
  aws_cryptosdk_materials_cache_release(cache);
  aws_cryptosdk_keyring_release(kms_keyring);
  ```

### Java

次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java はデータキーキャッシュをサポートしていませんが、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングをサポートしています。

```
/*
 * Security thresholds
 *   Max entry age is required.
 *   Max messages (and max bytes) per entry are optional
 */
int MAX_ENTRY_AGE_SECONDS = 60;
int MAX_ENTRY_MSGS = 10;

//Create a caching CMM
CryptoMaterialsManager cachingCmm =
    CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                                 .withCache(cache)
                                 .withMaxAge(MAX_ENTRY_AGE_SECONDS,
 TimeUnit.SECONDS)
                                 .withMessageUseLimit(MAX_ENTRY_MSGS)
                                 .build();
```

### JavaScript Browser

```
/*
 * Security thresholds
 *   Max age (in milliseconds) is required.
 *   Max messages (and max bytes) per entry are optional.
 */
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring  */
const cachingCmm = new WebCryptoCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
})
```

JavaScript Node.js

```
/*
 * Security thresholds
 *   Max age (in milliseconds) is required.
 *   Max messages (and max bytes) per entry are optional.
 */
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring  */
const cachingCmm = new NodeCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
})
```

Python

```
# Security thresholds
#   Max entry age is required.
#   Max messages (and max bytes) per entry are optional
#
MAX_ENTRY_AGE_SECONDS = 60.0
MAX_ENTRY_MESSAGES = 10

# Create a caching CMM
caching_cmm = CachingCryptoMaterialsManager(
    master_key_provider=key_provider,
    cache=cache,
    max_age=MAX_ENTRY_AGE_SECONDS,
    max_messages_encrypted=MAX_ENTRY_MESSAGES
)
```

必要な作業はこれだけです。次に、 にキャッシュ AWS Encryption SDK を管理させるか、独自のキャッシュ管理ロジックを追加します。

呼び出しでデータキーキャッシュを使用してデータを暗号化または復号するためには、マスターキープロバイダーやその他の CMM の代わりにキャッシュ CMM を指定します。

> **Note**
>
> データストリーム、または不明なサイズのデータを暗号化している場合、必ずリクエストでデータサイズを指定してください。 AWS Encryption SDK は、サイズが不明なデータを暗号化するときにデータキーキャッシュを使用しません。

C

では AWS Encryption SDK for C、キャッシュ CMM を使用してセッションを作成し、セッションを処理します。

デフォルトでは、メッセージサイズが不明で無制限の場合、 AWS Encryption SDK はデータキーをキャッシュしません。正確なデータサイズが不明な場合にキャッシュを許可するには、aws_cryptosdk_session_set_message_bound メソッドを使用してメッセージに最大サイズを設定します。推定メッセージサイズよりも大きい境界を設定します。実際のメッセージサイズが制限を超えると、暗号化オペレーションは失敗します。

```
/* Create a session with the caching CMM. Set the session mode to encrypt. */
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_cmm_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 caching_cmm);

/* Set a message bound of 1000 bytes */
aws_status = aws_cryptosdk_session_set_message_bound(session, 1000);

/* Encrypt the message using the session with the caching CMM */
aws_status = aws_cryptosdk_session_process(
              session, output_buffer, output_capacity, &output_produced,
 input_buffer, input_len, &input_consumed);

/* Release your references to the caching CMM and the session. */
aws_cryptosdk_cmm_release(caching_cmm);
aws_cryptosdk_session_destroy(session);
```

Java

次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java は、データキーキャッシュ CMM を廃止します。バージョン 3.x では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングを使用することもできます。

```
// When the call to encryptData specifies a caching CMM,
// the encryption operation uses the data key cache
final AwsCrypto encryptionSdk = AwsCrypto.standard();
return encryptionSdk.encryptData(cachingCmm, plaintext_source).getResult();
```

JavaScript Browser

```
const { result } = await encrypt(cachingCmm, plaintext)
```

JavaScript Node.js

AWS Encryption SDK for JavaScript for Node.js でキャッシュ CMM を使用する場合、`encrypt`メソッドにはプレーンテキストの長さが必要です。これを指定しないと、データキーはキャッシュされません。長さを指定しても、指定した平文データがその長さを超えると、暗号化オペレーションは失敗します。データのストリーミング時など、平文の正確な長さがわからない場合は、期待される最大の値を指定します。

```
const { result } = await encrypt(cachingCmm, plaintext, { plaintextLength:
 plaintext.length })
```

Python

```
# Set up an encryption client
client = aws_encryption_sdk.EncryptionSDKClient()

# When the call to encrypt specifies a caching CMM,
# the encryption operation uses the data key cache
#
encrypted_message, header = client.encrypt(
    source=plaintext_source,
    materials_manager=caching_cmm
)
```

## データキーキャッシュの例: 文字列を暗号化する

このシンプルなコード例では、文字列を暗号化するときにデータキーキャッシュを使用します。段階的な手順のコードを実行可能なテストコードに組み合わせます。

この例では、ローカルキャッシュとマスターキープロバイダーまたはキーリングを AWS KMS keyのために作成します。次に、ローカルキャッシュとマスターキープロバイダーまたはキーリングを使用

して、適切なセキュリティしきい値でキャッシュ CMM を作成します。Java および Python では、暗号化リクエストは、キャッシュ CMM、暗号化する平文のデータ、および暗号化コンテキストを指定します。Cでは、キャッシュ CMMがセッションで指定され、セッションが暗号化要求に提供されます。

これらの例を実行するには、AWS KMS keyの Amazon リソースネーム (ARN) を指定する必要があります。データキーを生成するために、AWS KMS keyを使用するためのアクセス許可 があるかどうか必ず確認してください。

データキーキャッシュの作成と使用に関するさらに詳細な実例については、「データキーキャッシュのコード例」を参照してください。

C

```
/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use
 * this file except in compliance with the License. A copy of the License is
 * located at
 *
 *     http://aws.amazon.com/apache2.0/
 *
 * or in the "license" file accompanying this file. This file is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 * implied. See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <aws/cryptosdk/cache.h>
#include <aws/cryptosdk/cpp/kms_keyring.h>
#include <aws/cryptosdk/session.h>

void encrypt_with_caching(
    uint8_t *ciphertext,     // output will go here (assumes ciphertext_capacity
 bytes already allocated)
    size_t *ciphertext_len,  // length of output will go here
    size_t ciphertext_capacity,
    const char *kms_key_arn,
    int max_entry_age,
    int cache_capacity) {
    const uint64_t MAX_ENTRY_MSGS = 100;
```

```
   struct aws_allocator *allocator = aws_default_allocator();

   // Load error strings for debugging
   aws_cryptosdk_load_error_strings();

   // Create a keyring
   struct aws_cryptosdk_keyring *kms_keyring =
Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_arn);

   // Create a cache
   struct aws_cryptosdk_materials_cache *cache =
aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);

   // Create a caching CMM
   struct aws_cryptosdk_cmm *caching_cmm =
aws_cryptosdk_caching_cmm_new_from_keyring(
       allocator, cache, kms_keyring, NULL, max_entry_age, AWS_TIMESTAMP_SECS);
   if (!caching_cmm) abort();

   if (aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, MAX_ENTRY_MSGS))
abort();

   // Create a session
   struct aws_cryptosdk_session *session =
       aws_cryptosdk_session_new_from_cmm_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
caching_cmm);
   if (!session) abort();

   // Encryption context
   struct aws_hash_table *enc_ctx =
aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);
   if (!enc_ctx) abort();
   AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key, "purpose");
   AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value, "test");
   if (aws_hash_table_put(enc_ctx, enc_ctx_key, (void *)enc_ctx_value, NULL))
abort();

   // Plaintext data to be encrypted
   const char *my_data = "My plaintext data";
   size_t my_data_len  = strlen(my_data);
   if (aws_cryptosdk_session_set_message_size(session, my_data_len)) abort();

   // When the session uses a caching CMM, the encryption operation uses the data
key cache
```

```
    // specified in the caching CMM.
    size_t bytes_read;
    if (aws_cryptosdk_session_process(
            session,
            ciphertext,
            ciphertext_capacity,
            ciphertext_len,
            (const uint8_t *)my_data,
            my_data_len,
            &bytes_read))
        abort();
    if (!aws_cryptosdk_session_is_done(session) || bytes_read != my_data_len)
 abort();

    aws_cryptosdk_session_destroy(session);
    aws_cryptosdk_cmm_release(caching_cmm);
    aws_cryptosdk_materials_cache_release(cache);
    aws_cryptosdk_keyring_release(kms_keyring);
}
```

Java

次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java は、データキーキャッシュ CMM を廃止します。バージョン 3.x では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングを使用することもできます。

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.examples;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoMaterialsManager;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.CryptoMaterialsCache;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import java.nio.charset.StandardCharsets;
import java.util.Collections;
import java.util.Map;
```

```
import java.util.concurrent.TimeUnit;

/**
 * <p>
 * Encrypts a string using an &KMS; key and data key caching
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>KMS Key ARN: To find the Amazon Resource Name of your &KMS; key,
 *     see 'Find the key ID and ARN' at https://docs.aws.amazon.com/kms/latest/
developerguide/find-cmk-id-arn.html
 * <li>Max entry age: Maximum time (in seconds) that a cached entry can be used
 * <li>Cache capacity: Maximum number of entries in the cache
 * </ol>
 */
public class SimpleDataKeyCachingExample {

    /*
     * Security thresholds
     *   Max entry age is required.
     *   Max messages (and max bytes) per data key are optional
     */
    private static final int MAX_ENTRY_MSGS = 100;

    public static byte[] encryptWithCaching(String kmsKeyArn, int maxEntryAge, int
 cacheCapacity) {
        // Plaintext data to be encrypted
        byte[] myData = "My plaintext data".getBytes(StandardCharsets.UTF_8);

        // Encryption context
        // Most encrypted data should have an associated encryption context
        // to protect integrity. This sample uses placeholder values.
        // For more information see:
        // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-
Integrity-of-Your-Encrypted-Data-by-Using-AWS-Key-Management
        final Map<String, String> encryptionContext =
 Collections.singletonMap("purpose", "test");

        // Create a master key provider
        MasterKeyProvider<KmsMasterKey> keyProvider =
 KmsMasterKeyProvider.builder()
            .buildStrict(kmsKeyArn);
```

```
        // Create a cache
        CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(cacheCapacity);

        // Create a caching CMM
        CryptoMaterialsManager cachingCmm =

 CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                .withCache(cache)
                .withMaxAge(maxEntryAge, TimeUnit.SECONDS)
                .withMessageUseLimit(MAX_ENTRY_MSGS)
                .build();

        // When the call to encryptData specifies a caching CMM,
        // the encryption operation uses the data key cache
        final AwsCrypto encryptionSdk = AwsCrypto.standard();
        return encryptionSdk.encryptData(cachingCmm, myData,
 encryptionContext).getResult();
    }
}
```

### JavaScript Browser

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

/* This is a simple example of using a caching CMM with a KMS keyring
 * to encrypt and decrypt using the AWS Encryption SDK for Javascript in a browser.
 */

import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
  WebCryptoCachingMaterialsManager,
  getLocalCryptographicMaterialsCache,
} from '@aws-crypto/client-browser'
import { toBase64 } from '@aws-sdk/util-base64-browser'

/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
 * which enforces that this client only encrypts using committing algorithm suites
```

```
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* This is injected by webpack.
 * The webpack.DefinePlugin or @aws-sdk/karma-credential-loader will replace the
 values when bundling.
 * The credential values are pulled from @aws-sdk/credential-provider-node
 * Use any method you like to get credentials into the browser.
 * See kms.webpack.config
 */
declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}

/* This is done to facilitate testing. */
export async function testCachingCMMExample() {
  /* This example uses an &KMS; keyring. The generator key in a &KMS; keyring
 generates and encrypts the data key.
   * The caller needs kms:GenerateDataKey permission on the &KMS; key in
 generatorKeyId.
   */
  const generatorKeyId =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* Adding additional KMS keys that can decrypt.
   * The caller must have kms:Encrypt permission for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
   * In `generatorId`, this &KMS; key is identified by its alias ARN.
   * In `keyIds`, this &KMS; key is identified by its key ARN.
   * In practice, you would specify different &KMS; keys,
   * or omit the `keyIds` parameter.
   * This is *only* to demonstrate how the &KMS; key ARNs are configured.
```

```
  */
 const keyIds = [
   'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
 ]

 /* Need a client provider that will inject correct credentials.
  * The credentials here are injected by webpack from your environment bundle is
created
  * The credential values are pulled using @aws-sdk/credential-provider-node.
  * See kms.webpack.config
  * You should inject your credential into the browser in a secure manner
  * that works with your application.
  */
 const { accessKeyId, secretAccessKey, sessionToken } = credentials

 /* getClient takes a KMS client constructor
  * and optional configuration values.
  * The credentials can be injected here,
  * because browsers do not have a standard credential discovery process the way
Node.js does.
  */
 const clientProvider = getClient(KMS, {
   credentials: {
     accessKeyId,
     secretAccessKey,
     sessionToken,
   },
 })

 /* You must configure the KMS keyring with your &KMS; keys */
 const keyring = new KmsKeyringBrowser({
   clientProvider,
   generatorKeyId,
   keyIds,
 })

 /* Create a cache to hold the data keys (and related cryptographic material).
  * This example uses the local cache provided by the Encryption SDK.
  * The `capacity` value represents the maximum number of entries
  * that the cache can hold.
  * To make room for an additional entry,
  * the cache evicts the oldest cached entry.
  * Both encrypt and decrypt requests count independently towards this threshold.
  * Entries that exceed any cache threshold are actively removed from the cache.
```

```
  * By default, the SDK checks one item in the cache every 60 seconds (60,000
milliseconds).
  * To change this frequency, pass in a `proactiveFrequency` value
  * as the second parameter. This value is in milliseconds.
  */
 const capacity = 100
 const cache = getLocalCryptographicMaterialsCache(capacity)

 /* The partition name lets multiple caching CMMs share the same local
cryptographic cache.
  * By default, the entries for each CMM are cached separately. However, if you
want these CMMs to share the cache,
  * use the same partition name for both caching CMMs.
  * If you don't supply a partition name, the Encryption SDK generates a random
name for each caching CMM.
  * As a result, sharing elements in the cache MUST be an intentional operation.
  */
 const partition = 'local partition name'

 /* maxAge is the time in milliseconds that an entry will be cached.
  * Elements are actively removed from the cache.
  */
 const maxAge = 1000 * 60

 /* The maximum number of bytes that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest practical value.
  */
 const maxBytesEncrypted = 100

 /* The maximum number of messages that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest practical value.
  */
 const maxMessagesEncrypted = 10

 const cachingCMM = new WebCryptoCachingMaterialsManager({
   backingMaterials: keyring,
   cache,
   partition,
   maxAge,
   maxBytesEncrypted,
   maxMessagesEncrypted,
 })
```

```
  /* Encryption context is a *very* powerful tool for controlling
   * and managing access.
   * When you pass an encryption context to the encrypt function,
   * the encryption context is cryptographically bound to the ciphertext.
   * If you don't pass in the same encryption context when decrypting,
   * the decrypt function fails.
   * The encryption context is ***not*** secret!
   * Encrypted data is opaque.
   * You can use an encryption context to assert things about the encrypted data.
   * The encryption context helps you to determine
   * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
   * For example, if you are are only expecting data from 'us-west-2',
   * the appearance of a different AWS Region in the encryption context can indicate
 malicious interference.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/
concepts.html#encryption-context
   *
   * Also, cached data keys are reused ***only*** when the encryption contexts
 passed into the functions are an exact case-sensitive match.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
caching-details.html#caching-encryption-context
   */
  const encryptionContext = {
    stage: 'demo',
    purpose: 'simple demonstration app',
    origin: 'us-west-2',
  }

  /* Find data to encrypt. */
  const plainText = new Uint8Array([1, 2, 3, 4, 5])

  /* Encrypt the data.
   * The caching CMM only reuses data keys
   * when it know the length (or an estimate) of the plaintext.
   * However, in the browser,
   * you must provide all of the plaintext to the encrypt function.
   * Therefore, the encrypt function in the browser knows the length of the
 plaintext
   * and does not accept a plaintextLength option.
   */
  const { result } = await encrypt(cachingCMM, plainText, { encryptionContext })

  /* Log the plain text
```

```
  * only for testing and to show that it works.
  */
 console.log('plainText:', plainText)
 document.write('</br>plainText:' + plainText + '</br>')

 /* Log the base64-encoded result
  * so that you can try decrypting it with another AWS Encryption SDK
implementation.
  */
 const resultBase64 = toBase64(result)
 console.log(resultBase64)
 document.write(resultBase64)

 /* Decrypt the data.
  * NOTE: This decrypt request will not use the data key
  * that was cached during the encrypt operation.
  * Data keys for encrypt and decrypt operations are cached separately.
  */
 const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

 /* Grab the encryption context so you can verify it. */
 const { encryptionContext: decryptedContext } = messageHeader

 /* Verify the encryption context.
  * If you use an algorithm suite with signing,
  * the Encryption SDK adds a name-value pair to the encryption context that
contains the public key.
  * Because the encryption context might contain additional key-value pairs,
  * do not include a test that requires that all key-value pairs match.
  * Instead, verify that the key-value pairs that you supplied to the `encrypt`
function are included in the encryption context that the `decrypt` function
returns.
  */
 Object.entries(encryptionContext).forEach(([key, value]) => {
   if (decryptedContext[key] !== value)
     throw new Error('Encryption Context does not match expected values')
 })

 /* Log the clear message
  * only for testing and to show that it works.
  */
 document.write('</br>Decrypted:' + plaintext)
 console.log(plaintext)
```

```
  /* Return the values to make testing easy. */
  return { plainText, plaintext }
}
```

### JavaScript Node.js

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

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

/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
 * which enforces that this client only encrypts using committing algorithm suites
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

export async function cachingCMMNodeSimpleTest() {
  /* An &KMS; key is required to generate the data key.
   * You need kms:GenerateDataKey permission on the &KMS; key in generatorKeyId.
   */
  const generatorKeyId =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* Adding alternate &KMS; keys that can decrypt.
   * Access to kms:Encrypt is required for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
```

```
  * In `generatorId`, this &KMS; key is identified by its alias ARN.
  * In `keyIds`, this &KMS; key is identified by its key ARN.
  * In practice, you would specify different &KMS; keys,
  * or omit the `keyIds` parameter.
  * This is *only* to demonstrate how the &KMS; key ARNs are configured.
  */
 const keyIds = [
   'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
 ]

 /* The &KMS; keyring must be configured with the desired &KMS; keys
  * This example passes the keyring to the caching CMM
  * instead of using it directly.
  */
 const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

 /* Create a cache to hold the data keys (and related cryptographic material).
  * This example uses the local cache provided by the Encryption SDK.
  * The `capacity` value represents the maximum number of entries
  * that the cache can hold.
  * To make room for an additional entry,
  * the cache evicts the oldest cached entry.
  * Both encrypt and decrypt requests count independently towards this threshold.
  * Entries that exceed any cache threshold are actively removed from the cache.
  * By default, the SDK checks one item in the cache every 60 seconds (60,000
milliseconds).
  * To change this frequency, pass in a `proactiveFrequency` value
  * as the second parameter. This value is in milliseconds.
  */
 const capacity = 100
 const cache = getLocalCryptographicMaterialsCache(capacity)

 /* The partition name lets multiple caching CMMs share the same local
cryptographic cache.
  * By default, the entries for each CMM are cached separately. However, if you
want these CMMs to share the cache,
  * use the same partition name for both caching CMMs.
  * If you don't supply a partition name, the Encryption SDK generates a random
name for each caching CMM.
  * As a result, sharing elements in the cache MUST be an intentional operation.
  */
 const partition = 'local partition name'

 /* maxAge is the time in milliseconds that an entry will be cached.
```

```
   * Elements are actively removed from the cache.
   */
  const maxAge = 1000 * 60

  /* The maximum amount of bytes that will be encrypted under a single data key.
   * This value is optional,
   * but you should configure the lowest value possible.
   */
  const maxBytesEncrypted = 100

  /* The maximum number of messages that will be encrypted under a single data key.
   * This value is optional,
   * but you should configure the lowest value possible.
   */
  const maxMessagesEncrypted = 10

  const cachingCMM = new NodeCachingMaterialsManager({
    backingMaterials: keyring,
    cache,
    partition,
    maxAge,
    maxBytesEncrypted,
    maxMessagesEncrypted,
  })

  /* Encryption context is a *very* powerful tool for controlling
   * and managing access.
   * When you pass an encryption context to the encrypt function,
   * the encryption context is cryptographically bound to the ciphertext.
   * If you don't pass in the same encryption context when decrypting,
   * the decrypt function fails.
   * The encryption context is ***not*** secret!
   * Encrypted data is opaque.
   * You can use an encryption context to assert things about the encrypted data.
   * The encryption context helps you to determine
   * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
   * For example, if you are are only expecting data from 'us-west-2',
   * the appearance of a different AWS Region in the encryption context can indicate
 malicious interference.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/
concepts.html#encryption-context
   *
   * Also, cached data keys are reused ***only*** when the encryption contexts
 passed into the functions are an exact case-sensitive match.
```

```
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
caching-details.html#caching-encryption-context
   */
  const encryptionContext = {
    stage: 'demo',
    purpose: 'simple demonstration app',
    origin: 'us-west-2',
  }

  /* Find data to encrypt.  A simple string. */
  const cleartext = 'asdf'

  /* Encrypt the data.
   * The caching CMM only reuses data keys
   * when it know the length (or an estimate) of the plaintext.
   * If you do not know the length,
   * because the data is a stream
   * provide an estimate of the largest expected value.
   *
   * If your estimate is smaller than the actual plaintext length
   * the AWS Encryption SDK will throw an exception.
   *
   * If the plaintext is not a stream,
   * the AWS Encryption SDK uses the actual plaintext length
   * instead of any length you provide.
   */
  const { result } = await encrypt(cachingCMM, cleartext, {
    encryptionContext,
    plaintextLength: 4,
  })

  /* Decrypt the data.
   * NOTE: This decrypt request will not use the data key
   * that was cached during the encrypt operation.
   * Data keys for encrypt and decrypt operations are cached separately.
   */
  const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

  /* Grab the encryption context so you can verify it. */
  const { encryptionContext: decryptedContext } = messageHeader

  /* Verify the encryption context.
   * If you use an algorithm suite with signing,
```

```
   * the Encryption SDK adds a name-value pair to the encryption context that
 contains the public key.
   * Because the encryption context might contain additional key-value pairs,
   * do not include a test that requires that all key-value pairs match.
   * Instead, verify that the key-value pairs that you supplied to the `encrypt`
 function are included in the encryption context that the `decrypt` function
 returns.
   */
  Object.entries(encryptionContext).forEach(([key, value]) => {
    if (decryptedContext[key] !== value)
      throw new Error('Encryption Context does not match expected values')
  })

  /* Return the values so the code can be tested. */
  return { plaintext, result, cleartext, messageHeader }
}
```

### Python

```
# Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"). You
# may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# or in the "license" file accompanying this file. This file is
# distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF
# ANY KIND, either express or implied. See the License for the specific
# language governing permissions and limitations under the License.
"""Example of encryption with data key caching."""
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_with_caching(kms_key_arn, max_age_in_cache, cache_capacity):
    """Encrypts a string using an &KMS; key and data key caching.

    :param str kms_key_arn: Amazon Resource Name (ARN) of the &KMS; key
    :param float max_age_in_cache: Maximum time in seconds that a cached entry can
 be used
    :param int cache_capacity: Maximum number of entries to retain in cache at once
```

```
    """
    # Data to be encrypted
    my_data = "My plaintext data"

    # Security thresholds
    #   Max messages (or max bytes per) data key are optional
    MAX_ENTRY_MESSAGES = 100

    # Create an encryption context
    encryption_context = {"purpose": "test"}

    # Set up an encryption client with an explicit commitment policy. Note that if
you do not explicitly choose a
    # commitment policy, REQUIRE_ENCRYPT_REQUIRE_DECRYPT is used by default.
    client =
aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R

    # Create a master key provider for the &KMS; key
    key_provider =
aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])

    # Create a local cache
    cache = aws_encryption_sdk.LocalCryptoMaterialsCache(cache_capacity)

    # Create a caching CMM
    caching_cmm = aws_encryption_sdk.CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=max_age_in_cache,
        max_messages_encrypted=MAX_ENTRY_MESSAGES,
    )

    # When the call to encrypt data specifies a caching CMM,
    # the encryption operation uses the data key cache specified
    # in the caching CMM
    encrypted_message, _header = client.encrypt(
        source=my_data, materials_manager=caching_cmm,
encryption_context=encryption_context
    )

    return encrypted_message
```

# キャッシュセキュリティのしきい値の設定

データキーキャッシュを実装するときは、キャッシュ CMM が実施するセキュリティのしきい値を設定する必要があります。

セキュリティのしきい値は、各キャッシュデータキーの使用期間および各データキーで保護されるデータ量を制限するのに役立ちます。キャッシュ CMM は、キャッシュエントリがすべてのセキュリティしきい値に準拠している場合にのみ、キャッシュされたデータキーを返します。キャッシュエントリがしきい値を超えた場合、そのエントリは現在のオペレーションには使用されず、キャッシュから速やかに削除されます。各データキーの最初の使用 (キャッシュ前) は、これらのしきい値から除外されます。

