Developer Guide

AWS Database Encryption SDK

AWS Database Encryption SDK: Developer Guide

Table of Contents

What is the AWS Database Encryption SDK?

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK is a set of software libraries that enable you to include client-side encryption in your database design. The AWS Database Encryption SDK provides record-level encryption solutions. You specify which fields are encrypted and which fields are included in the signatures that ensure the authenticity of your data. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn't available to any third party, including AWS. The AWS Database Encryption SDK is provided free of charge under the Apache 2.0 license.

This developer guide provides a conceptual overview of the AWS Database Encryption SDK, including an introduction to its architecture, details about how it protects your data, how it differs from server-side encryption, and guidance on selecting critical components for your application to help you get started.

The AWS Database Encryption SDK supports Amazon DynamoDB with *attribute*-level encryption.

The AWS Database Encryption SDK has the following benefits:

Designed especially for database applications

You don't need to be a cryptography expert to use the AWS Database Encryption SDK. The implementations include helper methods that are designed to work with your existing applications.

After you create and configure the required components, the encryption client transparently encrypts and signs your records when you add them to a database, and verifies and decrypts them when you retrieve them.

Includes secure encryption and signing

The AWS Database Encryption SDK includes secure implementations that encrypt the field values in each record using a unique data encryption key, and then sign the record to protect it against unauthorized changes, such as adding or deleting fields, or swapping encrypted values.

Uses cryptographic materials from any source

The AWS Database Encryption SDK uses keyrings to generate, encrypt, and decrypt the unique data encryption key that protects your record. Keyrings determine the wrapping keys that encrypt that data key.

You can use wrapping keys from any source, including cryptography services, such as AWS Key Management Service (AWS KMS) or AWS CloudHSM. The AWS Database Encryption SDK doesn't require an AWS account or any AWS service.

Support for cryptographic materials caching

The AWS KMS Hierarchical keyring is a cryptographic materials caching solution that reduces the number of AWS KMS calls by using AWS KMS protected *branch keys* persisted in an Amazon DynamoDB table, and then locally caching branch key materials used in encrypt and decrypt operations. It allows you to protect your cryptographic materials under a symmetric encryption KMS key without calling AWS KMS every time you encrypt or decrypt a record. The AWS KMS Hierarchical keyring is a good choice for applications that need to minimize calls to AWS KMS.

Searchable encryption

You can design databases that can search encrypted records without decrypting the entire database. Depending on your threat model and query requirements, you can use searchable encryption to perform exact match searches or more customized complex queries on your encrypted database.

Support for multitenant database schemas

The AWS Database Encryption SDK enables you to protect data stored in databases with a shared schema by isolating each tenant with distinct encryption materials. If you have multiple users performing encrypt operations within your database, use one of the AWS KMS keyrings to provide each user with a distinct key to use in their cryptographic operations. For more information, see Working with multitenant databases.

Support for seamless schema updates

When you configure the AWS Database Encryption SDK, you provide cryptographic actions that tell the client which fields to encrypt and sign, which fields to sign (but not encrypt), and which to ignore. After you have used the AWS Database Encryption SDK to protect your records, you can still make changes to your data model. You can update your cryptographic actions, such as adding or removing encrypted fields, in a single deployment.

Developed in open-source repositories

The AWS Database Encryption SDK is developed in open-source repositories on GitHub. You can use these repositories to view the code, read and submit issues, and find information that is specific to your implementation.

The AWS Database Encryption SDK for DynamoDB

- The aws-database-encryption-sdk-dynamodb repository on GitHub supports the latest versions of the AWS Database Encryption SDK for DynamoDB in Java, .NET, and Rust.

  The AWS Database Encryption SDK for DynamoDB is a product of Dafny, a verification-aware language in which you write specifications, the code to implement them, and the proofs to test them. The result is a library that implements the features of the AWS Database Encryption SDK for DynamoDB in a framework that assures functional correctness.

Support and maintenance

The AWS Database Encryption SDK uses the same maintenance policy that the AWS SDK and Tools use, including its versioning and lifecycle phases. As a best practice, we recommend that you use the latest available version of the AWS Database Encryption SDK for your database implementation, and upgrade as new versions are released.

For more information, see the AWS SDKs and Tools maintenance policy in the AWS SDKs and Tools Reference Guide.

Sending feedback

We welcome your feedback! If you have a question or comment, or an issue to report, please use the following resources.

If you discover a potential security vulnerability in the AWS Database Encryption SDK, please notify AWS security. Do not create a public GitHub issue.

To provide feedback on this documentation, use the feedback link on any page.

AWS Database Encryption SDK concepts

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

This topic explains the concepts and terminology used in the AWS Database Encryption SDK.

To learn how the components of the AWS Database Encryption SDK interact, see How the AWS Database Encryption SDK works.

To learn more about the AWS Database Encryption SDK, see the following topics.

- Learn how the AWS Database Encryption SDK uses envelope encryption to protect your data.
- Learn about the elements of envelope encryption: the data keys that protect your records and the wrapping keys that protect your data keys.
- Learn about the keyrings that determine which wrapping keys you use.
- Learn about the encryption context that adds integrity to your encryption process.
- Learn about the material description that the encryption methods add to your record.
- Learn about the cryptographic actions that tell the AWS Database Encryption SDK what fields to encrypt and sign.

Topics

- Key commitment
- Digital signatures

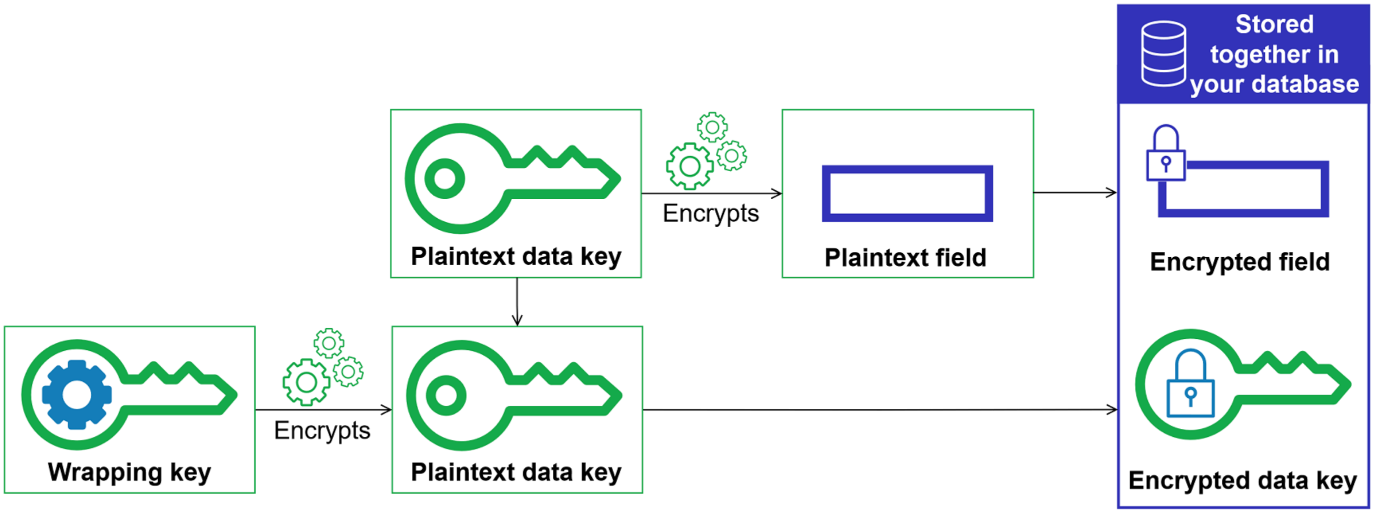
Envelope encryption

The security of your encrypted data depends in part on protecting the data key that can decrypt it. One accepted best practice for protecting the data key is to encrypt it. To do this, you need another encryption key, known as a *key-encryption key* or wrapping key. The practice of using a wrapping key to encrypt data keys is known as *envelope encryption*.

Protecting data keys

The AWS Database Encryption SDK encrypts each field with a unique data key. Then it encrypts each data key under the wrapping key you specify. It stores the encrypted data keys in the material description.

To specify your wrapping key, you use a keyring.



Encrypting the same data under multiple wrapping keys

You can encrypt the data key with multiple wrapping keys. You might want to provide different wrapping keys for different users, or wrapping keys of different types, or in different locations. Each of the wrapping keys encrypts the same data key. The AWS Database Encryption SDK stores all of the encrypted data keys alongside the encrypted fields in the material description.

To decrypt the data, you need to provide at least one wrapping key that can decrypt the encrypted data keys.

Combining the strengths of multiple algorithms

To encrypt your data, by default, the AWS Database Encryption SDK uses an algorithm suite with AES-GCM symmetric encryption, an HMAC-based key derivation function (HKDF), and ECDSA signing. To encrypt the data key, you can specify a symmetric or asymmetric encryption algorithm appropriate to your wrapping key.

In general, symmetric key encryption algorithms are faster and produce smaller ciphertexts than asymmetric or *public key encryption*. But public key algorithms provide inherent separation of roles. To combine the strengths of each, you can encrypt the data key with public key encryption.

We recommend using one of the AWS KMS keyrings whenever possible. When you use the AWS KMS keyring, you can choose to combine the strengths of multiple algorithms by specifying an asymmetric RSA AWS KMS key as your wrapping key. You can also use a symmetric encryption KMS key.

Data key

A *data key* is an encryption key that the AWS Database Encryption SDK uses to encrypt the fields in a record that are marked `ENCRYPT_AND_SIGN` in the cryptographic actions. Each data key is a byte array that conforms to the requirements for cryptographic keys. The AWS Database Encryption SDK uses a unique data key to encrypt each attribute.

You don't need to specify, generate, implement, extend, protect, or use data keys. The AWS Database Encryption SDK does that work for you when you call the encrypt and decrypt operations.

To protect your data keys, the AWS Database Encryption SDK encrypts them under one or more *key-encryption keys* known as wrapping keys. After the AWS Database Encryption SDK uses your plaintext data keys to encrypt your data, it removes them from memory as soon as possible. Then stores the encrypted data key in the material description. For details, see How the AWS Database Encryption SDK works.

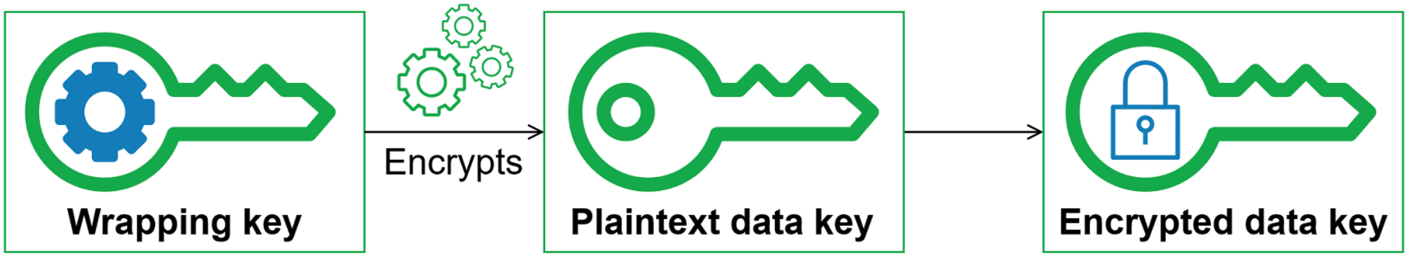
> **Tip**
>
> In the AWS Database Encryption SDK, we distinguish *data keys* from *data encryption keys*. As a best practice, all of the supported algorithm suites use a key derivation function. The

> key derivation function takes a data key as input and returns the data encryption keys that are actually used to encrypt your records. For this reason, we often say that data is encrypted "under" a data key rather than "by" the data key.

Each encrypted data key includes metadata, including the identifier of the wrapping key that encrypted it. This metadata makes it possible for the AWS Database Encryption SDK to identify valid wrapping keys when decrypting.

Wrapping key

A *wrapping key* is a key-encryption key that the AWS Database Encryption SDK uses to encrypt the data key that encrypts your records. Each data key can be encrypted under one or more wrapping keys. You determine which wrapping keys are used to protect your data when you configure a keyring.



The AWS Database Encryption SDK supports several commonly used wrapping keys, such as AWS Key Management Service (AWS KMS) symmetric encryption KMS keys (including multi-Region AWS KMS keys) and asymmetric RSA KMS keys, raw AES-GCM (Advanced Encryption Standard/Galois Counter Mode) keys, and raw RSA keys. We recommend using KMS keys whenever possible. To decide which wrapping key you should use, see Selecting wrapping keys.

When you use envelope encryption, you need to protect your wrapping keys from unauthorized access. You can do this in any of the following ways:

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

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

Keyrings

To specify the wrapping keys you use for encryption and decryption, you use a keyring. You can use the keyrings that the AWS Database Encryption SDK provides or design your own implementations.

A *keyring* generates, encrypts, and decrypts data keys. It also generates the MAC keys used to calculate the Hash-Based Message Authentication Codes (HMACs) in the signature. When you define a keyring, you can specify the wrapping keys that encrypt your data keys. Most keyrings specify at least one wrapping key or a service that provides and protects wrapping keys. When encrypting, the AWS Database Encryption SDK uses all of the wrapping keys specified in the keyring to encrypt the data key. For help with choosing and using the keyrings that the AWS Database Encryption SDK defines, see Using keyrings.

Cryptographic actions

Cryptographic actions tell the encryptor which actions to perform on each field in a record.

The cryptographic action values can be one of the following:

- **Encrypt and sign** – Encrypt the field. Include the encrypted field in the signature.
- **Sign only** – Include the field in the signature.
- **Sign and include in encryption context** – Include the field in the signature and encryption context.

  By default, the partition and sort keys are the only attribute included in the encryption context. You might consider defining additional fields as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` so that the branch key ID supplier for your AWS KMS Hierarchical keyring can identify which branch key is required for decryption from the encryption context. For more information, see branch key ID supplier.

  > **Note**
  >
  > To use the `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action, you must use version 3.3 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before updating your data model to include `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

- **Do nothing** – Do not encrypt or include the field in the signature.

For any field that can store sensitive data, use **Encrypt and sign**. For primary key values (for example, a partition key and sort key in a DynamoDB table), use **Sign only** or **Sign and include in encryption context**. If you specify any **Sign and include in encryption context** attributes, then the partition and sort attributes must also be **Sign and include in encryption context**. You do not need to specify cryptographic actions for the material description. The AWS Database Encryption SDK automatically signs the field that the material description is stored in.

Choose your cryptographic actions carefully. When in doubt, use **Encrypt and sign**. After you have used the AWS Database Encryption SDK to protect your records, you cannot change an existing `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` field to `DO_NOTHING`, or change the cryptographic action assigned to an existing `DO_NOTHING` field. However, you can still make other changes to your data model. For example, you can add or remove encrypted fields, in a single deployment.

Material description

The material description serves as the header for an encrypted record. When you encrypt and sign fields with the AWS Database Encryption SDK, the encryptor records the material description as it assembles the cryptographic materials and stores the material description in a new field (`aws_dbe_head`) that the encryptor adds to your record.

The material description is a portable formatted data structure that contains encrypted copies of the data keys and other information, such as encryption algorithms, encryption context, and encryption and signing instructions. The encryptor records the material description as it assembles the cryptographic materials for encryption and signing. Later, when it needs to assemble cryptographic materials to verify and decrypt a field, it uses the material description as its guide.

Storing the encrypted data keys alongside the encrypted field streamlines the decryption operation and frees you from having to store and manage encrypted data keys independently of the data that they encrypt.

For technical information about the material description, see Material description format.

Encryption context

To improve the security of your cryptographic operations, the AWS Database Encryption SDK includes an encryption context in all requests to encrypt and sign a record.

An *encryption context* is a set of name-value pairs that contain arbitrary, non-secret additional authenticated data. The AWS Database Encryption SDK includes the logical name for your database and primary key values (for example, a partition key and sort key in a DynamoDB table) in the encryption context. When you encrypt and sign a field, the encryption context is cryptographically bound to the encrypted record so that the same encryption context is required to decrypt the field.

If you use an AWS KMS keyring, the AWS Database Encryption SDK also uses the encryption context to provide additional authenticated data (AAD) in the calls the keyring makes to AWS KMS.

Whenever you use the default algorithm suite, the cryptographic materials manager (CMM) adds a name-value pair to the encryption context that consists of a reserved name, `aws-crypto-public-key`, and a value that represents the public verification key. The public verification key is stored in the material description.

Cryptographic materials manager

The cryptographic materials manager (CMM) assembles the cryptographic materials that are used to encrypt, decrypt, and sign your data. Whenever you use the default algorithm suite, the *cryptographic materials* include plaintext and encrypted data keys, symmetric signing keys, and an asymmetric signing key. You never interact with the CMM directly. The encryption and decryption methods handle it for you.

Because the CMM acts as a liaison between the AWS Database Encryption SDK and a keyring, it is an ideal point for customization and extension, such as support for policy enforcement. You can explicitly specify a CMM, but it's not required. When you specify a keyring, the AWS Database Encryption SDK creates a default CMM for you. The default CMM gets the encryption or decryption materials from the keyring that you specify. This might involve a call to a cryptographic service, such as AWS Key Management Service (AWS KMS).

Symmetric and asymmetric encryption

Symmetric encryption uses the same key to encrypt and decrypt data.

Asymmetric encryption uses a mathematically related data key pair. One key in the pair encrypts the data; only the other key in the pair can decrypt the data.

The AWS Database Encryption SDK uses envelope encryption. It encrypts your data with a symmetric data key. It encrypts the symmetric data key with one or more symmetric or asymmetric wrapping keys. It adds a material description to the record that includes at least one encrypted copy of the data key.

Encrypting your data (symmetric encryption)

To encrypt your data, the AWS Database Encryption SDK uses a symmetric data key and an algorithm suite that includes a symmetric encryption algorithm. To decrypt the data, the AWS Database Encryption SDK uses the same data key and the same algorithm suite.

Encrypting your data key (symmetric or asymmetric encryption)

The keyring that you supply to an encrypt and decrypt operation determines how the symmetric data key is encrypted and decrypted. You can choose a keyring that uses symmetric encryption, such as an AWS KMS keyring with a symmetric encryption KMS key, or one that uses asymmetric encryption, such as an AWS KMS keyring with an asymmetric RSA KMS key.

Key commitment

The AWS Database Encryption SDK supports *key commitment* (sometimes known as *robustness*), a security property that ensures that each ciphertext can be decrypted only to a single plaintext. To do this, key commitment ensures that only the data key that encrypted your record will be used to decrypt it. The AWS Database Encryption SDK includes key commitment for all encryption and decryption operations.

Most modern symmetric ciphers (including AES) encrypt plaintext under a single secret key, like the unique data key that the AWS Database Encryption SDK uses to encrypt each plaintext field marked `ENCRYPT_AND_SIGN` in a record. Decrypting this record with the same data key returns a plaintext that is identical to the original. Decrypting with a different key will usually fail. Although difficult, it's technically possible to decrypt a ciphertext under two different keys. In rare cases, it is feasible to find a key that can partially decrypt ciphertext into a different, but still intelligible, plaintext.

The AWS Database Encryption SDK always encrypts each attribute under one unique data key. It might encrypt that data key under multiple wrapping keys, but the wrapping keys always encrypt the same data key. Nonetheless, a sophisticated, manually crafted encrypted record might actually contain different data keys, each encrypted by a different wrapping key. For example, if one user decrypts the encrypted record it returns 0x0 (false) while another user decrypting the same encrypted record gets 0x1 (true).

To prevent this scenario, the AWS Database Encryption SDK includes key commitment when encrypting and decrypting. The encrypt method cryptographically binds the unique data key that produced the ciphertext to the *key commitment*, a Hash-Based Message Authentication Code

(HMAC) calculated over the material description using a derivation of the data key. Then it stores the key commitment in the material description. When it decrypts a record with key commitment, the AWS Database Encryption SDK verifies that the data key is the only key for that encrypted record. If data key verification fails, the decrypt operation fails.

Digital signatures

The AWS Database Encryption SDK encrypts your data using an authenticated encryption algorithm, AES-GCM, and the decryption process verifies the integrity and authenticity of an encrypted message without using a digital signature. But because AES-GCM uses symmetric keys, anyone who can decrypt the data key used to decrypt the ciphertext could also manually create a new encrypted ciphertext, causing a potential security concern. For instance, if you use an AWS KMS key as a wrapping key, a user with `kms:Decrypt` permissions could create encrypted ciphertexts without calling `kms:Encrypt`.

To avoid this issue, the default algorithm suite adds an Elliptic Curve Digital Signature Algorithm (ECDSA) signature to encrypted records. The default algorithm suite encrypts the fields in your record marked `ENCRYPT_AND_SIGN` using an authenticated encryption algorithm, AES-GCM. Then, it calculates both Hash-Based Message Authentication Codes (HMACs) and asymmetric ECDSA signatures over the fields in your record marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. The decryption process uses the signatures to verify that an authorized user encrypted the record.

When the default algorithm suite is used, the AWS Database Encryption SDK generates a temporary private key and public key pair for each encrypted record. The AWS Database Encryption SDK stores the public key in the material description and discards the private key. This ensures that no one can create another signature that verifies with the public key. The algorithm binds the public key to the encrypted data key as additional authenticated data in the material description, preventing users who can only decrypt fields from altering the public key or affecting signature verification.

The AWS Database Encryption SDK always includes HMAC verification. ECDSA digital signatures are enabled by default, but not required. If the users encrypting data and the users decrypting data are equally trusted, you might consider using an algorithm suite that does not include digital signatures to improve your performance. For more information on selecting alternative algorithm suites, see Choosing an algorithm suite.

> **Note**
>
> If a keyring doesn't delineate between encryptors and decryptors, digital signatures provide no cryptographic value.

AWS KMS keyrings, including the asymmetric RSA AWS KMS keyring, can delineate between encryptors and decryptors based on AWS KMS key policies and IAM policies.

Due to their cryptographic nature, the following keyrings cannot delineate between encryptors and decryptors:

- AWS KMS Hierarchical keyring
- AWS KMS ECDH keyring
- Raw AES keyring
- Raw RSA keyring
- Raw ECDH keyring

How the AWS Database Encryption SDK works

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK provides client-side encryption libraries that are designed specifically to protect the data that you store in databases. The libraries include secure implementations that you can extend or use unchanged. For more information about defining and using custom components, see the GitHub repository for your database implementation.

The workflows in this section explain how the AWS Database Encryption SDK encrypts and signs and decrypts and verifies the data in your database. These workflows describe the basic process using abstract elements and the default features. For details about how the AWS Database Encryption SDK works with your database implementation, see the *What is encrypted* topic for your database.

The AWS Database Encryption SDK uses envelope encryption to protect your data. Each record is encrypted under a unique data key. The data key is used to derive a unique *data encryption key* for each field marked `ENCRYPT_AND_SIGN` in your cryptographic actions. Then, a copy of data key is encrypted by the wrapping keys you specify. To decrypt the encrypted record, the AWS Database Encryption SDK uses the wrapping keys you specify to decrypt at least one encrypted data key. Then it can decrypt the ciphertext and return a plaintext entry.

For more information about the terms used in the AWS Database Encryption SDK, see AWS Database Encryption SDK concepts.

Encrypt and sign

At its core, the AWS Database Encryption SDK is a record encryptor that encrypts, signs, verifies, and decrypts the records in your database. It takes in information about your records and instructions about which fields to encrypt and sign. It gets the encryption materials, and instructions on how to use them, from a cryptographic materials manager configured from the wrapping key you specify.

The following walkthrough describes how the AWS Database Encryption SDK encrypts and signs your data entries.

1. The cryptographic materials manager provides the AWS Database Encryption SDK with unique data encryption keys: one plaintext data key, a copy of the data key encrypted by the specified wrapping key, and a MAC key.

   Note

   You can encrypt the data key under multiple wrapping keys. Each of the wrapping keys encrypt a separate copy of the data key. The AWS Database Encryption SDK stores all of the encrypted data keys in the material description. The AWS Database Encryption SDK adds a new field (`aws_dbe_head`) to the record that stores the material description.
   A MAC key is derived for each encrypted copy of the data key. The MAC keys are not stored in the material description. Instead, the decrypt method uses the wrapping keys to derive the MAC keys again.

2. The encryption method encrypts each field marked as `ENCRYPT_AND_SIGN` in the cryptographic actions you specified.
3. The encryption method derives a `commitKey` from the data key and uses it to generate a key commitment value, and then discards the data key.

4. The encryption method adds a material description to the record. The material description contains the encrypted data keys and the other information about the encrypted record. For a complete list of the information included in the material description, see Material description format.

5. The encryption method uses the MAC keys returned in **Step 1** to calculate Hash-Based Message Authentication Code (HMAC) values over the canonicalization of the material description, encryption context, and each field marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in the cryptographic actions. The HMAC values are stored in a new field (`aws_dbe_foot`) that the encryption method adds to the record.

6. The encryption method calculates an ECDSA signature over the canonicalization of the material description, encryption context, and each field marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` and stores the ECDSA signatures in the `aws_dbe_foot` field.

   > **Note**
   >
   > ECDSA signatures are enabled by default, but are not required.

7. The encryption method stores the encrypted and signed record in your database

Decrypt and verify

1. The cryptographic materials manager (CMM) provides the decryption method with the decryption materials stored in the material description, including the plaintext data key and the associated MAC key.

   - The CMM decrypts the encrypted data key with the wrapping keys in the specified keyring and returns the plaintext data key.

2. The decryption method compares and verifies the key commitment value in the material description.

3. The decryption method verifies the signatures in the signature field.

   It identifies which fields are marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` from the list of allowed unauthenticated fields that you defined. The decryption method uses the MAC key returned in **Step 1** to recalculate and compare HMAC values for the fields marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Then, it verifies the ECDSA signatures using the public key stored in the encryption context.

4. The decryption method uses the plaintext data key to decrypt each value marked `ENCRYPT_AND_SIGN`. The AWS Database Encryption SDK then discards the plaintext data key.
5. The decryption method returns the plaintext record.

Supported algorithm suites in the AWS Database Encryption SDK

> Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

An *algorithm suite* is a collection of cryptographic algorithms and related values. Cryptographic systems use the algorithm implementation to generate the ciphertext.

The AWS Database Encryption SDK uses an algorithm suite to encrypt and sign the fields in your database. All supported algorithm suites use the Advanced Encryption Standard (AES) algorithm with Galois/Counter Mode (GCM), known as AES-GCM, to encrypt raw data. The AWS Database Encryption SDK supports 256-bit encryption keys. The length of the authentication tag is always 16 bytes.

AWS Database Encryption SDK Algorithm Suites

Algorithm	Encryption algorithm	Data key length (in bits)	Key derivation algorithm	Symmetric signature algorithm	Asymmetric signature algorithm	Key commitment
Default	AES-GCM	256	HKDF with SHA-512	HMAC-SHA-384	ECDSA with P-384 and SHA-384	HKDF with SHA-512
AES-GCM without	AES-GCM	256	HKDF with SHA-512	HMAC-SHA-384	None	HKDF with SHA-512

Algorithm	Encryption algorithm	Data key length (in bits)	Key derivation algorithm	Symmetric signature algorithm	Asymmetric signature algorithm	Key commitment
ECDSA digital signatures						

Encryption algorithm

The name and mode of the encryption algorithm used. Algorithm suites in the AWS Database Encryption SDK use the Advanced Encryption Standard (AES) algorithm with Galois/Counter Mode (GCM).

Data key length

The length of the data key in bits. The AWS Database Encryption SDK supports 256-bit data keys. The data key is used as input to an HMAC-based extract-and-expand key derivation function (HKDF). The output of the HKDF is used as the data encryption key in the encryption algorithm.

Key derivation algorithm

The HMAC-based extract-and-expand key derivation function (HKDF) used to derive the data encryption key. The AWS Database Encryption SDK uses the HKDF defined in RFC 5869.

- The hash function used is SHA-512
- For the extract step:
  - No salt is used. Per the RFC, the salt is set to a string of zeros.
  - The input keying material is the data key from the keyring.
- For the expand step:
  - The input pseudorandom key is the output from the extract step.
  - The key label is the UTF-8-encoded bytes of the `DERIVEKEY` string in big endian byte order.
  - The input info is a concatenation of the algorithm ID and the key label (in that order).
  - The length of the output keying material is the **Data key length**. This output is used as the data encryption key in the encryption algorithm.

Symmetric signature algorithm

The Hash-Based Message Authentication Code (HMAC) algorithm used to generate a symmetric signature. All supported algorithm suites include HMAC verification.

The AWS Database Encryption SDK serializes the material description and all fields marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Then, it uses HMAC with a cryptographic hash function algorithm (SHA-384) to sign the canonicalization.

The symmetric HMAC signature is stored in a new field (`aws_dbe_foot`) that the AWS Database Encryption SDK adds to the record.

Asymmetric signature algorithm

The signature algorithm used to generate an asymmetric digital signature.

The AWS Database Encryption SDK serializes the material description and all fields marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Then, it uses the Elliptic Curve Digital Signature Algorithm (ECDSA) with the following specifics to sign the canonicalization:

- The elliptic curve used is the P-384, as defined in Digital Signature Standard (DSS) (FIPS PUB 186-4).
- The hash function used is SHA-384.

The asymmetric ECDSA signature is stored with the symmetric HMAC signature in the `aws_dbe_foot` field.

ECDSA digital signatures are included by default, but not required.

Key commitment

The HMAC-based extract-and-expand key derivation function (HKDF) used to derive the commit key.

- The hash function used is SHA-512
- For the extract step:
  - No salt is used. Per the RFC, the salt is set to a string of zeros.
  - The input keying material is the data key from the keyring.
- For the expand step:
  - The input pseudorandom key is the output from the extract step.

- The input info is the UTF-8-encoded bytes of the `COMMITKEY` string in big endian byte order.
- The length of the output keying material is 256 bits. This output is used as the commit key.

The commit key calculates the record commitment, a distinct 256-bit Hash-Based Message Authentication Code (HMAC) hash, over the material description. For a technical explanation of adding key commitment to an algorithm suite, see Key Committing AEADs in Cryptology ePrint Archive.

Default algorithm suite

By default, the AWS Database Encryption SDK uses an algorithm suite with AES-GCM, an HMAC-based extract-and-expand key derivation function (HKDF), HMAC verification, ECDSA digital signatures, key commitment, and a 256-bit encryption key.

The default algorithm suite includes HMAC verification (symmetric signatures) and ECDSA digital signatures (asymmetric signatures). These signatures are stored in a new field (`aws_dbe_foot`) that the AWS Database Encryption SDK adds to the record. ECDSA digital signatures are particularly useful when the authorization policy allows one set of users to encrypt data and a different set of users to decrypt data.

The default algorithm suite also derives a key commitment – an HMAC hash that ties the data key to the record. The key commitment value is an HMAC calculated from the material description and commit key. The key commitment value is then stored in the material description. Key commitment ensures that each ciphertext decrypts to only one plaintext. They do this by validating the data key used as input to the encryption algorithm. When encrypting, the algorithm suite derives a key commitment HMAC. Before decrypting, they validate that the data key produces the same key commitment HMAC. If it does not, the decrypt call fails.

AES-GCM without ECDSA digital signatures

Although the default algorithm suite is likely suitable for most applications, you can choose an alternate algorithm suite. For example, some trust models would be satisfied by an algorithm suite without ECDSA digital signatures. Use this suite only when the users who encrypt data and the users who decrypt data are equally trusted.

All AWS Database Encryption SDK algorithm suites include HMAC verification (symmetric signatures). The only difference, is that the AES-GCM algorithm suite without ECDSA digital signature lacks the asymmetric signature that provides an additional layer of authenticity and non-repudiation.

For example, if you have multiple wrapping keys in your keyring, `wrappingKeyA`, `wrappingKeyB`, and `wrappingKeyC`, and you decrypt a record using `wrappingKeyA`, the HMAC symmetric signature verifies that the record was encrypted by a user with access to `wrappingKeyA`. If you used the default algorithm suite, the HMACs provide the same verification of `wrappingKeyA`, and additionally use the ECDSA digital signature to ensure the record was encrypted by a user with encrypt permissions for `wrappingKeyA`.