通常、コストとパフォーマンスの目標を満たすために必要なキャッシュの最小量を使用します。

は、キー取得関数を使用して暗号化されたデータキー AWS Encryption SDK のみをキャッシュします。また、一部のしきい値の上限も確立します。これらの制限は暗号化の制限を超えてデータキーが再利用されないことを確認します。ただし、プレーンテキストデータキーはキャッシュされるため (デフォルトではメモリに)、キーが保存される期間を最小限にしてください。また、キーが侵害された場合に公開される可能性のあるデータを制限してください。

キャッシュセキュリティしきい値の設定例については、 AWS セキュリティブログのAWS Encryption SDK「: データキーキャッシュがアプリケーションに適しているかどうかを判断する方法」を参照してください。

> **Note**
>
> キャッシュ CMM には以下のすべてのしきい値が適用されます。オプションの値を指定していない場合、キャッシュ CMM はデフォルト値を使用します。
> データキーキャッシュを一時的に無効にするために、 AWS Encryption SDK の Java/Python 実装では、null の暗号化マテリアルキャッシュ (null キャッシュ) が用意されています。null キャッシュは、すべての GET リクエストのミスを返し、PUT リクエストに応答しません。キャッシュ容量またはセキュリティのしきい値を 0 に設定するのではなく、null キャッシュを使用することをお勧めします。詳細については、「Java」および「Python」の null キャッシュを参照してください。

最大期限 (必須)

キャッシュされたエントリが、追加された時点から使用できる時間を決定します。この値は必須です。0 より大きい値を入力します。 AWS Encryption SDK は最大経過時間の値を制限しません。

のすべての言語実装では、ミリ秒を使用する を除き AWS Encryption SDK for JavaScript、最大経過時間を秒単位で AWS Encryption SDK 定義します。

アプリケーションがキャッシュのメリットを得られる最短の間隔を使用します。最大期限しきい値をキーローテーションポリシーのように使用できます。それを使用してデータキーの再利用を制限し、暗号化マテリアルの公開を最小限に抑え、キャッシュされている間にポリシーが変更された可能性のあるデータキーを排除します。

暗号化されたメッセージの最大数 (オプション)

キャッシュされたデータキーが暗号化できるメッセージの最大数を指定します。この値はオプションです。1 ~ 2^32 の間のメッセージの値を入力します。デフォルト値は 2^32 メッセージです。

各キャッシュされたキーによって保護されるメッセージの数を、再利用の値を取得するのに十分な大きさに、しかし、キーが侵害された場合に公開される可能性のあるメッセージの数を制限できるほど小さく設定します。

暗号化されたバイトの最大数 (オプション)

キャッシュされたデータキーが暗号化できるバイトの最大数を指定します。この値はオプションです。0 ~ 2^63 - 1 の間の値を入力します。デフォルト値は 2^63 - 1 です。値を 0 に指定することで、空のメッセージ文字列を暗号化している場合にのみデータキーキャッシュを使用できます。

現在のリクエストのバイト数がこのしきい値を評価する際に含まれます。処理されたバイト数と現在のバイト数がしきい値を超えている場合、キャッシュされたデータキーは、より小さいリクエストで使用されたとしても、キャッシュから削除されます。

# データキーキャッシュの詳細

ほとんどのアプリケーションで、カスタムコードを記述することなくデータキーキャッシュのデフォルトの実装を使用できます。このセクションでは、デフォルトの実装とオプションの詳細について説明します。

トピック


- データキーキャッシュの仕組み
- 暗号化マテリアルキャッシュの作成
- キャッシュ暗号化マテリアルマネージャーの作成
- データキーキャッシュエントリとは
- 暗号化コンテキスト: キャッシュエントリを選択する方法
- アプリケーションはキャッシュされたデータキーを使用していますか？


## データキーキャッシュの仕組み

データキーキャッシュをリクエストで使用してデータを暗号化または復号すると、 AWS Encryption SDK はまずリクエストに一致するデータキーのキャッシュを検索します。有効な一致が見つかった場合、キャッシュされたデータキーを使用してデータを暗号化します。それ以外の場合は、新しいデータキーを生成します。キャッシュがない場合も同じ動作になります。

ストリームデータなど、不明なサイズのデータにデータキーキャッシュは使用されません。これにより、キャッシュ CMM が最大バイトしきい値を正しく適用できるようになります。この動作を避けるには、暗号化リクエストにメッセージサイズを追加します。

キャッシュに加えて、データキーキャッシュではキャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) が使用されます。キャッシュ CMM は、キャッシュおよび基盤となる CMM とのやり取りに特化した暗号化マテリアルマネージャー (CMM) です。(マスターキープロバイダーまたはキーリングを指定すると、 AWS Encryption SDK はデフォルトの CMM を自動で作成します。) キャッシュ CMM は、基盤となる CMM が返すデータキーをキャッシュします。また、キャッシュ CMM は、ユーザーが設定したキャッシュセキュリティしきい値を適用します。

キャッシュで誤ったデータキーが選択されないように、互換性のあるすべてのキャッシュ CMM では、キャッシュされた暗号化マテリアルの次のプロパティがマテリアルリクエストと一致している必要があります。

- アルゴリズムスイート
- 暗号化コンテキスト (空の場合も含む)
- パーティション名 (キャッシュ CMM を識別する文字列)
- (説明のみ) 暗号化されたデータキー

> **Note**
>
> は、アルゴリズムスイートがキー取得関数を使用する場合にのみデータキーを AWS Encryption SDK キャッシュします。

次のワークフローでは、データを暗号化するリクエストがデータキーキャッシュがある場合とない場合にどのように処理されるかを示します。キャッシュおよびキャッシュ CMM を含む、ユーザーが作成したキャッシュコンポーネントがプロセスの中でどのように使用されるかが示されます。

### キャッシュを使用しないでデータを暗号化する

キャッシュせずに暗号化マテリアルを取得するには:

1. アプリケーションは AWS Encryption SDK にデータの暗号化を要求します。

   リクエストは、マスターキープロバイダーまたはキーリングを指定します。 AWS Encryption SDK は、マスターキープロバイダーまたはキーリングとやり取りするデフォルト CMM を作成します。
2. AWS Encryption SDK は CMM に暗号化マテリアルを要求します (暗号化マテリアルを取得します )。
3. CMM によって、暗号化マテリアルのキーリング (C および JavaScript)、またはマスターキープロバイダー (Java および Python) を求められます。これには、 AWS Key Management Service () 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AWS KMS。CMM より AWS Encryption SDKに暗号化マテリアルが返ります。
4. AWS 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ます。また、暗号化されたデータキーと復号されたデータキーは暗号化されたメッセージに保存され、ユーザーに返ります。

### キャッシュを使用してデータを暗号化する

データキーキャッシュを使用して暗号化マテリアルを取得するには:

1. アプリケーションは AWS Encryption SDK にデータの暗号化を要求します。

   このリクエストは、基盤となる暗号化マテリアルマネージャー (CMM) と関連付けられているキャッシュ暗号化マテリアルマネージャー (キャッシュCMM) を指定します。マスターキープロ

バイダーまたはキーリングを指定すると、 AWS Encryption SDK はデフォルトの CMM を自動で作成します。

2. SDK によって、指定したキャッシュ CMM に対して暗号化マテリアルが求められます。
3. キャッシュ CMM は、キャッシュの暗号化マテリアルをリクエストします。
   a. キャッシュで一致が見つかった場合、期間が更新され、一致したキャッシュエントリの値を使用して、キャッシュされた暗号化マテリアルをキャッシュ CMM に返します。

      キャッシュエントリがセキュリティしきい値に準拠している場合、キャッシュ CMM はそれを SDK に返します。それ以外の場合は、一致がなかったものとしてキャッシュがエントリを削除し続行するように指示します。

   b. キャッシュで有効な一致が見つからない場合、キャッシュ CMM は基盤となる CMM に新しいデータキーを生成するよう要求します。

      基盤となる CMM は、そのキーリング (C および JavaScript)、またはマスターキープロバイダー (Java および Python) から暗号化マテリアルを取得します。これには、 AWS Key Management Serviceなどのサービスの呼び出しが含まれる場合があります。基盤となる CMM は、データキーのプレーンテキストおよび暗号化されたコピーをキャッシュ CMM に返します。

      キャッシュ CMM は、新しい暗号化マテリアルをキャッシュに保存します。
4. キャッシュ CMM より AWS Encryption SDKに暗号化マテリアルが返ります。
5. AWS 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ます。また、暗号化されたデータキーと復号されたデータキーは暗号化されたメッセージに保存され、ユーザーに返ります。

## 暗号化マテリアルキャッシュの作成

は、データキーキャッシュで使用される暗号化マテリアルキャッシュの要件 AWS Encryption SDK を定義します。また、ローカルキャッシュを提供します。これは、設定可能なインメモリの最も長く使用されていない (LRU) キャッシュです。ローカルキャッシュのインスタンスを作成するには、Java および Python の `LocalCryptoMaterialsCache` コンストラクタ、JavaScript の getLocalCryptographicMaterialsCache 関数、または C の `aws_cryptosdk_materials_cache_local_new` コンストラクタを使用します。

ローカルキャッシュには、キャッシュされたエントリの追加、削除、一致とキャッシュの保持を含む基本的なキャッシュ管理のロジックが含まれます。カスタムキャッシュ管理ロジックを作成する必要はありません。ローカルキャッシュはそのまま使用したり、カスタマイズしたり、任意の互換性のあるキャッシュを置き換えたりすることができます。

ローカルキャッシュの作成時に、その容量 (キャッシュが保持できるエントリの最大数) を設定します。この設定は、データキーの再利用が制限されている際に効率的なキャッシュを設計するのに役立ちます。

AWS Encryption SDK for Java と は、null 暗号化マテリアルキャッシュ (NullCryptoMaterialsCache) AWS Encryption SDK for Python も提供します。NullCryptoMaterialsCache は、すべての GET オペレーションのミスを返し、PUT オペレーションには応答しません。NullCryptoMaterialsCache はテストで使用したり、キャッシュコードを含むアプリケーションで一時的にキャッシュを無効にしたりするために使用できます。

では AWS Encryption SDK、各暗号化マテリアルキャッシュはキャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) に関連付けられます。キャッシュ CMM は、キャッシュからデータキーを取得し、キャッシュにデータキーを格納して、設定したセキュリティしきい値を適用します。キャッシュ CMM 作成時に、使用するキャッシュ、基盤となる CMM またはキャッシュするデータキーを生成するマスターキープロバイダーを指定します。

## キャッシュ暗号化マテリアルマネージャーの作成

データキーキャッシュを有効にするには、キャッシュとキャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) を作成します。次に、データを暗号化または復号するリクエストで、標準的な暗号化マテリアルマネージャー (CMM) の代わりにキャッシュ CMM、またはマスターキープロバイダーかキーリングを指定します。

CMM には 2 つのタイプがあります。いずれもデータキー (および関連する暗号化マテリアル) を取得しますが、以下のようにさまざまな方法があります。

- CMM は、キーリング (C または JavaScript) またはマスターキープロバイダー (Java および Python) に関連付けられています。SDK より CMM に暗号化マテリアルまたは復号マテリアルが求められると、CMM はそのキーリングまたはマスターキープロバイダーからそのマテリアルを取得します。Java および Python では、CMM はマスターキーを使用して、データキーを生成、暗号化、または復号します。C および JavaScript では、キーリングは暗号化マテリアルを生成し、暗号化して返します。
- キャッシュ CMM は、ローカルキャッシュなどの 1 つのキャッシュ、および基盤となる CMM に関連付けられています。SDK がキャッシュ CMM に暗号化マテリアルを要求すると、キャッシュ

CMM はキャッシュからそれらを取得しようとします。一致が見つからない場合、キャッシュ CMM は、基盤となる CMM にマテリアルを要求します。次に、発信者に返す前に、新しい暗号化マテリアルをキャッシュします。

また、キャッシュ CMM は、ユーザーが各キャッシュエントリに設定したセキュリティしきい値を適用します。セキュリティしきい値はキャッシュ CMM で設定され適用されるため、キャッシュが機密性情報向けに設計されていなくても、互換性があるすべてのキャッシュが使用できます。

## データキーキャッシュエントリとは

データキーキャッシュにより、キャッシュにデータキーおよび関連する暗号化マテリアルが保存されます。各エントリには、以下に示す要素が含まれます。この情報は、データキーキャッシュ機能を使用するかどうかを決定するときや、キャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) でセキュリティしきい値を設定するときに役立ちます。

暗号化リクエストのキャッシュされたエントリ

暗号化オペレーションの結果としてデータキーキャッシュに追加されたエントリには、次の要素が含まれます。

- プレーンテキストのデータキー
- 暗号化されたデータキー (1 つ以上)
- 暗号化コンテキスト
- メッセージ署名キー (使用している場合)
- アルゴリズムスイート
- セキュリティしきい値を適用するための使用量カウンターを含む、メタデータ

復号リクエストのキャッシュされたエントリ

復号オペレーションの結果としてデータキーキャッシュに追加されたエントリには、次の要素が含まれます。

- プレーンテキストのデータキー
- 署名の検証キー (使用している場合)
- セキュリティしきい値を適用するための使用量カウンターを含む、メタデータ

# 暗号化コンテキスト: キャッシュエントリを選択する方法

任意のリクエストで暗号化リクエストを指定してデータを暗号化できます。ただし、暗号化コンテキストはデータキーキャッシュで特別な役割を果たします。データキーが同じキャッシュ CMM から発生している場合であっても、キャッシュ内でデータキーのサブグループを作成できます。

暗号化コンテキストは、任意のシークレットデータを含まない、一連のキーと値のペアです。暗号化中、暗号化コンテキストは暗号化されたデータに暗号化されてバインドされます。これにより、データ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。では AWS Encryption SDK、暗号化コンテキストは、暗号化されたデータとデータキーを使用して暗号化されたメッセージに保存されます。

データキーキャッシュを使用する場合、暗号化コンテキストを使用して、暗号化オペレーションに特定のキャッシュされたデータキーを選択することもできます。暗号化コンテキストは、データキー (キャッシュエントリ ID の一部) を使用してキャッシュエントリに保存されます。キャッシュされたデータキーは、その暗号化がコンテキストと一致する場合にのみ再利用されます。暗号化リクエストに特定のデータキーを再利用する場合、同じ暗号化コンテキストを指定します。これらのデータキーを回避するには、別の暗号化コンテキストを指定します。

暗号化コンテキストは常にオプションですが、推奨されています。リクエストで暗号化コンテキストを指定しない場合、空の暗号化コンテキストがキャッシュエントリ ID に含められ、各リクエストに照合されます。

## アプリケーションはキャッシュされたデータキーを使用していますか？

データキーキャッシュは、特定のアプリケーションやワークロードに対して非常に効果的な最適化戦略です。ただし、リスクが伴うため、状況にどれほど効果があるかを判断し、その利点がリスクを上回るかどうかを判断することが重要です。

データキーキャッシュはデータキーを再利用するため、最も明白な効果は、新しいデータキーを生成するための呼び出し回数を減らすことです。データキーキャッシュが実装されている場合、 は オペレーションを AWS Encryption SDK AWS KMS `GenerateDataKey`呼び出して、最初のデータキーとキャッシュが欠落したときのみを作成します。しかし、同じ暗号化コンテキストとアルゴリズムスイートを含む、同じ特性を持つ多数のデータキーを生成するアプリケーションでのみキャッシュのパフォーマンスが知覚的に向上します。

の実装 AWS Encryption SDK でキャッシュのデータキーが実際に使用されているかどうかを確認するには、次の方法を試してください。

- マスターキーインフラストラクチャのログで、新しいデータキーを作成する呼び出しの頻度を確認します。データキーキャッシュが有効な場合、新しいキーを作成するための呼び出しの数は目に見えて低下します。たとえば、 AWS KMS マスターキープロバイダーまたはキーリングを使用している場合は、GenerateDataKey 呼び出しの CloudTrail ログを検索します。
- さまざまな暗号化リクエストに応じて AWS Encryption SDK が返す暗号化されたメッセージを比較します。例えば、 を使用している場合は AWS Encryption SDK for Java、異なる暗号化呼び出しの ParsedCiphertext オブジェクトを比較します。 AWS Encryption SDK for JavaScriptでは、MessageHeader の `encryptedDataKeys` プロパティの内容を比較します。データキーを再利用すると、暗号化されたメッセージ内の暗号化されたデータキーは同じになります。

# データキーキャッシュの例

この例では、ローカルキャッシュでデータキーキャッシュを使用して、複数のデバイスによって生成されたデータが暗号化されて異なるリージョンに保存されるアプリケーションを高速化します。

この例では、複数のデータのプロデューサーがデータを作成して暗号化し、各リージョンの Kinesis ストリームに書き込みます。AWS Lambda 関数 (コンシューマー) はそのストリームを復号して、プレーンテキストのデータをそのリージョンの DynamoDB のテーブルに書き込みます。データプロデューサーおよびコンシューマーは、 AWS Encryption SDK と AWS KMS マスターキープロバイダーを使用します。KMS への呼び出しを減らすために、各プロデューサーおよびコンシューマーには独自のローカルキャッシュがあります。

これらの例のソースコードは Java と Python で用意されています。サンプルには、サンプルのリソースを定義する AWS CloudFormation テンプレートも含まれています。

## ローカルキャッシュの結果

以下の表は、ローカルキャッシュによって、この例の KMS への合計呼び出し回数 (1 秒あたり、リージョンあたり) が元の値の 1% まで減少していることを示しています。

プロデューサーリクエスト

| | 1 秒あたり、クライアントあたりのリクエスト | | | リージョンあたりのクライアント | 1 秒あたり、リージョンあたりの平均リクエスト |
|---|---|---|---|---|---|
| | データキーの生成 (us-west-2) | データキーの暗号化 (eu-central-1) | 合計 (リージョンあたり) | | |

| | | | | | |
|---|---|---|---|---|---|
| キャッシュなし | 1 | 1 | 1 | 500 | 500 |
| ローカルキャッシュ | 1 rps/100 を使用 | 1 rps/100 を使用 | 1 rps/100 を使用 | 500 | 5 |

コンシューマーリクエスト

| | 1 秒あたり、クライアントあたりのリクエスト | | | リージョンあたりのクライアント | 1 秒あたり、リージョンあたりの平均リクエスト |
|---|---|---|---|---|---|
| | データキーを復号 | プロデューサー | Total | | |
| キャッシュなし | 1 rps/プロデューサー | 500 | 500 | 2 | 1,000 |
| ローカルキャッシュ | 1 rps/プロデューサー/100 を使用 | 500 | 5 | 2 | 10 |

## データキーキャッシュのコード例

このコード例は、Java と Python でローカルキャッシュを使用するデータキーキャッシュの単純な実装を作成します。このコードは、ローカルキャッシュの 2 つのインスタンスを作成します。1 つはデータを暗号化するデータプロデューサー用で、もう 1 つはデータを復号するデータコンシューマー (AWS Lambda 関数) 用です。言語別のデータキーキャッシュの実装の詳細については、 AWS Encryption SDKの Javadoc および Python ドキュメントを参照してください。

データキーキャッシュは、 が AWS Encryption SDK サポートするすべてのプログラミング言語で使用できます。

でデータキーキャッシュを使用するための完全でテスト済みの例については AWS Encryption SDK、以下を参照してください。

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java

- JavaScript Browser: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

## プロデューサー

プロデューサーはマップを取得して JSON に変換し、 AWS Encryption SDK を使用して暗号化し、それぞれで暗号文レコードを Kinesis ストリームにプッシュします AWS リージョン。

このコードでは、キャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) を定義し、ローカルキャッシュおよび基盤となる AWS KMS マスターキープロバイダーと関連付けます。キャッシュ CMM は、マスターキープロバイダーからデータキー (および関連する暗号化マテリアル) をキャッシュします。また、SDK に代わってキャッシュとのやり取りを行い、設定したセキュリティしきい値を適用します。

暗号化メソッドの呼び出しでは、通常の暗号化マテリアルマネージャー (CMM) やマスターキープロバイダーではなく、キャッシュ CMM が指定されるため、暗号化ではデータキーキャッシュが使用されます。

Java

次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java は、データキーキャッシュ CMM を廃止します。バージョン 3.x では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングを使用することもできます。

```
/*
 * Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use
 this file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed on an
 "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
 * specific language governing permissions and limitations under the License.
```

```
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.util.json.Jackson;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kms.KmsClient;

/**
 * Pushes data to Kinesis Streams in multiple Regions.
 */
public class MultiRegionRecordPusher {

    private static final long MAX_ENTRY_AGE_MILLISECONDS = 300000;
    private static final long MAX_ENTRY_USES = 100;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final String streamName_;
    private final ArrayList<KinesisClient> kinesisClients_;
    private final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;

    /**
     * Creates an instance of this object with Kinesis clients for all target
 Regions and a cached
     * key provider containing KMS master keys in all target Regions.
     */
```

```
   public MultiRegionRecordPusher(final Region[] regions, final String
kmsAliasName,
       final String streamName) {
       streamName_ = streamName;
       crypto_ = AwsCrypto.builder()
           .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
           .build();
       kinesisClients_ = new ArrayList<>();

       AwsCredentialsProvider credentialsProvider =
DefaultCredentialsProvider.builder().build();

       // Build KmsMasterKey and AmazonKinesisClient objects for each target region
       List<KmsMasterKey> masterKeys = new ArrayList<>();
       for (Region region : regions) {
           kinesisClients_.add(KinesisClient.builder()
               .credentialsProvider(credentialsProvider)
               .region(region)
               .build());

           KmsMasterKey regionMasterKey = KmsMasterKeyProvider.builder()
               .defaultRegion(region)
               .builderSupplier(() ->
KmsClient.builder().credentialsProvider(credentialsProvider))
               .buildStrict(kmsAliasName)
               .getMasterKey(kmsAliasName);

           masterKeys.add(regionMasterKey);
       }

       // Collect KmsMasterKey objects into single provider and add cache
       MasterKeyProvider<?> masterKeyProvider =
MultipleProviderFactory.buildMultiProvider(
           KmsMasterKey.class,
           masterKeys
       );

       cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()
           .withMasterKeyProvider(masterKeyProvider)
           .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
           .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
           .withMessageUseLimit(MAX_ENTRY_USES)
           .build();
   }
```

```
    /**
     * JSON serializes and encrypts the received record data and pushes it to all
 target streams.
     */
    public void putRecord(final Map<Object, Object> data) {
        String partitionKey = UUID.randomUUID().toString();
        Map<String, String> encryptionContext = new HashMap<>();
        encryptionContext.put("stream", streamName_);

        // JSON serialize data
        String jsonData = Jackson.toJsonString(data);

        // Encrypt data
        CryptoResult<byte[], ?> result = crypto_.encryptData(
            cachingMaterialsManager_,
            jsonData.getBytes(),
            encryptionContext
        );
        byte[] encryptedData = result.getResult();

        // Put records to Kinesis stream in all Regions
        for (KinesisClient regionalKinesisClient : kinesisClients_) {
            regionalKinesisClient.putRecord(builder ->
                builder.streamName(streamName_)
                    .data(SdkBytes.fromByteArray(encryptedData))
                    .partitionKey(partitionKey));
        }
    }
}
```

## Python

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

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

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

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

from aws_encryption_sdk import EncryptionSDKClient, StrictAwsKmsMasterKeyProvider,
 CachingCryptoMaterialsManager, LocalCryptoMaterialsCache, CommitmentPolicy
from aws_encryption_sdk.key_providers.kms import KMSMasterKey
import boto3


class MultiRegionRecordPusher(object):
    """Pushes data to Kinesis Streams in multiple Regions."""
    CACHE_CAPACITY = 100
    MAX_ENTRY_AGE_SECONDS = 300.0
    MAX_ENTRY_MESSAGES_ENCRYPTED = 100

    def __init__(self, regions, kms_alias_name, stream_name):
        self._kinesis_clients = []
        self._stream_name = stream_name

        # Set up EncryptionSDKClient
        _client =
 EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

        # Set up KMSMasterKeyProvider with cache
        _key_provider = StrictAwsKmsMasterKeyProvider(kms_alias_name)

        # Add MasterKey and Kinesis client for each Region
        for region in regions:
            self._kinesis_clients.append(boto3.client('kinesis',
 region_name=region))
            regional_master_key = KMSMasterKey(
                client=boto3.client('kms', region_name=region),
                key_id=kms_alias_name
            )
            _key_provider.add_master_key_provider(regional_master_key)

        cache = LocalCryptoMaterialsCache(capacity=self.CACHE_CAPACITY)
        self._materials_manager = CachingCryptoMaterialsManager(
```

```
            master_key_provider=_key_provider,
            cache=cache,
            max_age=self.MAX_ENTRY_AGE_SECONDS,
            max_messages_encrypted=self.MAX_ENTRY_MESSAGES_ENCRYPTED
        )

    def put_record(self, record_data):
        """JSON serializes and encrypts the received record data and pushes it to
all target streams.

        :param dict record_data: Data to write to stream
        """
        # Kinesis partition key to randomize write load across stream shards
        partition_key = uuid.uuid4().hex

        encryption_context = {'stream': self._stream_name}

        # JSON serialize data
        json_data = json.dumps(record_data)

        # Encrypt data
        encrypted_data, _header = _client.encrypt(
            source=json_data,
            materials_manager=self._materials_manager,
            encryption_context=encryption_context
        )

        # Put records to Kinesis stream in all Regions
        for client in self._kinesis_clients:
            client.put_record(
                StreamName=self._stream_name,
                Data=encrypted_data,
                PartitionKey=partition_key
            )
```

## コンシューマー

データコンシューマーは Kinesis イベントによってトリガーされる AWS Lambda 関数です。これは、それぞれのレコードを復号および逆シリアル化し、そのプレーンテキストのレコードを同じリージョンの Amazon DynamoDB のテーブルに書き込みます。

プロデューサーコードと同様に、復号メソッドの呼び出でキャッシュ暗号化マテリアルマネージャー (キャッシュ CMM) を使用することで、コンシューマーコードはデータキーキャッシュを有効にします。

Java コードは、指定された を使用して Strict モードでマスターキープロバイダーを構築します AWS KMS key。Strict モードは復号時に必須ではありませんが、ベストプラクティスです。Python コードは検出モードを使用します。これにより、 はデータキーを暗号化したラッピングキー AWS Encryption SDK を使用して復号できます。

Java

次の例では、 のバージョン 2.x を使用しています AWS Encryption SDK for Java。のバージョン 3.x AWS Encryption SDK for Java は、データキーキャッシュ CMM を廃止します。バージョン 3.x では、代替の暗号化マテリアルキャッシュソリューションであるAWS KMS 階層キーリングを使用することもできます。

このコードは、Strict モードで復号するためのマスターキープロバイダーを作成します。 AWS Encryption SDK は、 AWS KMS keys 指定した のみを使用してメッセージを復号できます。

```
/*
 * Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use
 this file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed on an
 "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
 * specific language governing permissions and limitations under the License.
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
```

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;
import com.amazonaws.util.BinaryUtils;
import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;

/**
 * Decrypts all incoming Kinesis records and writes records to DynamoDB.
 */
public class LambdaDecryptAndWrite {

    private static final long MAX_ENTRY_AGE_MILLISECONDS = 600000;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;
    private final DynamoDbTable<Item> table_;

    /**
     * Because the cache is used only for decryption, the code doesn't set the max
 bytes or max
     * message security thresholds that are enforced only on on data keys used for
 encryption.
     */
    public LambdaDecryptAndWrite() {
        String kmsKeyArn = System.getenv("CMK_ARN");
        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()

 .withMasterKeyProvider(KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn))
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .build();

        crypto_ = AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();

        String tableName = System.getenv("TABLE_NAME");
        DynamoDbEnhancedClient dynamodb = DynamoDbEnhancedClient.builder().build();
```

```
        table_ = dynamodb.table(tableName, TableSchema.fromClass(Item.class));
    }

    /**
     * @param event
     * @param context
     */
    public void handleRequest(KinesisEvent event, Context context)
        throws UnsupportedEncodingException {
        for (KinesisEventRecord record : event.getRecords()) {
            ByteBuffer ciphertextBuffer = record.getKinesis().getData();
            byte[] ciphertext = BinaryUtils.copyAllBytesFrom(ciphertextBuffer);

            // Decrypt and unpack record
            CryptoResult<byte[], ?> plaintextResult =
 crypto_.decryptData(cachingMaterialsManager_,
                ciphertext);

            // Verify the encryption context value
            String streamArn = record.getEventSourceARN();
            String streamName = streamArn.substring(streamArn.indexOf("/") + 1);
            if (!
streamName.equals(plaintextResult.getEncryptionContext().get("stream"))) {
                throw new IllegalStateException("Wrong Encryption Context!");
            }

            // Write record to DynamoDB
            String jsonItem = new String(plaintextResult.getResult(),
 StandardCharsets.UTF_8);
            System.out.println(jsonItem);
            table_.putItem(Item.fromJSON(jsonItem));
        }
    }

    private static class Item {

        static Item fromJSON(String jsonText) {
            // Parse JSON and create new Item
            return new Item();
        }
    }
}
```

### Python

この Python コードでは、Discovery モードでマスターキープロバイダーを使用して復号します。AWS Encryption SDK では、データキーを暗号化したラッピングキーを使用して復号できます。復号に使用できるラッピングキーを指定する Strict モードがベストプラクティスです。

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

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

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

or in the "license" file accompanying this file. This file is distributed on an "AS
 IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
specific language governing permissions and limitations under the License.
"""
import base64
import json
import logging
import os

from aws_encryption_sdk import EncryptionSDKClient,
 DiscoveryAwsKmsMasterKeyProvider, CachingCryptoMaterialsManager,
 LocalCryptoMaterialsCache, CommitmentPolicy
import boto3

_LOGGER = logging.getLogger(__name__)
_is_setup = False
CACHE_CAPACITY = 100
MAX_ENTRY_AGE_SECONDS = 600.0

def setup():
    """Sets up clients that should persist across Lambda invocations."""
    global encryption_sdk_client
    encryption_sdk_client =
 EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

    global materials_manager
```

```
    key_provider = DiscoveryAwsKmsMasterKeyProvider()
    cache = LocalCryptoMaterialsCache(capacity=CACHE_CAPACITY)

    #  Because the cache is used only for decryption, the code doesn't set
    #   the max bytes or max message security thresholds that are enforced
    #   only on on data keys used for encryption.
    materials_manager = CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=MAX_ENTRY_AGE_SECONDS
    )
    global table
    table_name = os.environ.get('TABLE_NAME')
    table = boto3.resource('dynamodb').Table(table_name)
    global _is_setup
    _is_setup = True


def lambda_handler(event, context):
    """Decrypts all incoming Kinesis records and writes records to DynamoDB."""
    _LOGGER.debug('New event:')
    _LOGGER.debug(event)
    if not _is_setup:
        setup()
    with table.batch_writer() as batch:
        for record in event.get('Records', []):
            # Record data base64-encoded by Kinesis
            ciphertext = base64.b64decode(record['kinesis']['data'])

            # Decrypt and unpack record
            plaintext, header = encryption_sdk_client.decrypt(
                source=ciphertext,
                materials_manager=materials_manager
            )
            item = json.loads(plaintext)

            # Verify the encryption context value
            stream_name = record['eventSourceARN'].split('/', 1)[1]
            if stream_name != header.encryption_context['stream']:
                raise ValueError('Wrong Encryption Context!')

            # Write record to DynamoDB
            batch.put_item(Item=item)
```

## データキーキャッシュの例: AWS CloudFormation テンプレート

この AWS CloudFormation テンプレートは、データキーキャッシュの例を再現するために必要なすべての AWS リソースを設定します。

JSON

```
{
    "Parameters": {
        "SourceCodeBucket": {
            "Type": "String",
            "Description": "S3 bucket containing Lambda source code zip files"
        },
        "PythonLambdaS3Key": {
            "Type": "String",
            "Description": "S3 key containing Python Lambda source code zip file"
        },
        "PythonLambdaObjectVersionId": {
            "Type": "String",
            "Description": "S3 version id for S3 key containing Python Lambda source
 code zip file"
        },
        "JavaLambdaS3Key": {
            "Type": "String",
            "Description": "S3 key containing Python Lambda source code zip file"
        },
        "JavaLambdaObjectVersionId": {
            "Type": "String",
            "Description": "S3 version id for S3 key containing Python Lambda source
 code zip file"
        },
        "KeyAliasSuffix": {
            "Type": "String",
            "Description": "Suffix to use for KMS key Alias (ie: alias/
<KeyAliasSuffix>)"
        },
        "StreamName": {
            "Type": "String",
            "Description": "Name to use for Kinesis Stream"
        }
```

```
    },
    "Resources": {
        "InputStream": {
            "Type": "AWS::Kinesis::Stream",
            "Properties": {
                "Name": {
                    "Ref": "StreamName"
                },
                "ShardCount": 2
            }
        },
        "PythonLambdaOutputTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "id",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "id",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        },
        "PythonLambdaRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
```

```
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                ],
                "Policies": [
                    {
                        "PolicyName": "PythonLambdaAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:DescribeTable",
                                        "dynamodb:BatchWriteItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:PutItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "kinesis:GetRecords",
                                        "kinesis:GetShardIterator",
                                        "kinesis:DescribeStream",
                                        "kinesis:ListStreams"
                                    ],
                                    "Resource": {
```

```
                                        "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "PythonLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Python consumer",
                "Runtime": "python2.7",
                "MemorySize": 512,
                "Timeout": 90,
                "Role": {
                    "Fn::GetAtt": [
                        "PythonLambdaRole",
                        "Arn"
                    ]
                },
                "Handler":
 "aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler",
                "Code": {
                    "S3Bucket": {
                        "Ref": "SourceCodeBucket"
                    },
                    "S3Key": {
                        "Ref": "PythonLambdaS3Key"
                    },
                    "S3ObjectVersion": {
                        "Ref": "PythonLambdaObjectVersionId"
                    }
                },
                "Environment": {
                    "Variables": {
                        "TABLE_NAME": {
                            "Ref": "PythonLambdaOutputTable"
                        }
                    }
                }
            }
```

```
        },
        "PythonLambdaSourceMapping": {
            "Type": "AWS::Lambda::EventSourceMapping",
            "Properties": {
                "BatchSize": 1,
                "Enabled": true,
                "EventSourceArn": {
                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                },
                "FunctionName": {
                    "Ref": "PythonLambdaFunction"
                },
                "StartingPosition": "TRIM_HORIZON"
            }
        },
        "JavaLambdaOutputTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "id",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "id",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        },
        "JavaLambdaRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
```

```
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                ],
                "Policies": [
                    {
                        "PolicyName": "JavaLambdaAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:DescribeTable",
                                        "dynamodb:BatchWriteItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:PutItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "kinesis:GetRecords",
                                        "kinesis:GetShardIterator",
```

```
                                    "kinesis:DescribeStream",
                                    "kinesis:ListStreams"
                                ],
                                "Resource": {
                                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                                }
                            }
                        ]
                    }
                }
            ]
        }
    },
    "JavaLambdaFunction": {
        "Type": "AWS::Lambda::Function",
        "Properties": {
            "Description": "Java consumer",
            "Runtime": "java8",
            "MemorySize": 512,
            "Timeout": 90,
            "Role": {
                "Fn::GetAtt": [
                    "JavaLambdaRole",
                    "Arn"
                ]
            },
            "Handler":
 "com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest",
            "Code": {
                "S3Bucket": {
                    "Ref": "SourceCodeBucket"
                },
                "S3Key": {
                    "Ref": "JavaLambdaS3Key"
                },
                "S3ObjectVersion": {
                    "Ref": "JavaLambdaObjectVersionId"
                }
            },
            "Environment": {
                "Variables": {
                    "TABLE_NAME": {
                        "Ref": "JavaLambdaOutputTable"
```

```
                        },
                        "CMK_ARN": {
                            "Fn::GetAtt": [
                                "RegionKinesisCMK",
                                "Arn"
                            ]
                        }
                    }
                }
            }
        },
        "JavaLambdaSourceMapping": {
            "Type": "AWS::Lambda::EventSourceMapping",
            "Properties": {
                "BatchSize": 1,
                "Enabled": true,
                "EventSourceArn": {
                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                },
                "FunctionName": {
                    "Ref": "JavaLambdaFunction"
                },
                "StartingPosition": "TRIM_HORIZON"
            }
        },
        "RegionKinesisCMK": {
            "Type": "AWS::KMS::Key",
            "Properties": {
                "Description": "Used to encrypt data passing through Kinesis Stream
 in this region",
                "Enabled": true,
                "KeyPolicy": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "AWS": {
                                    "Fn::Sub": "arn:aws:iam::${AWS::AccountId}:root"
                                }
                            },
                            "Action": [
                                "kms:Encrypt",
```

```
                            "kms:GenerateDataKey",
                            "kms:CreateAlias",
                            "kms:DeleteAlias",
                            "kms:DescribeKey",
                            "kms:DisableKey",
                            "kms:EnableKey",
                            "kms:PutKeyPolicy",
                            "kms:ScheduleKeyDeletion",
                            "kms:UpdateAlias",
                            "kms:UpdateKeyDescription"
                        ],
                        "Resource": "*"
                    },
                    {
                        "Effect": "Allow",
                        "Principal": {
                            "AWS": [
                                {
                                    "Fn::GetAtt": [
                                        "PythonLambdaRole",
                                        "Arn"
                                    ]
                                },
                                {
                                    "Fn::GetAtt": [
                                        "JavaLambdaRole",
                                        "Arn"
                                    ]
                                }
                            ]
                        },
                        "Action": "kms:Decrypt",
                        "Resource": "*"
                    }
                ]
            }
        }
    },
    "RegionKinesisCMKAlias": {
        "Type": "AWS::KMS::Alias",
        "Properties": {
            "AliasName": {
                "Fn::Sub": "alias/${KeyAliasSuffix}"
            },
```

```
                "TargetKeyId": {
                    "Ref": "RegionKinesisCMK"
                }
            }
        }
    }
}
```

YAML

```
Parameters:
    SourceCodeBucket:
        Type: String
        Description: S3 bucket containing Lambda source code zip files
    PythonLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    PythonLambdaObjectVersionId:
        Type: String
        Description: S3 version id for S3 key containing Python Lambda source code
 zip file
    JavaLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    JavaLambdaObjectVersionId:
        Type: String
        Description: S3 version id for S3 key containing Python Lambda source code
 zip file
    KeyAliasSuffix:
        Type: String
        Description: 'Suffix to use for KMS CMK Alias (ie: alias/<KeyAliasSuffix>)'
    StreamName:
        Type: String
        Description: Name to use for Kinesis Stream
Resources:
    InputStream:
        Type: AWS::Kinesis::Stream
        Properties:
            Name: !Ref StreamName
            ShardCount: 2
    PythonLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
```

```
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    PythonLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: PythonLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*
                            -
```

```
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    PythonLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Python consumer
            Runtime: python2.7
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt PythonLambdaRole.Arn
            Handler:
 aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref PythonLambdaS3Key
                S3ObjectVersion: !Ref PythonLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref PythonLambdaOutputTable
    PythonLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref PythonLambdaFunction
            StartingPosition: TRIM_HORIZON
    JavaLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
```

```
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    JavaLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: JavaLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
```

```
                                    Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    JavaLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Java consumer
            Runtime: java8
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt JavaLambdaRole.Arn
            Handler:
 com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref JavaLambdaS3Key
                S3ObjectVersion: !Ref JavaLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref JavaLambdaOutputTable
                    CMK_ARN: !GetAtt RegionKinesisCMK.Arn
    JavaLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref JavaLambdaFunction
            StartingPosition: TRIM_HORIZON
    RegionKinesisCMK:
        Type: AWS::KMS::Key
        Properties:
            Description: Used to encrypt data passing through Kinesis Stream in this
 region
            Enabled: true
            KeyPolicy:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            AWS: !Sub arn:aws:iam::${AWS::AccountId}:root
                        Action:
                            # Data plane actions
```