To select the AES-GCM algorithm suite without digital signatures, include the following snippet in your encryption configuration.

Java

The following snippet specifies the AES-GCM algorithm suite without ECDSA digital signatures. For more information, see the section called "Encryption configuration".

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

C# / .NET

The following snippet specifies the AES-GCM algorithm suite without ECDSA digital signatures. For more information, see the section called "Encryption configuration".

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

The following snippet specifies the AES-GCM algorithm suite without ECDSA digital signatures. For more information, see the section called "Encryption configuration".

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

Using the AWS Database Encryption SDK with AWS KMS

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

To use the AWS Database Encryption SDK, you need to configure a keyring and specify one or more wrapping keys. If you don't have a key infrastructure, we recommend using AWS Key Management Service (AWS KMS).

The AWS Database Encryption SDK supports two types of AWS KMS keyrings. The traditional AWS KMS keyring uses AWS KMS keys to generate, encrypt, and decrypt data keys. You can use either symmetric encryption (`SYMMETRIC_DEFAULT`) or asymmetric RSA KMS keys. Since the AWS Database Encryption SDK encrypts and signs every record with a unique data key, the AWS KMS keyring must call AWS KMS for every encrypt and decrypt operation. For applications that need to minimize the number of calls to AWS KMS, the AWS Database Encryption SDK also supports the AWS KMS Hierarchical keyring. The Hierarchical keyring is a cryptographic materials caching solution that reduces the number of AWS KMS calls by using AWS KMS protected *branch keys* persisted in an Amazon DynamoDB table, and then locally caching branch key materials used in encrypt and decrypt operations. We recommend using the AWS KMS keyrings whenever possible.

To interact with AWS KMS, the AWS Database Encryption SDK requires the AWS KMS module of the AWS SDK for Java.

To prepare to use the AWS Database Encryption SDK with AWS KMS

1. Create an AWS account. To learn how, see How do I create and activate a new Amazon Web Services account? in the AWS Knowledge Center.
2. Create a symmetric encryption AWS KMS key. For help, see Creating Keys in the *AWS Key Management Service Developer Guide*.

> **Tip**
>
> To use the AWS KMS key programmatically, you will need the Amazon Resource Name (ARN) of the AWS KMS key. For help finding the ARN of an AWS KMS key, see Finding the Key ID and ARN in the *AWS Key Management Service Developer Guide*.

3. Generate an access key ID and security access key. You can use either the access key ID and secret access key for an IAM user or you can use the AWS Security Token Service to create a new session with temporary security credentials that include an access key ID, secret access key, and session token. As a security best practice, we recommend that you use temporary credentials instead of the long-term credentials associated with your IAM user or AWS (root) user accounts.

   To create an IAM user with an access key, see Creating IAM Users in the *IAM User Guide*.

   To generate temporary security credentials, see Requesting temporary security credentials in the *IAM User Guide*.
4. Set your AWS credentials using the instructions in the AWS SDK for Java and the access key ID and secret access key that you generated in step 3. If you generated temporary credentials, you will also need to specify the session token.

   This procedure allows AWS SDKs to sign requests to AWS for you. Code samples in the AWS Database Encryption SDK that interact with AWS KMS assume that you have completed this step.

Configuring the AWS Database Encryption SDK

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK is designed to be easy to use. Although the AWS Database Encryption SDK has several configuration options, the default values are carefully chosen to be practical and secure for most applications. However, you might need to adjust your configuration to improve performance or include a custom feature in your design.

Topics

Selecting a programming language

The AWS Database Encryption SDK for DynamoDB is available in multiple programming languages. The language implementations are designed to be fully interoperable and to offer the same features, although they might be implemented in different ways. Typically, you use the library that is compatible with your application.

Selecting wrapping keys

The AWS Database Encryption SDK generates a unique symmetric data key to encrypt each field. You don't need to configure, manage, or use the data keys. The AWS Database Encryption SDK does it for you.

However, you must select one or more wrapping keys to encrypt each data key. The AWS Database Encryption SDK supports AWS Key Management Service (AWS KMS) symmetric encryption KMS keys and asymmetric RSA KMS keys. It also supports AES symmetric keys and RSA asymmetric keys that you provide in different sizes. You are responsible for the safety and durability of your wrapping keys, so we recommend that you use an encryption key in a hardware security module or a key infrastructure service, such as AWS KMS.

To specify your wrapping keys for encryption and decryption, you use a keyring. Depending on the type of keyring you use, you can specify one wrapping key or multiple wrapping keys of the same or different types. If you use multiple wrapping keys to wrap a data key, each wrapping key will encrypt a copy of the same data key. The encrypted data keys (one per wrapping key) are stored in the material description stored alongside the encrypted field. To decrypt the data, the AWS Database Encryption SDK must first use one of your wrapping keys to decrypt an encrypted data key.

We recommend using one of the AWS KMS keyrings whenever possible. The AWS Database Encryption SDK provides the AWS KMS keyring and the AWS KMS Hierarchical keyring, which reduces the number of calls made to AWS KMS. To specify an AWS KMS key in a keyring, use a supported AWS KMS key identifier. If you use the AWS KMS Hierarchical keyring, you must specify the key ARN. For details about the key identifiers for an AWS KMS key, see Key Identifiers in the *AWS Key Management Service Developer Guide*.

- When you encrypt with an AWS KMS keyring, you can specify any valid key identifier (key ARN, alias name, alias ARN, or key ID) for a symmetric encryption KMS key. If you use an asymmetric RSA KMS key, you must specify the key ARN.

  If you specify an alias name or alias ARN for a KMS key when encrypting, the AWS Database Encryption SDK saves the key ARN currently associated with that alias; it does not save the alias. Changes to the alias don't affect the KMS key used to decrypt your data keys.
- By default, the AWS KMS keyring decrypts records in strict mode (where you specify particular KMS keys). You must use a key ARN to identify AWS KMS keys for decryption.

  When you encrypt with an AWS KMS keyring, the AWS Database Encryption SDK stores the key ARN of the AWS KMS key in the material description with the encrypted data key. When decrypting in strict mode, the AWS Database Encryption SDK verifies that the same key ARN appears in the keyring before it attempts to use the wrapping key to decrypt the encrypted data key. If you use a different key identifier, the AWS Database Encryption SDK will not recognize or use the AWS KMS key, even if the identifiers refer to the same key.
- When decrypting in discovery mode, you don't specify any wrapping keys. First, the AWS Database Encryption SDK attempts to decrypt the record with the key ARN stored in the material description. If that doesn't work, the AWS Database Encryption SDK asks AWS KMS to decrypt

the record using the KMS key that encrypted it, regardless of who owns or has access to that KMS key.

To specify a raw AES key or a raw RSA key pair as a wrapping key in a keyring, you must specify a namespace and a name. When decrypting, you must use the exact same namespace and name for each raw wrapping key as you used when encrypting. If you use a different namespace or name, the AWS Database Encryption SDK will not recognize or use the wrapping key, even if the key material is the same.

Creating a discovery filter

When decrypting data encrypted with KMS keys, it's a best practice to decrypt in *strict mode*, that is, to limit the wrapping keys used to only those that you specify. However, if necessary, you can also decrypt in *discovery mode*, where you don't specify any wrapping keys. In this mode, AWS KMS can decrypt the encrypted data key using the KMS key that encrypted it, regardless of who owns or has access to that KMS key.

If you must decrypt in discovery mode, we recommend that you always use a *discovery filter*, which limits the KMS keys that can be used to those in a specified AWS account and partition. The discovery filter is optional, but it's a best practice.

Use the following table to determine the partition value for your discovery filter.

Region	Partition
AWS Regions	`aws`
China Regions	`aws-cn`
AWS GovCloud (US) Regions	`aws-us-gov`

The following example shows how to create a discovery filter. Before using the code, replace the example values with valid values for your AWS account and partition.

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
```

```
        .partition("aws")
        .accountIds(111122223333)
        .build();
```

C# / .NET

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

Rust

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

Working with multitenant databases

With the AWS Database Encryption SDK, you can configure client-side encryption for databases with a shared schema by isolating each tenant with distinct encryption materials. When considering a multitenant database, take some time to review your security requirements and how multitenancy might impact them. For example, using a multitenant database might impact your ability to combine the AWS Database Encryption SDK with another server-side encryption solution.

If you have multiple users performing encrypt operations within your database, you can use one of the AWS KMS keyrings to provide each user with a distinct key to use in their cryptographic operations. Managing the data keys for a multitenant client-side encryption solution can be complicated. We recommend organizing your data by tenant whenever possible. If the tenant is identified by the primary key values (for example, the partition key in an Amazon DynamoDB table), then managing your keys is easier.

You can use the AWS KMS keyring to isolate each tenant with a distinct AWS KMS keyring and AWS KMS keys. Based on the volume of AWS KMS calls made per tenant, you might want to use the AWS KMS Hierarchical keyring to minimize your calls to AWS KMS. The AWS KMS Hierarchical keyring is a cryptographic materials caching solution that reduces the number of AWS KMS calls by using AWS KMS protected *branch keys* persisted in an Amazon DynamoDB table, and then locally caching branch key materials used in encrypt and decrypt operations. You must use the AWS KMS Hierarchical keyring to implement searchable encryption in your database.

Creating signed beacons

The AWS Database Encryption SDK uses standard beacons and compound beacons to provide searchable encryption solutions that enable you to search encrypted records without decrypting the entire database queried. However, the AWS Database Encryption SDK also supports *signed beacons* that can be configured entirely from plaintext signed fields. Signed beacons are a type of compound beacon that index and perform complex queries on `SIGN_ONLY` and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` fields.

For example, if you have a multitenant database, you might want to create a signed beacon that enables you to query your database for records encrypted by a specific tenant's key. For more information, see Querying beacons in a multitenant database.

You must use the AWS KMS Hierarchical keyring to create signed beacons.

To configure a signed beacon, provide the following values.

Java

Compound beacon configuration

The following example defines the signed parts lists locally within the signed beacon configuration.

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

Beacon version definition

The following example defines the signed parts lists globally in the beacon version. For more information on defining the beacon version, see Using beacons.

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

C# / .NET

See the complete code sample: [BeaconConfig.cs]

Signed beacon configuration

The following example defines the signed parts lists locally within the signed beacon configuration.

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

Beacon version definition

The following example defines the signed parts lists globally in the beacon version. For more information on defining the beacon version, see [Using beacons].

```
var beaconVersions = new List<BeaconVersion>
```

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

You can define your signed parts in locally or globally defined lists. We recommend defining your signed parts in a global list in the beacon version whenever possible. By defining signed parts globally, you can define each part once and then reuse the parts in multiple compound beacon configurations. If you only intend to use a signed part once, you can define it in a local list in the signed beacon configuration. You can reference both local and global parts in your constructor list.

If you define your signed parts lists globally, you must provide a list of constructor parts that identify all of the possible ways the signed beacon can assemble the fields in your beacon configuration.

Note

To define signed parts lists globally, you must use version 3.2 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before defining any new parts globally.
You cannot update existing beacon configurations to define signed parts lists globally.

Beacon name

The name you use when querying the beacon.

A signed beacon name cannot be the same name as an unencrypted field. No two beacons can have the same beacon name.

Split character

The character used to separate the parts that make up your signed beacon.

The split character cannot appear in the plaintext values of any of the fields that the signed beacon is constructed from.

Signed parts list

Identifies the signed fields included in the signed beacon.

Each part must include a name, source, and prefix. The source is the `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` field that the part identifies. The source must be a field name or an index referring to the value of a nested field. If your part name identifies the source, you can omit the source and the AWS Database Encryption SDK will automatically use the name as its source. We recommend specifying the source as the part name whenever possible. The prefix can be any string, but it must be unique. No two signed parts in a signed beacon can have the same prefix. We recommend using a short value that distinguishes the part from other parts served by the compound beacon.

We recommend defining your signed parts globally whenever possible. You might consider defining a signed part locally if you only intend on using it in one compound beacon. A locally defined part cannot have the same prefix or name as a globally defined part.

Java

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

C# / .NET

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

Constructor list (Optional)

Identifies the *constructors* that define the different ways that the signed parts can be assembled by the signed beacon.

If you do not specify a constructor list, the AWS Database Encryption SDK assembles the signed beacon with the following default constructor.

- All signed parts in the order they were added to the signed parts list
- All parts are required

Constructors

Each constructor is an ordered list of *constructor parts* that defines one way that the signed beacon can be assembled. The constructor parts are joined together in the order they are added to the list, with each part separated by the specified split character.

Each constructor part names a signed part, and defines whether that part is required or optional within the constructor. For example, if you want to query a signed beacon on `Field1`, `Field1.Field2`, and `Field1.Field2.Field3`, mark `Field2` and `Field3` as optional and create one constructor.

Each constructor must have at least one required part. We recommend making the first part in each constructor required so that you can use the `BEGINS_WITH` operator in your queries.

A constructor succeeds if all its required parts are present in the record. When you write a new record, the signed beacon uses the constructor list to determine if the beacon can be assembled from the values provided. It attempts to assemble the beacon in the order that the constructors were added to the constructor list, and it uses the first constructor that succeeds. If no constructors succeed, the beacon is not written to the record.

All readers and writers should specify the same order of constructors to ensure that their query results are correct.

Use the following procedures to specify your own constructor list.

1. Create a constructor part for each signed part to define whether or not that part is required.

   The constructor part name must be the name of the signed field.

   The following example demonstrates how to create constructor part for one signed field.

Java

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

C# / .NET

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

2. Create a constructor for each possible way that the signed beacon can be assembled using the constructor parts you created in **Step 1**.

   For example, if you want to query on `Field1.Field2.Field3` and `Field4.Field2.Field3`, then you must create two constructors. `Field1` and `Field4` can both be required because they are defined in two separate constructors.

   Java

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

   C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
```

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

3. Create a constructor list that includes all of the constructors that you created in **Step 2**.

   Java

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

   C# / .NET

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

4. Specify the `constructorList` when you create your signed beacon.

Key stores in the AWS Database Encryption SDK

In the AWS Database Encryption SDK, a *key store* is a Amazon DynamoDB table that persists hierarchical data used by the AWS KMS Hierarchical keyring. The key store helps reduce the number of calls that you need to make to AWS KMS to perform cryptographic operations with the Hierarchical keyring.

The key store persists and manages the *branch keys* that the Hierarchical keyring uses to perform envelope encryption and protect data encryption keys. The key store stores the active branch key and all previous versions of the branch key. The *active* branch key is the most recent branch key version. The Hierarchical keyring uses a unique data encryption key for each encrypt request and encrypts each data encryption key with a unique wrapping key derived from the active branch key. The Hierarchical keyring is dependent on the hierarchy established between active branch keys and their derived wrapping keys.

Key store terminology and concepts

Key store

The DynamoDB table that persists hierarchical data, such as branch keys and beacon keys.

Root key

A symmetric encryption KMS key that generates and protects the branch keys and beacon keys in your key store.

Branch key

A data key that is reused to derive unique wrapping key for envelope encryption. You can create multiple branch keys in one key store, but each branch key can only have one active branch key version at a time. The *active* branch key is the most recent branch key version.

Branch keys are derived from AWS KMS keys using the kms:GenerateDataKeyWithoutPlaintext operation.

Wrapping key

A unique data key that is used to encrypt the data encryption key used in encrypt operations.

Wrapping keys are derived from branch keys. For more information on the key derivation process, see AWS KMS Hierarchical keyring technical details.

Data encryption key

A data key that is used in encrypt operations. The Hierarchical keyring uses a unique data encryption key for each encrypt request.

Beacon key

A data key that is used to generate beacons for searchable encryption. For more information, see Searchable encryption.

Implementing least privileged permissions

When using a key store and AWS KMS Hierarchical keyrings, we recommend that you follow the principle of least privilege by defining the following roles:

Key store administrator

Key store administrators are responsible for creating and managing the key store and the branch keys that it that persists and protects. Key store administrators should be the only users with write permissions to the Amazon DynamoDB table that serves as your key store. They should be the only users with access to privileged, administrator operations, such as `CreateKey` and `VersionKey`. You can only perform these operations when you statically configure your key store actions.

`CreateKey` is a privileged operation that can add a new KMS key ARN to your key store allowlist. This KMS key can create new active branch keys. We recommend limiting access to this operation because once a KMS key is added to the branch key store, it cannot be deleted.

Key store user

In most use cases, the key store user only interacts with key store via the Hierarchical keyring as they encrypt, decrypt, sign, and verify data. As a result, they only need read permissions to the Amazon DynamoDB table that serves as your key store. Key store users should only need access to the usage operations that make cryptographic operations possible, such as `GetActiveBranchKey`, `GetBranchKeyVersion`, and `GetBeaconKey`. They don't need permissions to create or manage the branch keys that they use.

You can perform usage operations when your key store actions are statically configured, or when they're configured for discovery. You cannot perform administrator operations (`CreateKey` and `VersionKey`) when your key store actions is configured for discovery.

If your branch key store administrator allowlisted multiple KMS keys in your branch key store, we recommend that your key store users configure their key store actions for discovery so that their Hierarchical keyring can use multiple KMS keys.

Create a key store

Before you can create branch keys or use an AWS KMS Hierarchical keyring, you must create your key store, a Amazon DynamoDB table that manages and protects your branch keys.

⚠ **Important**

Do not delete the DynamoDB table that persists your branch keys. If you delete this table, you will be unable to decrypt any data encrypted using the Hierarchical keyring.

Follow the Create a table procedures in the *Amazon DynamoDB Developer Guide*, using the following required string values for the partition key and sort key.

	Partition key	**Sort key**
Base table	`branch-key-id`	`type`

Logical key store name

When naming the DynamoDB table that serves as your key store, it's important to carefully consider the *logical key store name* that you'll specify when configuring your key store actions. The logical key store name acts as an identifier for your key store and cannot be changed after it is initially defined by the first user. You must always specify the same logical key store name in your key store actions.

There must be a one-to-one mapping between the DynamoDB table name and the logical key store name. The logical key store name is cryptographically bound to all data stored in the table to simplify DynamoDB restore operations. While the logical key store name can be different from your DynamoDB table name, we strongly recommend specifying your DynamoDB table name as the logical key store name. In the event that your table name changes after restoring your DynamoDB table from a backup, the logical key store name can be mapped to the new DynamoDB table name to ensure that the Hierarchical keyring can still access your key store.

Do not include confidential or sensitive information in your logical key store name. The logical key store name is displayed in plaintext in AWS KMS CloudTrail events as the `tablename`.

Next steps

1. the section called “Configure key store actions”
2. the section called “Create branch keys”
3. Create an AWS KMS Hierarchical keyring

Configure key store actions

Key store actions determine what operations your users can perform and how their AWS KMS Hierarchical keyring uses the KMS keys allowlisted in your key store. The AWS Database Encryption SDK supports the following key store action configurations.

Static

When you statically configure your key store, the key store can only use the KMS key associated with the KMS key ARN you provide in the `kmsConfiguration` when you configure your key store actions. An exception is thrown if a different KMS key ARN is encountered when creating, versioning, or getting a branch key.

You can specify a multi-Region KMS key in your `kmsConfiguration`, but the key's entire ARN, including the region, is persisted in the branch keys derived from the KMS key. You cannot specify a key in a different region, you must provide the exact same multi-region key for the values to match.

When you statically configure your key store actions, you can perform usage operations (`GetActiveBranchKey`, `GetBranchKeyVersion`, `GetBeaconKey`) and administrative operations (`CreateKey` and `VersionKey`). `CreateKey` is a privileged operation that can add a new KMS key ARN to your key store allowlist. This KMS key can create new active branch keys. We recommend limiting access to this operation because once a KMS key is added to the key store, it cannot be deleted.

Discovery

When you configure your key store actions for discovery, the key store can use any AWS KMS key ARN that is allowlisted in your key store. However, an exception is thrown when a multi-Region KMS key is encountered and the region in the key's ARN does not match the region of the AWS KMS client being used.

When you configure your key store for discovery, you cannot perform administrative operations, such as `CreateKey` and `VersionKey`. You can only perform the usage operations that enable encrypt, decrypt, sign, and verify operations. For more information, see the section called “Implementing least privileged permissions”.

Configure your key store actions

Before you configure your key store actions, ensure the following prerequisites are met.

- Determine what operations you need to perform. For more information, see the section called “Implementing least privileged permissions”.
- Choose a logical key store name

  There must be a one-to-one mapping between the DynamoDB table name and the logical key store name. The logical key store name is cryptographically bound to all data stored in the table to simplify DynamoDB restore operations, it cannot be changed after it is initially defined by the first user. You must always specify the same logical key store name in your key store actions. For more information, see logical key store name.

Static configuration

The following example statically configures key store actions. You must specify the name of the DynamoDB table that serves as your key store, a logical name for the key store, and the KMS key ARN that identifies a symmetric encryption KMS key.

> **Note**
>
> Carefully consider the KMS key ARN that you specify when statically configuring your key store service. The `CreateKey` operation adds the KMS key ARN to your branch key store allowlist. Once a KMS key is added to the branch key store, it cannot be deleted.

Java

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

C# / .NET

```
var kmsConfig = new KMSConfiguration { KmsKeyArn = kmsKeyArn };
  var keystoreConfig = new KeyStoreConfig
  {
      KmsClient = new AmazonKeyManagementServiceClient(),
      KmsConfiguration = kmsConfig,
      DdbTableName = keyStoreName,
      DdbClient = new AmazonDynamoDBClient(),
      LogicalKeyStoreName = logicalKeyStoreName
  };
  var keystore = new KeyStore(keystoreConfig);
```

Rust

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

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

Discovery configuration

The following example configures key store actions for discovery. You must specify the name of the DynamoDB table that serves as your key store and a logical key store name.

Java

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

C# / .NET

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

Rust

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

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

Create an active branch key

A *branch key* is a data key derived from an AWS KMS key that the AWS KMS Hierarchical keyring uses to reduce the number of calls made to AWS KMS. The *active* branch key is the most recent branch key version. The Hierarchical keyring generates a unique data key for every encrypt request and encrypts each data key with a unique wrapping key derived from the active branch key.

To create a new active branch key, you must statically configure your key store actions. `CreateKey` is a privileged operation that adds the KMS key ARN specified in your key store actions configuration to your key store allowlist. Then, the KMS key is used to generate the new active branch key. We recommend limiting access to this operation because once a KMS key is added to the key store, it cannot be deleted.

We recommend using the `CreateKey` operation through the KeyStore Admin interface in your application's control plane. This approach aligns with best practices for key management.

Don't create branch keys in the data plane. This practice can result in:

- Unnecessary calls to AWS KMS
- Multiple concurrent calls to AWS KMS in high-concurrency environments
- Multiple TransactWriteItems calls to the backing DynamoDB table.

The `CreateKey` operation includes a condition check in the `TransactWriteItems` call to prevent overwriting existing branch keys. However, creating keys in the data plane can still lead to inefficient resource usage and potential performance issues.

You can allowlist one KMS key in your key store, or you can allowlist multiple KMS keys by updating the KMS key ARN that you specify in your key store actions configuration and calling `CreateKey` again. If you allowlist multiple KMS keys, your key store users should configure their key store actions for discovery so that they can use any of the allowlisted keys in the key store that they have access to. For more information, see the section called "Configure key store actions".

Required permissions

To create branch keys, you need kms:GenerateDataKeyWithoutPlaintext and kms:ReEncrypt permissions on the KMS key specified in your key store actions.

Create a branch key

The following operation creates a new active branch key using the KMS key that youspecified in your key store actions configuration, and adds the active branch key to the DynamoDB table that serves as your key store.

When you call `CreateKey`, you can choose to specify the following optional values.

- `branchKeyIdentifier`: defines a custom `branch-key-id`.

  To create a custom `branch-key-id`, you must also include an additional encryption context with the `encryptionContext` parameter.
- `encryptionContext`: defines an optional set of non-secret key–value pairs that provides additional authenticated data (AAD) in the encryption context included in the kms:GenerateDataKeyWithoutPlaintext call.

  This additional encryption context is displayed with the `aws-crypto-ec:` prefix.

Java

```
final Map<String, String> additionalEncryptionContext =
 Collections.singletonMap("Additional Encryption Context for",
          "custom branch key id");

  final String BranchKey = keystore.CreateKey(
          CreateKeyInput.builder()
                  .branchKeyIdentifier(custom-branch-key-id) //OPTIONAL
                  .encryptionContext(additionalEncryptionContext) //OPTIONAL

                  .build()).branchKeyIdentifier();
```

C# / .NET

```
var additionalEncryptionContext = new Dictionary<string, string>();
  additionalEncryptionContext.Add("Additional Encryption Context for", "custom
 branch key id");

  var branchKeyId = keystore.CreateKey(new CreateKeyInput
  {
      BranchKeyIdentifier = "custom-branch-key-id", // OPTIONAL
      EncryptionContext = additionalEncryptionContext // OPTIONAL
  });
```

Rust

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

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

First, the `CreateKey` operation generates the following values.

- A version 4 Universally Unique Identifier (UUID) for the `branch-key-id` (unless you specified a custom `branch-key-id`).
- A version 4 UUID for the branch key version
- A `timestamp` in the ISO 8601 date and time format in Coordinated Universal Time (UTC).

Then, the `CreateKey` operation calls kms:GenerateDataKeyWithoutPlaintext using the following request.

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

> **Note**
>
> The `CreateKey` operation creates an active branch key and a beacon key, even if you have not configured your database for searchable encryption. Both keys are stored in your key store. For more information, see Using the Hierarchical keyring for searchable encryption.

Next, the `CreateKey` operation calls kms:ReEncrypt to create an active record for the branch key by updating the encryption context.

Last, the `CreateKey` operation calls ddb:TransactWriteItems to write a new item that will persist the branch key in the table you created in **Step 2**. The item has the following attributes.

```
{
      "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"
 }
```

Rotate your active branch key

There can only be one active version for each branch key at a time. Typically, each active branch key version is used to satisfy multiple requests. But you control the extent to which active branch keys are reused and determine how often the active branch key is rotated.

Branch keys are not used to encrypt plaintext data keys. They are used to derive the unique wrapping keys that encrypt plaintext data keys. The wrapping key derivation process produces a unique 32 byte wrapping key with 28 bytes of randomness. This means that a branch key can derive more than 79 octillion, or 2^{96}, unique wrapping keys before cryptographic wear-out occurs. Despite this very low exhaustion risk, you might be required to rotate your active branch keys due to business or contract rules or government regulations.

The active version of the branch key remains active until you rotate it. Previous versions of the active branch key will not be used to perform encrypt operations and cannot be used to derive new wrapping keys, but they can still be queried and provide wrapping keys to decrypt the data keys that they encrypted while active.

⚠ **Warning**

Deleting branch keys in test environments is irreversible. You cannot recover deleted branch keys. When you delete and recreate branch keys with the same ID in test environments, the following issues can occur:

- Materials from previous test runs might remain in the cache
- Some test hosts or threads might encrypt data using deleted branch keys
- Data encrypted with deleted branches can't be decrypted

To prevent encryption failures in integration tests:

- Reset the hierarchical keyring reference before creating new branch keys OR
- Use unique branch key IDs for each test

Required permissions

To rotate branch keys, you need kms:GenerateDataKeyWithoutPlaintext and kms:ReEncrypt permissions on the KMS key specified in your key store actions.

Rotate an active branch key

Use the `VersionKey` operation to rotate your active branch key. When you rotate the active branch key, a new branch key is created to replace the previous version. The `branch-key-id` does not change when you rotate the active branch key. You must specify the `branch-key-id` that identifies the current active branch key when you call `VersionKey`.

Java

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

C# / .NET

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

Rust

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

Keyrings

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK uses *keyrings* to perform envelope encryption. Keyrings generate, encrypt, and decrypt data keys. Keyrings determine the source of the unique data keys that protect each encrypted record, and the wrapping keys that encrypt that data key. You specify a keyring when encrypting and the same or a different keyring when decrypting.

You can use each keyring individually or combine keyrings into a multi-keyring. Although most keyrings can generate, encrypt, and decrypt data keys, you might create a keyring that performs only one particular operation, such as a keyring that only generates data keys, and use that keyring in combination with others.

We recommend that you use a keyring that protects your wrapping keys and performs cryptographic operations within a secure boundary, such as the AWS KMS keyring, which uses AWS KMS keys that never leave AWS Key Management Service (AWS KMS) unencrypted. You can also write a keyring that uses wrapping keys that are stored in your hardware security modules (HSMs) or protected by other master key services.

Your keyring determines the wrapping keys that protect your data keys, and ultimately, your data. Use the most secure wrapping keys that are practical for your task. Whenever possible use wrapping keys that are protected by a hardware security module (HSM) or a key management infrastructure, such as KMS keys in AWS Key Management Service (AWS KMS) or encryption keys in AWS CloudHSM.

The AWS Database Encryption SDK provides several keyrings and keyring configurations, and you can create your own custom keyrings. You can also create a multi-keyring that includes one or more keyrings of the same or a different type.

Topics

- AWS KMS ECDH keyrings
- Raw AES keyrings
- Raw RSA keyrings
- Raw ECDH keyrings
- Multi-keyrings

How keyrings work

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

When you encrypt and sign a field in your database, the AWS Database Encryption SDK asks the keyring for encryption materials. The keyring returns a plaintext data key, a copy of the data key that's encrypted by each of the wrapping keys in the keyring, and a MAC key that is associated with the data key. The AWS Database Encryption SDK uses the plaintext key to encrypt the data, and then removes the plaintext data key from memory as soon as possible. Then, the AWS Database Encryption SDK adds a material description that includes the encrypted data keys and other information, such as encryption and signing instructions. The AWS Database Encryption SDK uses the MAC key to calculate Hash-Based Message Authentication Codes (HMACs) over the canonicalization of the material description and all fields marked `ENCRYPT_AND_SIGN` or `SIGN_ONLY`.

When you decrypt data, you can use the same keyring that you used to encrypt the data, or a different one. To decrypt the data, a decryption keyring must have access to at least one wrapping key in the encryption keyring.

The AWS Database Encryption SDK passes the encrypted data keys from the material description to the keyring, and asks the keyring to decrypt any one of them. The keyring uses its wrapping keys to decrypt one of the encrypted data keys and returns a plaintext data key. The AWS Database Encryption SDK uses the plaintext data key to decrypt the data. If none of the wrapping keys in the keyring can decrypt any of the encrypted data keys, the decrypt operation fails.

You can use a single keyring or also combine keyrings of the same type or a different type into a multi-keyring. When you encrypt data, the multi-keyring returns a copy of the data key encrypted by all of the wrapping keys in all of the keyrings that comprise the multi-keyring and a MAC key that is associated with the data key. You can decrypt the data using a keyring with any one of the wrapping keys in the multi-keyring.

AWS KMS keyrings

> Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

An AWS KMS keyring uses symmetric encryption or asymmetric RSA AWS KMS keys to generate, encrypt, and decrypt data keys. AWS Key Management Service (AWS KMS) protects your KMS keys and performs cryptographic operations within the FIPS boundary. We recommend that you use a AWS KMS keyring, or a keyring with similar security properties, whenever possible.

You can also use a symmetric multi-Region KMS key in an AWS KMS keyring. For more details and examples using multi-Region AWS KMS keys, see Using multi-Region AWS KMS keys. For information about multi-Region keys, see Using multi-Region keys in the *AWS Key Management Service Developer Guide*.

AWS KMS keyrings can include two types of wrapping keys:

- **Generator key**: Generates a plaintext data key and encrypts it. A keyring that encrypts data must have one generator key.
- **Additional keys**: Encrypts the plaintext data key that the generator key generated. AWS KMS keyrings can have zero or more additional keys.

You must have a generator key to encrypt records. When an AWS KMS keyring has just one AWS KMS key, that key is used to generate and encrypt the data key.

Like all keyrings, AWS KMS keyrings can be used independently or in a multi-keyring with other keyrings of the same or a different type.