```
                            - kms:Encrypt
                            - kms:GenerateDataKey
                            # Control plane actions
                            - kms:CreateAlias
                            - kms:DeleteAlias
                            - kms:DescribeKey
                            - kms:DisableKey
                            - kms:EnableKey
                            - kms:PutKeyPolicy
                            - kms:ScheduleKeyDeletion
                            - kms:UpdateAlias
                            - kms:UpdateKeyDescription
                        Resource: '*'
                    -
                        Effect: Allow
                        Principal:
                            AWS:
                                - !GetAtt PythonLambdaRole.Arn
                                - !GetAtt JavaLambdaRole.Arn
                        Action: kms:Decrypt
                        Resource: '*'
    RegionKinesisCMKAlias:
        Type: AWS::KMS::Alias
        Properties:
            AliasName: !Sub alias/${KeyAliasSuffix}
            TargetKeyId: !Ref RegionKinesisCMK
```

# のバージョン AWS Encryption SDK

AWS Encryption SDK 言語実装では、セマンティックバージョニングを使用して、各リリースの変更の大きさを簡単に特定できます。1.x.x から 2.x.x のようなメジャーバージョン番号の変更は、コードの変更と計画的デプロイが必要になる可能性のある重大な変更を示します。新しいバージョンの変更がすべてのユースケースに影響を与えるとは限りません。リリースノートを確認して、影響を受けるかどうかを確認してください。x.1.x から x.2.x のようなマイナーバージョンの変更では、常に下位互換性がありますが、非推奨の要素が含まれている可能性があります。

可能な限り、 AWS Encryption SDK 選択したプログラミング言語で最新バージョンの を使用します。各バージョンの メンテナンスとサポートのポリシー は、プログラミング言語の実装によって異なります。お好みのプログラミング言語でサポートされているバージョンの詳細については、GitHub リポジトリ の `SUPPORT_POLICY.rst` ファイルを参照してください。

アップグレードに暗号化や復号化エラーを回避するための特別な設定を必要とする新機能が含まれる場合は、中間バージョンとその使用方法の詳細な説明を提供します。例えば、バージョン 1.7.x と 1.8.x は、1.7.x より前のバージョンからバージョン 2.0.x 以降へのアップグレードに役立つ移行バージョンになるように設計されています。　詳細については、「の移行 AWS Encryption SDK」を参照してください。

> **Note**
>
> バージョン番号の x は、メジャーバージョンとマイナーバージョンのパッチを示します。例えば、バージョン 1.7.x は 1.7 で始まるすべてのバージョンを表し、1.7.1 および 1.7.9 が含まれます。
>
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x AWS に置き換わり、Encryption CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

次の表は、各プログラミング言語の でサポートされているバージョン間の主な違いの概要 AWS Encryption SDK を示しています。

# C

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk-c リポジトリの CHANGELOG.md を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Supportフェーズ |
| | 1.7 | Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see バージョン 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see バージョン 2.0.x. | 一般提供 (GA) |
| | 2.2 | Improvements to the message decryption process. | |
| | 2.3 | Adds support for AWS KMS multi-Region keys. | |

# C# / .NET

すべての変更の詳細な説明については、GitHub の `aws-encryption-sdk-net`リポジトリの CHANGELOG.md を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 3.x | 3.0 | Initial release. | 一般提供 (GA)<br><br>.NET AWS Encryption SDK 用 のバージョン 3.x は、2024 年 5 月 13 日にメンテナンスモードになります。 |
| 4.x | 4.0 | Adds support for the AWS KMS Hierarchical keyring, the required encryption context CMM, and asymmetric RSA AWS KMS keyrings. | 一般提供 (GA) |

# コマンドラインインターフェイス (CLI)

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk-cli リポジトリのEncryption AWS CLI のバージョン「」と「CHANGELOG.rst」を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Supportフェーズ |
| | 1.7 | Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more | |

| | | | |
|---|---|---|---|
| | | information, see バージョン 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see バージョン 2.0.x. | End-of-Supportフェーズ |
| | 2.1 | `--discovery` パラメータを削除し、`--wrapping-keys` パラメータの `discovery` 属性に置き換えます。<br><br>Encryption AWS CLI のバージョン 2.1.0 は、他のプログラミング言語のバージョン 2.0 と同等です。 | |
| | 2.2 | Improvements to the message decryption process. | |
| 3.x | 3.0 | Adds support for AWS KMS multi-Region keys. | End-of-Supportフェーズ |
| 4.x | 4.0 | The AWS Encryption CLI no longer supports Python 2 or Python 3.4. As of major version 4.x of the AWS Encryption CLI, only Python 3.5 or later is supported. | 一般提供 (GA) |

| | | | |
|---|---|---|---|
| | 4.1 | The AWS Encryption CLI no longer supports Python 3.5. As of version 4.1.x of the AWS Encryption CLI, only Python 3.6 or later is supported. | |
| | 4.2 | The AWS Encryption CLI no longer supports Python 3.6. As of version 4.2.x of the AWS Encryption CLI, only Python 3.7 or later is supported. | |

## Java

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk-java リポジトリの CHANGELOG.rst を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Supportフェーズ |
| | 1.3 | Adds support for cryptographic materials manager and data key caching. Moved to deterministic IV generation. | |
| | 1.6.1 | `AwsCrypto.encryptString()` と を | |

| | | | |
|---|---|---|---|
| | | 廃止`AwsCrypto.decryptString()` し、`AwsCrypto.encryptData()` および に置き換えます`AwsCrypto.decryptData()` 。 | |
| | 1.7 | Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see バージョン 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see バージョン 2.0.x. | 一般提供 (GA)<br><br>のバージョン 2.x AWS Encryption SDK for Java は 2024 年にメンテナンスモードになります。 |
| | 2.2 | Improvements to the message decryption process. | |
| | 2.3 | Adds support for AWS KMS multi-Region keys. | |
| | 2.4 | Adds support for AWS SDK for Java 2.x. | |

| | | | |
|---|---|---|---|
| 3.x | 3.0 | をマテリアルプロバイダーライブラリ (MPL) AWS Encryption SDK for Java と統合します。 https://github.com/aws/aws-cryptographic-material-providers-library<br><br>対称および非対称 RSA AWS KMS キーリング、 AWS KMS ECDH キーリング、AWS KMS 階層キーリング、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、および必要な暗号化コンテキスト CMM のサポートが追加されました。 | 一般提供 (GA) |

## Go

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk リポジトリの Go ディレクトリにある CHANGELOG.md を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 0.1.x | 0.1.0 | Initial release. | 一般提供 (GA) |

# JavaScript

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk-javascript リポジトリの CHANGELOG.md を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Supportフェーズ |
| | 1.7 | Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see バージョン 1.7.x. | |
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see バージョン 2.0.x. | End-of-Supportフェーズ |
| | 2.2 | Improvements to the message decryption process. | |
| | 2.3 | Adds support for AWS KMS multi-Region keys. | |
| 3.x | 3.0 | Removes CI coverage for Node 10. Upgrades dependencies to no longer | メンテナンス<br>のバージョン 3.x のサポート AWS Encryption SDK for JavaScript は、2024 |

| | | | |
|---|---|---|---|
| | | support Node 8 and Node 10. | 年 1 月 17 日に終了します。 |
| 4.x | 4.0 | Requires version 3 of the AWS Encryption SDK for JavaScript's `kms ######` to use the AWS KMS keyring. | 一般提供 (GA) |

## Python

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk-python リポジトリの CHANGELOG.rst を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | End-of-Supportフェーズ |
| | 1.3 | Adds support for cryptographic materials manager and data key caching. Moved to deterministic IV generation. | |
| | 1.7 | Updates to the AWS Encryption SDK that help users of earlier versions upgrade to versions 2.0.x and later. For more information, see バージョン 1.7.x. | |

| | | | |
|---|---|---|---|
| 2.x | 2.0 | Updates to the AWS Encryption SDK. For more information, see バージョン 2.0.x. | End-of-Supportフェーズ |
| | 2.2 | Improvements to the message decryption process. | |
| | 2.3 | Adds support for AWS KMS multi-Region keys. | |
| 3.x | 3.0 | The AWS Encryption SDK for Python no longer supports Python 2 or Python 3.4. As of major version 3.x of the AWS Encryption SDK for Python, only Python 3.5 or later is supported. | 一般提供 (GA) |
| 4.x | 4.0 | をマテリアルプロバイダーライブラリ (MPL) AWS Encryption SDK for Python と統合します。 https://github.com/aws/aws-cryptographic-material-providers-library | 一般提供 (GA) |

## Rust

すべての変更の詳細な説明については、GitHub の aws-encryption-sdk リポジトリの Rust ディレクトリにある CHANGELOG.md を参照してください。

| メジャーバージョン | 詳細 | | SDK メジャーバージョンのライフサイクルフェーズ |
|---|---|---|---|
| 1.x | 1.0 | Initial release. | 一般提供 (GA) |

# バージョンの詳細

以下のリストでは、 AWS Encryption SDKでサポートされているバージョンの主な相違点を示します。

トピック


- 1.7.x より前のバージョン
- バージョン 1.7.x
- バージョン 2.0.x
- バージョン 2.2.x
- バージョン 2.3.x


## 1.7.x より前のバージョン

> **Note**
>
> のすべての 1.x.x バージョン AWS Encryption SDK はend-of-supportフェーズにあります。可能な限り早く、プログラミング言語 AWS Encryption SDK の の最新バージョンにアップグレードします。1.7.x より前の AWS Encryption SDK バージョンからアップグレードするには、まず 1.7.x にアップグレードする必要があります。詳細については、「の移行 AWS Encryption SDK」を参照してください。

1.7.x より AWS Encryption SDK 前のバージョンでは、Galois/Counter Mode (AES-GCM) の Advanced Encryption Standard アルゴリズムによる暗号化、HMAC extract-and-expandキー取得関数 (HKDF)、署名、256 ビット暗号化キーなど、重要なセキュリティ機能が提供されます。ただし、これらのバージョンでは、キーコミットメントなど、推奨ベストプラクティスがサポートされません。

# バージョン 1.7.x

> **Note**
>
> のすべての 1.x.x バージョン AWS Encryption SDK はend-of-supportフェーズにあります。

バージョン 1.7.x は、 の以前のバージョンのユーザーがバージョン 2.0.x 以降に AWS Encryption SDK アップグレードするのに役立つように設計されています。を初めて使用する場合は AWS Encryption SDK、このバージョンをスキップして、プログラミング言語で利用可能な最新バージョンから始めることができます。

バージョン 1.7.x には完全な下位互換性があり、重大な変更の導入や AWS Encryption SDKの動作の変更はありません。上位互換性もあり、バージョン 2.0.x と互換性があるようにコードを更新できます。これには新機能が含まれますが、完全に有効になるわけではありません。また、準備が整うまで、すべての新機能をすぐには採用できないようにする設定値が必要です。

バージョン 1.7.x には次の変更が含まれています。

AWS KMS マスターキープロバイダーの更新 (必須 )

バージョン 1.7.x では、 AWS Encryption SDK for Java と に新しいコンストラクタが導入され AWS Encryption SDK for Python 、厳格モードまたは検出モードで AWS KMS マスターキープロバイダーが明示的に作成されます。このバージョンでは、 AWS Encryption SDK コマンドラインインターフェイス (CLI) に同様の変更が追加されています。詳細については、「AWS KMS マスターキープロバイダーの更新」を参照してください。

- AWS KMS マスターキープロバイダーでは、Strict モードの場合、ラッピングキーのリストが必要で、指定したラッピングキーのみで暗号化と復号化が行われます。これが AWS Encryption SDK のベストプラクティスで、使用を意図したラッピングキーを使用していることが保証されます。
- AWS KMS マスターキープロバイダーでは、Discovery モードの場合、ラッピングキーが使用されません。ラッピングキーを暗号化に使用することはできません。復号時には、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できます。ただし、復号化に使用するラッピングキーは、特定の AWS アカウントのものに制限できます。アカウントのフィルタリングはオプションですが、お勧めのベストプラクティスです。

以前のバージョンの AWS KMS マスターキープロバイダーを作成するコンストラクタは、バージョン 1.7.x では廃止され、バージョン 2.0.x では削除されます。これらのコンストラクタは、指定したラッピングキーを使用して暗号化するマスターキープロバイダーをインスタンス化し

ます。ただし、指定したラッピングキーに関係なく、暗号化した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化します。ユーザーは、 AWS KMS keys 他の AWS アカウント やリージョンなど、使用しないラッピングキーを使用してメッセージを意図せずに復号する場合があります。

AWS KMS マスターキーのコンストラクタに変更はありません。暗号化および復号時に、 AWS KMS マスターキーは AWS KMS key 指定した のみを使用します。

AWS KMS キーリングの更新 (オプション )

バージョン 1.7.x では、AWS KMS 検出キーリングを特定の に制限する新しいフィルターが AWS Encryption SDK for C および AWS Encryption SDK for JavaScript 実装に追加されます AWS アカウント。この新しいアカウントフィルターはオプションですが、お勧めのベストプラクティスです。詳細については、「AWS KMS キーリングの更新」を参照してください。

AWS KMS キーリングのコンストラクタに変更はありません。標準 AWS KMS キーリングは、strict モードでマスターキープロバイダーのように動作します。 AWS KMS discovery キーリングは、discovery モードで明示的に作成されます。

キー ID を AWS KMS 復号化する

バージョン 1.7.x 以降では、暗号化されたデータキーを復号するときに、 は AWS KMS Decrypt オペレーションの呼び出し AWS KMS key で AWS Encryption SDK 常に を指定します。は、暗号化された各データキーのメタデータ AWS KMS key から のキー ID 値 AWS Encryption SDK を取得します。この機能では、コードの変更は必要ありません。

対称暗号化 AWS KMS key KMS キーで暗号化された暗号文を復号するために のキー ID を指定する必要はありませんが、AWS KMS ベストプラクティスです。キープロバイダーでラッピングキーを指定する場合と同様に、この方法では、使用するラッピングキーを使用して AWS KMS のみ が復号化されるようにします。

キーコミットメントで暗号化テキストを復号化する

バージョン 1.7.x では、キーコミットメントを使用しているかどうかに関係なく、暗号化された暗号化テキストを復号化できます。ただし、キーコミットメントによって暗号化テキストを暗号化することはできません。このプロパティを使用すると、キーコミットメントで暗号化された暗号化テキストを復号化できるアプリケーションを完全にデプロイしてから、そのような暗号化テキストを処理できます。このバージョンでは、キーコミットメントなしで暗号化されたメッセージを復号化するため、暗号化テキストを再暗号化する必要はありません。

この動作を実装するために、バージョン 1.7.x には、 がキーコミットメントで暗号化または復号化できるかどうかを決定する新しいコミットメントポリシー設定が含まれてい

ます。 AWS Encryption SDK バージョン 1.7.x では、コミットメントポリシーの有効な値、`ForbidEncryptAllowDecrypt` が暗号化と復号化のすべてのオペレーションで使用されます。この値により、 AWS Encryption SDK がキーコミットメントを含む新しいアルゴリズムスイートのいずれかで暗号化することが防止されます。これにより、 AWS Encryption SDK はキーコミットメントの有無にかかわらず暗号文を復号できます。

バージョン 1.7.x には有効なコミットメントポリシーの値が 1 つしかありませんが、このリリースで導入された新しい API を使用する場合は、この値を明示的に設定してください。値を明示的に設定すると、バージョン 2.1.x へのアップグレード時にコミットメントポリシーが自動的に `require-encrypt-require-decrypt` に変更されなくなります。その代わりに、コミットメントポリシーを段階的に移行できます。

キーコミットメントを使用するアルゴリズムスイート

バージョン 1.7.xには新しい 2 つのアルゴリズムスイートが組み込まれて、キーコミットメントがサポートされます。一方は署名を含み、もう一方は署名を含みません。以前サポートされていたアルゴリズムスイートと同様に、これらの両方の新しいアルゴリズムスイートには、AES-GCM による暗号化、256 ビット暗号化キー、HMAC ベースの抽出および展開キー取得関数 (HKDF) が含まれます。

ただし、暗号化に使用されるデフォルトのアルゴリズムスイートは変更されません。これらのアルゴリズムスイートがバージョン 1.7.x に追加されるのは、バージョン 2.0.x 以降で使用するようにアプリケーションを準備するためです。

CMM 実装の変更

バージョン 1.7.x では、キーコミットメントをサポートするために、デフォルト暗号化マテリアルマネージャ (CMM) インターフェイスが変更されました。この変更は、カスタム CMM を作成した場合にのみ影響します。詳細については、使用しているプログラミング言語の API ドキュメントまたは GitHub リポジトリを参照してください。

## バージョン 2.0.x

バージョン 2.0.x は、指定されたラッピングキーやキーコミットメントなど AWS Encryption SDK、で提供される新しいセキュリティ機能をサポートしています。バージョン 2.0.x では、これらの機能をサポートするため、前バージョンの AWS Encryption SDKが大きく変更されています。バージョン 1.7.x をデプロイすれば、これらの変更に備えることができます。バージョン 2.0.x には、バージョン 1.7.x で導入されたすべての新機能が含まれており、以下の追加・変更点もあります。

> **Note**
>
> のバージョン 2.x.x AWS Encryption SDK for Python AWS Encryption SDK for JavaScript、および AWS Encryption CLI end-of-supportフェーズにあります。
> 任意のプログラミング言語でのこの AWS Encryption SDK バージョンのサポートとメンテナンスの詳細については、GitHub リポジトリの `SUPPORT_POLICY.rst` ファイルを参照してください。

### AWS KMS マスターキープロバイダー

バージョン 1.7.x で廃止された元の AWS KMS マスターキープロバイダーコンストラクタは、バージョン 2.0.x で削除されます。 AWS KMS マスターキープロバイダーは、Strict モードまたは Discovery モードで明示的に構築する必要があります。

### キーコミットメントによる暗号化テキストの暗号化と復号化

バージョン 2.0.x では、キーコミットメントを使用しているかどうかに関係なく、暗号化テキストの暗号化と復号化ができます。その動作は、コミットメントポリシー設定によって決まります。デフォルトでは、常にキーコミットメントで暗号化し、キーコミットメントで暗号化された暗号化テキストのみを復号します。コミットメントポリシーを変更しない限り、 AWS Encryption SDK では、バージョン 1.7.x を含む AWS Encryption SDKの旧バージョンで暗号化された暗号化テキストが復号化されません。

> **Important**
>
> デフォルトの場合、バージョン 2.0.x では、キーコミットメントなしで暗号化された暗号化テキストは復号化されません。キーコミットなしで暗号化された暗号化テキストをアプリケーションで処理する可能性がある場合は、コミットメントポリシーの値を `AllowDecrypt` で設定してください。

バージョン 2.0.x の場合、コミットメントポリシー設定には次の 3 つの有効な値があります。

- `ForbidEncryptAllowDecrypt` — AWS Encryption SDK では、キーコミットメントで暗号化することはできません。キーコミットメントが使用されているかどうかにかかわらず、暗号化された暗号化テキストを復号化できます。
- `RequireEncryptAllowDecrypt` — AWS Encryption SDK では、キーコミットメントで暗号化する必要があります。キーコミットメントが使用されているかどうかにかかわらず、暗号化された暗号化テキストを復号化できます。

- `RequireEncryptRequireDecrypt` (デフォルト) — はキーコミットメントで暗号化 AWS Encryption SDK する必要があります。キーコミットメントによる暗号化テキストのみを復号化します。

の以前のバージョンから AWS Encryption SDK バージョン 2.0.x に移行する場合は、コミットメントポリシーを、アプリケーションが遭遇する可能性のある既存の暗号文をすべて復号できる値に設定します。この設定は時間の経過とともに調整することになる可能性があります。

## バージョン 2.2.x

デジタル署名と暗号化データキーの制限のサポートを追加します。

> **Note**
>
> のバージョン 2.x.x AWS Encryption SDK for Python AWS Encryption SDK for JavaScript、および AWS Encryption CLI end-of-supportフェーズにあります。
> 任意のプログラミング言語でのこの AWS Encryption SDK バージョンのサポートとメンテナンスについては、GitHub リポジトリの `SUPPORT_POLICY.rst` ファイルを参照してください。

### デジタル署名

復号時のデジタル署名の処理を改善するために、 には次の機能 AWS Encryption SDK が含まれています。

- 非ストリーミングモード — デジタル署名が存在する場合の検証を含め、すべての入力を処理した後にのみプレーンテキストを返します。　この機能を使用すると、デジタル署名を検証するまでプレーンテキストを使用できなくなります。この機能は、デジタル署名 (デフォルトのアルゴリズムスイート) で暗号化されたデータを復号化するときに使用します。たとえば、Encryption AWS CLI は常にストリーミングモードでデータを処理するため、デジタル署名で暗号文を復号するときに `- -buffer`パラメータを使用します。
- 署名なし専用復号モード — この機能では署名されていない暗号文のみを復号化します。復号化で暗号化テキスト内にデジタル署名が検出されると、オペレーションは失敗します。この機能を使用して、署名を検証する前に、署名付きメッセージのプレーンテキストを意図せずに処理しないようにします。

暗号化されたデータキーの制限

暗号化されたメッセージ内の暗号化されたデータキーの数を制限できます。この機能は、暗号化時に誤って構成されたマスターキープロバイダーまたはキーリングを検出したり、復号時に悪意のある暗号化テキストを特定したりするのに役立ちます。

信頼できない送信元からのメッセージを復号する場合は、暗号化されたデータキーを制限してください。不必要でコストがかかり、潜在的に網羅的な方法によって、キーインフラストラクチャを呼び出すことを防止できます。

## バージョン 2.3.x

AWS KMS マルチリージョンキーのサポートを追加しました。詳細については、「マルチリージョンの使用 AWS KMS keys」を参照してください。

> **Note**
>
> AWS Encryption CLI は、バージョン 3.0.x 以降のマルチリージョンキーをサポートしています。
> のバージョン 2.x.x AWS Encryption SDK for Python AWS Encryption SDK for JavaScript、および AWS Encryption CLI はend-of-supportフェーズにあります。
> 任意のプログラミング言語でのこの AWS Encryption SDK バージョンのサポートとメンテナンスについては、GitHub リポジトリの `SUPPORT_POLICY.rst` ファイルを参照してください。

# の移行 AWS Encryption SDK

は、相互運用可能な複数のプログラミング言語の実装 AWS Encryption SDK をサポートしており、それぞれが GitHub のオープンソースリポジトリで開発されています。ベストプラクティスとして、言語 AWS Encryption SDK ごとに最新バージョンの を使用することをお勧めします。

のバージョン 2.0.x 以降から最新バージョン AWS Encryption SDK に安全にアップグレードできます。ただし、 の 2.0.x バージョンでは、重要な新しいセキュリティ機能 AWS Encryption SDK が導入されていますが、その一部は重大な変更です。1.7.x より前のバージョンからバージョン 2.0.x 以降へアップグレードするには、まず最新の 1.x バージョンにアップグレードする必要があります。このセクションのトピックは、変更を理解し、アプリケーションの正しいバージョンを選択し、 AWS Encryption SDKの最新バージョンに安全かつ正常に移行できるように設計されています。

の重要バージョンについては AWS Encryption SDK、「」を参照してくださいのバージョン AWS Encryption SDK。

**Important**

1.7.x より前のバージョンからバージョン 2.0.x 以降に直接アップグレードする場合は、最初に最新の 1.x バージョンをアップグレードしてから行ってください。バージョン 2.0.x 以降に直接アップグレードし、すべての新機能をすぐに有効にすると、 AWS Encryption SDK は古いバージョンの で暗号化された暗号文を復号できなくなります AWS Encryption SDK。

**Note**

.NET AWS Encryption SDK 用 の最も古いバージョンはバージョン 3.0.x です。.NET AWS Encryption SDK 用 のすべてのバージョンは、 の 2.0.x で導入されたセキュリティのベストプラクティスをサポートしています AWS Encryption SDK。コードやデータを変更することなく、最新バージョンに安全にアップグレードできます。

AWS Encryption CLI: この移行ガイドを読むときは、Encryption CLI 1.8.x AWS の 1.7.x 移行手順を使用し、Encryption CLI 2.1.x AWS の 2.0.x 移行手順を使用します。詳細については、「Encryption AWS CLI のバージョン」を参照してください。

新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細について

は、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

新規のユーザー

を初めて使用する場合は AWS Encryption SDK、プログラミング言語 AWS Encryption SDK の最新バージョンの をインストールします。デフォルト値は、署名による暗号化 AWS Encryption SDK、キー取得、 のキーコミットメントなど、 のすべてのセキュリティ機能を有効にします。AWS Encryption SDK

現在のユーザー

できるだけ早く現在のバージョンから利用可能な最新バージョンにアップグレードすることをお勧めします。のすべての 1.x バージョン AWS Encryption SDK はend-of-supportフェーズにあり、一部のプログラミング言語ではそれ以降のバージョンも同様です。プログラミング言語での AWS Encryption SDK のサポートとメンテナンスの状況の詳細については、「サポートとメンテナンス」を参照してください。

AWS Encryption SDK バージョン 2.0.x 以降では、データの保護に役立つ新しいセキュリティ機能が提供されます。ただし、 AWS Encryption SDK バージョン 2.0.x には、下位互換性のない重大な変更が含まれています。安全な移行を確実に行うには、まず現在のバージョンからプログラミング言語の最新の 1.x への移行から始めてください。最新 1.x バージョンが完全にデプロイされて正常に動作している場合は、バージョン 2.0.x 以降に安全に移行できます。この 2 段階のプロセスは、特に分散アプリケーションでは重要です。

これらの変更の基礎となる AWS Encryption SDK セキュリティ機能の詳細については、 AWS セキュリティブログの「クライアント側の暗号化の改善: 明示的な KeyIds とキーコミットメント」を参照してください。

AWS Encryption SDK for Java で を使用するためのヘルプをお探し AWS SDK for Java 2.xですか？「前提条件」を参照してください。

トピック

- を移行およびデプロイする方法 AWS Encryption SDK
- AWS KMS マスターキープロバイダーの更新
- AWS KMS キーリングの更新
- コミットメントポリシーの設定

- 最新バージョンへの移行に関するトラブルシューティング

# を移行およびデプロイする方法 AWS Encryption SDK

1.7.x より前の AWS Encryption SDK バージョンからバージョン 2.0.x 以降に移行する場合は、キーコミットメントによる暗号化に安全に移行する必要があります。そうしないと、アプリケーションは復号できない暗号化テキストを検出します。 AWS KMS マスターキープロバイダーを使用している場合は、マスターキープロバイダーを Strict モードまたは Discovery モードで作成する新しいコンストラクタに更新する必要があります。

> **Note**
>
> このトピックは、 AWS Encryption SDK の以前のバージョンからバージョン 2.0.x 以降に移行するユーザーを対象としています。を初めて使用する場合は AWS Encryption SDK、デフォルト設定で利用可能な最新バージョンをすぐに使用できます。

読む必要がある暗号化テキストを復号化できない重大な状況を回避するには、複数の異なるステージで移行およびデプロイすることをお勧めします。各ステージを完了して完全にデプロイしたことを確認してから、次のステージを開始してください。これは、複数のホストがある分散アプリケーションでは特に重要です。

## ステージ 1: アプリケーションを最新 1.x バージョンに更新

ご使用のプログラミング言語の最新 1.x バージョンに更新します。慎重にテストし、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てから、ステージ 2 を開始します。

> **Important**
>
> 最新 1.x バージョンが AWS Encryption SDKのバージョン 1.7.x 以降であることを確認します。

の最新の 1.x バージョン AWS Encryption SDK は、 のレガシーバージョンと下位互換性 AWS Encryption SDK があり、バージョン 2.0.x 以降と前方互換性があります。これらはバージョン 2.0.x にある新機能が含まれていますが、この移行用に設計された安全なデフォルトが含まれています。こ

れにより、必要に応じて AWS KMS マスターキープロバイダーをアップグレードし、キーコミットメントで暗号文を復号できるアルゴリズムスイートを使用して完全にデプロイできます。

- レガシー AWS KMS マスターキープロバイダーのコンストラクタなど、非推奨の要素を置き換えます。Python では、非推奨の警告をオンにしてください。最新 1.x バージョンで非推奨になったコード要素は、バージョン 2.0.x 以降で削除されます。
- コミットメントポリシーを明示的に `ForbidEncryptAllowDecrypt` に設定してください。最新 1.x バージョンではこれが唯一の有効な値ですが、この設定は、このリリースで導入された API を使用する場合に必要です。これにより、バージョン 2.0.x 以降への移行時に、キーコミットメントなしで暗号化された暗号化テキストがアプリケーションで拒否されるのを防ぎます。詳細については、「the section called “コミットメントポリシーの設定”」を参照してください。
- AWS KMS マスターキープロバイダーを使用する場合は、レガシーマスターキープロバイダーを Strict モードと Discovery モードをサポートするマスターキープロバイダーに更新する必要があります。この更新は AWS Encryption SDK for Java、、 AWS Encryption SDK for Python、および AWS Encryption CLI に必要です。Discovery モードでマスターキープロバイダーを使用する場合は、Discovery フィルターを実装して、使用するラッピングキーを特に AWS アカウントのものに制限することをお勧めします。この更新はオプションですが、お勧めのベストプラクティスです。詳細については、「AWS KMS マスターキープロバイダーの更新」を参照してください。
- AWS KMS 検出キーリング を使用する場合は、復号化に使用するラッピングキーを特に AWS アカウントの制限する検出フィルターを含めることをお勧めします。この更新はオプションですが、お勧めのベストプラクティスです。詳細については、「AWS KMS キーリングの更新」を参照してください。

## ステージ 2: アプリケーションを最新バージョンに更新

最新 1.x バージョンをすべてのホストに正常にデプロイしたら、バージョン 2.0.x 以降にアップグレードできます。バージョン 2.0.x には、 のすべての以前のバージョンに対する重大な変更が含まれています AWS Encryption SDK。ただし、ステージ 1 で推奨されるコードの変更を行うと、最新バージョンに移行するときにエラーを回避できます。

最新バージョンに更新する前に、コミットメントポリシーを一貫して `ForbidEncryptAllowDecrypt` に設定していることを確認してください。次に、データ構成に応じて、自分のペースで `RequireEncryptAllowDecrypt` に移行してからデフォルト設定の `RequireEncryptRequireDecrypt` に移行できます。次のパターンのような一連の移行手順を推奨します。

1. 最初はコミットメントポリシーを ForbidEncryptAllowDecrypt に設定します。 AWS Encryption SDK ではキーコミットメントによるメッセージを復号化できますが、キーコミットメントではまだ暗号化しません。
2. 準備ができたら、コミットメントポリシーを RequireEncryptAllowDecrypt に更新します。は、キーコミットメントを使用してデータの暗号化 AWS Encryption SDK を開始します。キーコミットメントを使用しているかどうかにかかわらず、暗号化テキストを復号化できます。

   コミットメントポリシーを RequireEncryptAllowDecrypt に更新する前に、生成した暗号化テキストを復号するアプリケーションのホストを含め、すべてのホストに最新 1.x バージョンがデプロイされていることを確認します。バージョン 1.7.x AWS Encryption SDK より前のバージョンの では、キーコミットメントで暗号化されたメッセージを復号できません。

   この時点でアプリケーションにメトリクスを追加し、キーコミットメントによらない暗号化テキストをまだ処理しているかどうかを調査することもお勧めします。これにより、いつコミットメントポリシー設定を RequireEncryptRequireDecrypt に更新しても安全かを判断できるようになります。Amazon SQS キュー内のメッセージを暗号化するアプリケーションなど、一部のアプリケーションでは、古いバージョンで暗号化されたすべての暗号化テキストが再暗号化または削除されるのに時間がかかることがあります。暗号化された S3 オブジェクトなどの他のアプリケーションでは、すべてのオブジェクトをダウンロード、再暗号化、および再アップロードする必要がある場合があります。
3. キーコミットメントなしで暗号化されたメッセージがないことが確認できたら、コミットメントポリシーを RequireEncryptRequireDecrypt に更新できます。この値により、キーコミットメントでデータが常に暗号化、復号化されます。この設定はデフォルトであるため、明示的に設定する必要はありませんが、推奨されています。明示的に設定すると、アプリケーションがキーコミットメントなしで暗号化された暗号化テキストを検出した場合に必要となる可能性のあるデバッグとロールバックが容易になります。

# AWS KMS マスターキープロバイダーの更新

の最新の 1.x バージョンに移行し AWS Encryption SDK、次にバージョン 2.0.x 以降に移行するには、レガシー AWS KMS マスターキープロバイダーを Strict モードまたは Discovery モードで明示的に作成されたマスターキープロバイダーに置き換える必要があります。レガシーマスターキープロバイダーは、バージョン 1.7.x で非推奨となり、バージョン 2.0.x で削除されます。この変更は、AWS Encryption SDK for Java、AWS Encryption SDK for Python、AWS Encryption CLI を使用するアプリケーションとスクリプトで必要となります。このセクションの例では、コードの更新方法について説明します。

> **Note**
>
> Python では、非推奨の警告をオンにしてください。コードの更新が必要な部分を特定できるようになります。

AWS KMS マスターキー (マスターキープロバイダーではない) を使用している場合は、このステップをスキップできます。 AWS KMS マスターキーは廃止または削除されません。このマスターキーでは、指定したラッピングキーでのみ暗号化および復号化が行われます。

このセクションの例では、変更する必要があるコードの要素に焦点を当てています。更新されたコードの完全な例については、使用しているプログラミング言語の GitHub リポジトリの例セクションを参照してください。また、これらの例では、通常、キー ARNsします AWS KMS keys。暗号化用のマスターキープロバイダーを作成するときは、任意の有効な AWS KMS キー識別子を使用して を表すことができます AWS KMS key 。復号用のマスターキープロバイダーを作成するときは、キー ARN を使用する必要があります。

移行の詳細

すべての AWS Encryption SDK ユーザーについて、 でのコミットメントポリシーの設定について説明しますthe section called “コミットメントポリシーの設定”。

AWS Encryption SDK for C および AWS Encryption SDK for JavaScript ユーザーについては、 のキーリングに対するオプションの更新について説明しますAWS KMS キーリングの更新。

トピック

- Strict モードへの移行
- Discovery モードへの移行

## Strict モードへの移行

を最新の 1.x バージョンに更新したら AWS Encryption SDK、レガシーマスターキープロバイダーを Strict モードのマスターキープロバイダーに置き換えます。Strict モードでは、暗号化時および復号化時に使用するラッピングキーを指定する必要があります。は、指定したラッピングキーのみ AWS Encryption SDK を使用します。非推奨のマスターキープロバイダーは、異なる AWS アカウント やリージョン AWS KMS keys など、データキーを暗号化 AWS KMS key した を使用してデータを復号できます。

Strict モードのマスターキープロバイダーは、 AWS Encryption SDK バージョン 1.7.x で導入されました。1.7.x で非推奨となって 2.0.x で削除されるレガシーマスターキープロバイダーは置き換えられます。マスターキープロバイダーを Strict モードで使用することが AWS Encryption SDK ベストプラクティスです。

次のコードでは Strict モードでマスターキープロバイダーを作成し、暗号化と復号に使用できるようにしています。

Java

この例は、 AWS Encryption SDK for Javaのバージョン 1.6.2 以前を使用するアプリケーションのコードを表しています。

このコードは、 KmsMasterKeyProvider.builder()メソッドを使用して、ラッピングキー AWS KMS key として マスターキープロバイダーを使用する AWS KMS マスターキープロバイダーをインスタンス化します。

```
// Create a master key provider
// Replace the example key ARN with a valid one
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .withKeysForEncryption(awsKmsKey)
    .build();
```

この例は、 AWS Encryption SDK for Java のバージョン 1.7.x 以降を使用するアプリケーションのコードを表しています。詳しい例については、「BasicEncryptionExample.java」を参照してください。

前の例で使用した Builder.build() および Builder.withKeysForEncryption() メソッドは、バージョン 1.7.x で非推奨となり、バージョン 2.0.x で削除されます。

Strict モードのマスターキープロバイダーに更新するため、このコードでは非推奨メソッドの呼び出しを新しい Builder.buildStrict() メソッドの呼び出しに置き換えます。この例では、ラッピングキー AWS KMS key として 1 つを指定していますが、 Builder.buildStrict()メソッドは複数の のリストを取得できます AWS KMS keys。