Topics

- Required permissions for AWS KMS keyrings
- Identifying AWS KMS keys in an AWS KMS keyring
- Creating an AWS KMS keyring
- Using multi-Region AWS KMS keys

- Using an AWS KMS discovery keyring
- Using an AWS KMS regional discovery keyring

Required permissions for AWS KMS keyrings

The AWS Database Encryption SDK doesn't require an AWS account and it doesn't depend on any AWS service. However, to use an AWS KMS keyring, you need an AWS account and the following minimum permissions on the AWS KMS keys in your keyring.

- To encrypt with an AWS KMS keyring, you need kms:GenerateDataKey permission on the generator key. You need kms:Encrypt permission on all additional keys in the AWS KMS keyring.
- To decrypt with an AWS KMS keyring, you need kms:Decrypt permission on at least one key in the AWS KMS keyring.
- To encrypt with a multi-keyring comprised of AWS KMS keyrings, you need kms:GenerateDataKey permission on the generator key in the generator keyring. You need kms:Encrypt permission on all other keys in all other AWS KMS keyrings.
- To encrypt with an asymmetric RSA AWS KMS keyring, you do not need kms:GenerateDataKey or kms:Encrypt because you must specify the public key material that you want to use for encryption when you create the keyring. No AWS KMS calls are made when encrypting with this keyring. To decrypt with an asymmetric RSA AWS KMS keyring, you need kms:Decrypt permission.

For detailed information about permissions for AWS KMS keys, see Authentication and access control in the *AWS Key Management Service Developer Guide*.

Identifying AWS KMS keys in an AWS KMS keyring

An AWS KMS keyring can include one or more AWS KMS keys. To specify an AWS KMS key in an AWS KMS keyring, use a supported AWS KMS key identifier. The key identifiers you can use to identify an AWS KMS key in a keyring vary with the operation and the language implementation. For details about the key identifiers for an AWS KMS key, see Key Identifiers in the *AWS Key Management Service Developer Guide*.

As a best practice, use the most specific key identifier that is practical for your task.

- To encrypt with an AWS KMS keyring, you can use a key ID, key ARN, alias name, or alias ARN to encrypt data.

Note

If you specify an alias name or alias ARN for a KMS key in an encryption keyring, the encrypt operation saves the key ARN currently associated with the alias in the metadata of the encrypted data key. It does not save the alias. Changes to the alias don't affect the KMS key used to decrypt your encrypted data keys.

- To decrypt with an AWS KMS keyring, you must use a key ARN to identify AWS KMS keys. For details, see Selecting wrapping keys.
- In a keyring used for encryption and decryption, you must use a key ARN to identify AWS KMS keys.

When decrypting, the AWS Database Encryption SDK searches the AWS KMS keyring for an AWS KMS key that can decrypt one of the encrypted data keys. Specifically, the AWS Database Encryption SDK uses the following pattern for each encrypted data key in the material description.

- The AWS Database Encryption SDK gets the key ARN of the AWS KMS key that encrypted the data key from the metadata of the material description.
- The AWS Database Encryption SDK searches the decryption keyring for an AWS KMS key with a matching key ARN.
- If it finds an AWS KMS key with a matching key ARN in the keyring, the AWS Database Encryption SDK asks AWS KMS to use the KMS key to decrypt the encrypted data key.
- Otherwise, it skips to the next encrypted data key, if any.

Creating an AWS KMS keyring

You can configure each AWS KMS keyring with a single AWS KMS key or multiple AWS KMS keys in the same or different AWS accounts and AWS Regions. The AWS KMS key must be a symmetric encryption key (`SYMMETRIC_DEFAULT`) or an asymmetric RSA KMS key. You can also use a symmetric encryption multi-Region KMS key. You can use one or more AWS KMS keyrings in a multi-keyring.

You can create an AWS KMS keyring that encrypts and decrypts data, or you can create AWS KMS keyrings specifically for encrypting or decrypting. When you create an AWS KMS keyring to encrypt data, you must specify a *generator key*, which is an AWS KMS key that is used to generate a plaintext data key and encrypt it. The data key is mathematically unrelated to the KMS key. Then, if you choose, you can specify additional AWS KMS keys that encrypt the same plaintext data key. To decrypt an encrypted field protected by this keyring, the decryption keyring that you use must include at least one of the AWS KMS keys defined in the keyring, or no AWS KMS keys. (An AWS KMS keyring with no AWS KMS keys is known as an AWS KMS discovery keyring.)

All wrapping keys in an encryption keyring or multi-keyring must be able to encrypt the data key. If any wrapping key fails to encrypt, the encrypt method fails. As a result, the caller must have the required permissions for all keys in the keyring. If you use a discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails.

The following examples use the `CreateAwsKmsMrkMultiKeyring` method to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method automatically creates the AWS KMS client and ensures that the keyring will correctly handle both single-Region and multi-Region keys. These examples use a key ARNs to identify the KMS keys. For details, see Identifying AWS KMS keys in an AWS KMS keyring

Java

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

C# / .NET

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

Rust

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

The following examples use the `CreateAwsKmsRsaKeyring` method to create an AWS KMS keyring with an asymmetric RSA KMS key. To create an asymmetric RSA AWS KMS keyring, provide the following values.

- `kmsClient`: create a new AWS KMS client
- `kmsKeyID`: the key ARN that identifies your asymmetric RSA KMS key
- `publicKey`: a ByteBuffer of a UTF-8 encoded PEM file that represents the public key of the key you passed to `kmsKeyID`
- `encryptionAlgorithm`: the encryption algorithm must be `RSAES_OAEP_SHA_256` or `RSAES_OAEP_SHA_1`

Java

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

C# / .NET

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

Rust

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

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

Using multi-Region AWS KMS keys

You can use multi-Region AWS KMS keys as wrapping keys in the AWS Database Encryption SDK. If you encrypt with a multi-Region key in one AWS Region, you can decrypt using a related multi-Region key in a different AWS Region.

Multi-Region KMS keys are a set of AWS KMS keys in different AWS Regions that have the same key material and key ID. You can use these *related* keys as though they were the same key in different Regions. Multi-Region keys support common disaster recovery and backup scenarios that require encrypting in one Region and decrypting in a different Region without making a cross-Region call to AWS KMS. For information about multi-Region keys, see Using multi-Region keys in the *AWS Key Management Service Developer Guide*.

To support multi-Region keys, the AWS Database Encryption SDK includes AWS KMS multi-Region-aware keyrings. The `CreateAwsKmsMrkMultiKeyring` method supports both single-Region and multi-Region keys.

- For single-Region keys, the multi-Region-aware symbol behaves just like the single-Region AWS KMS keyring. It attempts to decrypt ciphertext only with the single-Region key that encrypted the data. To simplify your AWS KMS keyring experience, we recommend using the `CreateAwsKmsMrkMultiKeyring` method whenever you use a symmetric encryption KMS key.
- For multi-Region keys, the multi-Region-aware symbol attempts to decrypt ciphertext with the same multi-Region key that encrypted the data or with the related multi-Region key in the Region you specify.

In the multi-Region-aware keyrings that take more than one KMS key, you can specify multiple single-Region and multi-Region keys. However, you can specify only one key from each set of related multi-Region keys. If you specify more than one key identifier with the same key ID, the constructor call fails.

The following examples create an AWS KMS keyring with a multi-Region KMS key. The examples specify a multi-Region key as the generator key and a single-Region key as the child key.

Java

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

C# / .NET

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

Rust

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

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

When you use multi-Region AWS KMS keyrings, you can decrypt ciphertext in strict mode or discover mode. To decrypt the ciphertext in strict mode, instantiate the multi-Region-aware symbol with the key ARN of the related multi-Region key in the region you are decrypting the ciphertext. If you specify the key ARN of a related multi-Region key in a different Region (for example, the region where the record was encrypted), the multi-Region-aware symbol will make a cross-Region call for that AWS KMS key.

When decrypting in strict mode, the multi-Region-aware symbol requires a key ARN. It accepts only one key ARN from each set of related multi-Region keys.

You can also decrypt in *discovery mode* with AWS KMS multi-Region keys. When decrypting in discovery mode, you don't specify any AWS KMS keys. (For information about single-Region AWS KMS discovery keyrings, see Using an AWS KMS discovery keyring.)

If you encrypted with a multi-Region key, the multi-Region-aware symbol in discovery mode will try to decrypt by using a related multi-Region key in the local Region. If none exists; the call fails. In discovery mode, the AWS Database Encryption SDK will not attempt a cross-Region call for the multi-Region key used for encryption.

Using an AWS KMS discovery keyring

When decrypting, it's a best practice to specify the wrapping keys that the AWS Database Encryption SDK can use. To follow this best practice, use an AWS KMS decryption keyring that limits the AWS KMS wrapping keys to those that you specify. However, you can also create an *AWS KMS discovery keyring*, that is, an AWS KMS keyring that doesn't specify any wrapping keys.

The AWS Database Encryption SDK provides a standard AWS KMS discovery keyring and a discovery keyring for AWS KMS multi-Region keys. For information about using multi-Region keys with the AWS Database Encryption SDK, see Using multi-Region AWS KMS keys.

Because it doesn't specify any wrapping keys, a discovery keyring can't encrypt data. If you use a discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails.

When decrypting, a discovery keyring allows the AWS Database Encryption SDK to ask AWS KMS to decrypt any encrypted data key by using the AWS KMS key that encrypted it, regardless of who owns or has access to that AWS KMS key. The call succeeds only when the caller has `kms:Decrypt` permission on the AWS KMS key.

> ⚠ **Important**
>
> If you include an AWS KMS discovery keyring in a decryption multi-keyring, the discovery keyring overrides all KMS key restrictions specified by other keyrings in the multi-keyring. The multi-keyring behaves like its least restrictive keyring. If you use a discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails

The AWS Database Encryption SDK provides an AWS KMS discovery keyring for convenience. However, we recommend that you use a more limited keyring whenever possible for the following reasons.

- **Authenticity** – An AWS KMS discovery keyring can use any AWS KMS key that was used to encrypt a data key in the material description, so long as the caller has permission to use that AWS KMS key to decrypt. This might not be the AWS KMS key that the caller intends to use. For example, one of the encrypted data keys might have been encrypted under a less secure AWS KMS key that anyone can use.
- **Latency and performance** – An AWS KMS discovery keyring might be perceptibly slower than other keyrings because the AWS Database Encryption SDK tries to decrypt all of the encrypted data keys, including those encrypted by AWS KMS keys in other AWS accounts and Regions, and AWS KMS keys that the caller doesn't have permission to use for decryption.

If you use a discovery keyring, we recommend that you use a *discovery filter* to limit the KMS keys that can be used to those in specified AWS accounts and partitions. For help finding your account ID and partition, see Your AWS account identifiers and ARN format in the *AWS General Reference*.

The following code examples instantiate an AWS KMS discovery keyring with a discovery filter that limits the KMS keys that the AWS Database Encryption SDK can use to those in the `aws` partition and 111122223333 example account.

Before using this code, replace the example AWS account and partition values with valid values for your AWS account and partition. If your KMS keys are in China Regions, use the `aws-cn` partition value. If your KMS keys are in AWS GovCloud (US) Regions, use the `aws-us-gov` partition value. For all other AWS Regions, use the `aws` partition value.

Java

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

C# / .NET

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

Rust

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

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

Using an AWS KMS regional discovery keyring

An *AWS KMS regional discovery keyring* is a keyring that doesn't specify the ARNs of KMS keys. Instead, it allows the AWS Database Encryption SDK to decrypt using only the KMS keys in particular AWS Regions.

When decrypting with an AWS KMS regional discovery keyring, the AWS Database Encryption SDK decrypts any encrypted data key that was encrypted under an AWS KMS key in the specified AWS Region. To succeed, the caller must have `kms:Decrypt` permission on at least one of the AWS KMS keys in the specified AWS Region that encrypted a data key.

Like other discovery keyrings, the regional discovery keyring has no effect on encryption. It works only when decrypting encrypted fields. If you use a regional discovery keyring in a multi-keyring that is used for encrypting and decrypting, it is effective only when decrypting. If you use a multi-Region discovery keyring to encrypt data, alone or in a multi-keyring, the encrypt operation fails.

> ⚠ **Important**
>
> If you include an AWS KMS regional discovery keyring in a decryption multi-keyring, the regional discovery keyring overrides all KMS key restrictions specified by other keyrings in the multi-keyring. The multi-keyring behaves like its least restrictive keyring. An AWS KMS discovery keyring has no effect on encryption when used by itself or in a multi-keyring.

The regional discovery keyring in the AWS Database Encryption SDK attempts to decrypt only with KMS keys in the specified Region. When you use a discovery keyring, you configure the Region on the AWS KMS client. These AWS Database Encryption SDK implementations don't filter KMS keys by Region, but AWS KMS will fail a decrypt request for KMS keys outside of the specified Region.

If you use a discovery keyring, we recommend that you use a *discovery filter* to limit the KMS keys used in decryption to those in specified AWS accounts and partitions.

For example, the following code creates an AWS KMS regional discovery keyring with a discovery filter. This keyring limits the AWS Database Encryption SDK to KMS keys in account 111122223333 in the US West (Oregon) Region (us-west-2).

Java

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

C# / .NET

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

Rust

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

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

AWS KMS Hierarchical keyrings

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

Note

As of July 24, 2023, branch keys created during the developer preview are not supported. Create new branch keys to continue using the key store that you created during the developer preview.

With the AWS KMS Hierarchical keyring, you can protect your cryptographic materials under a symmetric encryption KMS key without calling AWS KMS every time you encrypt or decrypt a record. It is a good choice for applications that need to minimize calls to AWS KMS, and applications that can reuse some cryptographic materials without violating their security requirements.

The Hierarchical keyring is a cryptographic materials caching solution that reduces the number of AWS KMS calls by using AWS KMS protected *branch keys* persisted in an Amazon DynamoDB table, and then locally caching branch key materials used in encrypt and decrypt operations. The DynamoDB table serves as the key store that manages and protects branch keys. It stores the active branch key and all previous versions of the branch key. The *active* branch key is the most recent branch key version. The Hierarchical keyring uses a unique data encryption key for each encrypt request and encrypts each data encryption key with a unique wrapping key derived from the active branch key. The Hierarchical keyring is dependent on the hierarchy established between active branch keys and their derived wrapping keys.

The Hierarchical keyring typically uses each branch key version to satisfy multiple requests. But you control the extent to which active branch keys are reused and determine how often the active branch key is rotated. The active version of the branch key remains active until you rotate it. Previous versions of the active branch key will not be used to perform encrypt operations, but they can still be queried and used in decrypt operations.

When you instantiate the Hierarchical keyring, it creates a local cache. You specify a cache limit that defines the maximum amount of time that the branch key materials are stored within the local cache before they expire and are evicted from the cache. The Hierarchical keyring makes one AWS KMS call to decrypt the branch key and assemble the branch key materials the first time a `branch-key-id` is specified in an operation. Then, the branch key materials are stored in the local cache and reused for all encrypt and decrypt operations that specify that `branch-key-id` until the cache limit expires. Storing branch key materials in the local cache reduces AWS KMS calls. For example, consider a cache limit of 15 minutes. If you perform 10,000 encrypt operations within that cache limit, the traditional AWS KMS keyring would need to make 10,000 AWS KMS calls to satisfy 10,000 encrypt operations. If you have one active `branch-key-id`, the Hierarchical keyring only needs to make one AWS KMS call to satisfy 10,000 encrypt operations.

The local cache separates encryption materials from decryption materials. The encryption materials are assembled from the active branch key and reused for all encrypt operations until the cache limit expires. The decryption materials are assembled from the branch key ID and version that is identified in the encrypted field's metadata, and they are reused for all decrypt operations related to the branch key ID and version until the cache limit expires. The local cache can store multiple versions of the same branch key at a time. When the local cache is configured to use a branch key ID supplier, it can also store branch key materials from multiple active branch keys at a time.

> **Note**
>
> All mentions of *Hierarchical keyring* in the AWS Database Encryption SDK refer to the AWS KMS Hierarchical keyring.

Topics

How it works

The following walkthroughs describe how the Hierarchical keyring assembles encryption and decryption materials, and the different calls that the keyring makes for encrypt and decrypt operations. For technical details on the wrapping key derivation and plaintext data key encryption processes, see AWS KMS Hierarchical keyring technical details.

Encrypt and sign

The following walkthrough describes how the Hierarchical keyring assembles encryption materials and derives a unique wrapping key.

1. The encryption method asks the Hierarchical keyring for encryption materials. The keyring generates a plaintext data key, then checks to see if there are valid branch key materials in the local cache to generate the wrapping key. If there are valid branch key materials, the keyring proceeds to **Step 4**.
2. If there are no valid branch key materials, the Hierarchical keyring queries the key store for the active branch key.
    a. The key store calls AWS KMS to decrypt the active branch key and returns the plaintext active branch key. Data identifying the active branch key is serialized to provide additional authenticated data (AAD) in the decrypt call to AWS KMS.

   - b. The key store returns the plaintext branch key and data that identifies it, such as the branch key version.

3. The Hierarchical keyring assembles branch key materials (the plaintext branch key and branch key version) and stores a copy of them in the local cache.
4. The Hierarchical keyring derives a unique wrapping key from the plaintext branch key and a 16-byte random salt. It uses the derived wrapping key to encrypt a copy of the plaintext data key.

The encryption method uses the encryption materials to encrypt and sign the record. For more information on how records are encrypted and signed in the AWS Database Encryption SDK, see Encrypt and sign.

Decrypt and verify

The following walkthrough describes how the Hierarchical keyring assembles decryption materials and decrypts the encrypted data key.

1. The decryption method identifies the encrypted data key from the material description field of the encrypted record, and passes it to the Hierarchical keyring.
2. The Hierarchical keyring deserializes data identifying the encrypted data key, including the branch key version, the 16-byte salt, and other information describing how the data key was encrypted.

   For more information, see AWS KMS Hierarchical keyring technical details.
3. The Hierarchical keyring checks to see if there are valid branch key materials in the local cache that match the branch key version identified in **Step 2**. If there are valid branch key materials, the keyring proceeds to **Step 6**.
4. If there are no valid branch key materials, the Hierarchical keyring queries the key store for the branch key that matches the branch key version identified in **Step 2**.
   - a. The key store calls AWS KMS to decrypt the branch key and returns the plaintext active branch key. Data identifying the active branch key is serialized to provide additional authenticated data (AAD) in the decrypt call to AWS KMS.
   - b. The key store returns the plaintext branch key and data that identifies it, such as the branch key version.

5. The Hierarchical keyring assembles branch key materials (the plaintext branch key and branch key version) and stores a copy of them in the local cache.
6. The Hierarchical keyring uses the assembled branch key materials and the 16-byte salt identified in **Step 2** to reproduce the unique wrapping key that encrypted the data key.
7. The Hierarchical keyring uses the reproduced wrapping key to decrypt the data key and returns the plaintext data key.

The decryption method uses the decryption materials and plaintext data key to decrypt and verify the record. For more information on how records are decrypted and verified in the AWS Database Encryption SDK, see Decrypt and verify.

Prerequisites

Before you create and use a Hierarchical keyring, ensure the following prerequisites are met.

- You, or your key store administrator, have created a key store and created at least one active branch key.
- You have configured your key store actions.

> **Note**
>
> How you configure your key store actions determines what operations you can perform and what KMS keys the Hierarchical keyring can use. For more information, see Key store actions.

- You have the required AWS KMS permissions to access and use the key store and branch keys. For more information, see the section called “Required permissions”.
- You have reviewed the supported cache types and configured the cache type that best fits your needs. For more information, see the section called “Choose a cache”

Required permissions

The AWS Database Encryption SDK doesn't require an AWS account and it doesn't depend on any AWS service. However, to use an Hierarchical keyring, you need an AWS account and the following minimum permissions on the symmetric encryption AWS KMS key(s) in your key store.

- To encrypt and decrypt data with the Hierarchical keyring, you need kms:Decrypt.

- To create and rotate branch keys, you need kms:GenerateDataKeyWithoutPlaintext and kms:ReEncrypt.

For more information on controlling access to your branch keys and key store, see the section called “Implementing least privileged permissions”.

Choose a cache

The Hierarchical keyring reduces the number of calls made to AWS KMS by locally caching the branch key materials used in encrypt and decrypt operations. Before you create your Hierarchical keyring, you need to decide what type of cache you want to use. You can use the default cache or customize the cache to best fits your needs.

The Hierarchical keyring supports the following cache types:

- the section called “Default cache”
- the section called “MultiThreaded cache”
- the section called “StormTracking cache”
- the section called “Shared cache”

Default cache

For most users, the Default cache fulfills their threading requirements. The Default cache is designed to support heavily multithreaded environments. When a branch key materials entry expires, the Default cache prevents multiple threads from calling AWS KMS by notifying one thread that the branch key materials entry is going to expire 10 seconds in advance. This ensures that only one thread sends a request to AWS KMS to refresh the cache.

The Default and StormTracking caches support the same threading model, but you only need to specify the entry capacity to use the Default cache. For more granular cache customizations, use the the section called “StormTracking cache”.

Unless you want to customize the number of branch key materials entries that can be stored in the local cache, you do not need to specify a cache type when you create the Hierarchical keyring. If you do not specify a cache type, the Hierarchical keyring uses the Default cache type and sets the entry capacity to 1000.

To customize the Default cache, specify the following values:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

Java

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

C# / .NET

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

Rust

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

MultiThreaded cache

The MultiThreaded cache is safe to use in multithreaded environments, but it does not provide any functionality to minimize AWS KMS or Amazon DynamoDB calls. As a result, when a branch key materials entry expires, all threads will be notified at the same time. This can result in multiple AWS KMS calls to refresh the cache.

To use the MultiThreaded cache, specify the following values:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.
- **Entry pruning tail size**: defines the number of entries to prune if the entry capacity is reached.

Java

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

C# / .NET

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

Rust

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

StormTracking cache

The StormTracking cache is designed to support heavily multithreaded environments. When a branch key materials entry expires, the StormTracking cache prevents multiple threads from calling AWS KMS by notifying one thread that the branch key materials entry is going to expire in advance. This ensures that only one thread sends a request to AWS KMS to refresh the cache.

To use the StormTracking cache, specify the following values:

- **Entry capacity**: limits the number of branch key materials entries that can be stored in the local cache.

  Default value: 1000 entries

- **Entry pruning tail size**: defines the number of branch key materials entries to prune at a time.

  Default value: 1 entry
- **Grace period**: defines the number of seconds before expiration that an attempt to refresh branch key materials is made.

  Default value: 10 seconds
- **Grace interval**: defines the number of seconds between attempts to refresh the branch key materials.

  Default value: 1 seconds
- **Fan out**: defines the number of simultaneous attempts that can be made to refresh the branch key materials.

  Default value: 20 attempts
- **In flight time to live (TTL)**: defines the number of seconds until an attempt to refresh the branch key materials times out. Any time the cache returns `NoSuchEntry` in response to a `GetCacheEntry`, that branch key is considered to be *in flight* until the same key is written with a `PutCache` entry.

  Default value: 10 seconds
- **Sleep**: defines the number of seconds that a thread should sleep if the `fanOut` is exceeded.

  Default value: 20 milliseconds

Java

```
.cache(CacheType.builder()
        .StormTracking(StormTrackingCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(10)
        .sleepMilli(20)
        .build())
```

C# / .NET

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

Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

Shared cache

By default, the Hierarchical keyring creates a new local cache every time you instantiate the keyring. However, the Shared cache can help conserve memory by enabling you to share a cache across multiple Hierarchical keyrings. Rather than creating a new cryptographic materials cache for each Hierarchical keyring you instantiate, the Shared cache stores only one cache in memory, which can be used by all the Hierarchical keyrings that reference it. The Shared cache helps optimize memory usage by avoiding the duplication of cryptographic materials across keyrings. Instead, the Hierarchical keyrings can access the same underlying cache, reducing the overall memory footprint.

When you create your Shared cache, you still define the cache type. You can specify a the section called "Default cache", the section called "MultiThreaded cache", or the section called "StormTracking cache" as the cache type, or substitute any compatible custom cache.

Partitions

Multiple Hierarchical keyrings can use a single Shared cache. When you create a Hierarchical keyring with a Shared cache you can define an optional **partition ID**. The partition ID distinguishes which Hierarchical keyring is writing to the cache. If two Hierarchical keyrings reference the same partition ID, logical key store name, and branch key ID the two keyrings will share the same cache entries in the cache. If you create two Hierarchical keyrings with the same Shared cache, but different partition IDs, each keyring will only access the cache entries from its own designated partition within the Shared cache. The partitions act as logical divisions within the shared cache, allowing each Hierarchical keyring to operate independently on its own designated partition, without interfering with the data stored in the other partition.

If you intend to reuse or share the cache entries in a partition, you must define your own partition ID. When you pass the partition ID to your Hierarchical keyring, the keyring can reuse the cache entries that are already present in the Shared cache, rather than having to retrieve and re-authorize the branch key materials again. If you do not specify a partition ID, a unique partition ID is automatically assigned to the keyring each time you instantiate the Hierarchical keyring.

The following procedures demonstrate how to create a Shared cache with the Default cache type and pass it to a Hierarchical keyring.

1. Create a `CryptographicMaterialsCache` (CMC) using the Material Providers Library (MPL).

   Java

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

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

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

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

C# / .NET

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

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

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

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

Rust

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

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

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

2. Create a `CacheType` object for the Shared cache.

   Pass the `sharedCryptographicMaterialsCache` you created in **Step 1** to the new `CacheType` object.

   Java

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

   C# / .NET

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

   Rust

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

3. Pass the `sharedCache` object from **Step 2** to your Hierarchical keyring.

   When you create a Hierarchical keyring with a Shared cache, you can optionally define a `partitionID` to share cache entries across multiple Hierarchical keyrings. If you do not specify a partition ID, the Hierarchical keyring automatically assigns the keyring a unique partition ID.

   > **Note**
   >
   > Your Hierarchical keyrings will share the same cache entries in a Shared cache if you create two or more keyrings that reference the same partition ID, logical key store

name, and branch key ID. If you do not want multiple keyrings to share the same cache entries, you must use a unique partition ID for each Hierarchical keyring.

The following example creates a Hierarchical keyring with a branch key ID supplier, and a cache limit of 600 seconds. For more information on the values defined in following Hierarchical keyring configuration, see the section called "Create a Hierarchical keyring".

Java

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

C# / .NET

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

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
clone it to
    // pass it to different Hierarchical Keyrings, it will still point to the
same
    // underlying cache, and increment the reference count accordingly.
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

Create a Hierarchical keyring

To create a Hierarchical keyring, you must provide the following values:

- **A key store name**

  The name of the DynamoDB table you, or your key store administrator, created to serve as your key store.

-

  A cache limit time to live (TTL)

  The amount of time in seconds that a branch key materials entry within the local cache can be used before it expires. The cache limit TTL dictates how often the client calls AWS KMS to authorize use of the branch keys. This value must be greater than zero. After the cache limit TTL expires, the entry is never served, and will be evicted from the local cache.

- **A branch key identifier**

  You can either statically configure the `branch-key-id` that identifies a single active branch key in your key store, or provide a branch key ID supplier.

  The *branch key ID supplier* uses the fields stored in the encryption context to determine which branch key is required to decrypt a record. By default, only the partition and sort keys are included in the encryption context. However, you can use the

`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action to include additional fields in the encryption context.

We strongly recommend using a branch key ID supplier for multitenant databases where each tenant has their own branch key. You can use the branch key ID supplier to create a friendly name for your branch key IDs to make it easy to recognize the correct branch key ID for a specific tenant. For example, the friendly name lets you refer to a branch key as `tenant1` instead of `b3f61619-4d35-48ad-a275-050f87e15122`.

For decrypt operations, you can either statically configure a single Hierarchical keyring to restrict decryption to a single tenant, or you can use the branch key ID supplier to identify which tenant is responsible for decrypting a record.

- **(Optional) A cache**

  If you want to customize your cache type or the number of branch key materials entries that can be stored in the local cache, specify the cache type and entry capacity when you initialize the keyring.

  The Hierarchical keyring supports the following cache types: Default, MultiThreaded, StormTracking, and Shared. For more information and examples demonstrating how to define each cache type, see the section called “Choose a cache”.

  If you do not specify a cache, the Hierarchical keyring automatically uses the Default cache type and sets the entry capacity to 1000.

- **(Optional) A partition ID**

  If you specify the the section called “Shared cache”, you can optionally define a partition ID. The partition ID distinguishes which Hierarchical keyring is writing to the cache. If you intend to reuse or share the cache entries in a partition, you must define your own partition ID. You can specify any string for the partition ID. If you do not specify a partition ID, a unique partition ID is automatically assigned to the keyring at creation.

  For more information, see Partitions.

  > **Note**
  >
  > Your Hierarchical keyrings will share the same cache entries in a Shared cache if you create two or more keyrings that reference the same partition ID, logical key store name,

> and branch key ID. If you do not want multiple keyrings to share the same cache entries, you must use a unique partition ID for each Hierarchical keyring.

- **(Optional) A list of Grant Tokens**

  If you control access to the KMS key in your Hierarchical keyring with grants, you must provide all necessary grant tokens when you initialize the keyring.

Create a Hierarchical keyring with a static branch key ID

The following examples demonstrate how to create a Hierarchical keyring with a static branch key ID, the the section called "Default cache", and a cache limit TTL of 600 seconds.

Java

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

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

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

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

Create a Hierarchical keyring with a branch key ID supplier

The following procedures demonstrate how to create a Hierarchical keyring with a branch key ID supplier.

1. Create a branch key ID supplier

   The following example creates friendly names for the two branch keys created in **Step 1**, and calls `CreateDynamoDbEncryptionBranchKeyIdSupplier` to create a branch key ID supplier with the AWS Database Encryption SDK for DynamoDB client.

   Java

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

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

```
            .build()).branchKeyIdSupplier();
```

C# / .NET

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

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

Rust

```
// Create friendly names for each branch_key_id
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant1: String,
    branch_key_id_for_tenant2: String,
}

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

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

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

2. Create a Hierarchical keyring

   The following examples initialize a Hierarchical keyring with the branch key ID supplier created in **Step 1**, a cache limit TLL of 600 seconds, and a maximum cache size of 1000.

   Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(100)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

   C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
```

```
    {
         Default = new DefaultCache { EntryCapacity = 100 }
    }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

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

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

Using the Hierarchical keyring for searchable encryption

Searchable encryption enables you to search encrypted records without decrypting the entire database. This is accomplished by indexing the plaintext value of an encrypted field with a beacon. To implement searchable encryption, you must use a Hierarchical keyring.

The key store `CreateKey` operation generates both a branch key and *beacon key*. The branch key is used in record encryption and decryption operations. The beacon key is used to generate beacons.

The branch key and beacon key are protected by the same AWS KMS key that you specify when creating your key store service. After the `CreateKey` operation calls AWS KMS to generate the branch key, it calls kms:GenerateDataKeyWithoutPlaintext a second time to generate the beacon key using the following request.

```
{
   "EncryptionContext": {
      "branch-key-id" : "branch-key-id",
      "type" : type,
      "create-time" : "timestamp",
      "logical-key-store-name" : "the logical table name for your key store",
      "kms-arn" : the KMS key ARN,
      "hierarchy-version" : 1
   },
   "KeyId": "the KMS key ARN",
   "NumberOfBytes": "32"
}
```

After generating both keys, the `CreateKey` operation calls ddb:TransactWriteItems to write two new items that will persist the branch key and beacon key in your branch key store.

When you configure a standard beacon, the AWS Database Encryption SDK queries the key store for the beacon key. Then, it uses an HMAC-based extract-and-expand key derivation function (HKDF) to combine the beacon key with the name of the standard beacon to create the HMAC key for a given beacon.

Unlike branch keys, there is only one beacon key version per `branch-key-id` in a key store. The beacon key is never rotated.

Defining your beacon key source

When you define the beacon version for your standard and compound beacons, you must identify the beacon key and define a cache limit time to live (TTL) for the beacon key materials. Beacon key materials are stored in a separate local cache from the branch keys. The following snippet demonstrates how to define the `keySource` for a single-tenant database. Identify your beacon key by the `branch-key-id` it is associated with.

Java

```
keySource(BeaconKeySource.builder()
        .single(SingleKeyStore.builder()
                .keyId(branch-key-id)
                .cacheTTL(6000)
                .build())
        .build())
```

C# / .NET

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

Rust

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

Defining beacon source in a multitenant database

If you have a multitenant database, you must specify the following values when configuring the `keySource`.

- **keyFieldName**

  Defines the name of the field that stores the `branch-key-id` associated with the beacon key used to generated beacons for a given tenant. The `keyFieldName` can be any string, but it must be unique to all other fields in your database. When you write new records to your database, the `branch-key-id` that identifies the beacon key used to generate any beacons for that record is stored in this field. You must include this field in your beacon queries and identify the appropriate beacon key materials required to recalculate the beacon. For more information, see Querying beacons in a multitenant database.

- **cacheTTL**

  The amount of time in seconds that a beacon key materials entry within the local beacon cache can be used before it expires. This value must be greater than zero. When the cache limit TTL expires, the entry is evicted from the local cache.

- **(Optional) A cache**

  If you want to customize your cache type or the number of branch key materials entries that can be stored in the local cache, specify the cache type and entry capacity when you initialize the keyring.

  The Hierarchical keyring supports the following cache types: Default, MultiThreaded, StormTracking, and Shared. For more information and examples demonstrating how to define each cache type, see the section called “Choose a cache”.

  If you do not specify a cache, the Hierarchical keyring automatically uses the Default cache type and sets the entry capacity to 1000.

The following example creates a Hierarchical keyring with a branch key ID supplier a cache limit TLL of 600 seconds, and an entry capacity of 1000.

Java

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

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
    Cache = new CacheType
    {
        Default = new DefaultCache { EntryCapacity = 1000 }
    }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

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

AWS KMS ECDH keyrings

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

⚠ **Important**

The AWS KMS ECDH keyring is only available with version 1.5.0 or later of the Material Providers Library.

An AWS KMS ECDH keyring uses asymmetric key agreement AWS KMS keys to derive a shared symmetric wrapping key between two parties. First, the keyring uses the Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm to derive a shared secret from the private key in the sender's KMS key pair and the recipient's public key. Then, the keyring uses the shared secret to derive the shared wrapping key that protects your data encryption keys. The key derivation function that the AWS Database Encryption SDK uses (`KDF_CTR_HMAC_SHA384`) to derive the shared wrapping key conforms to NIST recommendations for key derivation.

The key derivation function returns 64 bytes of keying material. To ensure that both parties use the correct keying material, the AWS Database Encryption SDK uses the first 32 bytes as a commitment key and the last 32 bytes as the shared wrapping key. On decrypt, if the keyring cannot reproduce the same commitment key and shared wrapping key that is stored in the material description field of the encrypted record, the operation fails. For example, if you encrypt a record with a keyring configured with **Alice's** private key and **Bob's** public key, a keyring configured with **Bob's** private key and **Alice's** public key will reproduce the same commitment key and shared wrapping key and be able to decrypt the record. If Bob's public key is not from a KMS key pair, then Bob can create a Raw ECDH keyring to decrypt the record.

The AWS KMS ECDH keyring encrypts records with a symmetric key using AES-GCM. The data key is then envelope encrypted with the derived shared wrapping key using AES-GCM. Each AWS KMS ECDH keyring can have only one shared wrapping key, but you can include multiple AWS KMS ECDH keyrings, alone or with other keyrings, in a multi-keyring.

Topics

Required permissions for AWS KMS ECDH keyrings

The AWS Database Encryption SDK doesn't require an AWS account and it doesn't depend on any AWS service. However, to use an AWS KMS ECDH keyring, you need an AWS account and the following minimum permissions on the AWS KMS keys in your keyring. The permissions vary based on which key agreement schema you use.

- To encrypt and decrypt records using the `KmsPrivateKeyToStaticPublicKey` key agreement schema, you need kms:GetPublicKey and kms:DeriveSharedSecret on the *sender's* asymmetric KMS key pair. If you directly provide the sender's DER-encoded public key when you instantiate your keyring, you only need kms:DeriveSharedSecret permission on the sender's asymmetric KMS key pair.
- To decrypt records using the `KmsPublicKeyDiscovery` key agreement schema, you need kms:DeriveSharedSecret and kms:GetPublicKey permissions on the specified asymmetric KMS key pair.

Creating an AWS KMS ECDH keyring

To create an AWS KMS ECDH keyring that encrypts and decrypts data, you must use the `KmsPrivateKeyToStaticPublicKey` key agreement schema. To initialize an AWS KMS ECDH keyring with the `KmsPrivateKeyToStaticPublicKey` key agreement schema, provide the following values:

- **Sender's AWS KMS key ID**

  Must identify an asymmetric NIST-recommended elliptic curve (ECC)KMS key pair with a `KeyUsage` value of `KEY_AGREEMENT`. The sender's private key is used to derive the shared secret.
- **(Optional) Sender's public key**

  Must be a DER-encoded X.509 public key, also known as `SubjectPublicKeyInfo` (SPKI), as defined in RFC 5280.

  The AWS KMS GetPublicKey operation returns the public key of an asymmetric KMS key pair in the required DER-encoded format.

  To reduce the number of AWS KMS calls that your keyring makes, you can directly provide the sender's public key. If no value is provided for the sender's public key, the keyring calls AWS KMS to retrieve the sender's public key.
- **Recipient's public key**

  You must provide the recipient's DER-encoded X.509 public key, also known as `SubjectPublicKeyInfo` (SPKI), as defined in RFC 5280.

  The AWS KMS GetPublicKey operation returns the public key of an asymmetric KMS key pair in the required DER-encoded format.
- **Curve specification**

  Identifies the elliptic curve specification in the specified key pairs. Both the sender and recipient's key pairs must have the same curve specification.

  Valid values: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`
- **(Optional) A list of Grant Tokens**

  If you control access to the KMS key in your AWS KMS ECDH keyring with grants, you must provide all necessary grant tokens when you initialize the keyring.

C# / .NET

The following example creates an AWS KMS ECDH keyring with the with the sender's KMS key, the sender's public key, and the recipient's public key. This example uses the optional `senderPublicKey` parameter to provide the sender's public key. If you do not provide the sender's public key, the keyring calls AWS KMS to retrieve the sender's public key. Both the sender and recipient's key pairs are on the `ECC_NIST_P256` curve.

```
// 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

The following example creates an AWS KMS ECDH keyring with the with the sender's KMS key, the sender's public key, and the recipient's public key. This example uses the optional `senderPublicKey` parameter to provide the sender's public key. If you do not provide the sender's public key, the keyring calls AWS KMS to retrieve the sender's public key. Both the sender and recipient's key pairs are on the `ECC_NIST_P256` curve.

```
// Retrieve public keys
// Must be DER-encoded X.509 public keys
ByteBuffer BobPublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab");
        ByteBuffer AlicePublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");

// Create the AWS KMS ECDH static keyring
        final CreateAwsKmsEcdhKeyringInput senderKeyringInput =
          CreateAwsKmsEcdhKeyringInput.builder()
            .kmsClient(KmsClient.create())
            .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
            .KeyAgreementScheme(
              KmsEcdhStaticConfigurations.builder()
                .KmsPrivateKeyToStaticPublicKey(
                  KmsPrivateKeyToStaticPublicKeyInput.builder()
                    .senderKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
                    .senderPublicKey(BobPublicKey)
                    .recipientPublicKey(AlicePublicKey)
                    .build()).build()).build();
```

Rust

The following example creates an AWS KMS ECDH keyring with the with the sender's KMS key, the sender's public key, and the recipient's public key. This example uses the optional `sender_public_key` parameter to provide the sender's public key. If you do not provide the sender's public key, the keyring calls AWS KMS to retrieve the sender's public key.

```
// 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?;
```

Creating an AWS KMS ECDH discovery keyring

When decrypting, it's a best practice to specify the keys that the AWS Database Encryption SDK can use. To follow this best practice, use an AWS KMS ECDH keyring with the `KmsPrivateKeyToStaticPublicKey` key agreement schema. However, you can also create an AWS KMS ECDH discovery keyring, that is, an AWS KMS ECDH keyring that can decrypt any record where the public key of the specified KMS key pair matches the *recipient's* public key stored in the material description field of the encrypted record.

> **Important**
>
> When you decrypt records using the `KmsPublicKeyDiscovery` key agreement schema, you accept all public keys, regardless of who owns it.

To initialize an AWS KMS ECDH keyring with the `KmsPublicKeyDiscovery` key agreement schema, provide the following values:

- **Recipient's AWS KMS key ID**

  Must identify an asymmetric NIST-recommended elliptic curve (ECC)KMS key pair with a `KeyUsage` value of `KEY_AGREEMENT`.
- **Curve specification**

  Identifies the elliptic curve specification in the recipient's KMS key pair.

  Valid values: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`
- **(Optional) A list of Grant Tokens**

  If you control access to the KMS key in your AWS KMS ECDH keyring with grants, you must provide all necessary grant tokens when you initialize the keyring.

C# / .NET

The following example creates an AWS KMS ECDH discovery keyring with a KMS key pair on the `ECC_NIST_P256` curve. You must have kms:GetPublicKey and kms:DeriveSharedSecret permissions on the specified KMS key pair. This keyring can decrypt any record where the public key of the specified KMS key pair matches the recipient's public key stored in the material description field of the encrypted record.

```
// 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

The following example creates an AWS KMS ECDH discovery keyring with a KMS key pair on the ECC_NIST_P256 curve. You must have kms:GetPublicKey and kms:DeriveSharedSecret permissions on the specified KMS key pair. This keyring can decrypt any record where the public key of the specified KMS key pair matches the recipient's public key stored in the material description field of the encrypted record.

```
// Create the AWS KMS ECDH discovery keyring
final CreateAwsKmsEcdhKeyringInput recipientKeyringInput =
  CreateAwsKmsEcdhKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
    .KeyAgreementScheme(
      KmsEcdhStaticConfigurations.builder()
        .KmsPublicKeyDiscovery(
          KmsPublicKeyDiscoveryInput.builder()
            .recipientKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321").build()
        ).build())
    .build();
```

Rust

```
// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;
```

```
let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

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

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
    .await?;
```

Raw AES keyrings

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK lets you use an AES symmetric key that you provide as a wrapping key that protects your data key. You need to generate, store, and protect the key material, preferably in a hardware security module (HSM) or key management system. Use a Raw AES keyring when you need to provide the wrapping key and encrypt the data keys locally or offline.

The Raw AES keyring encrypts data by using the AES-GCM algorithm and a wrapping key that you specify as a byte array. You can specify only one wrapping key in each Raw AES keyring, but you can include multiple Raw AES keyrings, alone or with other keyrings, in a multi-keyring.

Key namespaces and names

To identify the AES key in a keyring, the Raw AES keyring uses a *key namespace* and *key name* that you provide. These values are not secret. They appear in plain text in the material description that the AWS Database Encryption SDK adds to the record. We recommend using a key namespace your HSM or key management system and a key name that identifies the AES key in that system.

Note

The key namespace and key name are equivalent to the *Provider ID* (or *Provider*) and *Key ID* fields in the `JceMasterKey`.

If you construct different keyrings to encrypt and decrypt a given field, the namespace and name values are critical. If the key namespace and key name in the decryption keyring isn't an exact, case-sensitive match for the key namespace and key name in the encryption keyring, the decryption keyring isn't used, even if the key material bytes are identical.

For example, you might define a Raw AES keyring with key namespace `HSM_01` and key name `AES_256_012`. Then, you use that keyring to encrypt some data. To decrypt that data, construct a Raw AES keyring with the same key namespace, key name, and key material.

The following examples show how to create a Raw AES keyring. The `AESWrappingKey` variable represents the key material you provide.

Java

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

C# / .NET

```
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

// This example uses the key generator in Bouncy Castle to generate the key
 material.
// In production, use key material from a secure source.
var aesWrappingKey = new
 MemoryStream(GeneratorUtilities.GetKeyGenerator("AES256").GenerateKey());
```

```
// Create the keyring
var keyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

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

Rust

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

Raw RSA keyrings

> Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The Raw RSA keyring performs asymmetric encryption and decryption of data keys in local memory with an RSA public and private keys that you provide. You need to generate, store, and protect the private key, preferably in a hardware security module (HSM) or key management system. The encryption function encrypts the data key under the RSA public key. The decryption function decrypts the data key using the private key. You can select from among the several RSA padding modes.

A Raw RSA keyring that encrypts and decrypts must include an asymmetric public key and private key pair. However, you can encrypt data with a Raw RSA keyring that has only a public key, and you can decrypt data with a Raw RSA keyring that has only a private key. You can include any Raw RSA keyring in a multi-keyring. If you configure a Raw RSA keyring with a public and private key, be sure that they are part of the same key pair.

The Raw RSA keyring is equivalent to and interoperates with the JceMasterKey in the AWS Encryption SDK for Java when they are used with RSA asymmetric encryption keys.

> **Note**
>
> The Raw RSA keyring does not support asymmetric KMS keys. To use asymmetric RSA KMS keys, construct an AWS KMS keyring.

Namespaces and names

To identify the RSA key material in a keyring, the Raw RSA keyring uses a *key namespace* and *key name* that you provide. These values are not secret. They appear in plain text in the material description that the AWS Database Encryption SDK adds to the record. We recommend using the key namespace and key name that identifies the RSA key pair (or its private key) in your HSM or key management system.

> **Note**
>
> The key namespace and key name are equivalent to the *Provider ID* (or *Provider*) and *Key ID* fields in the `JceMasterKey`.

If you construct different keyrings to encrypt and decrypt a given record, the namespace and name values are critical. If the key namespace and key name in the decryption keyring isn't an exact, case-sensitive match for the key namespace and key name in the encryption keyring, the decryption keyring isn't used, even if the keys are from the same key pair.

The key namespace and key name of the key material in the encryption and decryption keyrings must be same whether the keyring contains the RSA public key, the RSA private key, or both keys in the key pair. For example, suppose you encrypt data with a Raw RSA keyring for an RSA public key with key namespace `HSM_01` and key name `RSA_2048_06`. To decrypt that data, construct a Raw RSA keyring with the private key (or key pair), and the same key namespace and name.

Padding mode

You must specify a padding mode for Raw RSA keyrings used for encryption and decryption, or use features of your language implementation that specify it for you.

The AWS Encryption SDK supports the following padding modes, subjects to the constraints of each language. We recommend an OAEP padding mode, particularly OAEP with SHA-256 and MGF1 with SHA-256 Padding. The PKCS1 padding mode is supported only for backward compatibility.

- OAEP with SHA-1 and MGF1 with SHA-1 Padding
- OAEP with SHA-256 and MGF1 with SHA-256 Padding
- OAEP with SHA-384 and MGF1 with SHA-384 Padding
- OAEP with SHA-512 and MGF1 with SHA-512 Padding
- PKCS1 v1.5 Padding

The following Java example shows how to create a Raw RSA keyring with the public and private key of an RSA key pair and the OAEP with SHA-256 and MGF1 with SHA-256 padding mode. The `RSAPublicKey` and `RSAPrivateKey` variables represent the key material you provide.

Java

```
final CreateRawRsaKeyringInput keyringInput = CreateRawRsaKeyringInput.builder()
        .keyName("RSA_2048_06")
        .keyNamespace("HSM_01")
        .paddingScheme(PaddingScheme.OAEP_SHA256_MGF1)
        .publicKey(RSAPublicKey)
        .privateKey(RSAPrivateKey)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

C# / .NET

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

```
// Get public and private keys from PEM files
var publicKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePublicKey.pem"));
var privateKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePrivateKey.pem"));

// Create the keyring input
var keyringInput = new CreateRawRsaKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
    PrivateKey = privateKey
};

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

Rust

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

Raw ECDH keyrings

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

⚠ **Important**

The Raw ECDH keyring is only available with version 1.5.0 of the Material Providers Library.

The Raw ECDH keyring uses the elliptic curve public-private key pairs that you provide to derive a shared wrapping key between two parties. First, the keyring derives a shared secret using the sender's private key, the recipient's public key, and the Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm. Then, the keyring uses the shared secret to derive the shared wrapping key that protects your data encryption keys. The key derivation function that the AWS Database Encryption SDK uses (`KDF_CTR_HMAC_SHA384`) to derive the shared wrapping key conforms to NIST recommendations for key derivation.

The key derivation function returns 64 bytes of keying material. To ensure that both parties use the correct keying material, the AWS Database Encryption SDK uses the first 32 bytes as a commitment key and the last 32 bytes as the shared wrapping key. On decrypt, if the keyring cannot reproduce the same commitment key and shared wrapping key that is stored in the material description field of the encrypted record, the operation fails. For example, if you encrypt a record with a keyring configured with **Alice's** private key and **Bob's** public key, a keyring configured with **Bob's** private key and **Alice's** public key will reproduce the same commitment key and shared wrapping key and be able to decrypt the record. If Bob's public key is from an AWS KMS key pair, then Bob can create an AWS KMS ECDH keyring to decrypt the record.

The Raw ECDH keyring encrypts records with a symmetric key using AES-GCM. The data key is then envelope encrypted with the derived shared wrapping key using AES-GCM. Each Raw ECDH keyring can have only one shared wrapping key, but you can include multiple Raw ECDH keyrings, alone or with other keyrings, in a multi-keyring.

You are responsible for generating, storing, and protecting your private keys, preferably in a hardware security module (HSM) or key management system. The sender and recipient's key pairs much be on the same elliptic curve. The AWS Database Encryption SDK supports the following elliptic cuve specifications:

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

Creating a Raw ECDH keyring

The Raw ECDH keyring supports three key agreement schemas: `RawPrivateKeyToStaticPublicKey`, `EphemeralPrivateKeyToStaticPublicKey`, and `PublicKeyDiscovery`. The key agreement schema that you select determines which cryptographic operations you can perform and how the keying materials are assembled.

Topics

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

Use the `RawPrivateKeyToStaticPublicKey` key agreement schema to statically configure the sender's private key and the recipient's public key in the keyring. This key agreement schema can encrypt and decrypt records.

To initialize a Raw ECDH keyring with the `RawPrivateKeyToStaticPublicKey` key agreement schema, provide the following values:

- **Sender's private key**

  You must provide the sender's PEM-encoded private key (PKCS #8 PrivateKeyInfo structures), as defined in RFC 5958.
- **Recipient's public key**

  You must provide the recipient's DER-encoded X.509 public key, also known as `SubjectPublicKeyInfo` (SPKI), as defined in RFC 5280.

  You can specify the public key of an asymmetric key agreement KMS key pair or the public key from a key pair generated outside of AWS.
- **Curve specification**

  Identifies the elliptic curve specification in the specified key pairs. Both the sender and recipient's key pairs must have the same curve specification.

  Valid values: `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

The following Java example uses the `RawPrivateKeyToStaticPublicKey` key agreement schema to statically configure the sender's private key and the recipient's public key. Both key pairs are on the `ECC_NIST_P256` curve.

```
private static void StaticRawKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair senderKeys = GetRawEccKey();
    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH static keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .RawPrivateKeyToStaticPublicKey(
                RawPrivateKeyToStaticPublicKeyInput.builder()
                  // Must be a PEM-encoded private key

 .senderStaticPrivateKey(ByteBuffer.wrap(senderKeys.getPrivate().getEncoded()))
                  // Must be a DER-encoded X.509 public key

 .recipientPublicKey(ByteBuffer.wrap(recipient.getPublic().getEncoded()))
                  .build()
            )
            .build()
        ).build();

    final IKeyring staticKeyring =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Rust

The following Python example uses the `raw_ecdh_static_configuration` key agreement schema to statically configure the sender's private key and the recipient's public key. Both key pairs must be on the same curve.

```
// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

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

```
// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
    .await?;
```

EphemeralPrivateKeyToStaticPublicKey

Keyrings configured with the `EphemeralPrivateKeyToStaticPublicKey` key agreement schema create a new key pair locally and derive a unique shared wrapping key for each encrypt call.

This key agreement schema can only encrypt records. To decrypt records encrypted with the `EphemeralPrivateKeyToStaticPublicKey` key agreement schema, you must use a discovery key agreement schema configured with the same recipient's public key. To decrypt, you can use a Raw ECDH keyring with the `PublicKeyDiscovery` key agreement algorithm, or, if the recipient's public key is from an asymmetric key agreement KMS key pair, you can use an AWS KMS ECDH keyring with the KmsPublicKeyDiscovery key agreement schema.

To initialize a Raw ECDH keyring with the `EphemeralPrivateKeyToStaticPublicKey` key agreement schema, provide the following values:

- **Recipient's public key**

  You must provide the recipient's DER-encoded X.509 public key, also known as `SubjectPublicKeyInfo` (SPKI), as defined in RFC 5280.

  You can specify the public key of an asymmetric key agreement KMS key pair or the public key from a key pair generated outside of AWS.

- **Curve specification**

  Identifies the elliptic curve specification in the specified public key.

  On encrypt, the keyring creates a new key pair on the specified curve and uses the new private key and specified public key to derive a shared wrapping key.

  Valid values: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

The following example creates a Raw ECDH keyring with the `EphemeralPrivateKeyToStaticPublicKey` key agreement schema. On encrypt, the keyring will create a new key pair locally on the specified `ECC_NIST_P256` curve.

```
// 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

The following example creates a Raw ECDH keyring with the `EphemeralPrivateKeyToStaticPublicKey` key agreement schema. On encrypt, the keyring will create a new key pair locally on the specified `ECC_NIST_P256` curve.

```
private static void EphemeralRawEcdhKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    ByteBuffer recipientPublicKey = getPublicKeyBytes();
    // Create the Raw ECDH ephemeral keyring
    final CreateRawEcdhKeyringInput ephemeralInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .EphemeralPrivateKeyToStaticPublicKey(
              EphemeralPrivateKeyToStaticPublicKeyInput.builder()
                .recipientPublicKey(recipientPublicKey)
                .build()
            )
            .build()
        ).build();

    final IKeyring ephemeralKeyring =
 materialProviders.CreateRawEcdhKeyring(ephemeralInput);
}
```

Rust

The following example creates a Raw ECDH keyring with the `ephemeral_raw_ecdh_static_configuration` key agreement schema. On encrypt, the keyring will create a new key pair locally on the specified curve.

```
// Create EphemeralPrivateKeyToStaticPublicKeyInput
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

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

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
```

```
        .curve_spec(ecdh_curve_spec)
        .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
        .send()
        .await?;
```

PublicKeyDiscovery

When decrypting, it's a best practice to specify the wrapping keys that the AWS Database Encryption SDK can use. To follow this best practice, use an ECDH keyring that specifies both a sender's private key and recipient's public key. However, you can also create a Raw ECDH discovery keyring, that is, a Raw ECDH keyring that can decrypt any record where the specified key's public key matches the recipient's public key stored in the material description field of the encrypted record. This key agreement schema can only decrypt records.

> ⚠ **Important**
>
> When you decrypt records using the `PublicKeyDiscovery` key agreement schema, you accept all public keys, regardless of who owns it.

To initialize a Raw ECDH keyring with the `PublicKeyDiscovery` key agreement schema, provide the following values:

- **Recipient's static private key**

  You must provide the recipient's PEM-encoded private key (PKCS #8 PrivateKeyInfo structures), as defined in RFC 5958.

- **Curve specification**

  Identifies the elliptic curve specification in the specified private key. Both the sender and recipient's key pairs must have the same curve specification.

  Valid values: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

The following example creates a Raw ECDH keyring with the `PublicKeyDiscovery` key agreement schema. This keyring can decrypt any record where the public key of the specified private key matches the recipient's public key stored in the material description field of the encrypted record.

```
// 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

The following example creates a Raw ECDH keyring with the `PublicKeyDiscovery` key agreement schema. This keyring can decrypt any record where the public key of the specified private key matches the recipient's public key stored in the material description field of the encrypted record.

```
private static void RawEcdhDiscovery() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
```

```
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
              PublicKeyDiscoveryInput.builder()
                // Must be a PEM-encoded private key

 .recipientStaticPrivateKey(ByteBuffer.wrap(sender.getPrivate().getEncoded()))
                .build()
            )
            .build()
        ).build();

    final IKeyring publicKeyDiscovery  =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Rust

The following example creates a Raw ECDH keyring with the `discovery_raw_ecdh_static_configuration` key agreement schema. This keyring can decrypt any message where the public key of the specified private key matches the recipient's public key stored on the message ciphertext.

```
// 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?;
```

Multi-keyrings

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

You can combine keyrings into a multi-keyring. A *multi-keyring* is a keyring that consists of one or more individual keyrings of the same or a different type. The effect is like using several keyrings in a series. When you use a multi-keyring to encrypt data, any of the wrapping keys in any of its keyrings can decrypt that data.

When you create a multi-keyring to encrypt data, you designate one of the keyrings as the *generator keyring*. All other keyrings are known as *child keyrings*. The generator keyring generates and encrypts the plaintext data key. Then, all of the wrapping keys in all of the child keyrings encrypt the same plaintext data key. The multi-keyring returns the plaintext key and one encrypted data key for each wrapping key in the multi-keyring. If the generator keyring is a KMS keyring, the generator key in the AWS KMS keyring generates and encrypts the plaintext key. Then, all additional AWS KMS keys in the AWS KMS keyring, and all wrapping keys in all child keyrings in the multi-keyring, encrypt the same plaintext key.

When decrypting, the AWS Database Encryption SDK uses the keyrings to try to decrypt one of the encrypted data keys. The keyrings are called in the order that they are specified in the multi-keyring. Processing stops as soon as any key in any keyring can decrypt an encrypted data key.

To create a multi-keyring, first instantiate the child keyrings. In this example, we use an AWS KMS keyring and a Raw AES keyring, but you can combine any supported keyrings in a multi-keyring.

Java

```
// 1. Create the raw AES keyring.
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateRawAesKeyringInput createRawAesKeyringInput =
 CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

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

C# / .NET

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

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

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

Rust

```
// 1. Create the raw AES keyring
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

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

Next, create the multi-keyring and specify its generator keyring, if any. In this example, we create a multi-keyring in which the AWS KMS keyring is the generator keyring and the AES keyring is the child keyring.

Java

The Java `CreateMultiKeyringInput` constructor lets you define a generator keyring and child keyrings. The resulting `createMultiKeyringInput` object is immutable.

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

C# / .NET

The .NET `CreateMultiKeyringInput` constructor lets you define a generator keyring and child keyrings. The resulting `CreateMultiKeyringInput` object is immutable.

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

Rust

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

Now, you can use the multi-keyring to encrypt and decrypt data.

Searchable encryption

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

Searchable encryption enables you to search encrypted records without decrypting the entire database. This is accomplished using *beacons*, which create a map between the plaintext value written to a field and the encrypted value that is actually stored in your database. The AWS Database Encryption SDK stores the beacon in a new field that it adds to the record. Depending on the type of beacon you use, you can perform exact match searches or more customized complex queries on your encrypted data.

Note

Searchable encryption in the AWS Database Encryption SDK differs from the searchable symmetric encryption defined in academic research, such as searchable symmetric encryption.

A beacon is a truncated Hash-Based Message Authentication Code (HMAC) tag that creates a map between the plaintext and encrypted values of a field. When you write a new value to an encrypted field that's configured for searchable encryption, the AWS Database Encryption SDK calculates an HMAC over the plaintext value. This HMAC output is a one-to-one (1:1) match for the plaintext value of that field. The HMAC output is truncated so that multiple, distinct plaintext values map to the same truncated HMAC tag. These false positives limit an unauthorized user's ability to identify distinguishing information about the plaintext value. When you query a beacon, the AWS Database Encryption SDK automatically filters out these false positives and returns the plaintext result of your query.

The average number of false positives generated for each beacon is determined by the beacon length remaining after truncation. For help determining the appropriate beacon length for your implementation, see Determining beacon length.

Note

Searchable encryption is designed to be implemented in new, unpopulated databases. Any beacon configured in an existing database will only map new records uploaded to the database, there is no way for a beacon to map existing data.

Topics

Are beacons right for my dataset?

Using beacons to perform queries on encrypted data reduces the performance costs associated with client-side encrypted databases. When you use beacons, there is an inherent tradeoff between how efficient your queries are and how much information is revealed about the distribution of your data. The beacon does not alter the encrypted state of the field. When you encrypt and sign a field with the AWS Database Encryption SDK, the plaintext value of the field is never exposed to the database. The database stores the randomized, encrypted value of the field.

Beacons are stored alongside the encrypted fields they are calculated from. This means that even if an unauthorized user cannot view the plaintext values of an encrypted field, they might be able to perform statistical analysis on the beacons to learn more about the distribution of your dataset, and, in extreme cases, identify the plaintext values that a beacon maps to. The way you configure your beacons can mitigate these risks. In particular, choosing the right beacon length can help you preserve the confidentiality of your dataset.

Security vs. Performance

- The shorter the beacon length, the more security is preserved.
- The longer the beacon length, the more performance is preserved.

Searchable encryption might not be able to provide the desired levels of both performance and security for all datasets. Review your threat model, security requirements, and performance needs before configuring any beacons.

Consider the following dataset uniqueness requirements as you determine whether searchable encryption is right for your dataset.

Distribution

The amount of security preserved by a beacon depends on the distribution of your dataset. When you configure an encrypted field for searchable encryption, the AWS Database Encryption SDK calculates an HMAC over the plaintext values written to that field. All of the beacons calculated for a given field are calculated using the same key, with the exception of multitenant databases that use a distinct key for each tenant. This means that if the same plaintext value is written to the field multiple times, the same HMAC tag is created for every instance of that plaintext value.

You should avoid constructing beacons from fields that contain very common values. For example, consider a database that stores the address of every resident of the state of Illinois. If you construct a beacon from the encrypted `City` field, the beacon calculated over "Chicago" will be overrepresented due to the large percentage of the Illinois population that lives in Chicago. Even if an unauthorized user can only read the encrypted values and beacon values, they might be able to identify which records contain data for residents of Chicago if the beacon preserves this distribution. To minimize the amount of distinguishing information revealed about your distribution, you must sufficiently truncate your beacon. The beacon length required to hide this uneven distribution has significant performance costs that might not meet the needs of your application.

You must carefully analyze the distribution of your dataset to determine how much your beacons need to be truncated. The beacon length remaining after truncation directly correlates to the amount of statistical information that can be identified about your distribution.
You might need to choose shorter beacon lengths to sufficiently minimize the amount of distinguishing information revealed about your dataset.

In extreme cases, you cannot calculate a beacon length for an unevenly distributed dataset that effectively balances performance and security. For example, you should not construct a beacon from a field that stores the result of a medical test for a rare disease. Since `NEGATIVE` results are expected to be significantly more prevalent within the dataset, `POSITIVE` results can be easily identified by how rare they are. It is very challenging to hide the distribution when the field only has two possible values. If you use a beacon length that is short enough to hide the distribution, all plaintext values map to the same HMAC tag. If you use a longer beacon length, it is obvious which beacons map to plaintext `POSITIVE` values.

Correlation

> We strongly recommend that you avoid constructing distinct beacons from fields with correlated values. Beacons constructed from correlated fields require shorter beacon lengths to sufficiently minimize the amount of information revealed about the distribution of each dataset to an unauthorized user. You must carefully analyze your dataset, including its entropy and the joint distribution of correlated values, to determine how much your beacons need to be truncated. If the resulting beacon length does not meet your performance needs, then beacons might not be a good fit for your dataset.
>
> For example, you should not construct two separate beacons from `City` and `ZIPCode` fields because the ZIP code will likely be associated with just one city. Typically, the false positives generated by a beacon limit an unauthorized user's ability to identify distinguishing information about your dataset. But the correlation between the `City` and `ZIPCode` fields means that an unauthorized user can easily identify which results are false positives and distinguish the different ZIP codes.
>
> You should also avoid constructing beacons from fields that contain the same plaintext values. For example, you should not construct a beacon from `mobilePhone` and `preferredPhone` fields because they likely hold the same values. If you construct distinct beacons from both fields, the AWS Database Encryption SDK creates the beacons for each field under different keys. This results in two different HMAC tags for the same plaintext value. The two distinct beacons are unlikely to have the same false positives and an unauthorized user might be able to distinguish different phone numbers.

Even if your dataset contains correlated fields or has an uneven distribution, you might be able to construct beacons that preserve the confidentiality of your dataset by using shorter beacon lengths. However, beacon length does not guarantee that every unique value in your dataset will produce a number of false positives that effectively minimizes the amount of distinguishing information revealed about your dataset. Beacon length only estimates the average number of false positives produced. The more unevenly distributed your dataset, the less effective beacon length is at determining the average number of false positives produced.

Carefully consider the distribution of the fields you construct beacons from and consider how much you will need to truncate the beacon length to meet your security requirements. The following topics in this chapter assume that your beacons are uniformly distributed and do not contain correlated data.

Searchable encryption scenario

The following example demonstrates a simple searchable encryption solution. In application, the example fields used in this example might not meet the distribution and correlation uniqueness recommendations for beacons. You can use this example for reference as you read about the searchable encryption concepts in this chapter.

Consider a database named `Employees` that tracks employee data for a company. Each record in the database contains fields called *EmployeeID*, *LastName*, *FirstName*, and *Address*. Each field in the `Employees` database is identified by the primary key `EmployeeID`.

The following is an example plaintext record in the database.

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

If you marked the `LastName` and `FirstName` fields as `ENCRYPT_AND_SIGN` in your [cryptographic actions](), the values in these fields are encrypted locally before they're uploaded to the database. The encrypted data that is uploaded is fully randomized, the database doesn't recognize this data as being protected. It just detects typical data entries. This means that the record that is actually stored in the database might look like the following.

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

If you need to query the database for exact matches in the `LastName` field, configure a standard beacon named *LastName* to map the plaintext values written to the `LastName` field to the encrypted values stored in the database.

This beacon calculates HMACs from the plaintext values in the `LastName` field. Each HMAC output is truncated so that it is no longer an exact match for the plaintext value. For example, the complete hash and the truncated hash for `Jones` might look like the following.

Complete hash

`2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f`

Truncated hash

`b35099d408c833`

After the standard beacon is configured, you can perform equality searches on the `LastName` field. For example, if you want to search for `Jones`, use the *LastName* beacon to perform the following query.