```
// Create a master key provider in strict mode
// Replace the example key ARN with a valid one from your AWS #####.
```

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

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);
```

Python

この例は、 AWS Encryption SDK for Pythonのバージョン 1.4.1 を使用するアプリケーションのコードを表しています。このコードでは KMSMasterKeyProvider を使用しますが、これはバージョン 1.7.x で非推奨となり、バージョン 2.0.x から削除されます。復号時には、 AWS KMS keys 指定した に関係なく、データキーを暗号化 AWS KMS key した が使用されます。

KMSMasterKey は非推奨にならず、削除されません。暗号化および復号時には、 AWS KMS key 指定した のみが使用されます。

```
# Create a master key provider
# Replace the example key ARN with a valid one
key_1 = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"

aws_kms_master_key_provider = KMSMasterKeyProvider(
   key_ids=[key_1, key_2]
)
```

この例は、 AWS Encryption SDK for Pythonのバージョン 1.7.x を使用するアプリケーションのコードを表しています。詳しい例については、「basic_encryption.py」を参照してください。

Strict モードのマスターキープロバイダーに更新するため、このコードでは KMSMasterKeyProvider() の呼び出しを StrictAwsKmsMasterKeyProvider() の呼び出しに置き換えます。

```
# Create a master key provider in strict mode
# Replace the example key ARNs with valid values from your AWS #####
key_1 = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
```

```
    key_ids=[key_1, key_2]
)
```

### AWS Encryption CLI

この例では、Encryption CLI バージョン AWS 1.1.7 以前を使用して暗号化および復号する方法を示します。

バージョン 1.1.7 以前では、暗号化時に AWS KMS keyなどの 1 つ以上のマスターキー (ラッピングキー) を指定します。復号化時には、カスタムのマスターキープロバイダーを使用していない限り、ラッピングキーを指定することはできません。 AWS Encryption CLI は、データキーを暗号化した任意のラッピングキーを使用できます。

```
\\ Replace the example key ARN with a valid one
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --master-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

この例では、Encryption CLI バージョン AWS 1.7.x 以降を使用して暗号化および復号する方法を示します。完全な例については、「Encryption AWS CLI の例」を参照してください。

`--master-keys` パラメータはバージョン 1.7.x で非推奨となり、バージョン 2.0.x で削除されます。これは `--wrapping-keys` パラメータに置き換わり、すべての暗号化コマンドと復号コマンドに必要となります。このパラメータでは、Strict モードと Discovery モードがサポートされます。厳格モードは、意図したラッピングキーを使用することを保証する AWS Encryption SDK ベストプラクティスです。

Strict モードにアップグレードするには、`--wrapping-keys`パラメータの key 属性を使用して、暗号化時および復号時のラッピングキーを指定します。

```
\\ Replace the example key ARN with a valid value
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

## Discovery モードへの移行

バージョン 1.7.x 以降では、マスターキープロバイダーに strict モード AWS KMS を使用する、つまり暗号化および復号時にラッピングキーを指定するの AWS Encryption SDK がベストプラクティスです。暗号化するときは、常にラッピングキーを指定する必要があります。ただし、復号 AWS KMS keys のために のキー ARNs を指定するのは実用的ではない場合があります。例えば、エイリアスを使用して暗号化 AWS KMS keys 時に を識別する場合、復号時にキー ARNs を一覧表示する必要がある場合、エイリアスの利点が失われます。また、Discovery モードのマスターキープロバイダーは元のマスターキープロバイダーと同様に動作するため、移行戦略の一部として一時的にそれを使用し、後で Strict モードのマスターキープロバイダーにアップグレードできます。

このような場合は、マスターキープロバイダーを Discovery モードで使用できます。これらのマスターキープロバイダーではラッピングキーを指定できないため、暗号化には使用できません。復号時には、データキーを暗号化したラッピングキーを使用できます。ただし、同じ動作をするレガシーマスターキープロバイダーとは異なり、Discovery モードで明示的に作成します。Discovery モードでマスターキープロバイダーを使用する場合、使用できるラッピングキーを特に AWS アカウントのものに制限できます。この検出フィルターはオプションですが、お勧めのベストプラクティスです。AWS パーティションとアカウントの詳細については、「AWS 全般のリファレンス」の「Amazon リソースネーム」を参照してください。

次の例では、暗号化用の Strict モードで AWS KMS マスターキープロバイダーを作成し、復号用の検出モードで AWS KMS マスターキープロバイダーを作成します。Discovery モードのマスターキープロバイダーは、検出フィルターを使用して、復号に使用するラッピングキーを aws パーティションと特定の AWS アカウント例に制限します。この単純な例ではアカウントフィルターは必要ありませんが、あるアプリケーションがデータを暗号化し、別のアプリケーションがデータを復号化する場合に非常に有益なベストプラクティスです。

Java

この例は、 AWS Encryption SDK for Javaのバージョン 1.7.x 以降を使用するアプリケーションのコードを表しています。詳しい例については、DiscoveryDecryptionExample.java を参照してください。

暗号化では Strict モードでマスターキープロバイダーをインスタンス化するために、この例では Builder.buildStrict() メソッドを使用します。復号では Discovery モードでマスターキープロバイダーをインスタンス化するため、Builder.buildDiscovery() メソッドを使用します。Builder.buildDiscovery() メソッドは、指定された AWS パーティションとアカウント AWS KMS keys で AWS Encryption SDK を DiscoveryFilterに制限する を取ります。

```
// Create a master key provider in strict mode for encrypting
// Replace the example alias ARN with a valid one from your AWS #####.
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias";

KmsMasterKeyProvider encryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

// Create a master key provider in discovery mode for decrypting
// Replace the example account IDs with valid values.
DiscoveryFilter accounts = new DiscoveryFilter("aws", Arrays.asList("111122223333",
 "444455556666"));

KmsMasterKeyProvider decryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildDiscovery(accounts);
```

Python

この例は、 AWS Encryption SDK for Python のバージョン 1.7.x 以降を使用するアプリケーションのコードを表しています。詳しい例については、discovery_kms_provider.py を参照してください。

暗号化では Strict モードでマスターキープロバイダーを作成するために、この例では `StrictAwsKmsMasterKeyProvider` を使用します。復号化のために検出モードでマスターキープロバイダーを作成するには、指定された AWS パーティションとアカウント AWS KMS keys で AWS Encryption SDK を に制限`DiscoveryFilter`する `DiscoveryAwsKmsMasterKeyProvider`で を使用します。

```
# Create a master key provider in strict mode
# Replace the example key ARN and alias ARNs with valid values from your AWS #####.
key_1 = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias"
key_2 = "arn:aws:kms:us-
west-2:444455556666:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[key_1, key_2]
)

# Create a master key provider in discovery mode for decrypting
# Replace the example account IDs with valid values
accounts = DiscoveryFilter(
    partition="aws",
    account_ids=["111122223333", "444455556666"]
)
aws_kms_master_key_provider = DiscoveryAwsKmsMasterKeyProvider(
        discovery_filter=accounts
)
```

### AWS Encryption CLI

この例では、Encryption CLI バージョン AWS 1.7.x 以降を使用して暗号化および復号する方法を示します。バージョン 1.7.x 以降は、`--wrapping-keys` パラメータが暗号化および復号化時に必要となります。`--wrapping-keys` パラメータでは、Strict モードと Discovery モードがサポートされます。完全な例については、「the section called “例”」を参照してください。

この例では、暗号化時に必須のラッピングキーを指定します。復号化時には、`--wrapping-keys` パラメータの `discovery` 属性の値を `true` にして、Discovery モードを明示的に選択します。

が検出モードで AWS Encryption SDK 使用できるラッピングキーを特にそれらに制限するために AWS アカウント、この例では `--wrapping-keys`パラメータの 属性`discovery-partition`と `discovery-account` 属性を使用します。これらのオプションの属性は、`discovery` 属性を `true` に設定しているときに限って有効です。`discovery-partition`

属性と discovery-account 属性は一緒に使用する必要があります。単独では有効ではありません。

```
\\ Replace the example key ARN with a valid value
$ keyAlias=arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyAlias \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
\\ Replace the example account IDs with valid values
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys discovery=true \
                                     discovery-partition=aws \
                                     discovery-account=111122223333 \
                                     discovery-account=444455556666 \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

# AWS KMS キーリングの更新

の AWS KMS キーリングAWS Encryption SDK for C、 AWS Encryption SDK for .NET、および は、暗号化および復号時にラッピングキーを指定できるようにすることで、ベストプラクティスAWS Encryption SDK for JavaScriptをサポートします。AWS KMS 検出キーリングを作成する場合は、明示的に作成します。

> **Note**
>
> .NET AWS Encryption SDK 用 の最も古いバージョンはバージョン 3.0.x です。for AWS Encryption SDK .NET のすべてのバージョンは、 の 2.0.x で導入されたセキュリティのベストプラクティスをサポートしています AWS Encryption SDK。コードやデータを変更することなく、最新バージョンに安全にアップグレードできます。

を最新の 1.x バージョンに更新すると AWS Encryption SDK、検出フィルターを使用して、 AWS KMS 検出キーリングまたはAWS KMS リージョン検出キーリングが復号するときに使用するラッピングキーを、特にそれらに制限できます AWS アカウント。検出キーリングのフィルタリングは AWS Encryption SDK ベストプラクティスです。

このセクションの例では、検出フィルターを AWS KMS リージョン検出キーリングに追加する方法を示します。

移行の詳細

すべての AWS Encryption SDK ユーザーについて、 でのコミットメントポリシーの設定について説明しますthe section called “コミットメントポリシーの設定”。

AWS Encryption SDK for Java、 AWS Encryption SDK for Python、および AWS Encryption CLI ユーザーの場合、 のマスターキープロバイダーに必要な更新について説明しますthe section called “AWS KMS マスターキープロバイダーの更新”。

アプリケーションでは、コードは次のようなものになります。この例では、米国西部 (オレゴン) (us-west-2) リージョンのラッピングキーのみを使用する AWS KMS リージョン検出キーリングを作成します。この例では、1.7.x より前の AWS Encryption SDK バージョンのコードを表します。ただし、バージョン 1.7.x 以降でも有効です。

C

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

JavaScript Browser

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

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser({ clientProvider, discovery })
```

JavaScript Node.js

```
const discovery = true
```

```
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({ clientProvider, discovery })
```

バージョン 1.7.x 以降では、任意の検出キーリングに AWS KMS 検出フィルターを追加できます。この検出フィルター AWS Encryption SDK は、 AWS KMS keys が復号に使用できる を、指定されたパーティションとアカウント内のものに制限します。このコードを使用する前に、必要に応じてパーティションを変更し、サンプルアカウント ID を有効なアカウント ID に置き換えます。

C

詳しい例については、kms_discovery.cpp を参照してください。

```
std::shared_ptr<KmsKeyring::DiscoveryFilter> discovery_filter(
    KmsKeyring::DiscoveryFilter::Builder("aws")
        .AddAccount("111122223333")
        .AddAccount("444455556666")
        .Build());

struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()

 .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery(discovery_filter))
```

JavaScript Browser

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

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

JavaScript Node.js

詳しい例については、kms_filtered_discovery.ts を参照してください。

```
const discovery = true
```

```
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

# コミットメントポリシーの設定

キーコミットメントにより、暗号化されたデータは常に同じプレーンテキストに復号化されるようになります。このセキュリティプロパティを提供するために、 バージョン 1.7.x 以降、 はキーコミットメントで新しいアルゴリズムスイート AWS Encryption SDK を使用します。データをキーコミットメントで暗号化および復号化するかどうかを決めるには、コミットメントポリシー構成設定を使用します。キーコミットメントによるデータの暗号化と復号化は、AWS Encryption SDK のベストプラクティスです。

コミットメントポリシーの設定は、 の最新の 1.x バージョンから AWS Encryption SDK バージョン 2.0.x 以降への移行プロセスにおける 2 番目のステップの重要な部分です。コミットメントポリシーを設定および変更したら、アプリケーションを徹底的にテストしてから本番環境にデプロイしてください。移行ガイダンスについては、「を移行およびデプロイする方法 AWS Encryption SDK」を参照してください。

バージョン 2.0.x 以降に、コミットメントポリシー設定には 3 つの有効な値があります。最新の 1.x バージョン (バージョン 1.7.x 以降) では、`ForbidEncryptAllowDecrypt` のみ有効です。

- `ForbidEncryptAllowDecrypt` — はキーコミットメントで暗号化 AWS Encryption SDK できません。キーコミットメントが使用されているかどうかにかかわらず、暗号化された暗号化テキストを復号化できます。

  最新の 1.x バージョンでは、これが唯一の有効な値です。これにより、キーコミットメントで復号化する準備が完全に整うまで、キーコミットメントで暗号化しないようになります。値を明示的に設定すると、バージョン 2.0.x 以降へアップグレード時にコミットメントポリシーが自動的に `require-encrypt-require-decrypt` へ変更されるのを防ぎます。その代わりに、コミットメントポリシーを段階的に移行できます。
- `RequireEncryptAllowDecrypt` — AWS Encryption SDK は常にキーコミットメントで暗号化します。キーコミットメントが使用されているかどうかにかかわらず、暗号化された暗号化テキストを復号化できます。この値はバージョン 2.0.x で追加されました。

- `RequireEncryptRequireDecrypt` — AWS Encryption SDK は常にキーコミットメントで暗号化および復号します。この値はバージョン 2.0.x で追加されました。バージョン 2.0.x 以降では、これがデフォルト値です。

最新の 1.x バージョンでは、唯一の有効なコミットメントポリシー値は `ForbidEncryptAllowDecrypt` です。バージョン 2.0.x 以降に移行した後、準備が整ったら、コミットメントポリシーを段階的に変更できます。すべてのメッセージがキーコミットメントで暗号化されることが確認できるまで、コミットメントポリシーを `RequireEncryptRequireDecrypt` に更新しないでください。

これらの例では、最新の 1.x バージョンおよび バージョン 2.0.x 以降でコミットメントポリシーを設定する方法を示します。この手法はプログラミング言語によって異なります。

移行の詳細

AWS Encryption SDK for Java AWS Encryption SDK for Pythonおよび AWS Encryption CLI で、 のマスターキープロバイダーに必要な変更について説明しますthe section called “AWS KMS マスターキープロバイダーの更新”。

AWS Encryption SDK for C および については AWS Encryption SDK for JavaScript、 のキーリングに対するオプションの更新について説明しますAWS KMS キーリングの更新。

## コミットメントポリシーの設定方法

コミットメントポリシーの設定に使用する手法は、言語実装ごとに若干異なります。以下の例ではその方法を示します。コミットメントポリシーを変更する前に、「移行してデプロイする方法」で多段階のアプローチを確認してください。

C

のバージョン 1.7.x 以降では AWS Encryption SDK for C、`aws_cryptosdk_session_set_commitment_policy`関数を使用して、暗号化および復号セッションにコミットメントポリシーを設定します。設定したコミットメントポリシーは、そのセッションで呼び出されるすべての暗号化および復号オペレーションに適用されます。

`aws_cryptosdk_session_new_from_keyring` 関数および `aws_cryptosdk_session_new_from_cmm` 関数は、バージョン 1.7.x で非推奨となり、バージョン 2.0.x で削除されます。これらの関数は、セッションを返す `aws_cryptosdk_session_new_from_keyring_2` 関数および `aws_cryptosdk_session_new_from_cmm_2` 関数に置き換わります。

最新の 1.x バージョンで aws_cryptosdk_session_new_from_keyring_2 および aws_cryptosdk_session_new_from_cmm_2 を使用している場合は、COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT コミットメントポリシーの値で aws_cryptosdk_session_set_commitment_policy 関数を呼び出す必要があります。バージョン 2.0.x 以降では、この関数の呼び出しはオプションであり、すべての有効な値を取ります。バージョン 2.0.x 以降のデフォルトコミットポリシーは COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT です。

詳しい例については、string.cpp を参照してください。

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

/* Create an AWS KMS keyring */
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* Create an encrypt session with a CommitmentPolicy setting */
struct aws_cryptosdk_session *encrypt_session =
 aws_cryptosdk_session_new_from_keyring_2(
    alloc, AWS_CRYPTOSDK_ENCRYPT, kms_keyring);

aws_cryptosdk_keyring_release(kms_keyring);
aws_cryptosdk_session_set_commitment_policy(encrypt_session,
    COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);

...
/* Encrypt your data */

size_t plaintext_consumed_output;
aws_cryptosdk_session_process(encrypt_session,
                              ciphertext_output,
                              ciphertext_buf_sz_output,
                              ciphertext_len_output,
                              plaintext_input,
                              plaintext_len_input,
                              &plaintext_consumed_output)
...

/* Create a decrypt session with a CommitmentPolicy setting */
```

```
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
struct aws_cryptosdk_session *decrypt_session =
 *aws_cryptosdk_session_new_from_keyring_2(
        alloc, AWS_CRYPTOSDK_DECRYPT, kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);
aws_cryptosdk_session_set_commitment_policy(decrypt_session,
        COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);

/* Decrypt your ciphertext */
size_t ciphertext_consumed_output;
aws_cryptosdk_session_process(decrypt_session,
                              plaintext_output,
                              plaintext_buf_sz_output,
                              plaintext_len_output,
                              ciphertext_input,
                              ciphertext_len_input,
                              &ciphertext_consumed_output)
```

C# / .NET

`require-encrypt-require-decrypt` 値は、.NET AWS Encryption SDK 用 のすべてのバージョンのデフォルトのコミットメントポリシーです。ベストプラクティスとして明示的に設定することもできますが、必須ではありません。ただし、 AWS Encryption SDK for .NET を使用して、キーコミットメント AWS Encryption SDK なしで の別の言語実装によって暗号化された暗号文を復号する場合は、コミットメントポリシー値を `REQUIRE_ENCRYPT_ALLOW_DECRYPT`または は に変更する必要がありますFORBID_ENCRYPT_ALLOW_DECRYPT。含まれていない場合、暗号文の復号は失敗します。

.NET AWS Encryption SDK 用 では、 のインスタンスにコミットメントポリシーを設定します AWS Encryption SDK。`CommitmentPolicy` パラメータを使用してAwsEncryptionSdkConfigオブジェクトをインスタンス化し、設定オブジェクトを使用して AWS Encryption SDK インスタンスを作成します。次に、設定された AWS Encryption SDK インスタンスの `Encrypt()`および `Decrypt()`メソッドを呼び出します。

この例では、コミットメントポリシーを `require-encrypt-allow-decrypt` に設定します。

```
// Instantiate the material providers
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();
```

```
// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{
    CommitmentPolicy = CommitmentPolicy.REQUIRE_ENCRYPT_ALLOW_DECRYPT
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

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

var encryptionContext = new Dictionary<string, string>()
{
    {"purpose", "test"}encryptionSdk
};

var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var keyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

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

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

### AWS Encryption CLI

Encryption CLI AWS でコミットメントポリシーを設定するには、 `--commitment-policy`パラメータを使用します。このパラメータはバージョン 1.8.x で導入されました。

最新の 1.x バージョンでは、--encrypt または --decrypt コマンドの --wrapping-keys パラメータを使用するときに、forbid-encrypt-allow-decrypt 値 を持つ --commitment-policy パラメータが必要です。そうでない場合、--commitment-policy パラメータは無効です。

バージョン 2.1.x 以降の場合、--commitment-policy パラメータはオプションであり、デフォルトで require-encrypt-require-decrypt 値になっており、キーコミットメントなしで暗号化された暗号化テキストは暗号化または復号されません。ただし、メンテナンスとトラブルシューティングに役立つように、すべての暗号化および復号呼び出しでコミットメントポリシーを明示的に設定することをお勧めします。

この例ではコミットメントポリシーを設定します。また、バージョン 1.8.x 以降、--master-keys パラメータを置き換える --wrapping-keys パラメータを使用します。詳細については、「the section called “AWS KMS マスターキープロバイダーの更新”」を参照してください。完全な例については、「Encryption AWS CLI の例」を参照してください。

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

\\ Encrypt your plaintext data - no change to algorithm suite used
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy forbid-encrypt-allow-decrypt \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext - supports key commitment on 1.7 and later
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy forbid-encrypt-allow-decrypt \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

Java

のバージョン 1.7.x 以降では AWS Encryption SDK for Java、 AWS Encryption SDK クライアントを表す オブジェクトAwsCryptoである のインスタンスにコミットメントポリシーを設定しま

す。このコミットメントポリシーは、そのクライアントで呼び出されるすべての暗号化および復号オペレーションに適用されます。

AwsCrypto() コンストラクタは、 の最新の 1.x バージョンでは非推奨 AWS Encryption SDK for Java となり、バージョン 2.0.x では削除されます。これは、新しい Builder クラス、Builder.withCommitmentPolicy() メソッド、CommitmentPolicy 列挙型に置き換わります。

最新 1.x バージョンでは、Builder クラスには Builder.withCommitmentPolicy() メソッドと CommitmentPolicy.ForbidEncryptAllowDecrypt 引数が必要です。バージョン 2.0.x 以降、Builder.withCommitmentPolicy() メソッドはオプションであり、デフォルト値は CommitmentPolicy.RequireEncryptRequireDecrypt です。

詳しい例については、「SetCommitmentPolicyExample.java」を参照してください。

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

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

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

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

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

### JavaScript

のバージョン 1.7.x 以降では AWS Encryption SDK for JavaScript、 AWS Encryption SDK クライアントをインスタンス化する新しい `buildClient`関数を呼び出すときに、コミットメントポリシーを設定できます。`buildClient` 関数は、コミットメントポリシーを表す列挙値を取ります。更新された `encrypt` 関数と `decrypt` 関数が返されて、暗号化および復号化時にコミットメントポリシーが適用されます。

最新 1.x バージョンでは、`buildClient` 関数には `CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT` 引数が必要です。バージョン 2.0.x 以降、コミットメントポリシー引数はオプションであり、デフォルト値は `CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT` です。

Node.js とブラウザのコードは、ブラウザが認証情報を設定するためのステートメントを必要とする点を除いて、この目的では同じです。

次の例では、 AWS KMS キーリングを使用してデータを暗号化します。新しい `buildClient` 関数は、コミットメントポリシーを最新 1.x バージョンでのデフォルト値の `FORBID_ENCRYPT_ALLOW_DECRYPT` に設定します。`buildClient` が返すアップグレード済みの `encrypt` 関数と `decrypt` 関数では、設定したコミットメントポリシーが適用されます。

```
import { buildClient } from '@aws-crypto/client-node'
const { encrypt, decrypt } =
 buildClient(CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT)

// Create an AWS KMS keyring
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias'
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

// Encrypt your plaintext data
const { ciphertext } = await encrypt(keyring, plaintext, { encryptionContext:
 context })

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

### Python

のバージョン 1.7.x 以降では AWS Encryption SDK for Python、 AWS Encryption SDK クライアントを表す新しいオブジェクト`EncryptionSDKClient`である のインスタンスにコミットメン

トポリシーを設定します。設定したコミットメントポリシーは、クライアントのインスタンスを使用するすべての encrypt 呼び出しと decrypt 呼び出しに適用されます。

最新 1.x バージョンでは、EncryptionSDKClient コンストラクタには CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT 列挙値が必要です。バージョン 2.0.x 以降、コミットメントポリシー引数はオプションであり、デフォルト値は CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT です。

この例では新しい EncryptionSDKClient コンストラクタを使用して、コミットメントポリシーを 1.7.x のデフォルト値に設定します。コンストラクタは、 AWS Encryption SDKを表すクライアントをインスタンス化します。このクライアントで、encrypt、decrypt、stream のいずれかのメソッドを呼び出すと、設定したコミットメントポリシーが適用されます。この例では、 StrictAwsKmsMasterKeyProvider クラスの新しいコンストラクタも使用します。このコンストラクタは、暗号化および復号 AWS KMS keys 時に を指定します。

詳しい例については、「set_commitment.py」を参照してください。

```
# Instantiate the client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.FORBID_ENCRYPT_AL

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

# Encrypt your plaintext data
ciphertext, encrypt_header = client.encrypt(
        source=source_plaintext,
        encryption_context=encryption_context,
        master_key_provider=aws_kms_strict_master_key_provider
)

# Decrypt your ciphertext
decrypted, decrypt_header = client.decrypt(
        source=ciphertext,
        master_key_provider=aws_kms_strict_master_key_provider
)
```

Rust

`require-encrypt-require-decrypt` 値は、 for AWS Encryption SDK Rust のすべてのバージョンのデフォルトのコミットメントポリシーです。ベストプラクティスとして明示的に設定することもできますが、必須ではありません。ただし、 AWS Encryption SDK for Rust を使用して、キーコミットメント AWS Encryption SDK なしで の別の言語実装によって暗号化された暗号文を復号する場合は、コミットメントポリシー値を `REQUIRE_ENCRYPT_ALLOW_DECRYPT`または は に変更する必要がありますFORBID_ENCRYPT_ALLOW_DECRYPT。含まれていない場合、暗号文の復号は失敗します。

AWS Encryption SDK for Rust で、 のインスタンスにコミットメントポリシーを設定します AWS Encryption SDK。`comitment_policy` パラメータを使用して`AwsEncryptionSdkConfig`オブジェクトをインスタンス化し、設定オブジェクトを使用して AWS Encryption SDK インスタンスを作成します。次に、設定された AWS Encryption SDK インスタンスの `Encrypt()`および `Decrypt()`メソッドを呼び出します。

この例では、コミットメントポリシーを `forbid-encrypt-allow-decrypt` に設定します。

```
// Configure the commitment policy on the AWS Encryption SDK instance
let esdk_config = AwsEncryptionSdkConfig::builder()
                    .commitment_policy(ForbidEncryptAllowDecrypt)
                    .build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

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

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

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

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

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

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

// Decrypt your ciphertext
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

```
)

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

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

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}

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

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

```
    panic(err)
}

// Encrypt your plaintext data
res, err := forbidEncryptClient.Encrypt(context.Background(),
 esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}

// Decrypt your ciphertext
decryptOutput, err := forbidEncryptClient.Decrypt(context.Background(),
 esdktypes.DecryptInput{
    Ciphertext:        res.Ciphertext,
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}
```

# 最新バージョンへの移行に関するトラブルシューティング

アプリケーションをバージョン 2.0.x 以降の に更新する前に AWS Encryption SDK、 を最新の 1.x バージョンに更新 AWS Encryption SDK して完全にデプロイします。これにより、バージョン 2.0.x 以降への更新時に発生する可能性のあるほとんどのエラーを回避できます。例を含む詳細なガイダンスについては、「の移行 AWS Encryption SDK」を参照してください。

> **Important**
>
> 最新 1.x バージョンが AWS Encryption SDKのバージョン 1.7.x 以降であることを確認します。

> **Note**
>
> AWS Encryption CLI: このガイドの のバージョン 1.7.x への参照は、 Encryption AWS CLI のバージョン 1.8.x AWS Encryption SDK に適用されます。このガイドの のバージョン 2.0.x への参照は、 Encryption CLI のバージョン AWS 2.1.x AWS Encryption SDK に適用されます。
> 新しいセキュリティ機能は、もともと AWS Encryption CLI バージョン 1.7.x および 2.0.x でリリースされました。ただし、 AWS Encryption CLI バージョン 1.8.x はバージョン 1.7.x に置き換わり、Encryption AWS CLI 2.1.x は 2.0.x に置き換わります。詳細については、GitHub の aws-encryption-sdk-cli リポジトリで関連するセキュリティアドバイザリを参照してください。

このトピックは、発生する可能性のある最も一般的なエラーを認識し、解決するのに役立つように設計されています。

トピック

- 非推奨または削除されたオブジェクト
- 構成の競合: コミットメントポリシーとアルゴリズムスイート
- 構成の競合: コミットメントポリシーと暗号化テキスト
- キーコミットメントの検証の失敗
- その他の暗号化の失敗
- その他の復号化の失敗
- ロールバックに関する考慮事項

## 非推奨または削除されたオブジェクト

バージョン 2.0.x には、バージョン 1.7.x で非推奨になったレガシーコンストラクタ、メソッド、関数、クラスの削除など、いくつかの重大な変更が含まれています。コンパイラエラー、インポートエラー、構文エラー、記号が見つかりませんエラー (プログラミング言語によって異なります) を回避するには、まずプログラミング言語の の最新バージョン 1.x AWS Encryption SDK にアップグレードします。(これはバージョン 1.7.x 以降である必要があります。) 最新 1.x バージョンを使用中、元のシンボルが削除される前に置換要素の使用を開始できます。

バージョン 2.0.x にすぐにアップグレードする必要がある場合は、使用中のプログラミング言語の変更履歴を参照し、レガシーシンボルを変更履歴が推奨するシンボルに置き換えます。

# 構成の競合: コミットメントポリシーとアルゴリズムスイート

コミットメントポリシーと競合するアルゴリズムスイートを指定した場合は、暗号化の呼び出しが構成の競合というエラーで失敗します。

このタイプのエラーを回避するには、アルゴリズムスイートを指定しないでください。デフォルトでは、 AWS Encryption SDK は、コミットメントポリシーと互換性のある最も安全なアルゴリズムを選択します。ただし、署名なしなどのアルゴリズムスイートを指定する必要がある場合は、コミットメントポリシーと互換性のあるアルゴリズムスイートを必ず選択してください。

| コミットメントポリシー | 互換性のあるアルゴリズムスイート |
| --- | --- |
| ForbidEncryptAllowDecrypt | 次のようにキーコミットメントのないアルゴリズムスイート<br>`AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384` (03 78) (署名付き)<br><br>`AES_256_GCM_IV12_TAG16_HKDF_SHA256` (01 78) (署名なし) |
| RequireEncryptAllowDecrypt<br><br>RequireEncryptRequireDecrypt | 次のようにキーコミットメントのあるアルゴリズムスイート<br>`AES_256_GCM_HKDF_SHA512_COMMIT_KEY_ECDSA_P384` (05 78) (署名付き)<br><br>`AES_256_GCM_HKDF_SHA512_COMMIT_KEY` (04 78) (署名なし) |

アルゴリズムスイートを指定していないときにこのエラーが発生した場合は、競合するアルゴリズムスイートが暗号化マテリアルマネージャー (CMM) によって選択されている可能性があります。デフォルト CMM では、競合するアルゴリズムスイートは選択されませんが、カスタム CMM はそれを選択する可能性があります。ヘルプについては、カスタム CMM のドキュメントを参照してください。

## 構成の競合: コミットメントポリシーと暗号化テキスト

`RequireEncryptRequireDecrypt` コミットメントポリシーでは、 AWS Encryption SDK はキーコミットメントなしで暗号化されたメッセージを復号できません。キーコミットメントなしでメッセージを復号 AWS Encryption SDK するように に要求すると、設定競合エラーが返されます。

このエラーを回避するには、`RequireEncryptRequireDecrypt` コミットメントポリシーの設定前に、キーコミットメントなしで暗号化されたすべての暗号化テキストを復号してキーコミットメントありで再暗号化するか、別のアプリケーションによって処理してください。このエラーが発生した場合は、競合する暗号化テキストのエラーを返すか、コミットメントポリシーを一時的に `RequireEncryptAllowDecrypt` に変更できます。

まず最新 1.x バージョン (バージョン 1.7.x 以降) へにアップグレードせずに、1.7.x より前のバージョンからバージョン 2.0.x 以降へアップグレードしたためにこのエラーが発生した場合、バージョン 2.0.x 以降にアップグレードする前に、最新 1.x バージョンにロールバックし、そのバージョンをすべてのホストにデプロイすることを検討してください。ヘルプについては、「を移行およびデプロイする方法 AWS Encryption SDK」を参照してください。

## キーコミットメントの検証の失敗

キーコミットメントで暗号化されたメッセージを復号するとき、「キーコミットメントの検証に失敗しました」というエラーメッセージが表示される場合があります。これは、暗号化されたメッセージのデータキーがメッセージの一意のデータキーと同一ではないために復号呼び出しが失敗したことを示します。復号時にデータキーを検証すると、キーコミットメントにより、複数のプレーンテキストが生成される可能性のあるメッセージを復号化しないように保護されます。

このエラーは、復号しようとした暗号化されたメッセージが、 AWS Encryption SDKによって返されなかったことを示します。これは、手動で作成されたメッセージであるか、データ破損の結果である可能性があります。このエラーが発生した場合、アプリケーションはメッセージを拒否して続行するか、新しいメッセージの処理を停止できます。

## その他の暗号化の失敗

暗号化は複数の理由で失敗する可能性があります。Discovery モードでは、AWS KMS 検出キーリングまたはマスターキープロバイダーを使用してメッセージを暗号化できません。

使用許可のあるラッピングキーを含むキーリングまたはマスターキープロバイダーを暗号化に指定してください。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「キーポリシーの表示」および「 へのアクセスの確認 AWS KMS key」を参照してください。

## その他の復号化の失敗

暗号化されたメッセージを復号化しようとして失敗した場合は、 AWS Encryption SDK がメッセージ内の暗号化されたデータキーを復号できなかった (またはしなかった) ことを示します。

ラッピングキーを指定するキーリングまたはマスターキープロバイダーを使用した場合、 AWS Encryption SDK は指定したラッピングキーのみを使用します。意図したラッピングキーを使用していて、ラッピングキーの少なくとも 1 つに対する `kms:Decrypt` 権限があることを確認してください。を使用している場合は AWS KMS keys、フォールバックとして、検出モードで AWS KMS 検出キーリングまたはマスターキープロバイダーを使用してメッセージを復号化できます。オペレーションが成功した場合は、プレーンテキストを返す前に、メッセージの復号に使用されるキーが信頼できるキーであることを確認します。

## ロールバックに関する考慮事項

アプリケーションがデータの暗号化または復号化に失敗した場合は、通常、コードシンボル、キーリング、マスターキープロバイダー、またはコミットメントポリシーを更新すると問題が解決することがあります。ただし、場合によっては、アプリケーションを以前のバージョンの AWS Encryption SDKにロールバックすることが最善であると判断することもあります。

ロールバックする必要がある場合は、注意して実行してください。1.7.x AWS Encryption SDK より前のバージョンの では、キーコミットメントで暗号化された暗号文を復号できません。

- 最新 1.x バージョンから AWS Encryption SDK の前のバージョンにロールバックすることは、一般的には安全です。以前のバージョンでサポートされていないシンボルやオブジェクトを使用するには、コードに加えた変更を元に戻さなければならない場合があります。
- バージョン 2.0.x 以降でキーコミットメントによる暗号化を開始した場合は (コミットメントポリシーを `RequireEncryptAllowDecrypt` に設定した場合は)、バージョン 1.7.x にロールバックできますが、それより前のバージョンにはロールバックできません。1.7.x AWS Encryption SDK より前のバージョンの では、キーコミットメントで暗号化された暗号文を復号できません。

すべてのホストがキーコミットメントで復号化できるようになる前に、誤ってキーコミットによる暗号化を有効にした場合は、ロールバックするのではなく、ロールアウトを続行することをお勧めします。メッセージが一時的であるか、安全にドロップできる場合は、メッセージの損失を伴うロールバックを検討してください。ロールバックが必要な場合は、すべてのメッセージを復号化して再暗号化するツールを作成することを検討してください。

# よくある質問

よくある質問


- は AWS SDKs とどのように AWS Encryption SDK 異なりますか？
- は Amazon S3 暗号化クライアントとどのように AWS Encryption SDK 異なりますか？
- でサポートされている暗号化アルゴリズム AWS Encryption SDKとデフォルトはどれですか？
- 初期化ベクター (IV) はどのように生成され、どこに保存されますか?
- 各データキーはどのように生成、暗号化、および復号されますか?
- データを暗号化するために使用されたデータキーを追跡するにはどうすればよいですか?
- では、暗号化されたデータキーを暗号化されたデータと共に AWS Encryption SDK 保存する方法を教えてください。
- AWS Encryption SDK メッセージ形式は暗号化されたデータにどのくらいのオーバーヘッドを追加しますか？
- 独自のマスターキープロバイダーを使用できますか?
- 複数のラッピングキーでデータを暗号化できますか?
- どのデータ型を で暗号化できますか AWS Encryption SDK？
- は入出力 (I/O) ストリームをどのように AWS Encryption SDK 暗号化および復号しますか？


## は AWS SDKs とどのように AWS Encryption SDK 異なりますか？

AWS SDKs は、 () を含む Amazon Web Services (AWS) とやり取りするためのライブラリを提供します AWS Key Management Service AWS KMS。for .NET など AWS Encryption SDK、 の一部の言語実装では、常に同じプログラミング言語で AWS SDK が必要です。 AWS Encryption SDK他の言語実装では、キーリングまたはマスターキープロバイダーでキーを使用する AWS KMS 場合にのみ、対応する AWS SDK が必要です。詳細については、AWS Encryption SDK プログラミング言語のプログラミング言語のトピックを参照してください。

AWS SDKs を使用して、少量のデータ (対称暗号化キーで最大 4,096 バイト) の暗号化と復号化 AWS KMS、クライアント側の暗号化用のデータキーの生成など、 を操作できます。ただし、データキーを生成するときは、暗号化と復号プロセス全体を管理する必要があります。これには、外部のデータキーを使用したデータの暗号化 AWS KMS、プレーンテキストのデータキーの安全な破棄、暗号化されたデータキーの保存、データキーの復号とデータの復号が含まれます。 AWS Encryption SDK が、このプロセスを処理します。

AWS Encryption SDK には、業界標準とベストプラクティスを使用してデータを暗号化および復号するライブラリが用意されています。データキーを生成し、指定したラッピングキーで暗号化し、暗号化されたメッセージ、暗号化されたデータと復号化に必要な暗号化データキーを含むポータブル・データ・オブジェクトを返します。復号するときは、暗号化されたメッセージと少なくとも 1 つのラッピングキー (オプション) を渡します。 はプレーンテキストデータ AWS Encryption SDK を返します。

でラッピングキー AWS KMS keys として を使用できますが AWS Encryption SDK、必須ではありません。自分で生成した暗号化キーと、キーマネージャまたはオンプレミスのハードウェアセキュリティモジュールから生成した暗号化キーを使用できます。 AWS アカウントがない場合 AWS Encryption SDK でも、 を使用できます。

## は Amazon S3 暗号化クライアントとどのように AWS Encryption SDK 異なりますか？

AWS SDKs の Amazon S3 暗号化クライアントは、Amazon Simple Storage Service (Amazon S3) に保存するデータの暗号化と復号化を提供します。これらのクライアントは、Amazon S3 と緊密に連携しており、そこに格納されているデータにのみ使用されることを意図しています。

AWS Encryption SDK は、どこにでも保存できるデータの暗号化と復号を提供します。 AWS Encryption SDK と Amazon S3 暗号化クライアントは、異なるデータ形式で暗号文を生成するため、互換性がありません。

## でサポートされている暗号化アルゴリズム AWS Encryption SDKとデフォルトはどれですか？

AWS Encryption SDK は、AES-GCM と呼ばれる Galois/Counter Mode (GCM) の Advanced Encryption Standard (AES) 対称アルゴリズムを使用してデータを暗号化します。データを暗号化するデータキーは、複数ある対称および非対称のアルゴリズムから選択することができます。

AES-GCM の場合、デフォルトのアルゴリズムスイートは、256 ビットのキー、キー取得 (HKDF)、デジタル署名、およびキーコミットメントを持つ AES-GCM です。 は、デジタル署名とキーコミットメントなしで 192 ビットおよび 128 ビットの暗号化キーと暗号化アルゴリズム AWS Encryption SDK もサポートしています。

すべてのケースで、初期化ベクター (IV) の長さは 12 バイトで、認証タグの長さは 16 バイトです。デフォルトでは、SDK はデータキーを HMAC ベースの抽出および展開キー取得関数 (HKDF) への入

力として使用して AES-GCM 暗号化キーを取得します。また、Elliptic Curve Digital 署名アルゴリズム (ECDSA) 署名を追加します。

使用するアルゴリズムの選択については、「サポートされているアルゴリズムスイート」を参照してください。

サポートされているアルゴリズムの実装の詳細については、「アルゴリズムのリファレンス」を参照してください。

## 初期化ベクター (IV) はどのように生成され、どこに保存されますか?

は決定論的な方法 AWS Encryption SDK を使用して、フレームごとに異なる IV 値を構築します。この手順により、メッセージ内で IV が繰り返されないことが保証されます。( AWS Encryption SDK for Java および のバージョン 1.3.0 以前は AWS Encryption SDK for Python、 はフレームごとに一意の IV 値を AWS Encryption SDK ランダムに生成していました )。

IV は、 が AWS Encryption SDK 返す暗号化されたメッセージに保存されます。詳細については、「AWS Encryption SDK メッセージ形式のリファレンス」を参照してください。

## 各データキーはどのように生成、暗号化、および復号されますか?

この方法は、使用するキーリングまたはマスターキープロバイダーによって異なります。

の AWS KMS キーリングとマスターキープロバイダーは、 AWS KMS GenerateDataKey API オペレーション AWS Encryption SDK を使用して各データキーを生成し、ラッピングキーで暗号化します。追加の KMS キーでデータキーのコピーを暗号化するには、 AWS KMS Encrypt オペレーションを使用します。データキーを復号するには、 AWS KMS Decrypt オペレーションを使用します。詳細については、GitHub の AWS Encryption SDK 「仕様」のAWS KMS 「キーリング」を参照してください。

他のキーリングは、各プログラミング言語のベストプラクティスメソッドを使用して、データキーを生成、暗号化、復号化します。詳細については、GitHub の「仕様」の「フレームワーク」セクションの「キーリングまたはマスターキープロバイダーの AWS Encryption SDK 仕様」を参照してください。

## データを暗号化するために使用されたデータキーを追跡するにはどうすればよいですか?

AWS Encryption SDK がこれを行います。データを暗号化する場合、SDK によってデータキーが暗号化され、返された暗号化されたメッセージの暗号化されたデータと共に暗号化されたキーが保存されます。データを復号する場合、 AWS Encryption SDK は暗号化されたメッセージから暗号化されたデータキーを抽出し、それを復号してデータの復号に使用します。

## では、暗号化されたデータキーを暗号化されたデータと共に AWS Encryption SDK 保存する方法を教えてください。

の暗号化オペレーションは、暗号化されたメッセージ、暗号化されたデータとその暗号化されたデータキーを含む単一のデータ構造 AWS Encryption SDK を返します。メッセージ形式は少なくとも 2 つの部分 (ヘッダーと本文) で構成されます。メッセージヘッダーには、暗号化されたデータキーと、メッセージ本文の構成に関する情報が含まれています。メッセージ本文には、暗号化データが含まれます。アルゴリズムスイートにデジタル署名が含まれる場合、メッセージ形式には、署名を含むフッターが含まれます。詳細については、「AWS Encryption SDK メッセージ形式のリファレンス」を参照してください。

## AWS Encryption SDK メッセージ形式は暗号化されたデータにどのくらいのオーバーヘッドを追加しますか？

によって追加されるオーバーヘッドの量は、次のようないくつかの要因 AWS Encryption SDK によって異なります。

- プレーンテキストデータのサイズ
- どのサポートされているアルゴリズムが使用されているか
- 追加認証データ (AAD) が提供されているかどうか、およびその AAD の長さ
- ラッピングキーまたはマスターキーの数と種類
- フレームサイズ (フレームデータが使用される場合)

デフォルト設定 (1 つはラッピングキー (またはマスターキー ) 、AAD なし、非フレームデータ、署名付き暗号化アルゴリズム AWS KMS key ) AWS Encryption SDK で を使用する場合、オーバーヘッドは約 600 バイトです。一般的に、 AWS Encryption SDK で生じるオーバーヘッドは、AAD がない

場合で 1 KB 以下と考えることができます。詳細については、「AWS Encryption SDK メッセージ形式のリファレンス」を参照してください。

## 独自のマスターキープロバイダーを使用できますか?

はい。実装の詳細は、サポートされているプログラミング言語のどれを使用するかによって異なります。ただし、サポートされているすべての言語では、カスタム暗号化マテリアルマネージャー (CMMs)Ms)、マスターキープロバイダー、キーリング、マスターキー、ラッピングキーを定義できます。

## 複数のラッピングキーでデータを暗号化できますか?

はい。追加のラッピングキー (またはマスターキー) を使用してデータキーを暗号化することで、キーが別のリージョンにある場合や復号のために使用できない場合に備えて冗長性を保つことができます。

複数のラッピングキーでデータを暗号化するには、複数のラッピングキーを使用してキーリングまたはマスターキープロバイダーを作成します。キーリングを使用する場合は、複数のラッピングキーを持つ 1 つのキーリングかマルチキーリングを作成できます。

複数のラッピングキーでデータを暗号化すると、 AWS Encryption SDK は 1 つのラッピングキーを使用してプレーンテキストのデータキーを生成します。データキーは固有で、ラッピングキーとは数学的に無関係です。このオペレーションでは、プレーンテキストデータキーとラッピングキーで暗号化されたデータキーのコピーが返されます。次に暗号化メソッドは、データキーを他のラッピングキーで暗号化します。結果として得られる暗号化されたメッセージには、暗号化されたデータと各ラッピングキーで 1 つずつ暗号化された一組のデータキーが含まれます。

暗号化されたメッセージは、暗号化オペレーションで使用されるラッピングキーのいずれかを使用して復号できます。 AWS 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ます。次に、そのプレーンテキストのデータキーを使用してデータを復号します。

## どのデータ型を で暗号化できますか AWS Encryption SDK?

のほとんどのプログラミング言語実装 AWS Encryption SDK では、raw バイト (バイト配列 ) 、I/O ストリーム (バイトストリーム ) 、および文字列を暗号化できます。 AWS Encryption SDK for .NET は I/O ストリームをサポートしていません。サポートされている各プログラミング言語 のサンプルコードを提供します。

# は入出力 (I/O) ストリームをどのように AWS Encryption SDK 暗号化および復号しますか？

は、基盤となる I/O ストリームをラップする暗号化ストリームまたは復号ストリーム AWS Encryption SDK を作成します。暗号化や復号のストリームは、読み取りまたは書き込みの呼び出しに対して暗号化オペレーションを実行します。たとえば、基盤となるストリームでプレーンテキストのデータを読み取り、結果を返す前に暗号化できます。または、基盤となるストリームから暗号化テキストを読み取り、結果を返す前に復号できます。ストリーミングをサポートする サポートされている各プログラミング言語 のストリームを暗号化および復号するためにサンプルコードを提供します。

AWS Encryption SDK for .NET は I/O ストリームをサポートしていません。

# AWS Encryption SDK リファレンス

このページの情報は、 AWS Encryption SDKと互換性のある独自の暗号化ライブラリを構築するためのリファレンスです。互換性のある独自の暗号化ライブラリを構築しない場合は、この情報は必要ありません。

サポートされているプログラミング言語のいずれか AWS Encryption SDK で を使用するには、「」を参照してくださいプログラミング言語。

適切な AWS Encryption SDK 実装の要素を定義する仕様については、GitHub のAWS Encryption SDK 「仕様」を参照してください。

AWS Encryption SDK は、サポートされているアルゴリズムを使用して、暗号化されたデータおよび対応する暗号化されたデータキーを含む単一のデータ構造またはメッセージを返します。以下のトピックでは、アルゴリズムおよびデータ構造について説明します。この情報を使用して、この SDK と互換性のある暗号化テキストを読み書きできるライブラリを構築します。

トピック

- AWS Encryption SDK メッセージ形式のリファレンス
- AWS Encryption SDK メッセージ形式の例
- AWS Encryption SDKの本文追加認証データ (AAD) のリファレンス
- AWS Encryption SDK アルゴリズムリファレンス
- AWS Encryption SDK 初期化ベクトルリファレンス
- AWS KMS 階層キーリングの技術的な詳細

# AWS Encryption SDK メッセージ形式のリファレンス

このページの情報は、 AWS Encryption SDKと互換性のある独自の暗号化ライブラリを構築するためのリファレンスです。互換性のある独自の暗号化ライブラリを構築しない場合は、この情報は必要ありません。

サポートされているプログラミング言語のいずれか AWS Encryption SDK で を使用するには、「」を参照してくださいプログラミング言語。

適切な AWS Encryption SDK 実装の要素を定義する仕様については、GitHub のAWS Encryption SDK 「仕様」を参照してください。

の暗号化オペレーションは、暗号化されたデータ (暗号文) とすべての暗号化されたデータキーを含む単一のデータ構造または暗号化されたメッセージ AWS Encryption SDK を返します。このデータ構造を理解したり、それを読み書きするライブラリを構築するには、メッセージ形式を理解しておく必要があります。

メッセージ形式は少なくとも 2 つの部分 (ヘッダーと本文) で構成されます。場合によって、メッセージ形式は 3 番目の部分、フッター で構成されます。メッセージ形式は、ビッグエンディアン形式とも呼ばれる、ネットワークバイト順で順序付けられたバイトシーケンスを定義します。メッセージ形式は、ヘッダーで始まり、その後に本文、続いてフッターの順に続きます (ある場合)。

AWS Encryption SDK によってサポートされるアルゴリズムスイートでは、2 つのメッセージ形式バージョンのいずれかを使用します。キーコミットメントがないアルゴリズムスイートでは、メッセージ形式バージョン 1 を使用します。キーコミットメントがあるアルゴリズムスイートでは、メッセージ形式バージョン 2 を使用します。

トピック

- ヘッダーの構造
- 本文の構造
- フッターの構造

## ヘッダーの構造

メッセージヘッダーには、暗号化されたデータキーと、メッセージ本文の構成に関する情報が含まれています。以下の表では、メッセージ形式バージョン 1 および 2 のヘッダ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。