```
LastName = Jones
```

The AWS Database Encryption SDK automatically filters out the false positives and returns the plaintext result of your query.

Beacons

> Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

A beacon is a truncated Hash-Based Message Authentication Code (HMAC) tag that creates a map between the plaintext value written to a field and the encrypted value that is actually stored in your database. The beacon does not alter the encrypted state of the field. The beacon calculates an HMAC over the field's plaintext value and stores it alongside the encrypted value. This HMAC output is a one-to-one (1:1) match for the plaintext value of that field. The HMAC output is truncated so that multiple, distinct plaintext values map to the same truncated HMAC tag. These false positives limit an unauthorized user's ability to identify distinguishing information about the plaintext value.

Beacons can only be constructed from fields that are marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in your cryptographic actions. The beacon itself is not signed or encrypted. You cannot construct a beacon with fields that are marked `DO_NOTHING`.

The type of beacon you configure determines the type of queries you are able to perform. There are two types of beacons that support searchable encryption. *Standard beacons* perform equality searches. *Compound beacons* combine literal plaintext strings and standard beacons to perform complex database operations. After you configure your beacons, you must configure a secondary index for each beacon before you can search on the encrypted fields. For more information, see Configuring secondary indexes with beacons.

Topics

Standard beacons

Standard beacons are the simplest way to implement searchable encryption in your database. They can only perform equality searches for a single encrypted or virtual field. To learn how to configure standard beacons, see Configuring standard beacons.

The field that a standard beacon is constructed from is called the *beacon source*. It identifies the location of the data that the beacon needs to map. The beacon source can be either an encrypted field or a *virtual field*. The beacon source in each standard beacon must be unique. You cannot configure two beacons with the same beacon source.

Standard beacons can be used to perform equality searches for an encrypted or virtual field. Or, they can be used to construct compound beacons to perform more complex database operations. To help you organize and manage standard beacons, the AWS Database Encryption SDK provides the following optional *beacon styles* that define the intended use of a standard beacon. For more information see, Defining beacon styles.

You can create a standard beacon that performs equality searches for a single encrypted field, or you can create a standard beacon that performs equality searches on the concatenation of multiple `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` fields by creating a virtual field.

Virtual fields

A virtual field is a conceptual field constructed from one or more source fields. Creating a virtual field does not write a new field to your record. The virtual field is not explicitly stored in your database. It is used in standard beacon configuration to give the beacon instructions on how to identify a specific segment of a field or concatenate multiple fields within a record to perform a specific query. A virtual field requires at least one encrypted field.

> **Note**
>
> The following example demonstrates the types of transformations and queries you can perform with a virtual field. In application, the example fields used in this example might not meet the distribution and correlation uniqueness recommendations for beacons.

For example, if you want to perform equality searches on the concatenation of `FirstName` and `LastName` fields, you might create one of the following virtual fields.

- A virtual `NameTag` field, constructed from the first letter of the `FirstName` field, followed by the `LastName` field, all in lowercase. This virtual field enables you to query `NameTag=mjones`.
- A virtual `LastFirst` field, which is constructed from the `LastName` field, followed by the `FirstName` field. This virtual field enables you to query `LastFirst=JonesMary`.

Or, if you want to perform equality searches on a specific segment of an encrypted field, create a virtual field that identifies the segment you want to query.

For example, if you want to query an encrypted `IPAddress` field using the first three segments of the IP address, create the following virtual field.

- A virtual `IPSegment` field, constructed from `Segments('.', 0, 3)`. This virtual field enables you to query `IPSegment=192.0.2`. The query returns all records with an `IPAddress` value that starts with "192.0.2".

Virtual fields must be unique. Two virtual fields cannot be constructed from the exact same source fields.

For help configuring virtual fields and the beacons that use them, see Creating a virtual field.

Compound beacons

Compound beacons create indexes that improve query performance and enable you to perform more complex database operations. You can use compound beacons to combine literal plaintext strings and standard beacons to perform complex queries on encrypted records, such as querying two different record types from a single index or querying a combination of fields with a sort key. For more compound beacon solution examples, see Choose a beacon type.

Compound beacons can be constructed from standard beacons or a combination of standard beacons and signed fields. They are constructed from a list of parts. All compound beacons should include a list of encrypted parts that identifies the `ENCRYPT_AND_SIGN` fields included in the beacon. Every `ENCRYPT_AND_SIGN` field must be identified by a standard beacon. More complex compound beacons might also include a list of signed parts that identifies the plaintext `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` fields included in the beacon, and a list of constructor parts that identify all of the possible ways the compound beacon can assemble the fields.

> **Note**
>
> The AWS Database Encryption SDK also supports *signed beacons* that can be configured entirely from plaintext `SIGN_ONLY` and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` fields. Signed beacons are a type of compound beacon that index and perform complex queries on signed, but not encrypted, fields. For more information, see Creating signed beacons.

For help configuring compound beacons, see Configuring compound beacons.

The way you configure your compound beacon determines the types of queries it can perform. For example, you can make some encrypted and signed parts optional to allow for more flexibility in your queries. For more information on the types of queries compound beacons can perform, see Querying beacons.

Planning beacons

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

Beacons are designed to be implemented in new, unpopulated databases. Any beacon configured in an existing database will only map new records written to the database. Beacons are calculated from the plaintext value of a field, once the field is encrypted there is no way for the beacon to map existing data. After you have written new records with a beacon, you cannot update the beacon's configuration. However, you can add new beacons for new fields that you add to your record.

To implement searchable encryption, you must use the AWS KMS Hierarchical keyring to generate, encrypt, and decrypt the data keys used to protect your records. For more information, see Using the Hierarchical keyring for searchable encryption.

Before you can configure beacons for searchable encryption, you need to review your encryption requirements, database access patterns, and threat model to determine the best solution for your database.

The type of beacon that you configure determines the type of queries that you can perform. The beacon length that you specify in standard beacon configuration determines the expected number of false positives produced for a given beacon. We strongly recommend identifying and planning the types of queries that you need to perform before you configure your beacons. Once you have used a beacon, the configuration cannot be updated.

We strongly recommend that you review and complete the following tasks before you configure any beacons.

- Determine if beacons are right for your dataset
- Choose a beacon type
- Choose a beacon length
- Choose a beacon name

Remember the following beacon uniqueness requirements as you plan the searchable encryption solution for your database.

- **Every standard beacon must have a unique beacon source**

  Multiple standard beacons cannot be constructed from the same encrypted or virtual field.

  However, a single standard beacon can be used to construct multiple compound beacons.

- **Avoid creating a virtual field with source fields that overlap with existing standard beacons**

  Constructing a standard beacon from a virtual field that contains a source field that was used to create another standard beacon can reduce the security of both beacons.

  For more information, see Security considerations for virtual fields.

Considerations for multitenant databases

To query beacons configured in a multitenant database, you must include the field that stores the `branch-key-id` associated with the tenant that encrypted the record in your query. You define this field when you define the beacon key source. For the query to succeed, the value in this field must identify the appropriate beacon key materials required to recalculate the beacon.

Before you configure your beacons, you must decide how you plan to include the `branch-key-id` in your queries. For more information on the different ways you can include the `branch-key-id` in your queries, see Querying beacons in a multitenant database.

Choosing a beacon type

> Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

With searchable encryption, you can search encrypted records by mapping the plaintext values in an encrypted field with a *beacon*. The type of beacon you configure determines the type of queries that you can perform.

We strongly recommend identifying and planning the types of queries that you need to perform before you configure your beacons. After you configure your beacons, you must configure a secondary index for each beacon before you can search on the encrypted fields. For more information, see Configuring secondary indexes with beacons.

Beacons create a map between the plaintext value written to a field and the encrypted value that is actually stored in your database. You cannot compare the values of two standard beacons, even if they contain the same underlying plaintext. The two standard beacons will produce two different HMAC tags for the same plaintext values. As a result, standard beacons cannot perform the following queries.

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

You can only perform the above queries if you compare the signed parts of compound beacons, with the exception of the `CONTAINS` operator, which you can use with compound beacons to identify the entire value of an encrypted or signed field that the assembled beacon contains. When you compare signed parts, you can optionally include the prefix of an encrypted part, but you cannot include the encrypted value of a field. For more information about the types of queries that standard and compound beacons can perform, see Querying beacons.

Consider the following searchable encryption solutions as you review your database access patterns. The following examples define which beacon to configure to satisfy different encryption and querying requirements.

Standard beacons

Standard beacons can only perform equality searches. You can use standard beacons to perform the following queries.

Query a single encrypted field

If you want to identify records that contain a specific value for an encrypted field, create a standard beacon.

Examples

For the following example, consider a database named `UnitInspection` that tracks inspection data for a production facility. Each record in the database contains fields called `work_id`, `inspection_date`, `inspector_id_last4`, and `unit`. The full inspector ID is a number between 0—99,999,999. However, to ensure that the dataset is uniformly distributed, the `inspector_id_last4` only stores the last four digits of the inspector's ID. Each field in the database is identified by the primary key `work_id`. The `inspector_id_last4` and `unit` fields are marked `ENCRYPT_AND_SIGN` in the cryptographic actions.

The following is an example of a plaintext entry in the `UnitInspection` database.

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

Query a single encrypted field in a record

> If the `inspector_id_last4` field needs to be encrypted, but you still need query it for exact matches, construct a standard beacon from the `inspector_id_last4` field. Then, use the standard beacon to create a secondary index. You can use this secondary index to query on the encrypted `inspector_id_last4` field.

For help configuring standard beacons, see Configuring standard beacons.

Query a virtual field

A virtual field is a conceptual field constructed from one or more source fields. If you want to perform equality searches for a specific segment of an encrypted field, or perform equality searches on the concatenation of multiple fields, construct a standard beacon from a virtual field. All virtual fields must include at least one encrypted source field.

Examples

The following examples create virtual fields for the `Employees` database. The following is an example plaintext record in the `Employees` database.

```
{
    "EmployeeID": 101,
    "SSN": 000-00-0000,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
```

```
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

Query a segment of an encrypted field

For this example, the `SSN` field is encrypted.

If you want to query the `SSN` field using the last four digits of a social security number, create a virtual field that identifies the segment you plan to query.

A virtual `Last4SSN` field, constructed from `Suffix(4)` enables you to query `Last4SSN=0000`. Use this virtual field to construct a standard beacon. Then, use the standard beacon to create a secondary index. You can use this secondary index to query on the virtual field. This query returns all records with an `SSN` value that ends with the last four digits you specified.

Query the concatenation of multiple fields

> **Note**
>
> The following example demonstrates the types of transformations and queries you can perform with a virtual field. In application, the example fields used in this example might not meet the distribution and correlation uniqueness recommendations for beacons.

If you want to perform equality searches on a concatenation of `FirstName` and `LastName` fields, you might create a virtual `NameTag` field, constructed from the first letter of the `FirstName` field, followed by the `LastName` field, all in lowercase. Use this virtual field to construct a standard beacon. Then, use the standard beacon to create a secondary index. You can use this secondary index to query `NameTag=mjones` on the virtual field.

At least one of the source fields must be encrypted. Either `FirstName` or `LastName` could be encrypted, or they could both be encrypted. Any plaintext source fields must be marked as `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in your cryptographic actions.

For help configuring virtual fields and the beacons that use them, see Creating a virtual field.

Compound beacons

Compound beacons create an index from literal plaintext strings and standard beacons to perform complex database operations. You can use compound beacons to perform the following queries.

Query a combination of encrypted fields on a single index

If you need to query a combination of encrypted fields on a single index, create a compound beacon that combines the individual standard beacons constructed for each encrypted field to form a single index.

After you configure the compound beacon, you can create a secondary index that specifies the compound beacon as the partition key to perform exact match queries or with a sort key to perform more complex queries. Secondary indexes that specify the compound beacon as the sort key can perform exact match queries and more customized complex queries.

Examples

For the following examples, consider a database named `UnitInspection` that tracks inspection data for a production facility. Each record in the database contains fields called `work_id`, `inspection_date`, `inspector_id_last4`, and `unit`. The full inspector ID is a number between 0—99,999,999. However, to ensure that the dataset is uniformly distributed, the `inspector_id_last4` only stores the last four digits of the inspector's ID. Each field in the database is identified by the primary key `work_id`. The `inspector_id_last4` and `unit` fields are marked `ENCRYPT_AND_SIGN` in the cryptographic actions.

The following is an example of a plaintext entry in the `UnitInspection` database.

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

Perform equality searches on a combination of encrypted fields

If you want to query the `UnitInspection` database for exact matches on `inspector_id_last4.unit`, first create distinct standard beacons for the `inspector_id_last4` and `unit` fields. Then, create a compound beacon from the two standard beacons.

After you configure the compound beacon, create a secondary index that specifies the compound beacon as the partition key. Use this secondary index to query for exact matches on `inspector_id_last4.unit`. For example, you could query this beacon to find a list of inspections that an inspector performed for a given unit.

Perform complex queries on a combination of encrypted fields

If you want to query the `UnitInspection` database on `inspector_id_last4` and `inspector_id_last4.unit`, first create distinct standard beacons for the `inspector_id_last4` and `unit` fields. Then, create a compound beacon from the two standard beacons.

After you configure the compound beacon, create a secondary index that specifies the compound beacon as the sort key. Use this secondary index to query the `UnitInspection` database for entries that start with a certain inspector or query the database for a list of all of the units within a specific unit ID range that were inspected by a certain inspector. You can also perform exact match searches on `inspector_id_last4.unit`.

For help configuring compound beacons, see Configuring compound beacons.

Query a combination of encrypted and plaintext fields on a single index

If you need to query a combination of encrypted and plaintext fields on a single index, create a compound beacon that combines individual standard beacons and plaintext fields to form a single index. The plaintext fields used to construct the compound beacon must be marked `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in your cryptographic actions.

After you configure the compound beacon, you can create a secondary index that specifies the compound beacon as the partition key to perform exact match queries or with a sort key to perform more complex queries. Secondary indexes that specify the compound beacon as the sort key can perform exact match queries and more customized complex queries.

Examples

For the following examples, consider a database named `UnitInspection` that tracks inspection data for a production facility. Each record in the database contains fields called `work_id`, `inspection_date`, `inspector_id_last4`, and `unit`. The full inspector ID is a number between 0—99,999,999. However, to ensure that the dataset is uniformly distributed, the `inspector_id_last4` only stores the last four digits of the inspector's ID. Each field in the database is identified by the primary key `work_id`. The `inspector_id_last4` and `unit` fields are marked `ENCRYPT_AND_SIGN` in the cryptographic actions.

The following is an example of a plaintext entry in the `UnitInspection` database.

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

Perform equality searches on a combination of fields

If you want to query the `UnitInspection` database for inspections conducted by a specific inspector on a specific date, first create a standard beacon for the `inspector_id_last4` field. The `inspector_id_last4` field is marked `ENCRYPT_AND_SIGN` in the cryptographic actions. All encrypted parts require their own standard beacon. The `inspection_date` field is marked `SIGN_ONLY` and does not require a standard beacon. Next, create a compound beacon from the `inspection_date` field and the `inspector_id_last4` standard beacon.

After you configure the compound beacon, create a secondary index that specifies the compound beacon as the partition key. Use this secondary index to query the databases for records with exact matches to a certain inspector and inspection date. For example, you can query the database for a list of all inspections that the inspector whose ID ends with `8744` conducted on a specific date.

Perform complex queries on a combination of fields

If you want to query the database for inspections conducted within an `inspection_date` range, or query the database for inspections conducted on a particular `inspection_date` constrained by `inspector_id_last4` or `inspector_id_last4.unit`, first create distinct standard beacons for the `inspector_id_last4` and `unit` fields. Then, create a compound beacon from the plaintext `inspection_date` field and the two standard beacons.

After you configure the compound beacon, create a secondary index that specifies the compound beacon as the sort key. Use this secondary index to perform queries for inspections conducted on specific dates by a specific inspector. For example, you can query the database for a list of all units inspected on the same date. Or, you can query the database for a list of all inspections performed on a specific unit between a given range of inspection dates.

For help configuring compound beacons, see Configuring compound beacons.

Choosing a beacon length

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

When you write a new value to an encrypted field that's configured for searchable encryption, the AWS Database Encryption SDK calculates an HMAC over the plaintext value. This HMAC output is a one-to-one (1:1) match for the plaintext value of that field. The HMAC output is truncated so that multiple, distinct plaintext values map to the same truncated HMAC tag. These collisions, or *false positives*, limit an unauthorized user's ability to identify distinguishing information about the plaintext value.

The average number of false positives generated for each beacon is determined by the beacon length remaining after truncation. You only need to define beacon length when configuring standard beacons. Compound beacons use the beacon lengths of the standard beacons they're constructed from.

The beacon does not alter the encrypted state of the field. However, when you use beacons, there is an inherent tradeoff between how efficient your queries are and how much information is revealed about the distribution of your data.

The goal of searchable encryption is to reduce the performance costs associated with client-side encrypted databases by using beacons to perform queries on encrypted data. Beacons are stored alongside the encrypted fields they are calculated from. This means that they can reveal distinguishing information about the distribution of your dataset. In extreme cases, an unauthorized user might be able to analyze the information revealed about your distribution and use it to identify a field's plaintext value. Choosing the right beacon length can help mitigate these risks and preserve the confidentiality of your distribution.

Review your threat model to determine the level of security that you need. For example, the more individuals who have access to your database, but should not have access to the plaintext data, the more you might want to protect the confidentiality of your dataset distribution. To increase confidentiality, a beacon needs to generate more false positives. Increased confidentiality results in reduced query performance.

Security vs. Performance

- A beacon length that is **too long** produces too few false positives and might reveal distinguishing information about the distribution of your dataset.
- A beacon length that is **too short** produces too many false positives and increases the performance cost of queries because it requires a broader scan of the database.

When determining the appropriate beacon length for your solution, you must find a length that adequately preserves the security of your data without impacting the performance of your queries more than absolutely necessary. The amount of security preserved by a beacon depends on the distribution of your dataset and the correlation of the fields that your beacons are constructed from. The following topics assume that your beacons are uniformly distributed and do not contain correlated data.

Topics

- Calculating beacon length
- Example

Calculating beacon length

Beacon length is defined in *bits* and refers to the number of bits of the HMAC tag that are kept after truncation. The recommended beacon length varies depending on the dataset distribution, presence of correlated values, and your specific security and performance requirements. If your dataset is uniformly distributed, you can use the following equations and procedures to help identify the best beacon length for your implementation. These equations only estimate the average number of false positives that the beacon will produce, they do not guarantee that every unique value in your dataset will produce a specific number of false positives.

> **Note**
>
> The effectiveness of these equations is dependent on the distribution of your dataset. If your dataset is not uniformly distributed, see Are beacons right for my dataset?.
> In general, the further your dataset is from a uniform distribution, the more you need to shorten your beacon length.

1.

Estimate the population

The population is the expected number of unique values in the field that your standard beacon is constructed from, it is not the total expected number of values stored in the field. For example, consider an encrypted `Room` field that identifies the location of employee meetings. The `Room` field is expected to store 100,000 total values, but there are only 50 different rooms that employees can reserve for meetings. This means that the population is 50 because there only 50 possible unique values that can be stored in the `Room` field.

> **Note**
>
> If your standard beacon is constructed from a virtual field, the population used to calculate beacon length is the number of unique combinations created by the virtual field.

When estimating your population, be sure to consider the projected growth of the dataset. After you have written new records with the beacon, you cannot update the beacon length. Review your threat model and any existing database solutions to create an estimate for the number of unique values you expect this field to store in the next five years.

Your population does not need to be precise. First, identify the number of unique values in your current database, or estimate the number of unique values that you expect to store in the first year. Next, use the following questions to help you determine the projected growth of unique values over the next five years.

- Do you expect the unique values to multiply by 10?
- Do you expect the unique values to multiply by 100?
- Do you expect the unique values to multiply by 1000?

The difference between 50,000 and 60,000 unique values is not significant and they will both result in the same recommended beacon length. However, the difference between 50,000 and 500,000 unique values will significantly impact the recommended beacon length.

Consider reviewing public data on the frequency of common data types, such as ZIP codes or last names. For example, there are 41,707 ZIP codes in the United States. The population you use should be proportional to your own database. If the `ZIPCode` field in your database includes data from across the entire United States, then you might define your population as 41,707, even if the `ZIPCode` field does not *currently* have 41,707 unique values. If the `ZIPCode` field in your database only includes data from a single state, and will only ever include data from a single state, then you might define your population as the total number of ZIP codes in that state instead of 41,704.

2. **Calculate the recommended range for the expected number of collisions**

   To determine the appropriate beacon length for a given field, you must first identify an appropriate range for the expected number of collisions. The expected number of collisions represents the average expected number of unique plaintext values that map to a particular HMAC tag. The expected number of false positives for one unique plaintext value is one less than the expected number of collisions.

   We recommend that the expected number of collisions is greater than or equal to two, and less than the square root of your population. The following equations only work if your population has 16 or more unique values.

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

   If the number of collisions is less than two, the beacon will produce too few false positives. We recommend two as the minimum number of expected collisions because it means, on average, every unique value in the field will generate at least one false positive by mapping to one other unique value.

3. **Calculate the recommended range for beacon lengths**

   After identifying the minimum and maximum number of expected collisions, use the following equation to identify a range of appropriate beacon lengths.

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

   First, solve for **beacon length** where the number of expected collisions equals two (the minimum recommended number of expected collisions).

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

   Then, solve for **beacon length** where the expected number of collisions equals the square root of your population (the maximum recommended number of expected collisions).

```
√(Population) = Population * 2^-(beacon length)
```

We recommend rounding the output produced by this equation down to the shorter beacon length. For example, if the equation produces a beacon length of 15.6, we recommend rounding that value down to 15 bits instead of rounding up to 16 bits.

4. **Choose a beacon length**

   These equations only identify a recommended range of beacon lengths for your field. We recommend using a shorter beacon length to preserve the security of your dataset whenever possible. However, the beacon length that you actually use is determined by your threat model. Consider your performance requirements as you review your threat model to determine the best beacon length for your field.

   Using a shorter beacon length reduces query performance, while using a longer beacon length decreases security. In general, if your dataset is unevenly distributed, or if you construct distinct beacons from correlated fields, you need to use shorter beacon lengths to minimize the amount of information revealed about the distribution of your datasets.

   If you review your threat model and decide that any distinguishing information revealed about the distribution of a field does not present a threat to your overall security, you might choose to use a beacon length that is longer than the recommended range you calculated. For example, if you calculated the recommended range of beacon lengths for a field as 9—16 bits, you might choose to use a beacon length of 24 bits to avoid any performance loss.

   Choose your beacon length carefully. After you have written new records with the beacon, you cannot update the beacon length.

Example

Consider a database that marked the `unit` field as `ENCRYPT_AND_SIGN` in the cryptographic actions. To configure a standard beacon for the `unit` field, we need to determine the expected number of false positives and beacon length for the `unit` field.

1. Estimate the population

   After reviewing our threat model and current database solution, we expect the `unit` field to eventually have 100,000 unique values.

This means that **Population = 100,000**.

2. Calculate the recommended range for the expected number of collisions.

   For this example, the expected number of collisions should be between 2—316.

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

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

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

3. Calculate the recommended range for beacon length.

   For this example, the beacon length should be between 9—16 bits.

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

   a. Calculate the beacon length where the expected number of collisions equals the minimum identified in **Step 2**.

   $$2 = 100{,}000 * 2^{-(\text{beacon length})}$$

   Beacon length = 15.6, or 15 bits

   b. Calculate the beacon length where the expected number of collisions equals the maximum identified in **Step 2**.

   $$316 = 100{,}000 * 2^{-(\text{beacon length})}$$

   Beacon length = 8.3, or 8 bits

4. Determine the beacon length appropriate for your security and performance requirements.

   For every bit below 15, the performance cost and the security double.

   - 16 bits
     - On average, each unique value will map to 1.5 other units.

    - Security: two records with the same truncated HMAC tag are 66% likely to have the same plaintext value.
    - Performance: a query will retrieve 15 records for every 10 records that you actually requested.
- 14 bits
    - On average, each unique value will map to 6.1 other units.
    - Security: two records with the same truncated HMAC tag are 33% likely to have the same plaintext value.
    - Performance: a query will retrieve 30 records for every 10 records that you actually requested.

Choosing a beacon name

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

Every beacon is identified by a unique *beacon name*. Once a beacon is configured, the beacon name is the name you use when querying an encrypted field. A beacon name can be the same name as an encrypted field or virtual field, but it cannot be the same name as an unencrypted field. Two different beacons cannot have the same beacon name.

For examples demonstrating how to name and configure beacons, see Configuring beacons.

Naming standard beacon

When naming standard beacons, we strongly recommend that your beacon name resolves to the *beacon source* whenever possible. This means that the beacon name and the name of the encrypted or virtual field that your standard beacon is constructed from are the same. For example, if you are creating a standard beacon for an encrypted field named `LastName`, your beacon name should also be `LastName`.

When your beacon name is the same as the beacon source, you can omit the beacon source from your configuration and the AWS Database Encryption SDK will automatically use the beacon name as the beacon source.

Configuring beacons

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

There are two types of beacons that support searchable encryption. Standard beacons perform equality searches. They are the simplest way to implement searchable encryption in your database. Compound beacons combine literal plaintext strings and standard beacons to perform more complex queries.

Beacons are designed to be implemented in new, unpopulated databases. Any beacon configured in an existing database will only map new records written to the database. Beacons are calculated from the plaintext value of a field, once the field is encrypted there is no way for the beacon to map existing data. After you have written new records with a beacon, you cannot update the beacon's configuration. However, you can add new beacons for new fields that you add to your record.

After determining your access patterns, configuring beacons should be the second step in your database implementation. Then, after you configure all of your beacons, you need to create an AWS KMS Hierarchical keyring, define the beacon version, configure a secondary index for each beacon, define your cryptographic actions, and configure your database and AWS Database Encryption SDK client. For more information, see Using beacons.

To make it easier to define the beacon version, we recommend creating lists for standard and compound beacons. Add each beacon you create to the respective standard or compound beacon list as you configure them.

Topics

Configuring standard beacons

Standard beacons are the simplest way to implement searchable encryption in your database. They can only perform equality searches for a single encrypted or virtual field.