「なし」は、フィールドがそのバージョンのメッセージ形式に存在しないことを示します。太字テキストは、各バージョンで異なる値を示します。

> **Note**
>
> この表のすべてのデータを表示するには、水平または垂直にスクロールする必要があります。

ヘッダーの構造

| フィールド | メッセージ形式バージョン 1<br><br>長さ (バイト) | メッセージ形式バージョン 2<br><br>長さ (バイト) |
|---|---|---|
| Version | 1 | 1 |
| Type | 1 | [なし] |
| Algorithm ID | 2 | 2 |
| Message ID | 16 | 32 |
| AAD Length | 2<br><br>暗号化コンテキストが空の場合、2 バイトの AAD の長さフィールドの値は 0 です。 | 2<br><br>暗号化コンテキストが空の場合、2 バイトの AAD の長さフィールドの値は 0 です。 |
| AAD | 変数。　このフィールドの長さは、前の 2 バイト (AAD の長さフィールド) に表示されます。<br><br>暗号化コンテキストが空の場合、ヘッダーに AAD フィールドはありません。 | 変数。　このフィールドの長さは、前の 2 バイト (AAD の長さフィールド) に表示されます。<br><br>暗号化コンテキストが空の場合、ヘッダーに AAD フィールドはありません。 |
| Encrypted Data Key Count | 2 | 2 |
| Encrypted Data Key(s) | 変数。　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。 | 変数。　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。 |
| Content Type | 1 | 1 |
| Reserved | 4 | [なし] |
| IV Length | 1 | [なし] |

| フィールド | メッセージ形式バージョン 1<br><br>長さ (バイト) | メッセージ形式バージョン 2<br><br>長さ (バイト) |
| --- | --- | --- |
| Frame Length | 4 | 4 |
| Algorithm Suite Data | [なし] | 変数。 メッセージを生成したアルゴリズムによって決まります。 |
| Header Authentication | 変数。 メッセージを生成したアルゴリズムによって決まります。 | 変数。 メッセージを生成したアルゴリズムによって決まります。 |

バージョン

このメッセージ形式のバージョン。バージョンは 1 または 2 で、16 進数表記のバイト `01` または `02` としてエンコードされます。

タイプ

このメッセージ形式のタイプ。タイプは構造の種類を示します。カスタマー認証暗号化データとして示されるタイプのみがサポートされています。そのタイプの値は 128 で、16 進数表記のバイト `80` でエンコードされます。

このフィールドは、メッセージ形式バージョン 2 では存在しません。

アルゴリズム ID

使用されるアルゴリズムの識別子。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。アルゴリズムの詳細については、「AWS Encryption SDK アルゴリズムリファレンス」を参照してください。

メッセージ ID

メッセージを識別するランダムに生成された値。メッセージ ID。

- 暗号化されたメッセージを一意に識別します。
- メッセージヘッダーを、メッセージ本文に弱くバインドします。
- 複数の暗号化されたメッセージでデータキーを安全に再利用するためのメカニズムを提供します。

- AWS Encryption SDKでのデータキーの誤った再利用や失効を防ぎます。

この値は、メッセージ形式バージョン 1 で 128 ビット、バージョン 2 では 256 ビットです。

AAD の長さ

追加認証データ (AAD) の長さ。これは、AAD 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。

暗号化コンテキストが空の場合、AAD の長さフィールドの値は 0 です。

AAD

追加認証データ。AAD は、暗号化コンテキストのエンコードです。キーと値の各ペアが UTF-8 エンコード文字の文字列のキーと値のペアの配列です。暗号化コンテキストはバイトシーケンスに変換され、AAD 値に使用されます。暗号化コンテキストが空の場合、ヘッダーに AAD フィールドはありません。

署名付きのアルゴリズムを使用する場合、暗号化コンテキストにはキーと値のペア `{'aws-crypto-public-key', Qtxt}` が含まれている必要があります。Qtxt は、SEC 1 バージョン 2.0 に基づいて圧縮され、その後 base64 でエンコードされた楕円曲線点 Q を表します。暗号化コンテキストには、追加の値を含めることができますが、構築された AAD の最大長は 2^16 - 1 バイトです。

以下の表では、AAD を形成するフィールドについて説明します。キーと値のペアは、UTF-8 文字コードに基づいて昇順でキーごとにソートされます。バイトは示されている順に追加されます。

AAD の構造

| フィールド | 長さ (バイト) |
| --- | --- |
| Key-Value Pair Count | 2 |
| Key Length | 2 |
| Key | 変数。　前の 2 バイト (キーの長さ) で指定された値と同じです。 |
| Value Length | 2 |
| Value | 変数。　前の 2 バイト (値の長さ) で指定された値と同じです。 |

キーと値のペアの数

AAD 内のキーと値のペアの数。これは、AAD でキーと値のペアの数を指定する 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。AAD 内のキーと値のペアの最大数は 2^16 - 1 です。

暗号化コンテキストが存在しない場合、または暗号化コンテキストが空の場合、このフィールドは AAD 構造内に存在しません。

キーの長さ

キーと値のペアのキーの長さ。これは、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。

キー

キーと値のペアのキー。UTF-8 でエンコードされたバイトのシーケンスです。

値の長さ

キーと値のペアの値の長さ。これは、値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。

値

キーと値のペアの値。UTF-8 でエンコードされたバイトのシーケンスです。

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 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。各メッセージの暗号化されたデータキーの最大数は 65,535 (2^16 - 1) です。

暗号化されたデータキー (複数可 )

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 がマスターキープロバイダーであるか、 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暗号化キーです。

コンテンツタイプ

暗号化されたデータのタイプ (フレーム化されていないデータまたはフレーム化されたデータ)。

> **Note**
>
> 可能な限り、フレーム化されたデータを使用してください。は、レガシー使用のためにのみフレーム化されていないデータ AWS Encryption SDK をサポートします。の一部の言語実装では、フレーム化されていない暗号文を生成 AWS Encryption SDK できます。サポートされているすべての言語実装では、フレーム化された暗号化テキストとフレーム化されていない暗号化文書を復号化できます。

フレーム化されたデータは同じ長さのパートに分割されます。各パートは別々に暗号化されます。フレーム化されたコンテンツはタイプ 2 で、16 進数表記のバイト `02` としてエンコードされます。

フレーム化されていないデータは分割されず、1 つの暗号化された BLOB になります。フレーム化されていないコンテンツはタイプ 1 で、16 進数表記のバイト `01` としてエンコードされます。

予約済み

予約された 4 バイトのシーケンスです。この値は、0 である必要があります。これは 16 進数でバイト `00 00 00 00` としてエンコードされます (つまり、0 と等しい 4 バイトシーケンスの 32 ビット整数値)。

このフィールドは、メッセージ形式バージョン 2 では存在しません。

IV の長さ

初期化ベクトル (IV) の長さ。これは、IV を含むバイト数を指定する 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。この値はメッセージを生成したアルゴリズムの IV バイト値によって決まります。

このフィールドはメッセージ形式バージョン 2 には存在しません。バージョン 2 では、メッセージヘッダーで確定的 IV 値を使用するアルゴリズムスイートのみがサポートされます。

フレームの長さ

フレーム化されたデータの各フレームの長さ。これは、各フレームのバイト数を指定する 32 ビットの符号なし整数として解釈される 4 バイトの値です。データがフレーム化されていないとき、つまり `Content Type` フィールドが 1 であるとき、この値は 0 である必要があります。

> **Note**
>
> 可能な限り、フレーム化されたデータを使用してください。は、レガシー使用のためにのみフレーム化されていないデータ AWS Encryption SDK をサポートします。の一部の言語実装では、フレーム化されていない暗号文を生成 AWS Encryption SDK できます。サポートされているすべての言語実装では、フレーム化された暗号化テキストとフレーム化されていない暗号化文書を復号化できます。

アルゴリズムスイートデータ

メッセージを生成したアルゴリズムが必要とする補足データ。長さと内容はアルゴリズムによって決定されます。その長さは 0 になる場合があります。

このフィールドは、メッセージ形式バージョン 1 では存在しません。

ヘッダー認証

ヘッダー認証は、メッセージを生成したアルゴリズムよって決まります。ヘッダー認証はヘッダー全体で計算されます。IV と認証タグで構成されています。バイトは示されている順に追加されます。

ヘッダー認証構造

| フィールド | バージョン 1.0 での長さ (バイト) | バージョン 2.0 での長さ (バイト) |
|---|---|---|
| IV | 変数。 メッセージを生成したアルゴリズムの IV バイト値によって決まります。 | 該当なし |
| Authentication Tag | 変数。 メッセージを生成したアルゴリズムの認証タグ | 変数。 メッセージを生成したアルゴリズムの認証タグ |

| フィールド | バージョン 1.0 での長さ (バイト) | バージョン 2.0 での長さ (バイト) |
| --- | --- | --- |
| | のバイト値によって決まります。 | のバイト値によって決まります。 |

IV

ヘッダー認証タグの計算に使用される初期化ベクトル (IV)。

このフィールドは、メッセージ形式バージョン 2 のヘッダーでは存在しません。メッセージ形式バージョン 2 では、メッセージヘッダーで確定的 IV 値を使用するアルゴリズムスイートのみがサポートされます。

認証タグ

ヘッダーの認証値。ヘッダーのコンテンツ全体を認証するために使用されます。

## 本文の構造

メッセージ本文には、暗号化テキストという暗号化されたデータが含まれています。本文の構造は、コンテンツタイプ (フレーム化されていないコンテンツまたはフレーム化されたコンテンツ) によって異なります。以下のセクションでは、各コンテンツタイプのメッセージ本文の形式について説明します。メッセージ本文の構造は、メッセージ形式バージョン 1 および 2 で同じです。

トピック

- フレーム化されていないデータ
- フレーム化されたデータ

### フレーム化されていないデータ

フレーム化されていないデータは、一意の IV と本文 AAD を含む 1 つの blob に暗号化されます。

> **Note**
>
> 可能な限り、フレーム化されたデータを使用してください。は、レガシー使用のためにのみフレーム化されていないデータ AWS Encryption SDK をサポートします。の一部の言語実装では、フレーム化されていない暗号文を生成 AWS Encryption SDK できます。サポートされ

ているすべての言語実装では、フレーム化された暗号化テキストとフレーム化されていない暗号化文書を復号化できます。

以下の表に、フレーム化されていないデータを構成するフィールドを示します。バイトは示されている順に追加されます。

フレーム化されていない本文構造

| フィールド | 長さ、バイト単位 |
| --- | --- |
| IV | 変数。 ヘッダーの IV Length バイトで指定された値と同じです。 |
| Encrypted Content Length | 8 |
| Encrypted Content | 変数。 前の 8 バイト (暗号化されたコンテンツの長さ) で指定された値と同じです。 |
| Authentication Tag | 変数。 使用されたアルゴリズムの実装によって決定されます。 |

IV

暗号化アルゴリズム で使用する初期化ベクトル (IV)。

暗号化されたコンテンツの長さ

暗号化されたコンテンツ、または暗号化テキストの長さ。これは、暗号化されたコンテンツを含むバイト数を指定する 64 ビットの符号なし整数として解釈される 8 バイトの値です。

技術的には、最大許容値は 2^63-1、または 8 エクスビバイト (8 EiB) です。ただし、実装されたアルゴリズムによって設定されている制限が原因で、実際の最大値は 2^36-32、または 64 ギビバイト (64 GiB) です。

> **Note**
>
> この SDK の Java 実装では、言語の制限により、この値はさらに 2^31-1 または 2 ギビバイト (2 GiB) に制限されます。

暗号化されたコンテンツ

暗号化アルゴリズムによって返される暗号化されたコンテンツ (暗号化テキスト)。

認証タグ

本文の認証値。メッセージ本文を認証するために使用されます。

## フレーム化されたデータ

フレーム化されたデータでは、プレーンテキストのデータはフレームと呼ばれる同じ長さのパートに分割されます。は AWS Encryption SDK 、一意の IV と本文 AAD を使用して各フレームを個別に暗号化します。

> **Note**
>
> 可能な限り、フレーム化されたデータを使用してください。は、レガシー使用のためにのみフレーム化されていないデータ AWS Encryption SDK をサポートします。の一部の言語実装では、フレーム化されていない暗号文を生成 AWS Encryption SDK できます。サポートされているすべての言語実装では、フレーム化された暗号化テキストとフレーム化されていない暗号化文書を復号化できます。

フレームの長さ (フレーム内の暗号化されたコンテンツの長さ) はメッセージごとに異なります。フレームの最大バイト数は 2^32 - 1 です。メッセージの最大フレーム数は 2^32 - 1 です。

フレームには、通常と最終の 2 種類があります。すべてのメッセージは、最終フレームで構成するか、最終フレームを含める必要があります。

1 つのメッセージのすべての通常フレームの長さは同じになります。最終フレームの長さは異なることができます。

フレーム化されたデータのフレームの構成は、暗号化されたコンテンツの長さによって異なります。

- フレームの長さと同じである場合 — 暗号化されたコンテンツの長さが通常フレームの長さと同じ場合、メッセージはデータを含む通常フレームとそれに続く長さがゼロ (0) の最終フレームで構成されます。または、メッセージはデータを含む最終フレームのみで構成されます。この場合、最終フレームの長さは通常フレームと同じになります。

- フレームの長さの倍数である場合 — 暗号化されたコンテンツの長さが通常フレームの長さの倍数である場合、メッセージはデータを含む通常フレームとそれに続く長さがゼロ (0) の最終フレームで終わります。または、メッセージはデータを含む最終フレームで終わります。この場合、最終フレームの長さは通常フレームと同じになります。
- フレームの長さの倍数ではない場合 — 暗号化されたコンテンツの長さが通常フレームの長さの倍数ではない場合、最終フレームには残りのデータが含まれます。最終フレームの長さは通常フレームよりも短くなります。
- フレームの長さよりも短い場合 — 暗号化されたコンテンツの長さが通常フレームの長さよりも短い場合、メッセージはすべてのデータを含む最終フレームで構成されます。最終フレームの長さは通常フレームよりも短くなります。

以下の表では、フレーム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。

フレーム化された本文構造、標準フレーム

| フィールド | 長さ、バイト単位 |
| --- | --- |
| Sequence Number | 4 |
| IV | 変数。 ヘッダーの IV Length バイトで指定された値と同じです。 |
| Encrypted Content | 変数。 ヘッダーの Frame Length で指定された値と同じです。 |
| Authentication Tag | 変数。 ヘッダーの Algorithm ID で指定された、使用されているアルゴリズムによって決定されます。 |

シーケンス番号

フレームシーケンス番号。これはフレームの増分カウンタです。これは 32 ビットの符号なし整数として解釈される 4 バイトの値です。

フレームデータはシーケンス番号 1 で始まる必要があります。後続のフレームは、順番に並んでいなければならず、1 つ前のフレームの増分を含む必要があります。それ以外の場合、復号プロセスは停止して、エラーが表示されます。

IV

フレームの初期化ベクトル (IV)。SDK は、決定的メソッドを使用して、メッセージ内のフレームごとに異なる IV を構築します。その長さは使用されるアルゴリズムスイートで指定されます。

暗号化されたコンテンツ

暗号化アルゴリズムによって返されるフレームの暗号化されたコンテンツ (暗号化テキスト)。

認証タグ

フレームの認証値。フレーム全体を認証するために使用されます。

フレーム化された本文構造、最終フレーム

| フィールド | 長さ、バイト単位 |
| --- | --- |
| Sequence Number End | 4 |
| Sequence Number | 4 |
| IV | 変数。 ヘッダーの IV Length バイトで指定された値と同じです。 |
| Encrypted Content Length | 4 |
| Encrypted Content | 変数。 前の 4 バイト (暗号化されたコンテンツの長さ) で指定された値と同じです。 |
| Authentication Tag | 変数。 ヘッダーの Algorithm ID で指定された、使用されているアルゴリズムによって決定されます。 |

シーケンス番号の終了

最終フレームのインジケータです。その値は 16 進数表記の 4 バイト FF FF FF FF として エンコードされます。

シーケンス番号

フレームシーケンス番号。これはフレームの増分カウンタです。これは 32 ビットの符号なし整数として解釈される 4 バイトの値です。

フレームデータはシーケンス番号 1 で始まる必要があります。後続のフレームは、順番に並んでいなければならず、1 つ前のフレームの増分を含む必要があります。それ以外の場合、復号プロセスは停止して、エラーが表示されます。

IV

フレームの初期化ベクトル (IV)。SDK は、決定的メソッドを使用して、メッセージ内のフレームごとに異なる IV を構築します。IV の長さはアルゴリズムスイートによって指定されます。

暗号化されたコンテンツの長さ

暗号化されたコンテンツの長さ。これは、フレームの暗号化されたコンテンツを含むバイト数を指定する 32 ビットの符号なし整数として解釈される 4 バイトの値です。

暗号化されたコンテンツ

暗号化アルゴリズムによって返されるフレームの暗号化されたコンテンツ (暗号化テキスト)。

認証タグ

フレームの認証値。フレーム全体を認証するために使用されます。

## フッターの構造

署名付きのアルゴリズムを使用する場合、メッセージ形式にはフッターが含まれます。メッセージフッターには、メッセージヘッダーおよび本文で計算されたデジタル署名が含まれています。以下の表では、フッタ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。メッセージフッターの構造は、メッセージ形式バージョン 1 および 2 で同じです。

フッターの構造

| フィールド | 長さ、バイト単位 |
| --- | --- |
| Signature Length | 2 |
| Signature | 変数。 前の 2 バイト (署名の長さ) で指定された値と同じです。 |

署名の長さ

署名の長さ。これは、署名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。

署名

署名

# AWS Encryption SDK メッセージ形式の例

このページの情報は、 AWS Encryption SDKと互換性のある独自の暗号化ライブラリを構築するためのリファレンスです。互換性のある独自の暗号化ライブラリを構築しない場合は、この情報は必要ありません。

サポートされているプログラミング言語のいずれか AWS Encryption SDK で を使用するには、「」を参照してくださいプログラミング言語。

適切な AWS Encryption SDK 実装の要素を定義する仕様については、GitHub のAWS Encryption SDK 「仕様」を参照してください。

以下のトピックでは、 AWS Encryption SDK メッセージ形式の例を示します。それぞれの例では、16 進数表記のローバイトを示し、それにこれらのバイト内容の説明文が続きます。

トピック

- フレーム化されたデータ (メッセージ形式バージョン 1)
- フレーム化されたデータ (メッセージ形式バージョン 2)
- フレーム化されていないデータ (メッセージ形式バージョン 1)

## フレーム化されたデータ (メッセージ形式バージョン 1)

以下の例は、メッセージ形式バージョン 1 のフレーム化されたデータのメッセージ形式を示しています。

```
+--------+
| Header |
+--------+
01                                      Version (1.0)
80                                      Type (128, customer authenticated encrypted
 data)
0378                                    Algorithm ID (see ##############)
6E7C0FBD 4DF4A999 717C22A2 DDFE1A27     Message ID (random 128-bit value)
```

```
008E                                    AAD Length (142)
0004                                    AAD Key-Value Pair Count (4)
0005                                    AAD Key-Value Pair 1, Key Length (5)
30746869 73                             AAD Key-Value Pair 1, Key ("0This")
0002                                    AAD Key-Value Pair 1, Value Length (2)
6973                                    AAD Key-Value Pair 1, Value ("is")
0003                                    AAD Key-Value Pair 2, Key Length (3)
31616E                                  AAD Key-Value Pair 2, Key ("1an")
000A                                    AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                  AAD Key-Value Pair 2, Value ("encryption")
0008                                    AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                       AAD Key-Value Pair 3, Key ("2context")
0007                                    AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                         AAD Key-Value Pair 3, Value ("example")
0015                                    AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69     AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632D6B65 79
0044                                    AAD Key-Value Pair 4, Value Length (68)
416A4173 7569326F 7430364C 4B77715A     AAD Key-Value Pair 4, Value
 ("AjAsui2ot06LKwqZXDJnU/Aqc2vD+0OkpOZ1cc8Tg2qd7rs5aLTg7lvfUEW/86+/5w==")
58444A6E 552F4171 63327644 2B304F6B
704F5A31 63633854 67327164 37727335
614C5467 376C7666 5545572F 38362B2F
35773D3D
0002                                    EncryptedDataKeyCount (2)
0007                                    Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                    Encrypted Data Key 1, Key Provider
 Information Length (75)
61726E3A 6177733A 6B6D733A 75732D77      Encrypted Data Key 1, Key Provider
 Information ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-
a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                    Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010200 7857A1C1 F7370545 4ECA7C83      Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
```

```
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C3F F02C897B
7A12EB19 8BF2D802 0110803B 24003D1F
A5474FBC 392360B5 CB9997E0 6A17DE4C
A6BD7332 6BF86DAB 60D8CCB8 8295DBE9
4707E356 ADA3735A 7C52D778 B3135A47
9F224BF9 E67E87
0007                                    Encrypted Data Key 2, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                    Encrypted Data Key 2, Key Provider
 Information Length (78)
61726E3A 6177733A 6B6D733A 63612D63      Encrypted Data Key 2, Key Provider
 Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-
be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                    Encrypted Data Key 2, Encrypted Data Key
 Length (167)
01010200 78FAFFFB D6DE06AF AC72F79B      Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C36 CD985E12
D218B674 5BBC6102 0110803B 0320E3CD
E470AA27 DEAB660B 3E0CE8E0 8B1A89E4
57DCC69B AAB1294F 21202C01 9A50D323
72EBAAFD E24E3ED8 7168E0FA DB40508F
556FBD58 9E621C
02                                      Content Type (2, framed data)
00000000                                Reserved
0C                                      IV Length (12)
00000100                                Frame Length (256)
4ECBD5C0 9899CA65 923D2347              IV
0B896144 0CA27950 CA571201 4DA58029      Authentication Tag
+------+
| Body |
+------+
00000001                                Frame 1, Sequence Number (1)
```

```
6BD3FE9C ADBCB213 5B89E8F1                Frame 1, IV
1F6471E0 A51AF310 10FA9EF6 F0C76EDF       Frame 1, Encrypted Content
F5AFA33C 7D2E8C6C 9C5D5175 A212AF8E
FBD9A0C3 C6E3FB59 C125DBF2 89AC7939
BDEE43A8 0F00F49E ACBBD8B2 1C785089
A90DB923 699A1495 C3B31B50 0A48A830
201E3AD9 1EA6DA14 7F6496DB 6BC104A4
DEB7F372 375ECB28 9BF84B6D 2863889F
CB80A167 9C361C4B 5EC07438 7A4822B4
A7D9D2CC 5150D414 AF75F509 FCE118BD
6D1E798B AEBA4CDB AD009E5F 1A571B77
0041BC78 3E5F2F41 8AF157FD 461E959A
BB732F27 D83DC36D CC9EBC05 00D87803
57F2BB80 066971C2 DEEA062F 4F36255D
E866C042 E1382369 12E9926B BA40E2FC
A820055F FB47E428 41876F14 3B6261D9
5262DB34 59F5D37E 76E46522 E8213640
04EE3CC5 379732B5 F56751FA 8E5F26AD       Frame 1, Authentication Tag
00000002                                  Frame 2, Sequence Number (2)
F1140984 FF25F943 959BE514                Frame 2, IV
216C7C6A 2234F395 F0D2D9B9 304670BF       Frame 2, Encrypted Content
A1042608 8A8BCB3F B58CF384 D72EC004
A41455B4 9A78BAC9 36E54E68 2709B7BD
A884C1E1 705FF696 E540D297 446A8285
23DFEE28 E74B225A 732F2C0C 27C6BDA2
7597C901 65EF3502 546575D4 6D5EBF22
1FF787AB 2E38FD77 125D129C 43D44B96
778D7CEE 3C36625F FF3A985C 76F7D320
ED70B1F3 79729B47 E7D9B5FC 02FCE9F5
C8760D55 7779520A 81D54F9B EC45219D
95941F7E 5CBAEAC8 CEC13B62 1464757D
AC65B6EF 08262D74 44670624 A3657F7F
2A57F1FD E7060503 AC37E197 2F297A84
DF1172C2 FA63CF54 E6E2B9B6 A86F582B
3B16F868 1BBC5E4D 0B6919B3 08D5ABCF
FECDC4A4 8577F08B 99D766A1 E5545670
A61F0A3B A3E45A84 4D151493 63ECA38F       Frame 2, Authentication Tag
FFFFFFFF                                  Final Frame, Sequence Number End
00000003                                  Final Frame, Sequence Number (3)
35F74F11 25410F01 DD9E04BF                Final Frame, IV
0000008E                                  Final Frame, Encrypted Content Length (142)
F7A53D37 2F467237 6FBD0B57 D1DFE830       Final Frame, Encrypted Content
B965AD1F A910AA5F 5EFFFFF4 BC7D431C
BA9FA7C4 B25AF82E 64A04E3A A0915526
```