Example configuration syntax

Java

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

C# / .NET

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

Rust

```
let standard_beacon_list = vec![

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

To configure a standard beacon, provide the following values.

Beacon name

The name you use when querying an encrypted field.

A beacon name can be the same name as an encrypted field or virtual field, but it cannot be the same name as an unencrypted field. We strongly recommend using the name of the encrypted field or virtual field that your standard beacon is constructed from whenever possible. Two different beacons cannot have the same beacon name. For help determining the best beacon name for your implementation, see Choosing a beacon name.

Beacon length

The number of bits of the beacon hash value that are kept after truncation.

The beacon length determines the average number of false positives produced by a given beacon. For more information and help determining the appropriate beacon length for your implementation, see Determining beacon length.

Beacon source (Optional)

The field that a standard beacon is constructed from.

The beacon source must be a field name or an index referring to the value of a nested field. When your beacon name is the same as the beacon source, you can omit the the beacon source from your configuration and the AWS Database Encryption SDK will automatically use the beacon name as the beacon source.

Creating a virtual field

To create a virtual field, you must provide a name for the virtual field and a list of the source fields. The order that you add source fields to the virtual part list determines the order that they are concatenated to build the virtual field. The following example concatenates two source fields in their entirety to create a virtual field.

> **Note**
>
> We recommend verifying that your virtual fields produce the expected outcome before you populate your database. For more information, see Testing beacon outputs.

Java

See the complete code example: VirtualBeaconSearchableEncryptionExample.java

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

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

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

C# / .NET

See the complete code example: VirtualBeaconSearchableEncryptionExample.cs

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

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

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

Rust

See the complete code example: virtual_beacon_searchable_encryption.rs

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

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

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

To create a virtual field with a specific segment of a source field, you must define that transformation before adding the source field to your virtual part list.

Security considerations for virtual fields

Beacons do not alter the encrypted state of the field. However, when you use beacons, there is an inherent tradeoff between how efficient your queries are and how much information is revealed about the distribution of your data. The way that you configure your beacon determines the level of security that is preserved by that beacon.

Avoid creating a virtual field with source fields that overlap with existing standard beacons. Creating virtual fields that include a source field that has already been used to create a standard beacon can reduce the level of security for both beacons. The extent that security is reduced is dependent on the level of entropy added by the additional source fields. The level of entropy is determined by the distribution of unique values in the additional source field and the number of bits that the additional source field contributes to the overall size of the virtual field.

You can use population and beacon length to determine if the source fields for a virtual field preserve the security of your dataset. The population is the expected number of unique values in a field. Your population does not need to be precise. For help estimating the population of a field, see Estimate the population.

Consider the following example as you review the security of your virtual fields.

- Beacon1 is constructed from `FieldA`. `FieldA` has a population greater than $\mathbf{2^{(Beacon1\ length)}}$.
- Beacon2 is constructed from `VirtualField`, which is constructed from `FieldA`, `FieldB`, `FieldC`, and `FieldD`. Together, `FieldB`, `FieldC`, and `FieldD` have a population greater than $\mathbf{2^N}$

Beacon2 preserves the security of both Beacon1 and Beacon2 if the following statements are true:

```
N ≥ (Beacon1 length)/2
```

and

```
N ≥ (Beacon2 length)/2
```

Defining beacon styles

Standard beacons can be used to perform equality searches for an encrypted or virtual field. Or, they can be used to construct compound beacons to perform more complex database operations. To help you organize and manage standard beacons, the AWS Database Encryption SDK provides the following optional *beacon styles* that define the intended use of a standard beacon.

> **Note**
>
> To define beacon styles, you must use version 3.2 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before adding beacon styles to your beacon configurations.

PartOnly

A standard beacon defined as `PartOnly` can only be used to define an encrypted part of a compound beacon. You cannot directly query a `PartOnly` standard beacon.

Java

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

C# / .NET

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

Rust

```
StandardBeacon::builder()
```

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

Shared

By default, every standard beacon generates a unique HMAC key for beacon calculation. As a result, you cannot perform an equality search on the encrypted fields from two separate standard beacons. A standard beacon defined as `Shared` uses the HMAC key from another standard beacon for its calculations.

For example, if you need to compare `beacon1` fields to `beacon2` fields, define `beacon2` as a `Shared` beacon that uses the HMAC key from `beacon1` for its calculations.

> **Note**
>
> Consider your security and performance needs before configuring any `Shared` beacons. `Shared` beacons might increase the amount of statistical information that can be identified about the distribution of your dataset. For example, they might reveal which shared fields contain the same plaintext value.

Java

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

C# / .NET

```
new StandardBeacon
```

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

Rust

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

AsSet

By default, if a field value is a set, the AWS Database Encryption SDK calculates a single standard beacon for the set. As a result, you cannot perform the query `CONTAINS(a, :value)` where *a* is an encrypted field. A standard beacon defined as `AsSet` calculates individual standard beacon values for each individual element of the set and stores the beacon value in the item as a set. This enables the AWS Database Encryption SDK to perform the query `CONTAINS(a, :value)`.

To define an `AsSet` standard beacon, the elements in the set must be from the same population so that they can all use the same beacon length. The beacon set might have fewer elements than the plaintext set if there were collisions when calculating the beacon values.

Note

Consider your security and performance needs before configuring any `AsSet` beacons. `AsSet` beacons might increase the amount of statistical information that can be identified about the distribution of your dataset. For example, they might reveal the size of the plaintext set.

Java

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

C# / .NET

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

Rust

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

SharedSet

A standard beacon defined as `SharedSet` combines the `Shared` and `AsSet` functions so that you can perform equality searches on the encrypted values of a set and field. This enables the AWS Database Encryption SDK to perform the query `CONTAINS(a, b)` where *a* is an encrypted set and *b* is an encrypted field.

Note

Consider your security and performance needs before configuring any `Shared` beacons. `SharedSet` beacons might increase the amount of statistical information that can be identified about the distribution of your dataset. For example, they might reveal the size of the plaintext set or which shared fields contain the same plaintext value.

Java

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

C# / .NET

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

Rust

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

Configuring compound beacons

Compound beacons combine literal plaintext strings and standard beacons to perform complex database operations, such as querying two different record types from a single index or querying a combination of fields with a sort key. Compound beacons can be constructed from `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` fields. You must create a standard beacon for every encrypted field included in the compound beacon.

> **Note**
>
> We recommend verifying that your compound beacons produce the expected outcome before you populate your database. For more information, see Testing beacon outputs.

Example configuration syntax

Java

Compound beacon configuration

The following example defines encrypted and signed parts lists locally within the compound beacon configuration.

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

Beacon version definition

The following example defines encrypted and signed parts lists globally in the beacon version. For more information on defining the beacon version, see Using beacons.

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

C# / .NET

See the complete code sample: BeaconConfig.cs

Compound beacon configuration

The following example defines encrypted and signed parts lists locally within the compound beacon configuration.

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

Beacon version definition

The following example defines encrypted and signed parts lists globally in the beacon version. For more information on defining the beacon version, see Using beacons.

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

Rust

See the complete code sample: beacon_config.rs

Compound beacon configuration

The following example defines encrypted and signed parts lists locally within the compound beacon configuration.

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

Beacon version definition

The following example defines encrypted and signed parts lists globally in the beacon version. For more information on defining the beacon version, see Using beacons.

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

You can define your encrypted parts and signed parts in locally or globally defined lists. We recommend defining your encrypted and signed parts in a global list in the beacon version whenever possible. By defining encrypted and signed parts globally, you can define each part once and then reuse the parts in multiple compound beacon configurations. If you only intend to use an encrypted or signed part once, you can define it in a local list in the compound beacon configuration. You can reference both local and global parts in your constructor list.

If you define your encrypted and signed parts lists globally, you must provide a list of constructor parts that identify all of the possible ways the compound beacon can assemble the fields in your compound beacon configuration.

> **Note**
>
> To define encrypted and signed parts lists globally, you must use version 3.2 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before defining any new parts globally.
> You cannot update existing beacon configurations to define encrypted and signed parts lists globally.

To configure a compound beacon, provide the following values.

Beacon name

The name you use when querying an encrypted field.

A beacon name can be the same name as an encrypted field or virtual field, but it cannot be the same name as an unencrypted field. No two beacons can have the same beacon name. For help determining the best beacon name for your implementation, see Choosing a beacon name.

Split character

The character used to separate the parts that make up your compound beacon.

The split character cannot appear in the plaintext values of any of the fields that the compound beacon is constructed from.

Encrypted parts list

Identifies the `ENCRYPT_AND_SIGN` fields included in the compound beacon.

Each part must include a name and prefix. The part name must be the name of the standard beacon constructed from the encrypted field. The prefix can be any string, but it must be unique. An encrypted part cannot have the same prefix as a signed part. We recommend using a short value that distinguishes the part from other parts served by the compound beacon.

We recommend defining your encrypted parts globally whenever possible. You might consider defining an encrypted part locally if you only intend on using it in one compound beacon. A locally defined encrypted part cannot have the same prefix or name as a globally defined encrypted part.

Java

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

C# / .NET

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

Rust

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

Signed parts list

Identifies the signed fields included in the compound beacon.

Note

Signed parts are optional. You can configure a compound beacon that does not reference any signed parts.

Each part must include a name, source, and prefix. The source is the `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` field that the part identifies. The source must be a field name or an index referring to the value of a nested field. If your part name identifies the source, you can omit the source and the AWS Database Encryption SDK will automatically use the name as its source. We recommend specifying the source as the part name whenever possible. The prefix can be any string, but it must be unique. A signed part cannot have the same prefix as an encrypted part. We recommend using a short value that distinguishes the part from other parts served by the compound beacon.

We recommend defining your signed parts globally whenever possible. You might consider defining a signed part locally if you only intend on using it in one compound beacon. A locally defined signed part cannot have the same prefix or name as a globally defined signed part.

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
```

```
SignedPart signedPartExample = SignedPart.builder()
    .name("signedFieldName")
    .prefix("S-")
    .build();
signedPartList.add(signedPartExample);
```

C# / .NET

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

Rust

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

Constructor list

Identifies the *constructors* that define the different ways that the encrypted and signed parts can be assembled by the compound beacon. You can reference both local and global parts in your constructor list.

If you construct your compound beacon from globally defined encrypted and signed parts, you must provide a constructor list.

If you do not use any globally defined encrypted or signed parts to construct your compound beacon, the constructor list is optional. If you do not specify a constructor list, the AWS Database Encryption SDK assembles the compound beacon with the following default constructor.

- All signed parts in the order they were added to the signed parts list

- All encrypted parts in the order they were added to the encrypted parts list
- All parts are required

Constructors

Each constructor is an ordered list of *constructor parts* that defines one way that the compound beacon can be assembled. The constructor parts are joined together in the order they are added to the list, with each part separated by the specified split character.

Each constructor part names an encrypted part or a signed part, and defines whether that part is required or optional within the constructor. For example, if you want to query a compound beacon on `Field1`, `Field1.Field2`, and `Field1.Field2.Field3`, mark `Field2` and `Field3` as optional and create one constructor.

Each constructor must have at least one required part. We recommend making the first part in each constructor required so that you can use the `BEGINS_WITH` operator in your queries.

A constructor succeeds if all its required parts are present in the record. When you write a new record, the compound beacon uses the constructor list to determine if the beacon can be assembled from the values provided. It attempts to assemble the beacon in the order that the constructors were added to the constructor list, and it uses the first constructor that succeeds. If no constructors succeed, the beacon is not written to the record.

All readers and writers should specify the same order of constructors to ensure that their query results are correct.

Use the following procedures to specify your own constructor list.

1. Create a constructor part for each encrypted part and signed part to define whether or not that part is required.

   The constructor part name must be the name of the standard beacon or signed field it represents.

   Java

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

C# / .NET

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

Rust

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

2. Create a constructor for each possible way that the compound beacon can be assembled using the constructor parts you created in **Step 1**.

   For example, if you want to query on `Field1.Field2.Field3` and `Field4.Field2.Field3`, then you must create two constructors. `Field1` and `Field4` can both be required because they are defined in two separate constructors.

   Java

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

   C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
```

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

Rust

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

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

3. Create a constructor list that includes all of the constructors that you created in **Step 2**.

   Java

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

   C# / .NET

```
var constructorList = new List<Constructor>
```

```
{
    field123Constructor,
    field421Constructor
};
```

Rust

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

4. Specify the `constructorList` when you create your compound beacon.

Example configurations

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The following examples demonstrate how to configure standard and compound beacons. The following configurations do not provide beacon lengths. For help determining the appropriate beacon length for your configuration, see Choose a beacon length.

To see complete code examples that demonstrate how to configure and use beacons, see the Java, .NET, and Rust searchable encryption examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Topics

- Standard beacons
- Compound beacons

Standard beacons

If you want to query the `inspector_id_last4` field for exact matches, create a standard beacon using the following configuration.

Java

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

C# / .NET

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

Rust

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

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

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

Compound beacons

If you want to query the `UnitInspection` database on `inspector_id_last4` and `inspector_id_last4.unit`, create a compound beacon with the following configuration. This compound beacon only requires encrypted parts.

Java

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
List<StandardBeacon> standardBeaconList = new ArrayList<>);
```

```
StandardBeacon inspectorBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(inspectorBeacon);

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

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

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

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

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

C# / .NET

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

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

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

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

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

Rust

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

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

let standard_beacon_list = vec![last4_beacon, unit_beacon];

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

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

Using beacons

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

Beacons enable you to search encrypted records without decrypting the entire database being queried. Beacons are designed to be implemented in new, unpopulated databases. Any beacon configured in an existing database will only map new records written to the database. Beacons are calculated from the plaintext value of a field, once the field is encrypted there is no way for the beacon to map existing data. After you have written new records with a beacon, you cannot update the beacon's configuration. However, you can add add new beacons for new fields that you add to your record.

After you configure your beacons, you must complete the following steps before you begin populating your database and performing queries on your beacons.

1. **Create an AWS KMS Hierarchical keyring**

   To use searchable encryption, you must use the AWS KMS Hierarchical keyring to generate, encrypt, and decrypt the data keys used to protect your records.

   After you configure your beacons, assemble the Hierarchical keyring prerequisites and create your Hierarchical keyring.

   For more details on why the Hierarchical keyring is required, see Using the Hierarchical keyring for searchable encryption.

2. **Define the beacon version**

   Specify your `keyStore`, `keySource`, a list of all standard beacons you configured, a list of all compound beacons you configured, a list of encrypted parts, a list of signed parts, and a beacon version. You must specify `1` for the beacon version. For guidance on defining your `keySource`, see Defining your beacon key source.

   The following Java example defines the beacon version for a single tenant database. For help defining the beacon version for a multitenant database, see Searchable encryption for multitenant databases.

Java

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

C# / .NET

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

Rust

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

3. **Configure secondary indexes**

   After you configure your beacons, you must configure a secondary index that reflects each beacon before you can search on the encrypted fields. For more information, see Configuring secondary indexes with beacons.

4. **Define your cryptographic actions**

   All fields used to construct a standard beacon must be marked `ENCRYPT_AND_SIGN`. All other fields used to construct beacons must be marked `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

5. **Configure an AWS Database Encryption SDK client**

   To configure an AWS Database Encryption SDK client that protects the table items in your DynamoDB table, see Java client-side encryption library for DynamoDB.

Querying beacons

The type of beacon you configure determines the type of queries you are able to perform. Standard beacons use filter expressions to perform equality searches. Compound beacons combine literal plaintext strings and standard beacons to perform complex queries. When you query encrypted data, you search on the beacon name.

You cannot compare the values of two standard beacons, even if they contain the same underlying plaintext. The two standard beacons will produce two different HMAC tags for the same plaintext values. As a result, standard beacons cannot perform the following queries.

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

Compound beacons can perform the following queries.

- `BEGINS_WITH(a)`, where *a* reflects the entire value of the field that the assembled compound beacon begins with. You cannot use the `BEGINS_WITH` operator to identify a value that begins with a particular substring. However, you can use `BEGINS_WITH(S_)`, where *S_* reflects the prefix for a part that the assembled compound beacon begins with.
- `CONTAINS(a)`, where *a* reflects the entire value of a field that the assembled compound beacon contains. You cannot use the `CONTAINS` operator to identify a record that contains a particular substring or a value within a set.

  For example, you cannot perform a query `CONTAINS(path, "a"` where *a* reflects the value in a set.
- You can compare signed parts of compound beacons. When you compare signed parts, you can optionally append the prefix of an encrypted part to one or more signed parts, but you cannot include the value of an encrypted field in any query.

  For example, you can compare signed parts and query on `signedField1 = signedField2` or `value IN (signedField1, signedField2, ...)`.

  You can also compare signed parts and the prefix of an encrypted part by query on `signedField1.A_ = signedField2.B_`.

- `field BETWEEN a AND b`, where *a* and *b* are signed parts. You can optionally append the prefix of an encrypted part to one or more signed parts, but you cannot include the value of an encrypted field in any query.

You must include the prefix for each part you include in a query on a compound beacon. For example, if you constructed a compound beacon, `compoundBeacon`, from two fields, `encryptedField` and `signedField`, you must include the prefixes configured for those two parts when you query the beacon.

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

Searchable encryption for multitenant databases

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

To implement searchable encryption in your database, you must use an AWS KMS Hierarchical keyring. The AWS KMS Hierarchical keyring generates, encrypts, and decrypts the data keys used to protect your records. It also creates the beacon key used to generate beacons. When using the AWS KMS Hierarchical keyring with multitenant databases, there is a distinct branch key and beacon key for each tenant. To query encrypted data in a multitenant database, you must identify the beacon key materials used to generate the beacon you are querying. For more information, see the section called "Using the Hierarchical keyring for searchable encryption".

When you define the beacon version for a multitenant database, specify a list of all standard beacons you configured, a list of all compound beacons you configured, a beacon version, and a `keySource`. You must define your beacon key source as a `MultiKeyStore`, and include a `keyFieldName`, a cache time to live for the local beacon key cache, and maximum cache size for the local beacon key cache.

If you configured any signed beacons, they must be included in your `compoundBeaconList`. Signed beacons are a type of compound beacon that index and perform complex queries on `SIGN_ONLY` and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`fields.

Java

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

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Multi = new MultiKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000,
                MaxCacheSize = 10
            }
        }
    }
};
```

Rust

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

keyFieldName

The `keyFieldName` defines the name of the field that stores the `branch-key-id` associated with the beacon key used to generated beacons for a given tenant.

When you write new records to your database, the `branch-key-id` that identifies the beacon key used to generate any beacons for that record is stored in this field.

By default, the `keyField` is a conceptual field that is not explicitly stored in your database. The AWS Database Encryption SDK identifies the `branch-key-id` from the encrypted data key in the material description and stores the value in the conceptual `keyField` for you to reference in your compound beacons and signed beacons. Since the material description is signed, the conceptual `keyField` is considered a signed part.

You can also include the `keyField` in your cryptographic actions as a `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` field to explicitly store the field in your database. If you do this, you must manually include the `branch-key-id` in the `keyField` every time you write a record to your database.

Querying beacons in a multitenant database

To query a beacon, you must include the `keyField` in your query to identify the appropriate beacon key materials required to recalculate the beacon. You must specify the `branch-key-id` associated with the beacon key used to generate the beacons for a record. You cannot specify the friendly name that identifies a tenant's `branch-key-id` in the branch key ID supplier. You can include the `keyField` in your queries in a following ways.

Compound beacons

Whether you explicitly store the `keyField` in your records or not, you can include the `keyField` directly in your compound beacons as a signed part. The `keyField` signed part must be required.

For example, if you want to construct a compound beacon, `compoundBeacon`, from two fields, `encryptedField` and `signedField`, you must also include the `keyField` as a signed part. This enables you to perform the following query on `compoundBeacon`.

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

Signed beacons

The AWS Database Encryption SDK uses standard and compound beacons to provide searchable encryption solutions. These beacons must include at least one encrypted field. However, the AWS Database Encryption SDK also supports signed beacons that can be configured entirely from plaintext `SIGN_ONLY` and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` fields.

Signed beacons can be constructed from a single part. Whether you explicitly store the `keyField` in your records or not, you can construct a signed beacon from the `keyField` and use it to create compound queries that combine a query on the `keyField` signed beacon with a query on one of your other beacons. For example, you could perform the following query.

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

For help configuring signed beacons, see Creating signed beacons

Query directly on the keyField

If you specified the `keyField` in your cryptographic actions and explicitly store the field in your record, you can create a compound query that combines a query on your beacon with a query on the `keyField`. You might choose to query directly on the `keyField` if you want to query a standard beacon. For example, you could perform the following query.

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

AWS Database Encryption SDK for DynamoDB

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK for DynamoDB is a software library that enables you to include client-side encryption in your Amazon DynamoDB design. The AWS Database Encryption SDK for DynamoDB provides attribute-level encryption and enables you to specify which items to encrypt and which items to include in the signatures that ensure the authenticity of your data. Encrypting your sensitive data in transit and at rest helps ensure that your plaintext data isn’t available to any third party, including AWS.

Note

The AWS Database Encryption SDK does not support PartiQL.

In DynamoDB, a table is a collection of items. Each *item* is a collection of *attributes*. Each attribute has a name and a value. The AWS Database Encryption SDK for DynamoDB encrypts the values of attributes. Then, it calculates a signature over the attributes. You specify which attribute values to encrypt and which to include in the signature in the cryptographic actions.

The topics in this chapter provide an overview of the AWS Database Encryption SDK for DynamoDB, including which fields are encrypted, guidance on client installation and configuration, and Java examples to help you get started.

Topics

- Client-side and server-side encryption
- Which fields are encrypted and signed?
- Searchable encryption in DynamoDB
- Updating your data model
- AWS Database Encryption SDK for DynamoDB available programming languages
- Legacy DynamoDB Encryption Client

Client-side and server-side encryption

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK for DynamoDB supports *client-side encryption*, where you encrypt your table data before you send it to your database. However, DynamoDB provides a server-side *encryption at rest* feature that transparently encrypts your table when it is persisted to disk and decrypts it when you access the table.

The tools that you choose depend on the sensitivity of your data and the security requirements of your application. You can use both the AWS Database Encryption SDK for DynamoDB and encryption at rest. When you send encrypted and signed items to DynamoDB, DynamoDB doesn't recognize the items as being protected. It just detects typical table items with binary attribute values.

Server-side encryption at rest

DynamoDB supports encryption at rest, a *server-side encryption* feature in which DynamoDB transparently encrypts your tables for you when the table is persisted to disk, and decrypts them when you access the table data.

When you use an AWS SDK to interact with DynamoDB, by default, your data is encrypted in transit over an HTTPS connection, decrypted at the DynamoDB endpoint, and then re-encrypted before being stored in DynamoDB.

- **Encryption by default.** DynamoDB transparently encrypts and decrypts all tables when they are written. There is no option to enable or disable encryption at rest.
- **DynamoDB creates and manages the cryptographic keys.** The unique key for each table is protected by an AWS KMS key that never leaves AWS Key Management Service (AWS KMS) unencrypted. By default, DynamoDB uses an AWS owned key in the DynamoDB service account, but you can choose an AWS managed key or customer managed key in your account to protect some or all of your tables.
- **All table data is encrypted on disk.** When an encrypted table is saved to disk, DynamoDB encrypts all table data, including the primary key and local and global secondary indexes. If your table has a sort key, some of the sort keys that mark range boundaries are stored in plaintext in the table metadata.

- **Objects related to tables are encrypted, too.** Encryption at rest protects DynamoDB streams, global tables, and backups whenever they are written to durable media.
- **Your items are decrypted when you access them.** When you access the table, DynamoDB decrypts the part of the table that includes your target item, and returns the plaintext item to you.

AWS Database Encryption SDK for DynamoDB

Client-side encryption provides end-to-end protection for your data, in transit and at rest, from its source to storage in DynamoDB. Your plaintext data is never exposed to any third party, including AWS. You can use the AWS Database Encryption SDK for DynamoDB with new DynamoDB tables, or you can migrate your existing Amazon DynamoDB tables to the latest version of the AWS Database Encryption SDK for DynamoDB.

- **Your data is protected in transit and at rest.** It is never exposed to any third party, including AWS.
- **You can sign your table Items.** You can direct the AWS Database Encryption SDK for DynamoDB to calculate a signature over all or part of a table item, including the primary key attributes. This signature allows you to detect unauthorized changes to the item as a whole, including adding or deleting attributes, or swapping attribute values.
- **You determine how your data is protected** by selecting a keyring. Your keyring determines the wrapping keys that protect your data keys, and ultimately, your data. Use the most secure wrapping keys that are practical for your task.
- **The AWS Database Encryption SDK for DynamoDB doesn't encrypt the entire table.** You choose which attributes are encrypted in your items. The AWS Database Encryption SDK for DynamoDB does not encrypt an entire item. It does not encrypt attribute names, or the names or values of the primary key (partition key and sort key) attributes.

AWS Encryption SDK

If you are encrypting data that you store in DynamoDB, we recommend the AWS Database Encryption SDK for DynamoDB.

The AWS Encryption SDK is a client-side encryption library that helps you to encrypt and decrypt generic data. Although it can protect any type of data, it isn't designed to work with structured data, like database records. Unlike the AWS Database Encryption SDK for DynamoDB, the AWS

Encryption SDK cannot provide item-level integrity checking and it has no logic to recognize attributes or prevent encryption of primary keys.

If you use the AWS Encryption SDK to encrypt any element of your table, remember that it isn't compatible with the AWS Database Encryption SDK for DynamoDB. You cannot encrypt with one library and decrypt with the other.

Which fields are encrypted and signed?

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The AWS Database Encryption SDK for DynamoDB is a client-side encryption library designed especially for Amazon DynamoDB applications. Amazon DynamoDB stores data in tables, which are a collection of items. Each *item* is a collection of *attributes*. Each attribute has a name and a value. The AWS Database Encryption SDK for DynamoDB encrypts the values of attributes. Then, it calculates a signature over the attributes. You can specify which attribute values to encrypt and which to include in the signature.

Encryption protects the confidentiality of the attribute value. Signing provides integrity of all signed attributes and their relationship to each other, and provides authentication. It enables you to detect unauthorized changes to the item as a whole, including adding or deleting attributes, or substituting one encrypted value for another.

In an encrypted item, some data remains in plaintext, including the table name, all attribute names, the attribute values that you don't encrypt, the names and values of the primary key (partition key and sort key) attributes, and the attribute types. Do not store sensitive data in these fields.

For more information on how the AWS Database Encryption SDK for DynamoDB works, see How the AWS Database Encryption SDK works.

Note

All mentions of *attribute actions* in the AWS Database Encryption SDK for DynamoDB topics refer to cryptographic actions.

Topics

- Encrypting attribute values
- Signing the item

Encrypting attribute values

The AWS Database Encryption SDK for DynamoDB encrypts the values (but not the attribute name or type) of the attributes that you specify. To determine which attribute values are encrypted, use attribute actions.

For example, this item includes `example` and `test` attributes.

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

If you encrypt the `example` attribute, but don't encrypt the `test` attribute, the results look like the following. The encrypted `example` attribute value is binary data, instead of a string.

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

The primary key attributes—partition key and sort key—of each item must remain in plaintext because DynamoDB uses them to find the item in the table. They should be signed, but not encrypted.

The AWS Database Encryption SDK for DynamoDB identifies the primary key attributes for you and ensures that their values are signed, but not encrypted. And, if you identify your primary key and then try to encrypt it, the client will throw an exception.

The client stores the material description in a new attribute (`aws_dbe_head`) that it adds to the item. The material description describes how the item was encrypted and signed. The client uses this information to verify and decrypt the item. The field that stores the material description is not encrypted.

Signing the item

After encrypting the specified attribute values, the AWS Database Encryption SDK for DynamoDB calculates Hash-Based Message Authentication Codes (HMACs) and a digital signature over the canonicalization of the material description, encryption context, and each field marked `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in the attribute actions. ECDSA signatures are enabled by default, but are not required. The client stores the HMACs and signatures in a new attribute (`aws_dbe_foot`) that it adds to the item.

Searchable encryption in DynamoDB

To configure your Amazon DynamoDB tables for searchable encryption, you must use the AWS KMS Hierarchical keyring to generate, encrypt, and decrypt the data keys used to protect your items. You must also include the `SearchConfig` in your table encryption configuration.

> **Note**
>
> If you're using the Java client-side encryption library for DynamoDB, you must use the low-level AWS Database Encryption SDK for DynamoDB API to encrypt, sign, verify, and decrypt your table items. The DynamoDB Enhanced Client and lower-level `DynamoDBItemEncryptor` do not support searchable encryption.

Topics

Configuring secondary indexes with beacons

After you configure your beacons, you must configure a secondary index that reflects each beacon before you can search on the encrypted attributes.

When you configure a standard or compound beacon, the AWS Database Encryption SDK adds the `aws_dbe_b_` prefix to the beacon name so that the server can easily identify beacons. For example, if you name a compound beacon, `compoundBeacon`, the full beacon name is actually `aws_dbe_b_compoundBeacon`. If you want to configure secondary indexes that include a standard or compound beacon, you must include the `aws_dbe_b_` prefix when you identify the beacon name.

Partition and sort keys

You cannot encrypt primary key values. Your partition and sort keys must be signed. Your primary key values cannot be a standard or compound beacon.

Your primary key values must be `SIGN_ONLY`, unless you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Your primary key values can be signed beacons. If you configured distinct signed beacons for each of your primary key values, you must specify the attribute name that identifies the primary key value as the signed beacon name. However, the AWS Database Encryption SDK does not add the `aws_dbe_b_` prefix to signed beacons. Even if you configured distinct signed beacons for your primary key values, you only need to specify the attribute names for the primary key values when you configure a secondary index.

Local secondary indexes

The sort key for a local secondary index can be a beacon.

If you specify a beacon for the sort key, the type must be String. If you specify a standard or compound beacon for the sort key, you must include the `aws_dbe_b_` prefix when you specify the beacon name. If you specify a signed beacon, specify the beacon name without any prefix.

Global secondary indexes

The partition and sort keys for a global secondary index can both be beacons.

If you specify a beacon for the partition or sort key, the type must be String. If you specify a standard or compound beacon for the sort key, you must include the `aws_dbe_b_` prefix when you specify the beacon name. If you specify a signed beacon, specify the beacon name without any prefix.

Attribute projections

A projection is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. DynamoDB provides three different options for attribute projections: `KEYS_ONLY`, `INCLUDE`, and `ALL`.

If you use the INCLUDE attribute projection to search on a beacon, you must specify the names for all of the attributes that the beacon is constructed from and the beacon name with the `aws_dbe_b_` prefix. For example, if you configured a compound beacon, `compoundBeacon`, from `field1`, `field2`, and `field3`, you must specify `aws_dbe_b_compoundBeacon`, `field1`, `field2`, and `field3` in the projection.

A global secondary index can only use the attributes explicitly specified in the projection, but a local secondary index can use any attribute.

Testing beacon outputs

If you configured compound beacons or constructed your beacons using virtual fields, we recommend verifying that these beacons produce the expected output before populating your DynamoDB table.

The AWS Database Encryption SDK provides the `DynamoDbEncryptionTransforms` service to help you troubleshoot virtual field and compound beacon outputs.

Testing virtual fields

The following snippet creates test items, defines the `DynamoDbEncryptionTransforms` service with the DynamoDB table encryption configuration, and demonstrates how to use `ResolveAttributes` to verify that the virtual field produces the expected output.

Java

See the complete code sample: VirtualBeaconSearchableEncryptionExample.java

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

final PutItemResponse itemWithHasTestResultPutResponse =
 ddb.putItem(itemWithHasTestResultPutRequest);

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

final PutItemResponse itemWithNoHasTestResultPutResponse =
 ddb.putItem(itemWithNoHasTestResultPutRequest);

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

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

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

C# / .NET

See the complete code sample: VirtualBeaconSearchableEncryptionExample.cs.

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

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

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

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

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

Rust

See the complete code sample: virtual_beacon_searchable_encryption.rs.

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

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

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

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

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

Testing compound beacons

The following snippet creates a test item, defines the `DynamoDbEncryptionTransforms` service with the DynamoDB table encryption configuration, and demonstrates how to use `ResolveAttributes` to verify that the compound beacon produces the expected output.

Java

See the complete code sample: CompoundBeaconSearchableEncryptionExample.java

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

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

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

final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that CompoundBeacons has the expected value
Map<String, String> cbs = new HashMap<>();
cbs.put("last4UnitCompound", "L-5678.U-011899988199");
assert resolveOutput.CompoundBeacons().equals(cbs);
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

C# / .NET

See the complete code sample: CompoundBeaconSearchableEncryptionExample.cs

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

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

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

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

Rust

See the complete code sample: compound_beacon_searchable_encryption.rs

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

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

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

// Verify that CompoundBeacons has the expected value
Dlet compound_beacons = resolve_output.compound_beacons.unwrap();
assert_eq!(compound_beacons.len(), 1);
assert_eq!(
    compound_beacons["last4UnitCompound"],
    "L-5678.U-011899988199"
);
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

Updating your data model

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

When you configure the AWS Database Encryption SDK for DynamoDB, you provide attribute actions. On encrypt, AWS Database Encryption SDK uses the attribute actions to identify which attributes to encrypt and sign, which attributes to sign (but not encrypt), and which to ignore. You also define allowed unsigned attributes to explicitly tell the client which attributes are excluded from the signatures. On decrypt, the AWS Database Encryption SDK uses the allowed unsigned attributes that you defined to identify which attributes are not included in the signatures. Attribute actions are not saved in the encrypted item and the AWS Database Encryption SDK does not update your attribute actions automatically.

Choose your attribute actions carefully. When in doubt, use **Encrypt and sign**. After you have used the AWS Database Encryption SDK to protect your items, you cannot change an existing `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `DO_NOTHING`. However, you can safely make the following changes.

- Add new `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes
- Remove existing attributes
- Change an existing `ENCRYPT_AND_SIGN` attribute to `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

- Change an existing `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `ENCRYPT_AND_SIGN`
- Add a new `DO_NOTHING` attribute
- Change an existing `SIGN_ONLY` attribute to `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- Change an existing `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `SIGN_ONLY`

Considerations for searchable encryption

Before you update your data model, carefully consider how your updates might impact any beacons you constructed from the attributes. After you have written new records with a beacon, you cannot update the beacon's configuration. You cannot update the attribute actions associated with the attributes you used to construct beacons. If you remove an existing attribute and its associated beacon, you will not be able to query existing records using that beacon. You can create new beacons for new fields that you add to your record, but you cannot update existing beacons to include the new field.

Considerations for `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes

By default, the partition and sort keys are the only attribute included in the encryption context. You might consider defining additional fields as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` so that the branch key ID supplier for your AWS KMS Hierarchical keyring can identify which branch key is required for decryption from the encryption context. For more information, see branch key ID supplier. If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> To use the `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action, you must use version 3.3 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before updating your data model to include `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Add new `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, and `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes

To add a new `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute, define the new attribute in your attribute actions.

You cannot remove an existing `DO_NOTHING` attribute and add it back as an `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute.

Using an annotated data class

If you defined your attribute actions with a `TableSchema`, add the new attribute to your annotated data class. If you do not specify an attribute action annotation for the new attribute, the client will encrypt and sign the new attribute by default (unless the attribute is part of the primary key). If you only want to sign the new attribute, you must add the new attribute with the `@DynamoDBEncryptionSignOnly` or `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` annotation.

Using an object model

If you manually defined your attribute actions, add the new attribute to the attribute actions in your object model and specify `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` as the attribute action.

Remove existing attributes

If you decide that you no longer need an attribute, you can stop writing data to that attribute or you can formally remove it from your attribute actions. When you stop writing new data to an attribute, the attribute still shows up in your attribute actions. This can be helpful if you need to start using the attribute again in the future. Formally removing the attribute from your attribute actions does not remove it from your dataset. Your dataset will still contain items that include that attribute.

To formally remove an existing `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, or `DO_NOTHING` attribute, update your attribute actions.

If you remove a `DO_NOTHING` attribute, you must not remove that attribute from your allowed unsigned attributes. Even if you are no longer writing new values to that attribute, the client still needs to know that the attribute is unsigned to read existing items that contain the attribute.

Using an annotated data class

If you defined your attribute actions with a `TableSchema`, remove the attribute from your annotated data class.

Using an object model

If you manually defined your attribute actions, remove the attribute from the attribute actions in your object model.

Change an existing ENCRYPT_AND_SIGN attribute to SIGN_ONLY or SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

To change an existing `ENCRYPT_AND_SIGN` attribute to `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, you must update your attribute actions. After you deploy the update, the client will be able to verify and decrypt existing values written to the attribute, but will only sign new values written to the attribute.

> **Note**
>
> Carefully consider your security requirements before changing an existing `ENCRYPT_AND_SIGN` attribute to `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Any attribute that can store sensitive data should be encrypted.

Using an annotated data class

If you defined your attribute actions with a `TableSchema`, update the existing attribute to include the `@DynamoDBEncryptionSignOnly` or `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` annotation in your annotated data class.

Using an object model

If you manually defined your attribute actions, update the attribute action associated with the existing attribute from `ENCRYPT_AND_SIGN` to `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in your object model.

Change an existing `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `ENCRYPT_AND_SIGN`

To change an existing `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `ENCRYPT_AND_SIGN`, you must update your attribute actions. After you deploy the update, the client will be able to verify the existing values written to the attribute, and will encrypt and sign new values written to the attribute.

Using an annotated data class

If you defined your attribute actions with a `TableSchema`, remove the `@DynamoDBEncryptionSignOnly` or `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` annotation from the existing attribute.

Using an object model

If you manually defined your attribute actions, update the attribute action associated with the attribute from `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` to `ENCRYPT_AND_SIGN` in your object model.

Add a new `DO_NOTHING` attribute

To reduce the risk of error when adding a new `DO_NOTHING` attribute, we recommend specifying a distinct prefix when naming your `DO_NOTHING` attributes, and then using that prefix to define your allowed unsigned attributes.

You cannot remove an existing `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute from your annotated data class and then add the attribute back as a `DO_NOTHING` attribute. You can only add entirely new `DO_NOTHING` attributes.

The steps you take to add a new `DO_NOTHING` attribute depend on whether your defined your allowed unsigned attributes explicitly in a list or with a prefix.

Using an allowed unsigned attributes prefix

If you defined your attribute actions with a `TableSchema`, add the new `DO_NOTHING` attribute to your annotated data class with the `@DynamoDBEncryptionDoNothing` annotation. If you manually defined your attribute actions, update your attribute actions to include the new attribute. Be sure to explicitly configure the new attribute with the `DO_NOTHING` attribute action. You must include the same distinct prefix in the new attribute's name.

Using an allowed unsigned attributes list

1. Add the new `DO_NOTHING` attribute to your allowed unsigned attributes list and deploy the updated list.
2. Deploy the change from **Step 1**.

   You cannot move on to **Step 3** until the change has propagated to all hosts that need to read this data.
3. Add the new `DO_NOTHING` attribute to your attribute actions.
   a. If you defined your attribute actions with a `TableSchema`, add the new `DO_NOTHING` attribute to your annotated data class with the `@DynamoDBEncryptionDoNothing` annotation.
   b. If you manually defined your attribute actions, update your attribute actions to include the new attribute. Be sure to explicitly configure the new attribute with the `DO_NOTHING` attribute action.
4. Deploy the change from **Step 3**.

Change an existing `SIGN_ONLY` attribute to `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

To change an existing `SIGN_ONLY` attribute to `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, you must update your attribute actions. After you deploy the update, the client will be able to verify the existing values written to the attribute, and will continue to sign new values written to the attribute. New values written to the attribute will be included in the encryption context.

If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Using an annotated data class

If you defined your attribute actions with a `TableSchema`, update the attribute action associated with the attribute from `@DynamoDBEncryptionSignOnly` to `@DynamoDBEncryptionSignAndIncludeInEncryptionContext`.

Using an object model

If you manually defined your attribute actions, update the attribute action associated with the attribute from `SIGN_ONLY` to `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` in your object model.

Change an existing `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `SIGN_ONLY`

To change an existing `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `SIGN_ONLY`, you must update your attribute actions. After you deploy the update, the client will be able to verify the existing values written to the attribute, and will continue to sign new values written to the attribute. New values written to the attribute will not be included in the encryption context.

Before changing an existing `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attribute to `SIGN_ONLY`, carefully consider how your updates might impact the functionality of your branch key ID supplier.

Using an annotated data class

If you defined your attribute actions with a `TableSchema`, update the attribute action associated with the attribute from `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` to `@DynamoDBEncryptionSignOnly`.

Using an object model

If you manually defined your attribute actions, update the attribute action associated with the attribute from `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` to `SIGN_ONLY` in your object model.

AWS Database Encryption SDK for DynamoDB available programming languages

The AWS Database Encryption SDK for DynamoDB is available for the following programming languages. The language-specific libraries vary, but the resulting implementations are interoperable. You can encrypt with one language implementation and decrypt with another.

Interoperability might be subject to language constraints. If so, these constraints are described in the topic about the language implementation.

Topics

- Java
- .NET
- Rust

Java

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

This topic explains how to install and use version 3.*x* of the Java client-side encryption library for DynamoDB. For details about programming with the AWS Database Encryption SDK for DynamoDB, see the Java examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Note

The following topics focus on version 3.*x* of the Java client-side encryption library for DynamoDB.
Our client-side encryption library was renamed to AWS Database Encryption SDK. The AWS Database Encryption SDK continues to support legacy DynamoDB Encryption Client versions.

Topics

- Prerequisites
- Installation
- Using the Java client-side encryption library for DynamoDB
- Java examples
- Configure an existing DynamoDB table to use the AWS Database Encryption SDK for DynamoDB
- Migrate to version 3.x of the Java client-side encryption library for DynamoDB

Prerequisites

Before you install version 3.*x* of the Java client-side encryption library for DynamoDB, be sure you have the following prerequisites.

A Java development environment

You will need Java 8 or later. On the Oracle website, go to Java SE Downloads, and then download and install the Java SE Development Kit (JDK).

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

AWS SDK for Java 2.x

The AWS Database Encryption SDK for DynamoDB requires the DynamoDB Enhanced Client module of the AWS SDK for Java 2.x. You can install the entire SDK or just this module.

For information about updating your version of the AWS SDK for Java, see Migrating from version 1.x to 2.x of the AWS SDK for Java.

The AWS SDK for Java is available through Apache Maven. You can declare a dependency for the entire AWS SDK for Java, or just the `dynamodb-enhanced` module.

Install the AWS SDK for Java using Apache Maven

- To import the entire AWS SDK for Java as a dependency, declare it in your `pom.xml` file.
- To create a dependency only for the Amazon DynamoDB module in the AWS SDK for Java, follow the instructions for specifying particular modules. Set the `groupId` to `software.amazon.awssdk` and the `artifactID` to `dynamodb-enhanced`.

> **Note**
>
> If you use the AWS KMS keyring or AWS KMS Hierarchical keyring, you also need to create a dependency for the AWS KMS module. Set the `groupId` to `software.amazon.awssdk` and the `artifactID` to `kms`.

Installation

You can install version 3.*x* of the Java client-side encryption library for DynamoDB in the following ways.

Using Apache Maven

The Amazon DynamoDB Encryption Client for Java is available through Apache Maven with the following dependency definition.

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

Using Gradle Kotlin

You can use Gradle to declare a dependency on The Amazon DynamoDB Encryption Client for Java by adding the following to the *dependencies* section of your Gradle project.

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

Manually

To install the Java client-side encryption library for DynamoDB, clone or download the aws-database-encryption-sdk-dynamodb GitHub repository.

After you install the SDK, get started by looking at the example code in this guide and the Java examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Using the Java client-side encryption library for DynamoDB

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

This topic explains some of the functions and helper classes in version 3.*x* of the Java client-side encryption library for DynamoDB.

For details about programming with the Java client-side encryption library for DynamoDB, see the Java examples, the Java examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Topics

Item encryptors

At its core, the AWS Database Encryption SDK for DynamoDB is an item encryptor. You can use version 3.*x* of the Java client-side encryption library for DynamoDB to encrypt, sign, verify, and decrypt your DynamoDB table items in the following ways.

The DynamoDB Enhanced Client

You can configure the DynamoDB Enhanced Client with the `DynamoDbEncryptionInterceptor` to automatically encrypt and sign items client-side with your DynamoDB `PutItem` requests. With the DynamoDB Enhanced Client, you can define your attribute actions using an annotated data class. We recommend using the DynamoDB Enhanced Client whenever possible.

The DynamoDB Enhanced Client does not support searchable encryption.

> **Note**
>
> The AWS Database Encryption SDK does not support annotations on nested attributes.

The low-level DynamoDB API

You can configure the low-level DynamoDB API with the `DynamoDbEncryptionInterceptor` to automatically encrypt and sign items client-side with your DynamoDB `PutItem` requests.

You must use the low-level DynamoDB API to use searchable encryption.

The lower-level `DynamoDbItemEncryptor`

The lower-level `DynamoDbItemEncryptor` directly encrypts and signs or decrypts and verifies your table items without calling DynamoDB. It does not make DynamoDB `PutItem` or `GetItem` requests. For example, you can use the lower-level `DynamoDbItemEncryptor` to directly decrypt and verify a DynamoDB item that you have already retrieved.

The lower-level `DynamoDbItemEncryptor` does not support searchable encryption.

Attribute actions in the AWS Database Encryption SDK for DynamoDB

Attribute actions determine which attribute values are encrypted and signed, which are only signed, which are signed and included in the encryption context, and which are ignored.

> **Note**
>
> To use the `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action, you must use version 3.3 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before updating your data model to include `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

If you use the low-level DynamoDB API or the lower-level `DynamoDbItemEncryptor`, you must manually define your attribute actions. If you use the DynamoDB Enhanced Client you can either manually define your attribute actions, or you can use an annotated data class to generate a `TableSchema`. To simplify the configuration process, we recommend using an annotated data class. When you use an annotated data class, you only have to model your object once.

> **Note**
>
> After you define your attribute actions, you must define which attributes are excluded from the signatures. To make it easier to add new unsigned attributes in the future, we recommend choosing a distinct prefix (such as "`:`") to identify your unsigned attributes. Include this prefix in the attribute name for all attributes marked `DO_NOTHING` as you define your DynamoDB schema and attribute actions.

Use an annotated data class

Use an annotated data class to specify your attribute actions with the DynamoDB Enhanced Client and `DynamoDbEncryptionInterceptor`. The AWS Database Encryption SDK for DynamoDB uses the standard DynamoDB attribute annotations that define the attribute type to determine how to protect an attribute. By default, all attributes are encrypted and signed except for primary keys, which are signed but not encrypted.

> **Note**
>
> To use the `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action, you must use version 3.3 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before updating your data model to include `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

See SimpleClass.java in the aws-database-encryption-sdk-dynamodb repository on GitHub for more guidance on the DynamoDB Enhanced Client annotations.

By default, primary key attributes are signed but not encrypted (`SIGN_ONLY`) and all other attributes are encrypted and signed (`ENCRYPT_AND_SIGN`). If you define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. To specify exceptions, use the encryption annotations defined in the Java client-side encryption library for DynamoDB. For example, if you want a particular attribute to only be signed use the `@DynamoDbEncryptionSignOnly` annotation. If you want a particular attribute to be signed and included in the encryption context, use the `@DynamoDbEncryptionSignAndIncludeInEncryptionContext`. If you want a particular attribute to be neither signed nor encrypted (`DO_NOTHING`), use the `@DynamoDbEncryptionDoNothing` annotation.

> **Note**
>
> The AWS Database Encryption SDK does not support annotations on nested attributes.

The following example shows the annotations used to define `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, and `DO_NOTHING`attribute actions. For an example that shows the annotations used to define `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, see SimpleClass4.java.

```
@DynamoDbBean
public class SimpleClass {

    private String partitionKey;
    private int sortKey;
    private String attribute1;
    private String attribute2;
    private String attribute3;

    @DynamoDbPartitionKey
    @DynamoDbAttribute(value = "partition_key")
    public String getPartitionKey() {
        return this.partitionKey;
    }

    public void setPartitionKey(String partitionKey) {
        this.partitionKey = partitionKey;
    }

    @DynamoDbSortKey
    @DynamoDbAttribute(value = "sort_key")
    public int getSortKey() {
        return this.sortKey;
    }

    public void setSortKey(int sortKey) {
        this.sortKey = sortKey;
    }

    public String getAttribute1() {
        return this.attribute1;
    }

    public void setAttribute1(String attribute1) {
        this.attribute1 = attribute1;
    }

    @DynamoDbEncryptionSignOnly
    public String getAttribute2() {
        return this.attribute2;
    }

    public void setAttribute2(String attribute2) {
        this.attribute2 = attribute2;
    }

    @DynamoDbEncryptionDoNothing
    public String getAttribute3() {
        return this.attribute3;
    }

    @DynamoDbAttribute(value = ":attribute3")
    public void setAttribute3(String attribute3) {
        this.attribute3 = attribute3;
    }

}
```

Use your annotated data class to create the `TableSchema` as shown in the following snippet.

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

Manually define your attribute actions

To manually specify attribute actions, create a `Map` object in which the name-value pairs represent attribute names and the specified actions.

Specify `ENCRYPT_AND_SIGN` to encrypt and sign an attribute. Specify `SIGN_ONLY` to sign, but not encrypt, an attribute. Specify `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` to sign an attribute and include it in the encryption context. You cannot encrypt an attribute without also signing it. Specify `DO_NOTHING` to ignore an attribute.

The partition and sort attributes must be either `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> To use the `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action, you must use version 3.3 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before updating your data model to include `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be signed
attributeActionsOnEncrypt.put("partition_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
```

```
// The sort attribute must be signed
attributeActionsOnEncrypt.put("sort_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute3",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put(":attribute4", CryptoAction.DO_NOTHING);
```

Encryption configuration in the AWS Database Encryption SDK for DynamoDB

When you use the AWS Database Encryption SDK, you must explicitly define an encryption configuration for your DynamoDB table. The values required in your encryption configuration depend on whether you defined your attribute actions manually or with an annotated data class.

The following snippet defines a DynamoDB table encryption configuration using the DynamoDB Enhanced Client, `TableSchema`, and allowed unsigned attributes defined by a distinct prefix.

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
        DynamoDbEnhancedTableEncryptionConfig.builder()
            .logicalTableName(ddbTableName)
            .keyring(kmsKeyring)
            .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
            .schemaOnEncrypt(tableSchema)
            // Optional: only required if you use beacons
            .search(SearchConfig.builder()
                    .writeVersion(1) // MUST be 1
                    .versions(beaconVersions)
                    .build())
            .build());
```

Logical table name

A logical table name for your DynamoDB table.

The logical table name is cryptographically bound to all data stored in the table to simplify DynamoDB restore operations. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. You must always specify the same logical table name. For decryption to succeed, the logical table name must match the name specified on encryption. In the event that your DynamoDB table name changes after restoring your DynamoDB table from a backup, the logical table name ensures that the decrypt operation still recognizes the table.

Allowed unsigned attributes

The attributes marked `DO_NOTHING` in your attribute actions.

The allowed unsigned attributes tell the client which attributes are excluded from the signatures. The client assumes that all other attributes are included in the signature. Then, when decrypting a record, the client determines which attributes it needs to verify and which to ignore from the allowed unsigned attributes you specified. You cannot remove an attribute from your allowed unsigned attributes.

You can define the allowed unsigned attributes explicitly by creating an array that lists all of your `DO_NOTHING` attributes. You can also specify a distinct prefix when naming your `DO_NOTHING` attributes and use the prefix to tell the client which attributes are unsigned. We strongly recommend specifying a distinct prefix because it simplifies the process of adding a new `DO_NOTHING` attribute in the future. For more information, see Updating your data model.

If you do not specify a prefix for all `DO_NOTHING` attributes, you can configure an `allowedUnsignedAttributes` array that explicitly lists all of the attributes that the client should expect to be unsigned when it encounters them on decryption. You should only explicitly define your allowed unsigned attributes if absolutely necessary.

Search Configuration (Optional)

The `SearchConfig` defines the beacon version.

The `SearchConfig` must be specified to use searchable encryption or signed beacons.

Algorithm Suite (Optional)

The `algorithmSuiteId` defines which algorithm suite the AWS Database Encryption SDK uses.

Unless you explicitly specify an alternative algorithm suite, the AWS Database Encryption SDK uses the default algorithm suite. The default algorithm suite uses the AES-GCM algorithm with key derivation, digital signatures, and key commitment. Although the default algorithm suite is likely to be suitable for most applications, you can choose an alternate algorithm suite. For example, some trust models would be satisfied by an algorithm suite without digital signatures. For information about the algorithm suites that the AWS Database Encryption SDK supports, see Supported algorithm suites in the AWS Database Encryption SDK.

To select the AES-GCM algorithm suite without ECDSA digital signatures, include the following snippet in your table encryption configuration.

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

Updating items with the AWS Database Encryption SDK

The AWS Database Encryption SDK does not support ddb:UpdateItem for items that have been encrypted or signed. To update an encrypted or signed item, you must use ddb:PutItem. When you specify the same primary key as an existing item in your `PutItem` request, the new item completely replaces the existing item. You can also use CLOBBER to clear and replace all attributes on save after updating your items.

Decrypting signed sets

In versions 3.0.0 and 3.1.0 of the AWS Database Encryption SDK, if you define a set type attribute as `SIGN_ONLY`, the values of the set are canonicalized in the order that they are provided. DynamoDB does not preserve the order of sets. As a result, there is a chance that signature validation of the item that contains the set will fail. Signature validation fails when the values of the set are returned in a different order than they were provided to the AWS Database Encryption SDK, even if the set attributes contain the same values.

> **Note**
>
> Versions 3.1.1 and later of the AWS Database Encryption SDK canonicalize the values of all set type attributes, so that the values are read in the same order that they were written to DynamoDB.

If signature validation fails, the decrypt operation fails and returns the following error message.

```
software.amazon.cryptography.dbencryptionsdk.structuredencryption.model.StructuredEncryptionException: No recipient tag matched.
```

If you receive the above error message, and believe that the item you are trying to decrypt includes a set that was signed using version 3.0.0 or 3.1.0, see the DecryptWithPermute directory of the aws-database-encryption-sdk-dynamodb-java repository on GitHub for details on how to successfully validate the set.

Java examples

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The following examples show you how to use the Java client-side encryption library for DynamoDB to protect the table items in your application. You can find more examples (and contribute your own) in the Java examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

The following examples demonstrate how to configure the Java client-side encryption library for DynamoDB in a new, unpopulated Amazon DynamoDB table. If you want to configure your existing Amazon DynamoDB tables for client-side encryption, see Add version 3.x to an existing table.

Topics

Using the DynamoDB enhanced client

The following example shows how to use the DynamoDB Enhanced Client and `DynamoDbEncryptionInterceptor` with an AWS KMS keyring to encrypt DynamoDB table items as part of your DynamoDB API calls.

You can use any supported keyring with the DynamoDB Enhanced Client, but we recommend using one of the AWS KMS keyrings whenever possible.

Note

The DynamoDB Enhanced Client does not support searchable encryption. Use the `DynamoDbEncryptionInterceptor` with the low-level DynamoDB API to use searchable encryption.

See the complete code sample: EnhancedPutGetExample.java

Step 1: Create the AWS KMS keyring

The following example uses `CreateAwsKmsMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

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

Step 2: Create a table schema from the annotated data class

The following example uses the annotated data class to create the `TableSchema`.

This example assumes that the annotated data class and attribute actions were defined using the SimpleClass.java. For more guidance on annotating your attribute actions, see Use an annotated data class.

Note

The AWS Database Encryption SDK does not support annotations on nested attributes.

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

Step 3: Define which attributes are excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client assumes that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Step 4: Create the encryption configuration

The following example defines a `tableConfigs` Map that represents the encryption configuration for the DynamoDB table.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. For more information, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

> **Note**
>
> To use searchable encryption or signed beacons, you must also include the `SearchConfig` in your encryption configuration.

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

Step 5: Creates the `DynamoDbEncryptionInterceptor`

The following example creates a new `DynamoDbEncryptionInterceptor` with the `tableConfigs` from **Step 4**.

```
final DynamoDbEncryptionInterceptor interceptor =
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

Step 6: Create a new AWS SDK DynamoDB client

The following example creates a new AWS SDK DynamoDB client using the `interceptor` from **Step 5**.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

Step 7: Create the DynamoDB Enhanced Client and create a table

The following example creates the DynamoDB Enhanced Client using the AWS SDK DynamoDB client created in **Step 6** and creates a table using the annotated data class.

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

Step 8: Encrypt and sign a table item

The following example puts an item into the DynamoDB table using the DynamoDB Enhanced Client. The item is encrypted and signed client-side before it is send to DynamoDB.

```
final SimpleClass item = new SimpleClass();
item.setPartitionKey("EnhancedPutGetExample");
item.setSortKey(0);
item.setAttribute1("encrypt and sign me!");
item.setAttribute2("sign me!");
item.setAttribute3("ignore me!");

table.putItem(item);
```

Using the low-level DynamoDB API

The following example shows how to use the low-level DynamoDB API with an AWS KMS keyring to automatically encrypt and sign items client-side with your DynamoDB `PutItem` requests.

You can use any supported keyring, but we recommend using one of the AWS KMS keyrings whenever possible.

See the complete code sample: BasicPutGetExample.java

Step 1: Create the AWS KMS keyring

The following example uses `CreateAwsKmsMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

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

Step 2: Configure your attribute actions

The following example defines an `attributeActionsOnEncrypt` Map that represents sample attribute actions for a table item.

Note

The following example does not define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

Step 3: Define which attributes are excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client assumes that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Step 4: Define the DynamoDB table encryption configuration

The following example defines a `tableConfigs` Map that represents the encryption configuration for this DynamoDB table.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. For more information, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

Note

To use searchable encryption or signed beacons, you must also include the `SearchConfig` in your encryption configuration.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
tableConfigs.put(ddbTableName, config);
```

Step 5: Create the `DynamoDbEncryptionInterceptor`

The following example creates the `DynamoDbEncryptionInterceptor` using the `tableConfigs` from **Step 4**.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

Step 6: Create a new AWS SDK DynamoDB client

The following example creates a new AWS SDK DynamoDB client using the `interceptor` from **Step 5**.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

Step 7: Encrypt and sign a DynamoDB table item

The following example defines an `item` Map that represents a sample table item and puts the item in the DynamoDB table. The item is encrypted and signed client-side before it is sent to DynamoDB.

```
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("partition_key", AttributeValue.builder().s("BasicPutGetExample").build());
item.put("sort_key", AttributeValue.builder().n("0").build());
item.put("attribute1", AttributeValue.builder().s("encrypt and sign me!").build());
item.put("attribute2", AttributeValue.builder().s("sign me!").build());
item.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

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

final PutItemResponse putResponse = ddb.putItem(putRequest);
```

Using the lower-level DynamoDbItemEncryptor

The following example shows how to use the lower-level `DynamoDbItemEncryptor` with an AWS KMS keyring to directly encrypt and sign table items. The `DynamoDbItemEncryptor` does not put the item in your DynamoDB table.

You can use any supported keyring with the DynamoDB Enhanced Client, but we recommend using one of the AWS KMS keyrings whenever possible.

Note

The lower-level `DynamoDbItemEncryptor` does not support searchable encryption. Use the `DynamoDbEncryptionInterceptor` with the low-level DynamoDB API to use searchable encryption.

See the complete code sample: ItemEncryptDecryptExample.java

Step 1: Create the AWS KMS keyring

The following example uses `CreateAwsKmsMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

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

Step 2: Configure your attribute actions

The following example defines an `attributeActionsOnEncrypt` Map that represents sample attribute actions for a table item.

Note

The following example does not define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

Step 3: Define which attributes are excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client assumes that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Step 4: Define the `DynamoDbItemEncryptor` configuration

The following example defines the configuration for the `DynamoDbItemEncryptor`.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. For more information, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

Step 5: Create the `DynamoDbItemEncryptor`

The following example creates a new `DynamoDbItemEncryptor` using the `config` from **Step 4**.

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

Step 6: Directly encrypt and sign a table item

The following example directly encrypts and signs an item using the `DynamoDbItemEncryptor`. The `DynamoDbItemEncryptor` does not put the item in your DynamoDB table.

```
final Map<String, AttributeValue> originalItem = new HashMap<>();
originalItem.put("partition_key",
 AttributeValue.builder().s("ItemEncryptDecryptExample").build());
originalItem.put("sort_key", AttributeValue.builder().n("0").build());
originalItem.put("attribute1", AttributeValue.builder().s("encrypt and sign
 me!").build());
originalItem.put("attribute2", AttributeValue.builder().s("sign me!").build());
originalItem.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final Map<String, AttributeValue> encryptedItem = itemEncryptor.EncryptItem(
        EncryptItemInput.builder()
                .plaintextItem(originalItem)
                .build()
).encryptedItem();
```

Configure an existing DynamoDB table to use the AWS Database Encryption SDK for DynamoDB

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

With version 3.*x* of the Java client-side encryption library for DynamoDB, you can configure your existing Amazon DynamoDB tables for client-side encryption. This topic provides guidance on the three steps you must take to add version 3.*x* to an existing, populated DynamoDB table.

Prerequisites

Version 3.*x* of the Java client-side encryption library for DynamoDB requires the DynamoDB Enhanced Client provided in AWS SDK for Java 2.x . If you still use the DynamoDBMapper, you must migrate to AWS SDK for Java 2.x to use the DynamoDB Enhanced Client.

Follow the instructions for migrating from version 1.x to 2.x of the AWS SDK for Java.

Then, follow the instructions to Get Started using the DynamoDB Enhanced Client API.

Before configuring your table to use the Java client-side encryption library for DynamoDB, you need to generate a `TableSchema` using an annotated data class and create an enhanced client.

Step 1: Prepare to read and write encrypted items

Complete the following steps to prepare your AWS Database Encryption SDK client to read and write encrypted items. After you deploy the following changes, your client will continue to read and write plaintext items. It will not encrypt or sign any new items written to the table, but it will be able to decrypt encrypted items as soon as they appear. These changes prepare the client to begin encrypting new items. The following changes must be deployed to each reader before you proceed to the next step.

1. Define your attribute actions

Update your annotated data class to include attribute actions that define which attribute values will be encrypted and signed, which will be only signed, and which will be ignored.

See the SimpleClass.java in the aws-database-encryption-sdk-dynamodb repository on GitHub for more guidance on the DynamoDB Enhanced Client annotations.

By default, primary key attributes are signed but not encrypted (`SIGN_ONLY`) and all other attributes are encrypted and signed (`ENCRYPT_AND_SIGN`). To specify exceptions, use the encryption annotations defined in the Java client-side encryption library for DynamoDB. For example, if you want a particular attribute to be sign only use the `@DynamoDbEncryptionSignOnly` annotation. If you want a particular attribute to be signed and included in the encryption context, use the

`@DynamoDbEncryptionSignAndIncludeInEncryptionContext` annotation. If you want a particular attribute to be neither signed nor encrypted (`DO_NOTHING`), use the `@DynamoDbEncryptionDoNothing` annotation.

> **Note**
>
> If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. For an example that shows the annotations used to define `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, see SimpleClass4.java.

For example annotations, see Use an annotated data class.

2. Define which attributes will be excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client will assume that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

3. Create a keyring

The following example creates an AWS KMS keyring. The AWS KMS keyring uses symmetric encryption or asymmetric RSA AWS KMS keys to generate, encrypt, and decrypt data keys.

This example uses `CreateMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

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

4. Define the DynamoDB table encryption configuration

The following example defines a `tableConfigs` Map that represents the encryption configuration for this DynamoDB table.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. For more information, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

You must specify `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` as the plaintext override. This policy continues to read and write plaintext items, reads encrypted items, and prepares the client to write encrypted items.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

5. Create the `DynamoDbEncryptionInterceptor`

The following example creates the `DynamoDbEncryptionInterceptor` using the `tableConfigs` from **Step 3**.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

Step 2: Write encrypted and signed items

Update the plaintext policy in your `DynamoDbEncryptionInterceptor` configuration to allow the client to write encrypted and signed items. After you deploy the following change, the client will encrypt and sign new items based on the attribute actions you configured in **Step 1**. The client will be able read plaintext items and encrypted and signed items.

Before you proceed to Step 3, you must encrypt and sign all existing plaintext items in your table. There is no single metric or query that you can run to quickly encrypt your existing plaintext items. Use the process that makes the most sense for your system. For example, you could use an asynchronous process that slowly scans the table and the rewrites the items using the attribute actions and encryption configuration you defined. To identify the plaintext items in your table, we recommend scanning for all items that do not contain the `aws_dbe_head` and `aws_dbe_foot` attributes that the AWS Database Encryption SDK adds to items when they're encrypted and signed.

The following example updates the table encryption configuration from **Step 1**. You must update the plaintext override with `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`. This policy continues to read plaintext items, but also reads and writes encrypted items. Create a new `DynamoDbEncryptionInterceptor` using the updated `tableConfigs`.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
         .logicalTableName(ddbTableName)
         .partitionKeyName("partition_key")
         .sortKeyName("sort_key")
         .schemaOnEncrypt(tableSchema)
         .keyring(kmsKeyring)
         .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
         .build();
tableConfigs.put(ddbTableName, config);
```

Step 3: Only read encrypted and signed items

After you have encrypted and signed all of your items, update the plaintext override in your `DynamoDbEncryptionInterceptor` configuration to only allow the client to read and write encrypted and signed items. After you deploy the following change, the client will encrypt and sign new items based on the attribute actions you configured in **Step 1**. The client will only be able read encrypted and signed items.

The following example updates the table encryption configuration from **Step 2**. You can either update the plaintext override with `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT` or remove the plaintext policy from your configuration. The client only reads and writes encrypted and signed items by default. Create a new `DynamoDbEncryptionInterceptor` using the updated `tableConfigs`.

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        // Optional: you can also remove the plaintext policy from your configuration

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

Migrate to version 3.x of the Java client-side encryption library for DynamoDB

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

Version 3.*x* of the Java client-side encryption library for DynamoDB is a major rewrite of the 2.*x* code base. It includes many updates, such as a new structured data format, improved multitenancy support, seamless schema changes, and searchable encryption support. This topic provides guidance on how to migrate your code to version 3.*x*.

Migrating from version 1.x to 2.x

Migrate to version 2.*x* before you migrate to version 3.*x*. Version 2.*x* changed the symbol for the Most Recent Provider from `MostRecentProvider` to `CachingMostRecentProvider`. If you currently use version 1.*x* of the Java client-side encryption library for DynamoDB with the `MostRecentProvider` symbol, you must update the symbol name in your code to `CachingMostRecentProvider`. For more information, see Updates to the Most Recent Provider.

Migrating from version 2.x to 3.x

The following procedures describe how to migrate your code from version 2.*x* to version 3.*x* of the Java client-side encryption library for DynamoDB.

Step 1. Prepare to read items in the new format

Complete the following steps to prepare your AWS Database Encryption SDK client to read items in the new format. After you deploy the following changes, your client will continue to behave in the same manner that it did in version 2.*x*. Your client will continue to read and write items in the version 2.*x* format, but these changes prepare the client to read items in the new format.

Update your AWS SDK for Java to version 2.x

Version 3.*x* of the Java client-side encryption library for DynamoDB requires the DynamoDB Enhanced Client. The DynamoDB Enhanced Client replaces the DynamoDBMapper used in previous versions. To use the enhanced client, you must use the AWS SDK for Java 2.x.

Follow the instructions for migrating from version 1.x to 2.x of the AWS SDK for Java.

For more information on what AWS SDK for Java 2.x modules are required, see Prerequisites.

Configure your client to read items encrypted by legacy versions

The following procedures provide an overview of the steps demonstrated in the code example below.

1. Create a keyring.

   Keyrings and cryptographic materials managers replace the cryptographic materials providers used in previous versions of the Java client-side encryption library for DynamoDB.

   > ⚠ **Important**
   >
   > The wrapping keys you specify when creating a keyring must be the same wrapping keys you used with your cryptographic materials provider in version 2.*x*.

2. Create a table schema over your annotated class.

This step defines the attribute actions that will be used when you begin writing items in the new format.

For guidance on using the new DynamoDB Enhanced Client, see the Generate a `TableSchema` in the *AWS SDK for Java Developer Guide*.

The following example assumes you updated your annotated class from version 2.*x* using the new attribute actions annotations. For more guidance on annotating your attribute actions, see Use an annotated data class.

> **Note**
>
> If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. For an example that shows the annotations used to define `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, see SimpleClass4.java.

3. Define which attributes are excluded from the signature.
4. Configure an explicit map of the attribute actions configured in your version 2.x modeled class.

   This step defines the attribute actions used to write items in the old format.
5. Configure the `DynamoDBEncryptor` you used in version 2.*x* of the Java client-side encryption library for DynamoDB.
6. Configure the legacy behavior.
7. Create a `DynamoDbEncryptionInterceptor`.
8. Create a new AWS SDK DynamoDB client.
9. Create the `DynamoDBEnhancedClient` and create a table with your modeled class.

   For more information on the DynamoDB Enhanced Client, see create an enhanced client.