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

## フレーム化されたデータ (メッセージ形式バージョン 2)

以下の例は、メッセージ形式バージョン 2 のフレーム化されたデータのメッセージ形式を示しています。

```
+--------+
| Header |
+--------+
02                                         Version (2.0)
0578                                       Algorithm ID (see Algorithms reference)
122747eb 21dfe39b 38631c61 7fad7340
cc621a30 32a11cc3 216d0204 fd148459        Message ID (random 256-bit value)
008e                                       AAD Length (142)
0004                                       AAD Key-Value Pair Count (4)
0005                                       AAD Key-Value Pair 1, Key Length (5)
30546869 73                                AAD Key-Value Pair 1, Key ("0This")
0002                                       AAD Key-Value Pair 1, Value Length (2)
6973                                       AAD Key-Value Pair 1, Value ("is")
0003                                       AAD Key-Value Pair 2, Key Length (3)
31616e                                     AAD Key-Value Pair 2, Key ("1an")
000a                                       AAD Key-Value Pair 2, Value Length (10)
656e6372 79707469 6f6e                     AAD Key-Value Pair 2, Value ("encryption")
0008                                       AAD Key-Value Pair 3, Key Length (8)
32636f6e 74657874                          AAD Key-Value Pair 3, Key ("2context")
```

```
0007                                    AAD Key-Value Pair 3, Value Length (7)
6578616d 706c65                         AAD Key-Value Pair 3, Value ("example")
0015                                    AAD Key-Value Pair 4, Key Length (21)
6177732d 63727970 746f2d70 75626c69     AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632d6b65 79
0044                                    AAD Key-Value Pair 4, Value Length (68)
41746733 72703845 41345161 36706669     AAD Key-Value Pair 4, Value
 ("QXRnM3JwOEVBNFFhNnBmaTk3MUlTNTk3NHpOMnlZWE5vSmtwRHFPc0dIYkVaVDRqME5OMlFkRStmbTFVY01WdThnPT0=
39373149 53353937 347a4e32 7959584e
6f4a6b70 44714f73 47486245 5a54346a
304e4e32 5164452b 666d3155 634d5675
38673d3d
0001                                    Encrypted Data Key Count (1)
0007                                    Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732d 6b6d73                         Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004b                                    Encrypted Data Key 1, Key Provider
 Information Length (75)
61726e3a 6177733a 6b6d733a 75732d77     Encrypted Data Key 1, Key
 Provider Information ("arn:aws:kms:us-west-2:658956600833:key/b3537ef1-
d8dc-4780-9f5a-55776cbb2f7f")
6573742d 323a3635 38393536 36303038
33333a6b 65792f62 33353337 6566312d
64386463 2d343738 302d3966 35612d35
35373736 63626232 663766
00a7                                    Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010100 7840f38c 275e3109 7416c107     Encrypted Data Key 1, Encrypted Data Key
29515057 1964ada3 ef1c21e9 4c8ba0bd
bc9d0fb4 14000000 7e307c06 092a8648
86f70d01 0706a06f 306d0201 00306806
092a8648 86f70d01 0701301e 06096086
48016503 04012e30 11040c39 32d75294
06063803 f8460802 0110803b 2a46bc23
413196d2 903bf1d7 3ed98fc8 a94ac6ed
e00ee216 74ec1349 12777577 7fa052a5
ba62e9e4 f2ac8df6 bcb1758f 2ce0fb21
cc9ee5c9 7203bb
02                                      Content Type (2, framed data)
00001000                                Frame Length (4096)
05cd035b 29d5499d 4587570b 87502afe     Algorithm Suite Data (key commitment)
634f7b2c c3df2aa9 88a10105 4a2c7687
```

```
76cb339f 2536741f 59a1c202 4f2594ab        Authentication Tag
+------+
| Body |
+------+
ffffffff                                   Final Frame, Sequence Number End
00000001                                   Final Frame, Sequence Number (1)
00000000 00000000 00000001                 Final Frame, IV
00000009                                   Final Frame, Encrypted Content Length (9)
fa6e39c6 02927399 3e                       Final Frame, Encrypted Content
f683a564 405d68db eeb0656c d57c9eb0        Final Frame, Authentication Tag
+--------+
| Footer |
+--------+
0067                                       Signature Length (103)
30650230 2a1647ad 98867925 c1712e8f        Signature
ade70b3f 2a2bc3b8 50eb91ef 56cfdd18
967d91d8 42d92baf 357bba48 f636c7a0
869cade2 023100aa ae12d08f 8a0afe85
e5054803 110c9ed8 11b2e08a c4a052a9
074217ea 3b01b660 534ac921 bf091d12
3657e2b0 9368bd
```

## フレーム化されていないデータ (メッセージ形式バージョン 1)

以下の例は、フレーム化されていないデータのメッセージ形式を示しています。

> **Note**
>
> 可能な限り、フレーム化されたデータを使用してください。は、レガシー使用のためにのみフレーム化されていないデータ AWS Encryption SDK をサポートします。の一部の言語実装では、フレーム化されていない暗号文を生成 AWS Encryption SDK できます。サポートされているすべての言語実装では、フレーム化された暗号化テキストとフレーム化されていない暗号化文書を復号化できます。

```
+--------+
| Header |
+--------+
01                                         Version (1.0)
```

```
80                                      Type (128, customer authenticated encrypted
 data)
0378                                    Algorithm ID (see ############)
B8929B01 753D4A45 C0217F39 404F70FF      Message ID (random 128-bit value)
008E                                    AAD Length (142)
0004                                    AAD Key-Value Pair Count (4)
0005                                    AAD Key-Value Pair 1, Key Length (5)
30746869 73                             AAD Key-Value Pair 1, Key ("0This")
0002                                    AAD Key-Value Pair 1, Value Length (2)
6973                                    AAD Key-Value Pair 1, Value ("is")
0003                                    AAD Key-Value Pair 2, Key Length (3)
31616E                                  AAD Key-Value Pair 2, Key ("1an")
000A                                    AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                  AAD Key-Value Pair 2, Value ("encryption")
0008                                    AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                       AAD Key-Value Pair 3, Key ("2context")
0007                                    AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                         AAD Key-Value Pair 3, Value ("example")
0015                                    AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69      AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632D6B65 79
0044                                    AAD Key-Value Pair 4, Value Length (68)
41734738 67473949 6E4C5075 3136594B      AAD Key-Value Pair 4, Value
 ("AsG8gG9InLPu16YKlqXTOD+nykG8YqHAhqecj8aXfD2e5B4gtVE73dZkyClA+rAMOQ==")
6C715854 4F442B6E 796B4738 59714841
68716563 6A386158 66443265 35423467
74564537 33645A6B 79436C41 2B72414D
4F513D3D
0002                                    Encrypted Data Key Count (2)
0007                                    Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                    Encrypted Data Key 1, Key Provider
 Information Length (75)
61726E3A 6177733A 6B6D733A 75732D77      Encrypted Data Key 1, Key Provider
 Information ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-
a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
```

```
00A7                                    Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010200 7857A1C1 F7370545 4ECA7C83      Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C28 4116449A
0F2A0383 659EF802 0110803B B23A8133
3A33605C 48840656 C38BCB1F 9CCE7369
E9A33EBE 33F46461 0591FECA 947262F3
418E1151 21311A75 E575ECC5 61A286E0
3E2DEBD5 CB005D
0007                                    Encrypted Data Key 2, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                    Encrypted Data Key 2, Key Provider
 Information Length (78)
61726E3A 6177733A 6B6D733A 63612D63      Encrypted Data Key 2, Key Provider
 Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-
be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                    Encrypted Data Key 2, Encrypted Data Key
 Length (167)
01010200 78FAFFFB D6DE06AF AC72F79B      Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040CB2 A820D0CC
76616EF2 A6B30D02 0110803B 8073D0F1
FDD01BD9 B0979082 099FDBFC F7B13548
3CC686D7 F3CF7C7A CCC52639 122A1495
71F18A46 80E2C43F A34C0E58 11D05114
2A363C2A E11397
01                                      Content Type (1, nonframed data)
00000000                                Reserved
0C                                      IV Length (12)
00000000                                Frame Length (0, nonframed data)
734C1BBE 032F7025 84CDA9D0              IV
```

```
2C82BB23 4CBF4AAB 8F5C6002 622E886C        Authentication Tag
+------+
| Body |
+------+
D39DD3E5 915E0201 77A4AB11                 IV
00000000 0000028E                          Encrypted Content Length (654)
E8B6F955 B5F22FE4 FD890224 4E1D5155        Encrypted Content
5871BA4C 93F78436 1085E4F8 D61ECE28
59455BD8 D76479DF C28D2E0B BDB3D5D3
E4159DFE C8A944B6 685643FC EA24122B
6766ECD5 E3F54653 DF205D30 0081D2D8
55FCDA5B 9F5318BC F4265B06 2FE7C741
C7D75BCC 10F05EA5 0E2F2F40 47A60344
ECE10AA7 559AF633 9DE2C21B 12AC8087
95FE9C58 C65329D1 377C4CD7 EA103EC1
31E4F48A 9B1CC047 EE5A0719 704211E5
B48A2068 8060DF60 B492A737 21B0DB21
C9B21A10 371E6179 78FAFB0B BAAEC3F4
9D86E334 701E1442 EA5DA288 64485077
54C0C231 AD43571A B9071925 609A4E59
B8178484 7EB73A4F AAE46B26 F5B374B8
12B0000C 8429F504 936B2492 AAF47E94
A5BA804F 7F190927 5D2DF651 B59D4C2F
A15D0551 DAEBA4AF 2060D0D5 CB1DA4E6
5E2034DB 4D19E7CD EEA6CF7E 549C86AC
46B2C979 AB84EE12 202FD6DF E7E3C09F
C2394012 AF20A97E 369BCBDA 62459D3E
C6FFB914 FEFD4DE5 88F5AFE1 98488557
1BABBAE4 BE55325E 4FB7E602 C1C04BEE
F3CB6B86 71666C06 6BF74E1B 0F881F31
B731839B CF711F6A 84CA95F5 958D3B44
E3862DF6 338E02B5 C345CFF8 A31D54F3
6920AA76 0BF8E903 552C5A04 917CCD11
D4E5DF5C 491EE86B 20C33FE1 5D21F0AD
6932E67C C64B3A26 B8988B25 CFA33E2B
63490741 3AB79D60 D8AEFBE9 2F48E25A
978A019C FE49EE0A 0E96BF0D D6074DDB
66DFF333 0E10226F 0A1B219C BE54E4C2
2C15100C 6A2AA3F1 88251874 FDC94F6B
9247EF61 3E7B7E0D 29F3AD89 FA14A29C
76E08E9B 9ADCDF8C C886D4FD A69F6CB4
E24FDE26 3044C856 BF08F051 1ADAD329
C4A46A1E B5AB72FE 096041F1 F3F3571B
2EAFD9CB B9EB8B83 AE05885A 8F2D2793
```

```
1E3305D9 0C9E2294 E8AD7E3B 8E4DEC96
6276C5F1 A3B7E51E 422D365D E4C0259C
50715406 822D1682 80B0F2E5 5C94
65B2E942 24BEEA6E A513F918 CCEC1DE3      Authentication Tag
+--------+
| Footer |
+--------+
0067                                     Signature Length (103)
30650230 7229DDF5 B86A5B64 54E4D627      Signature
CBE194F1 1CC0F8CF D27B7F8B F50658C0
BE84B355 3CED1721 A0BE2A1B 8E3F449E
1BEB8281 023100B2 0CB323EF 58A4ACE3
1559963B 889F72C3 B15D1700 5FB26E61
331F3614 BC407CEE B86A66FA CBF74D9E
34CB7E4B 363A38
```

# AWS Encryption SDKの本文追加認証データ (AAD) のリファレンス

このページの情報は、 AWS Encryption SDKと互換性のある独自の暗号化ライブラリを構築するためのリファレンスです。互換性のある独自の暗号化ライブラリを構築しない場合は、この情報は必要ありません。

サポートされているプログラミング言語のいずれか AWS Encryption SDK で を使用するには、「」を参照してくださいプログラミング言語。

適切な AWS Encryption SDK 実装の要素を定義する仕様については、GitHub のAWS Encryption SDK 「仕様」を参照してください。

暗号化オペレーションごとに、AES-GCM アルゴリズムに追加の認証データ (AAD) を指定する必要があります。これは、フレーム化された本文データとフレーム化されていない本文データの両方で必要です。AAD および Galois/Counter Mode での使用方法については、「ブロック暗号の動作モード: Galois/Counter Mode (GCM) および GMAC の推奨事項」を参照してください。

以下の表では、本文 AAD 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。

本文 AAD 構造

| フィールド | 長さ、バイト単位 |
| --- | --- |
| Message ID | 16 |
| Body AAD Content | 変数。 以下のリストの本文 AAD コンテンツを参照してください。 |
| Sequence Number | 4 |
| Content Length | 8 |

メッセージ ID

メッセージヘッダーの同じ Message ID 値のセット。

本文 AAD コンテンツ

使用する本文データのタイプによって決定される、UTF-8 でエンコードされた値。

フレーム化されていないデータの場合、`AWSKMSEncryptionClient Single Block` の値を使用します。

フレーム化されたデータの通常のフレーム。`AWSKMSEncryptionClient Frame` の値を使用します。

フレーム化されたデータの最終フレーム。`AWSKMSEncryptionClient Final Frame` の値を使用します。

シーケンス番号

32 ビットの符号なし整数として解釈される 4 バイトの値。

フレーム化されたデータの場合、これはフレームのシーケンス番号です。

フレーム化されていないデータの場合、1 の値 (4 バイトの 16 進数表記で `00 00 00 01` としてエンコード) を使用します。

コンテンツの長さ

暗号化のためにアルゴリズムに提供されるプレーンテキストデータの長さ (バイト単位)。これは 64 ビットの符号なし整数として解釈される 8 バイトの値です。

# AWS Encryption SDK アルゴリズムリファレンス

このページの情報は、 AWS Encryption SDKと互換性のある独自の暗号化ライブラリを構築するためのリファレンスです。互換性のある独自の暗号化ライブラリを構築しない場合は、この情報は必要ありません。

サポートされているプログラミング言語のいずれか AWS Encryption SDK で を使用するには、「」を参照してくださいプログラミング言語。

適切な AWS Encryption SDK 実装の要素を定義する仕様については、GitHub のAWS Encryption SDK 「仕様」を参照してください。

と互換性のある暗号文を読み書きできる独自のライブラリを構築する場合は AWS Encryption SDK、がサポートされているアルゴリズムスイート AWS Encryption SDK を実装して raw データを暗号化する方法を理解する必要があります。

は、次のアルゴリズムスイート AWS Encryption SDK をサポートしています。すべての AES-GCM アルゴリズムスイートには 12 バイトの初期化ベクトルおよび 16 バイトの AES-GCM 認証タグがあります。デフォルトのアルゴリズムスイートは、 AWS Encryption SDK バージョンと選択したキーコミットメントポリシーによって異なります。詳細については、「Commitment policy and algorithm suite」を参照してください。

AWS Encryption SDK アルゴリズムスイート

| アルゴリズム ID | メッセージ形式バージョン | 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 | 署名アルゴリズム | キーコミットメントアルゴリズム | アルゴリズムスイートのデータ長 (バイト) |
|---|---|---|---|---|---|---|---|
| 05 78 | 0x02 | AES-GCM | 256 | SHA-512 を使用する HKDF | P-384 および SHA-384 を使用する ECDSA | SHA-512 を使用する HKDF | 32 (キーコミットメント) |

| アルゴリズム ID | メッセージ形式バージョン | 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 | 署名アルゴリズム | キーコミットメントアルゴリズム | アルゴリズムスイートのデータ長 (バイト) |
|---|---|---|---|---|---|---|---|
| 04 78 | 0x02 | AES-GCM | 256 | SHA-512 を使用する HKDF | なし | SHA-512 を使用する HKDF | 32 (キーコミットメント) |
| 03 78 | 0x01 | AES-GCM | 256 | SHA-384 を使用する HKDF | P-384 および SHA-384 を使用する ECDSA | なし | 該当なし |
| 03 46 | 0x01 | AES-GCM | 192 | SHA-384 を使用する HKDF | P-384 および SHA-384 を使用する ECDSA | なし | 該当なし |
| 02 14 | 0x01 | AES-GCM | 128 | SHA-256 を使用する HKDF | P-256 および SHA-256 を使用する ECDSA | なし | 該当なし |
| 01 78 | 0x01 | AES-GCM | 256 | SHA-256 を使用する HKDF | なし | なし | 該当なし |
| 01 46 | 0x01 | AES-GCM | 192 | SHA-256 を使用する HKDF | なし | なし | 該当なし |

| アルゴリズム ID | メッセージ形式バージョン | 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 | 署名アルゴリズム | キーコミットメントアルゴリズム | アルゴリズムスイートのデータ長 (バイト) |
|---|---|---|---|---|---|---|---|
| 01 14 | 0x01 | AES-GCM | 128 | SHA-256 を使用する HKDF | なし | なし | 該当なし |
| 00 78 | 0x01 | AES-GCM | 256 | なし | なし | なし | 該当なし |
| 00 46 | 0x01 | AES-GCM | 192 | なし | なし | なし | 該当なし |
| 00 14 | 0x01 | AES-GCM | 128 | なし | なし | なし | 該当なし |

アルゴリズム ID

アルゴリズム実装を一意に識別する 2 バイトの 16 進値。この値は、暗号化テキストのメッセージヘッダーに保存されます。

メッセージ形式バージョン

メッセージ形式のバージョン。キーコミットメントがあるアルゴリズムスイートでは、メッセージ形式バージョン 2 (0x02) を使用します。キーコミットメントがないアルゴリズムスイートでは、メッセージ形式バージョン 1 (0x01) を使用します。

アルゴリズムスイートのデータ長

アルゴリズムスイートに固有のデータの長さ (バイト単位)。このフィールドは、メッセージ形式バージョン 2 (0x02) でのみサポートされます。メッセージ形式バージョン 2 (0x02) では、このデータはメッセージヘッダーの `Algorithm suite data` フィールドに表示されます。キーコミットメントをサポートするアルゴリズムスイートでは、キーコミットメント文字列に 32 バイトを使用します。詳細については、このリストのキーコミットメントアルゴリズムを参照してください。

データキーの長さ

データキーの長さ (ビット単位)。 AWS Encryption SDK では、256 ビット、192 ビット、128 ビットのキーをサポートしています。データキーは、キーリングまたはマスターキーによって生成されます。

一部の実装では、このデータキーは HMAC ベースの抽出および展開キー取得関数 (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。詳細については、このリストのキー取得アルゴリズムを参照してください。

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 Encryption SDK のアルゴリズムスイートでは、Advanced Encryption Standard (AES) 暗号化アルゴリズムを Galois/Counter Mode (GCM) と併用します。

キーコミットメントアルゴリズム

キーコミットメント文字列の計算に使用するアルゴリズム。出力は、メッセージヘッダーの `Algorithm suite data` フィールドに保存され、キーコミットメントのデータキーの検証に使用されます。

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。

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)。は RFC 5869 で定義された HKDF AWS Encryption SDK を使用します。

キーコミットメントのないアルゴリズムスイート (アルゴリズム ID `01xx` – `03xx`)

- 使用されるハッシュ関数は、SHA-384 または SHA-256 のいずれかで、アルゴリズムスイートによって決まります。
- 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。文字列の長さはハッシュ関数出力の長さと同じです。つまり、SHA-384 に対して 48 バイト、SHA-256 に対して 32 バイトです。
  - 入力キーマテリアルは、キーリングまたはマスターキープロバイダーからのデータキーです。
- 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、アルゴリズム ID とメッセージ ID 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。

キーコミットメントがあるアルゴリズムスイート (アルゴリズム ID `04xx` と `05xx`)

- 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 256 ビットの暗号化ランダム値です。メッセージ形式バージョン 2 (0x02) の場合、この値は `MessageID` フィールドに保存されます。
  - 初期キーマテリアルは、キーリングまたはマスターキープロバイダーからのデータキーです。
- 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。

メッセージ形式バージョン

アルゴリズムスイートで使用するメッセージ形式のバージョン。詳細については、「メッセージ形式のリファレンス」を参照してください。

署名アルゴリズム

暗号化テキストのヘッダーと本文へのデジタル署名の生成に使用される署名アルゴリズム。は、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 AWS Encryption SDK 使用します。

- 使用される楕円曲線のは、P-384 または P-256 のいずれかで、アルゴリズム ID によって指定されます。これらの曲線は、Digital Signature Standard (DSS) (FIPS PUB 186-4) で定義されています。
- 使用されるハッシュ関数は、SHA-384 (P-384 曲線を使用) または SHA-256 (P-256 曲線を使用) です。

# AWS Encryption SDK 初期化ベクトルリファレンス

このページの情報は、 AWS Encryption SDKと互換性のある独自の暗号化ライブラリを構築するためのリファレンスです。互換性のある独自の暗号化ライブラリを構築しない場合は、この情報は必要ありません。

サポートされているプログラミング言語のいずれか AWS Encryption SDK で を使用するには、「」を参照してくださいプログラミング言語。

適切な AWS Encryption SDK 実装の要素を定義する仕様については、GitHub のAWS Encryption SDK 「仕様」を参照してください。

は、サポートされているすべてのアルゴリズムスイートに必要な初期化ベクトル (IVs) AWS Encryption SDK を提供します。SDK は、フレームのシーケンス番号を使用して IV を構築し、同じメッセージ内の 2 つのフレームが同じ IV を持つことがないようにします。

各 96 ビット (12 バイト) IV は、以下の順序で連結された 2 つのビッグエンディアンバイト配列で構築されています。

- 64 ビット: 0 (将来の利用のために予約されています)
- 32 ビット: フレームシーケンス番号。ヘッダー認証タグの場合、この値はすべてゼロです。

データキーキャッシュが導入されるまで、 AWS Encryption SDK では、常に新しいデータキーを使用して各メッセージを暗号化し、すべての IV をランダムに生成していました。データキーが再利用されることはないので、ランダムに生成された IV は暗号論的に安全です。SDK で意図的にデータキーを再利用するデータキーキャッシュを導入した際、SDK が IV を生成する方法を変更しました。

メッセージ内で繰り返し使用できない決定的な IV を使用すると、単一のデータキーの下で安全に実行される呼び出しの数が大幅に増加します。さらに、キャッシュされたデータキーは常にキー取得関数と合わせてアルゴリズムスイートを使用します。擬似ランダムキー取得関数で決定論的な IV を使用してデータキーから暗号化キーを取得すると、 は暗号化境界を超えることなく 2^32 メッセージを暗号化 AWS Encryption SDK できます。

## 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、unqiue データキーを使用して各メッセージを暗号化し、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。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 Encryption SDK デベロッパーガイドのドキュメント履歴

このトピックでは、AWS Encryption SDK デベロッパーガイドの重要な更新を説明しています。

トピック


- 最新の更新
- 以前の更新


## 最新の更新

以下の表は、このドキュメントの 2017 年 11 月以降の大きな変更点をまとめたものです。ここに表示されている主要な変更に加えて、その内容の説明と例を向上し、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 |
| --- | --- | --- |
| 一般提供 | 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 |
| AWS Encryption SDK for Java バージョン 3.x | をマテリアルプロバイダーライブラリ AWS Encryption SDK for Java と統合します。キーリングと必要な暗号化コンテキスト CMM のサポートが追加されました。 | 2023 年 12 月 6 日 |
| AWS Encryption SDK for .NET バージョン 4.x | AWS KMS 階層キーリング、必要な暗号化コンテキスト CMM、非対称 RSA AWS KMS キーリングのサポートが追加されました。 | 2023 年 10 月 12 日 |

| | | |
|---|---|---|
| 一般提供 | for .NET AWS Encryption SDK のサポートを紹介します。 | 2022 年 5 月 17 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 |
| 一般提供 | 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 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る異なるのキーです。 | 2021 年 6 月 8 日 |
| 一般提供 | 改善されたメッセージ復号化プロセスに関するドキュメントが追加、更新されました。 | 2021 年 5 月 11 日 |
| 一般提供 | Encryption AWS CLI バージョン 1.7.x を置き換える Encryption CLI バージョン AWS 1.8.x の一般提供リリースと、Encryption AWS CLI 2.0.x を置き換える AWS Encryption CLI バージョン 2.1.x のドキュメントを追加および更新しました。 | 2020 年 10 月 27 日 |

| | | |
|---|---|---|
| 一般提供 | ベストプラクティスガイド、移行ガイド、概念の更新、プログラミング言語のトピックの更新、アルゴリズムスイートリファレンスの更新、メッセージ形式リファレンスの更新、新規のメッセージ形式例など、 AWS Encryption SDK バージョン 1.7.x および 2.0.x の一般公開リリースのドキュメントが追加、更新されました。 | 2020 年 9 月 24 日 |
| 一般提供 | AWS Encryption SDK for JavaScript の一般提供リリースのドキュメントを追加および更新しました。 | 2019 年 10 月 1 日 |
| プレビューリリース | AWS Encryption SDK for JavaScript のパブリックベータリリースのドキュメントを追加および更新しました。 | 2019 年 6 月 21 日 |
| 一般提供 | AWS Encryption SDK for C の一般提供リリースのドキュメントを追加および更新しました。 | 2019 年 5 月 16 日 |
| プレビューリリース | AWS Encryption SDK for C のプレビューリリースのドキュメントを追加しました。 | 2019 年 2 月 5 日 |
| 新規リリース | AWS Encryption SDKのコマンドラインインターフェイスのドキュメントが追加されました。 | 2017 年 11 月 20 日 |

# 以前の更新

以下の表は、2017 年 11 月より前に AWS Encryption SDK デベロッパーガイドに加えられた大きな変更点をまとめたものです。

| 変更 | 説明 | 日付 |
| --- | --- | --- |
| 新規リリース | 新しい機能に データキーキャッシュ 章を追加しました。<br><br>SDK が行った、ランダムな IV 生成から決定的な IV 構築への変更について説明する the section called “初期化ベクトルのリファレンス” トピックを追加しました。<br><br>新しい暗号化マテリアルマネージャーを含む概念について説明するトピック the section called “概念” が追加されました。 | 2017 年 7 月 31 日 |
| 更新 | 新しい メッセージ形式のリファレンス セクションに AWS Encryption SDK リファレンス ドキュメントを展開しました。<br><br>に関するセクションを追加しました AWS Encryption SDK サポートされているアルゴリズムスイート。 | 2017 年 3 月 21 日 |
| 新規リリース | は、 に加えて Pythonプログラミング言語をサポートする | 2017 年 3 月 21 日 |

| 変更 | 説明 | 日付 |
| --- | --- | --- |
| | AWS Encryption SDK ようになりましたJava。 | |
| 初回リリース | AWS Encryption SDK およびこのドキュメントの初回リリース。 | 2016 年 3 月 22 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。