```
public class MigrationExampleStep1 {

    public static void MigrationStep1(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Create a Keyring.
        //    This example creates an AWS KMS Keyring that specifies the
        //    same kmsKeyId previously used in the version 2.x configuration.
        //    It uses the 'CreateMrkMultiKeyring' method to create the
        //    keyring, so that the keyring can correctly handle both single
        //    region and Multi-Region KMS Keys.
        //    Note that this example uses the AWS SDK for Java v2 KMS client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        // 2. Create a Table Schema over your annotated class.
        //    For guidance on using the new attribute actions
        //    annotations, see SimpleClass.java in the
        //    aws-database-encryption-sdk-dynamodb GitHub repository.
        //    All primary key attributes must be signed but not encrypted
        //    and by default all non-primary key attributes
        //    are encrypted and signed (ENCRYPT_AND_SIGN).
        //    If you want a particular non-primary key attribute to be signed but
        //    not encrypted, use the 'DynamoDbEncryptionSignOnly' annotation.
        //    If you want a particular attribute to be neither signed nor encrypted
        //    (DO_NOTHING), use the 'DynamoDbEncryptionDoNothing' annotation.
        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        // 3. Define which attributes the client should expect to be excluded
        //    from the signature when reading items.
        //    This value represents all unsigned attributes across the entire
        //    dataset.
        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        // 4. Configure an explicit map of the attribute actions configured
        //    in your version 2.x modeled class.
        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);
        // 5. Configure the DynamoDBEncryptor that you used in version 2.x.
        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 6. Configure the legacy behavior.
        //    Input the DynamoDBEncryptor and attribute actions created in
        //    the previous steps. For Legacy Policy, use
        //    'FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This policy continues to
read
        //    and write items using the old format, but will be able to read
        //    items written in the new format as soon as they appear.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 7. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 8. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 7.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
```

```
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 9. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb client
        //    created in Step 8, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

Step 2. Write items in the new format

After you have deployed the changes from Step 1 to all readers, complete the following steps to configure your AWS Database Encryption SDK client to write items in the new format. After you deploy the following changes, your client will continue read items in the old format and start writing and reading items in the new format.

The following procedures provide an overview of the steps demonstrated in the code example below.

1. Continue configuring your keyring, table schema, legacy attribute actions, `allowedUnsignedAttributes`, and `DynamoDBEncryptor` as you did in **Step 1**.
2. Update your legacy behavior to only write new items using the new format.
3. Create a `DynamoDbEncryptionInterceptor`
4. Create a new AWS SDK DynamoDB client.
5. Create the `DynamoDBEnhancedClient` and create a table with your modeled class.

   For more information on the DynamoDB Enhanced Client, see create an enhanced client.

```
public class MigrationExampleStep2 {

    public static void MigrationStep2(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema, legacy
        //    attribute actions, allowedUnsignedAttributes, and
        //    DynamoDBEncryptor as you did in Step 1.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 2. Update your legacy behavior to only write new items using the new
        //    format.
        //    For Legacy Policy, use 'FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This
policy
        //    continues to read items in both formats, but will only write items
        //    using the new format.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
```

```
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb Client
 created
        //    in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

After deploying the Step 2 changes, you must re-encrypt all old items in your table with the new format before you can continue on to Step 3. There is no single metric or query that you can run to quickly encrypt your existing items. Use the process that makes the most sense for your system. For example, you could use an asynchronous process that slowly scans the table and the rewrites the items using the new attribute actions and encryption configuration you defined.

Step 3. Only read and write items in the new format

After re-encrypting all of the items in your table with the new format, you can remove the legacy behavior from your configuration. Complete the following steps to configure your client to only read and write items in the new format.

The following procedures provide an overview of the steps demonstrated in the code example below.

1. Continue configuring your keyring, table schema, and `allowedUnsignedAttributes` as you did in **Step 1**. Remove the legacy attribute actions and `DynamoDBEncryptor` from your configuration.
2. Create a `DynamoDbEncryptionInterceptor`.
3. Create a new AWS SDK DynamoDB client.
4. Create the `DynamoDBEnhancedClient` and create a table with your modeled class.

   For more information on the DynamoDB Enhanced Client, see create an enhanced client.

```
public class MigrationExampleStep3 {

    public static void MigrationStep3(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema,
        //    and allowedUnsignedAttributes as you did in Step 1.
        //    Do not include the configurations for the DynamoDBEncryptor or
        //    the legacy attribute actions.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");
        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        //    Do not configure any legacy behavior.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK Client
        //    created in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

.NET

This topic explains how to install and use version 3.*x* of the .NET client-side encryption library for DynamoDB. For details about programming with the AWS Database Encryption SDK for

DynamoDB, see the .NET examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

The .NET client-side encryption library for DynamoDB is for developers who are writing applications in C# and other .NET programming languages. It is supported on Windows, macOS, and Linux.

All programming language implementations of the AWS Database Encryption SDK for DynamoDB are interoperable. However, the SDK for .NET does not support empty values for list or map data types. This means that if you use the Java client-side encryption library for DynamoDB to write an item that contains empty values for a list or map data type, you cannot decrypt and read that item using the .NET client-side encryption library for DynamoDB.

Topics

Installing the .NET client-side encryption library for DynamoDB

The .NET client-side encryption library for DynamoDB is available as the AWS.Cryptography.DbEncryptionSDK.DynamoDb package in NuGet. For details about installing and building the library, see the .NET README.md file in the aws-database-encryption-sdk-dynamodb repository. The .NET client-side encryption library for DynamoDB requires the SDK for .NET even if you aren't using AWS Key Management Service (AWS KMS) keys. The SDK for .NET is installed with the NuGet package.

Version 3.*x* of the .NET client-side encryption library for DynamoDB supports .NET 6.0 and .NET Framework net48 and later.

Debugging with .NET

The .NET client-side encryption library for DynamoDB does not generate any logs. Exceptions in the .NET client-side encryption library for DynamoDB generate an exception message, but no stack traces.

To help you debug, be sure to enable logging in the SDK for .NET. The logs and error messages from the SDK for .NET can help you distinguish errors arising in the SDK for .NET from those in the .NET client-side encryption library for DynamoDB. For help with SDK for .NET logging, see AWSLogging in the *AWS SDK for .NET Developer Guide*. (To see the topic, expand the **Open to view .NET Framework content** section.)

Using the .NET client-side encryption library for DynamoDB

This topic explains some of the functions and helper classes in version 3.*x* of the .NET client-side encryption library for DynamoDB.

For details about programming with the .NET client-side encryption library for DynamoDB, see the .NET examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Topics

Item encryptors

At its core, the AWS Database Encryption SDK for DynamoDB is an item encryptor. You can use version 3.*x* of the .NET client-side encryption library for DynamoDB to encrypt, sign, verify, and decrypt your DynamoDB table items in the following ways.

The low-level AWS Database Encryption SDK for DynamoDB API

> You can use your table encryption configuration to construct a DynamoDB client that automatically encrypts and signs items client-side with your DynamoDB `PutItem` requests. You can use this client directly, or you can construct a document model or object persistence model.
>
> You must use the low-level AWS Database Encryption SDK for DynamoDB API to use searchable encryption.

The lower-level `DynamoDbItemEncryptor`

> The lower-level `DynamoDbItemEncryptor` directly encrypts and signs or decrypts and verifies your table items without calling DynamoDB. It does not make DynamoDB `PutItem` or `GetItem`

requests. For example, you can use the lower-level `DynamoDbItemEncryptor` to directly decrypt and verify a DynamoDB item that you have already retrieved. If you use the lower-level `DynamoDbItemEncryptor`, we recommend using the low-level programming modelthat the SDK for .NET provides for communicating with DynamoDB.

The lower-level `DynamoDbItemEncryptor` does not support searchable encryption.

Attribute actions in the AWS Database Encryption SDK for DynamoDB

Attribute actions determine which attribute values are encrypted and signed, which are only signed, which are signed and included in the encryption context, and which are ignored.

To specify attribute actions with the .NET client, manually define attribute actions using an object model. Specify your attribute actions by creating a `Dictionary` object in which the name-value pairs represent attribute names and the specified actions.

Specify `ENCRYPT_AND_SIGN` to encrypt and sign an attribute. Specify `SIGN_ONLY` to sign, but not encrypt, an attribute. Specify `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` to sign an attribute and include it in the encryption context. You cannot encrypt an attribute without also signing it. Specify `DO_NOTHING` to ignore an attribute.

The partition and sort attributes must be either `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> After you define your attribute actions, you must define which attributes are excluded from the signatures. To make it easier to add new unsigned attributes in the future, we recommend choosing a distinct prefix (such as "`:`") to identify your unsigned attributes. Include this prefix in the attribute name for all attributes marked `DO_NOTHING` as you define your DynamoDB schema and attribute actions.

The following object model demonstrates how to specify `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, and `DO_NOTHING` attribute actions with the .NET client. This example uses the prefix "`:`" to identify `DO_NOTHING` attributes.

Note

To use the `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` cryptographic action, you must use version 3.3 or later of the AWS Database Encryption SDK. Deploy the new version to all readers before updating your data model to include `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The
 partition attribute must be signed
    ["sort_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The sort
 attribute must be signed
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    ["attribute3"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT,
    [":attribute4"] = CryptoAction.DO_NOTHING
};
```

Encryption configuration in the AWS Database Encryption SDK for DynamoDB

When you use the AWS Database Encryption SDK, you must explicitly define an encryption configuration for your DynamoDB table. The values required in your encryption configuration depend on whether you defined your attribute actions manually or with an annotated data class.

The following snippet defines a DynamoDB table encryption configuration using the low-level AWS Database Encryption SDK for DynamoDB API and allowed unsigned attributes defined by a distinct prefix.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: SearchConfig only required if you use beacons
    Search = new SearchConfig
    {
        WriteVersion = 1, // MUST be 1
        Versions = beaconVersions
    }
};
tableConfigs.Add(ddbTableName, config);
```

Logical table name

A logical table name for your DynamoDB table.

The logical table name is cryptographically bound to all data stored in the table to simplify DynamoDB restore operations. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. You must always specify the same logical table name. For decryption to succeed, the logical table name must match the name specified on encryption. In the event that your DynamoDB table name changes after restoring your DynamoDB table from a backup, the logical table name ensures that the decrypt operation still recognizes the table.

Allowed unsigned attributes

The attributes marked `DO_NOTHING` in your attribute actions.

The allowed unsigned attributes tell the client which attributes are excluded from the signatures. The client assumes that all other attributes are included in the signature. Then, when decrypting a record, the client determines which attributes it needs to verify and which to ignore from the allowed unsigned attributes you specified. You cannot remove an attribute from your allowed unsigned attributes.

You can define the allowed unsigned attributes explicitly by creating an array that lists all of your `DO_NOTHING` attributes. You can also specify a distinct prefix when naming your `DO_NOTHING` attributes and use the prefix to tell the client which attributes are unsigned. We strongly recommend specifying a distinct prefix because it simplifies the process of adding a new `DO_NOTHING` attribute in the future. For more information, see Updating your data model.

If you do not specify a prefix for all `DO_NOTHING` attributes, you can configure an `allowedUnsignedAttributes` array that explicitly lists all of the attributes that the client should expect to be unsigned when it encounters them on decryption. You should only explicitly define your allowed unsigned attributes if absolutely necessary.

Search Configuration (Optional)

The `SearchConfig` defines the beacon version.

The `SearchConfig` must be specified to use searchable encryption or signed beacons.

Algorithm Suite (Optional)

The `algorithmSuiteId` defines which algorithm suite the AWS Database Encryption SDK uses.

Unless you explicitly specify an alternative algorithm suite, the AWS Database Encryption SDK uses the default algorithm suite. The default algorithm suite uses the AES-GCM algorithm with key derivation, digital signatures, and key commitment. Although the default algorithm suite is likely to be suitable for most applications, you can choose an alternate algorithm suite. For example, some trust models would be satisfied by an algorithm suite without digital signatures. For information about the algorithm suites that the AWS Database Encryption SDK supports, see Supported algorithm suites in the AWS Database Encryption SDK.

To select the AES-GCM algorithm suite without ECDSA digital signatures, include the following snippet in your table encryption configuration.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Updating items with the AWS Database Encryption SDK

The AWS Database Encryption SDK does not support ddb:UpdateItem for items that include encrypted or signed attributes. To update an encrypted or signed attribute, you must use ddb:PutItem. When you specify the same primary key as an existing item in your `PutItem` request, the new item completely replaces the existing item. You can also use CLOBBER to clear and replace all attributes on save after updating your items.

.NET examples

The following examples show you how to use the .NET client-side encryption library for DynamoDB to protect the table items in your application. To find more examples (and contribute your own), see the .NET examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

The following examples demonstrate how to configure the .NET client-side encryption library for DynamoDB in a new, unpopulated Amazon DynamoDB table. If you want to configure your existing Amazon DynamoDB tables for client-side encryption, see Add version 3.x to an existing table.

Topics

- Using the low-level AWS Database Encryption SDK for DynamoDB API
- Using the lower-level DynamoDbItemEncryptor

Using the low-level AWS Database Encryption SDK for DynamoDB API

The following example shows how to use the low-level AWS Database Encryption SDK for DynamoDB API with an AWS KMS keyring to automatically encrypt and sign items client-side with your DynamoDB `PutItem` requests.

You can use any supported keyring, but we recommend using one of the AWS KMS keyrings whenever possible.

See the complete code sample: BasicPutGetExample.cs

Step 1: Create the AWS KMS keyring

The following example uses `CreateAwsKmsMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

Step 2: Configure your attribute actions

The following example defines an `attributeActionsOnEncrypt` Dictionary that represents sample attribute actions for a table item.

> **Note**
>
> The following example does not define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you specify any

`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

Step 3: Define which attributes are excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client assumes that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
const String unsignAttrPrefix = ":";
```

Step 4: Define the DynamoDB table encryption configuration

The following example defines a `tableConfigs` Map that represents the encryption configuration for this DynamoDB table.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. For more information, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

Note

To use searchable encryption or signed beacons, you must also include the `SearchConfig` in your encryption configuration.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
tableConfigs.Add(ddbTableName, config);
```

Step 5: Create a new AWS SDK DynamoDB client

The following example creates a new AWS SDK DynamoDB client using the `TableEncryptionConfigs` from **Step 4**.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

Step 6: Encrypt and sign a DynamoDB table item

The following example defines an `item` Dictionary that represents a sample table item and puts the item in the DynamoDB table. The item is encrypted and signed client-side before it is sent to DynamoDB.

```
var item = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("BasicPutGetExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

PutItemRequest putRequest = new PutItemRequest
{
    TableName = ddbTableName,
    Item = item
};

PutItemResponse putResponse = await ddb.PutItemAsync(putRequest);
```

Using the lower-level `DynamoDbItemEncryptor`

The following example shows how to use the lower-level `DynamoDbItemEncryptor` with an AWS KMS keyring to directly encrypt and sign table items. The `DynamoDbItemEncryptor` does not put the item in your DynamoDB table.

You can use any supported keyring with the DynamoDB Enhanced Client, but we recommend using one of the AWS KMS keyrings whenever possible.

> **Note**
>
> The lower-level `DynamoDbItemEncryptor` does not support searchable encryption. Use the the low-level AWS Database Encryption SDK for DynamoDB API to use searchable encryption.

See the complete code sample: ItemEncryptDecryptExample.cs

Step 1: Create the AWS KMS keyring

The following example uses `CreateAwsKmsMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

Step 2: Configure your attribute actions

The following example defines an `attributeActionsOnEncrypt` Dictionary that represents sample attribute actions for a table item.

> **Note**
>
> The following example does not define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
var attributeActionsOnEncrypt = new Dictionary<String, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

Step 3: Define which attributes are excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client assumes that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
String unsignAttrPrefix = ":";
```

Step 4: Define the `DynamoDbItemEncryptor` configuration

The following example defines the configuration for the `DynamoDbItemEncryptor`.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. For more information, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
```

Step 5: Create the `DynamoDbItemEncryptor`

The following example creates a new `DynamoDbItemEncryptor` using the `config` from **Step 4**.

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

Step 6: Directly encrypt and sign a table item

The following example directly encrypts and signs an item using the `DynamoDbItemEncryptor`. The `DynamoDbItemEncryptor` does not put the item in your DynamoDB table.

```
var originalItem = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("ItemEncryptDecryptExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

var encryptedItem = itemEncryptor.EncryptItem(
    new EncryptItemInput { PlaintextItem = originalItem }
).EncryptedItem;
```

Configure an existing DynamoDB table to use the AWS Database Encryption SDK for DynamoDB

With version 3.*x* of the .NET client-side encryption library for DynamoDB, you can configure your existing Amazon DynamoDB tables for client-side encryption. This topic provides guidance on the three steps you must take to add version 3.*x* to an existing, populated DynamoDB table.

Step 1: Prepare to read and write encrypted items

Complete the following steps to prepare your AWS Database Encryption SDK client to read and write encrypted items. After you deploy the following changes, your client will continue to read and write plaintext items. It will not encrypt or sign any new items written to the table, but it will be able to decrypt encrypted items as soon as they appear. These changes prepare the client to begin encrypting new items. The following changes must be deployed to each reader before you proceed to the next step.

1. Define your attribute actions

Create an object model to define which attribute values will be encrypted and signed, which will be only signed, and which will be ignored.

By default, primary key attributes are signed but not encrypted (`SIGN_ONLY`) and all other attributes are encrypted and signed (`ENCRYPT_AND_SIGN`).

Specify `ENCRYPT_AND_SIGN` to encrypt and sign an attribute. Specify `SIGN_ONLY` to sign, but not encrypt, an attribute. Specify `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` to sign and attribute and include it in the encryption context. You cannot encrypt an attribute without also signing it. Specify `DO_NOTHING` to ignore an attribute. For more information, see Attribute actions in the AWS Database Encryption SDK for DynamoDB.

> **Note**
>
> If you specify any `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` attributes, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

2. Define which attributes will be excluded from the signatures

The following example assumes that all `DO_NOTHING` attributes share the distinct prefix "`:`", and uses the prefix to define the allowed unsigned attributes. The client will assume that any attribute name with the "`:`" prefix is excluded from the signatures. For more information, see Allowed unsigned attributes.

```
const String unsignAttrPrefix = ":";
```

3. Create a keyring

The following example creates an AWS KMS keyring. The AWS KMS keyring uses symmetric encryption or asymmetric RSA AWS KMS keys to generate, encrypt, and decrypt data keys.

This example uses `CreateMrkMultiKeyring` to create an AWS KMS keyring with a symmetric encryption KMS key. The `CreateAwsKmsMrkMultiKeyring` method ensures that the keyring will correctly handle both single-Region and multi-Region keys.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. Define the DynamoDB table encryption configuration

The following example defines a `tableConfigs` Map that represents the encryption configuration for this DynamoDB table.

This example specifies the DynamoDB table name as the logical table name. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration.

You must specify `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` as the plaintext override. This policy continues to read and write plaintext items, reads encrypted items, and prepares the client to write encrypted items.

For more information on the values included in the table encryption configuration, see Encryption configuration in the AWS Database Encryption SDK for DynamoDB.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

5. Create a new AWS SDK DynamoDB client

he following example creates a new AWS SDK DynamoDB client using the `TableEncryptionConfigs` from **Step 4**.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

Step 2: Write encrypted and signed items

Update the plaintext policy in your table encryption configuration to allow the client to write encrypted and signed items. After you deploy the following change, the client will encrypt and sign new items based on the attribute actions you configured in **Step 1**. The client will be able read plaintext items and encrypted and signed items.

Before you proceed to Step 3, you must encrypt and sign all existing plaintext items in your table. There is no single metric or query that you can run to quickly encrypt your existing plaintext items. Use the process that makes the most sense for your system. For example, you could use an asynchronous process that slowly scans the table and the rewrites the items using the attribute actions and encryption configuration you defined. To identify the plaintext items in your table, we recommend scanning for all items that do not contain the `aws_dbe_head` and `aws_dbe_foot` attributes that the AWS Database Encryption SDK adds to items when they're encrypted and signed.

The following example updates the table encryption configuration from **Step 1**. You must update the plaintext override with `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`. This policy continues to read plaintext items, but also reads and writes encrypted items. Create a new AWS SDK DynamoDB client using the updated `TableEncryptionConfigs`.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

Step 3: Only read encrypted and signed items

After you have encrypted and signed all of your items, update the plaintext override in your table encryption configuration to only allow the client to read and write encrypted and signed items. After you deploy the following change, the client will encrypt and sign new items based on the attribute actions you configured in **Step 1**. The client will only be able read encrypted and signed items.

The following example updates the table encryption configuration from **Step 2**. You can either update the plaintext override with `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT` or remove the plaintext policy from your configuration. The client only reads and writes encrypted and signed items by default. Create a new AWS SDK DynamoDB client using the updated `TableEncryptionConfigs`.

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: you can also remove the plaintext policy from your configuration
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

Rust

This topic explains how to install and use version 1.*x* of the Rust client-side encryption library for DynamoDB. For details about programming with the AWS Database Encryption SDK for DynamoDB, see the Rust examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

All programming language implementations of the AWS Database Encryption SDK for DynamoDB are interoperable.

Topics

- Prerequisites
- Installation
- Using the Rust client-side encryption library for DynamoDB

Prerequisites

Before you install the Rust client-side encryption library for DynamoDB, be sure you have the following prerequisites.

Install Rust and Cargo

Install the current stable release of Rust using rustup.

For more information on downloading and installing rustup, see the installation procedures in The Cargo Book.

Installation

The Rust client-side encryption library for DynamoDB is available as the aws-db-esdk crate on Crates.io. For details about installing and building the library, see the README.md file in the aws-database-encryption-sdk-dynamodb GitHub repository.

Manually

To install the Rust client-side encryption library for DynamoDB, clone or download the aws-database-encryption-sdk-dynamodb GitHub repository.

To install the latest version

Run the following Cargo command in your project directory:

```
cargo add aws-db-esdk
```

Or add the following line to your Cargo.toml:

```
aws-db-esdk = "<version>"
```

Using the Rust client-side encryption library for DynamoDB

This topic explains some of the functions and helper classes in version 1.*x* of the Rust client-side encryption library for DynamoDB.

For details about programming with the Rust client-side encryption library for DynamoDB, see the Rust examples in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Topics

- Item encryptors
- Attribute actions in the AWS Database Encryption SDK for DynamoDB
- Encryption configuration in the AWS Database Encryption SDK for DynamoDB
- Updating items with the AWS Database Encryption SDK

Item encryptors

At its core, the AWS Database Encryption SDK for DynamoDB is an item encryptor. You can use version 1.*x* of the Rust client-side encryption library for DynamoDB to encrypt, sign, verify, and decrypt your DynamoDB table items in the following ways.

The low-level AWS Database Encryption SDK for DynamoDB API

You can use your table encryption configuration to construct a DynamoDB client that automatically encrypts and signs items client-side with your DynamoDB `PutItem` requests.

You must use the low-level AWS Database Encryption SDK for DynamoDB API to use searchable encryption.

For an example demonstrating how to use the low-level AWS Database Encryption SDK for DynamoDB API, see basic_get_put_example.rs in the aws-database-encryption-sdk-dynamodb repository on GitHub.

The lower-level `DynamoDbItemEncryptor`

The lower-level `DynamoDbItemEncryptor` directly encrypts and signs or decrypts and verifies your table items without calling DynamoDB. It does not make DynamoDB `PutItem` or `GetItem` requests. For example, you can use the lower-level `DynamoDbItemEncryptor` to directly decrypt and verify a DynamoDB item that you have already retrieved.

The lower-level `DynamoDbItemEncryptor` does not support searchable encryption.

For an example demonstrating how to use the lower-level `DynamoDbItemEncryptor`, see item_encrypt_decrypt.rs in the aws-database-encryption-sdk-dynamodb repository on GitHub.

Attribute actions in the AWS Database Encryption SDK for DynamoDB

Attribute actions determine which attribute values are encrypted and signed, which are only signed, which are signed and included in the encryption context, and which are ignored.

To specify attribute actions with the Rust client, manually define attribute actions using an object model. Specify your attribute actions by creating a `HashMap` object in which the name-value pairs represent attribute names and the specified actions.

Specify `ENCRYPT_AND_SIGN` to encrypt and sign an attribute. Specify `SIGN_ONLY` to sign, but not encrypt, an attribute. Specify `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` to sign an attribute and include it in the encryption context. You cannot encrypt an attribute without also signing it. Specify `DO_NOTHING` to ignore an attribute.

The partition and sort attributes must be either `SIGN_ONLY` or `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. If you define any attributes as `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, then the partition and sort attributes must also be `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> After you define your attribute actions, you must define which attributes are excluded from the signatures. To make it easier to add new unsigned attributes in the future, we recommend choosing a distinct prefix (such as "`:`") to identify your unsigned attributes. Include this prefix in the attribute name for all attributes marked `DO_NOTHING` as you define your DynamoDB schema and attribute actions.

The following object model demonstrates how to specify `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, and `DO_NOTHING` attribute actions with the Rust client. This example uses the prefix "`:`" to identify `DO_NOTHING` attributes.

```
let attribute_actions_on_encrypt = HashMap::from([
    ("partition_key".to_string(), CryptoAction::SignOnly),
    ("sort_key".to_string(), CryptoAction::SignOnly),
    ("attribute1".to_string(), CryptoAction::EncryptAndSign),
    ("attribute2".to_string(), CryptoAction::SignOnly),
    (":attribute3".to_string(), CryptoAction::DoNothing),
]);
```

Encryption configuration in the AWS Database Encryption SDK for DynamoDB

When you use the AWS Database Encryption SDK, you must explicitly define an encryption configuration for your DynamoDB table. The values required in your encryption configuration depend on whether you defined your attribute actions manually or with an annotated data class.

The following snippet defines a DynamoDB table encryption configuration using the low-level AWS Database Encryption SDK for DynamoDB API and allowed unsigned attributes defined by a distinct prefix.

```
let table_config = DynamoDbTableEncryptionConfig::builder()
    .logical_table_name(ddb_table_name)
    .partition_key_name("partition_key")
    .sort_key_name("sort_key")
    .attribute_actions_on_encrypt(attribute_actions_on_encrypt)
    .keyring(kms_keyring)
    .allowed_unsigned_attribute_prefix(UNSIGNED_ATTR_PREFIX)
    // Specifying an algorithm suite is optional
    .algorithm_suite_id(
        DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
    )
    .build()?;

let table_configs = DynamoDbTablesEncryptionConfig::builder()
    .table_encryption_configs(HashMap::from([(ddb_table_name.to_string(),
 table_config)]))
    .build()?;
```

Logical table name

A logical table name for your DynamoDB table.

The logical table name is cryptographically bound to all data stored in the table to simplify DynamoDB restore operations. We strongly recommend specifying your DynamoDB table name as the logical table name when you first define your encryption configuration. You must always specify the same logical table name. For decryption to succeed, the logical table name must match the name specified on encryption. In the event that your DynamoDB table name changes after restoring your DynamoDB table from a backup, the logical table name ensures that the decrypt operation still recognizes the table.

Allowed unsigned attributes

The attributes marked `DO_NOTHING` in your attribute actions.

The allowed unsigned attributes tell the client which attributes are excluded from the signatures. The client assumes that all other attributes are included in the signature. Then, when decrypting a record, the client determines which attributes it needs to verify and which to ignore from the allowed unsigned attributes you specified. You cannot remove an attribute from your allowed unsigned attributes.

You can define the allowed unsigned attributes explicitly by creating an array that lists all of your `DO_NOTHING` attributes. You can also specify a distinct prefix when naming your `DO_NOTHING` attributes and use the prefix to tell the client which attributes are unsigned. We strongly recommend specifying a distinct prefix because it simplifies the process of adding a new `DO_NOTHING` attribute in the future. For more information, see Updating your data model.

If you do not specify a prefix for all `DO_NOTHING` attributes, you can configure an `allowedUnsignedAttributes` array that explicitly lists all of the attributes that the client should expect to be unsigned when it encounters them on decryption. You should only explicitly define your allowed unsigned attributes if absolutely necessary.

Search Configuration (Optional)

The `SearchConfig` defines the beacon version.

The `SearchConfig` must be specified to use searchable encryption or signed beacons.

Algorithm Suite (Optional)

The `algorithmSuiteId` defines which algorithm suite the AWS Database Encryption SDK uses.

Unless you explicitly specify an alternative algorithm suite, the AWS Database Encryption SDK uses the default algorithm suite. The default algorithm suite uses the AES-GCM algorithm with key derivation, digital signatures, and key commitment. Although the default algorithm suite is likely to be suitable for most applications, you can choose an alternate algorithm suite. For example, some trust models would be satisfied by an algorithm suite without digital signatures. For information about the algorithm suites that the AWS Database Encryption SDK supports, see Supported algorithm suites in the AWS Database Encryption SDK.

To select the AES-GCM algorithm suite without ECDSA digital signatures, include the following snippet in your table encryption configuration.

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

Updating items with the AWS Database Encryption SDK

The AWS Database Encryption SDK does not support ddb:UpdateItem for items that include encrypted or signed attributes. To update an encrypted or signed attribute, you must use ddb:PutItem. When you specify the same primary key as an existing item in your `PutItem` request, the new item completely replaces the existing item.

Legacy DynamoDB Encryption Client

On June 9, 2023, our client-side encryption library was renamed to AWS Database Encryption SDK. The AWS Database Encryption SDK continues to support legacy DynamoDB Encryption Client versions. For more information on the different parts of the client-side encryption library that changed with the rename, see Amazon DynamoDB Encryption Client rename.

To migrate to the latest version of the Java client-side encryption library for DynamoDB, see Migrate to version 3.x.

Topics

AWS Database Encryption SDK for DynamoDB version support

The topics in the Legacy chapter provide information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python.

The following table lists the languages and versions that support client-side encryption in Amazon DynamoDB.

Programming language	Version	SDK major version life-cycle phase
Java	Versions 1.*x*	End-of-Support phase, effective July 2022
Java	Versions 2.*x*	General Availability (GA)
Java	Version 3.*x*	General Availability (GA)
Python	Versions 1.*x*	End-of-Support phase, effective July 2022
Python	Versions 2.*x*	End-of-Support phase, effective July 2022
Python	Versions 3.*x*	General Availability (GA)

How the DynamoDB Encryption Client works

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

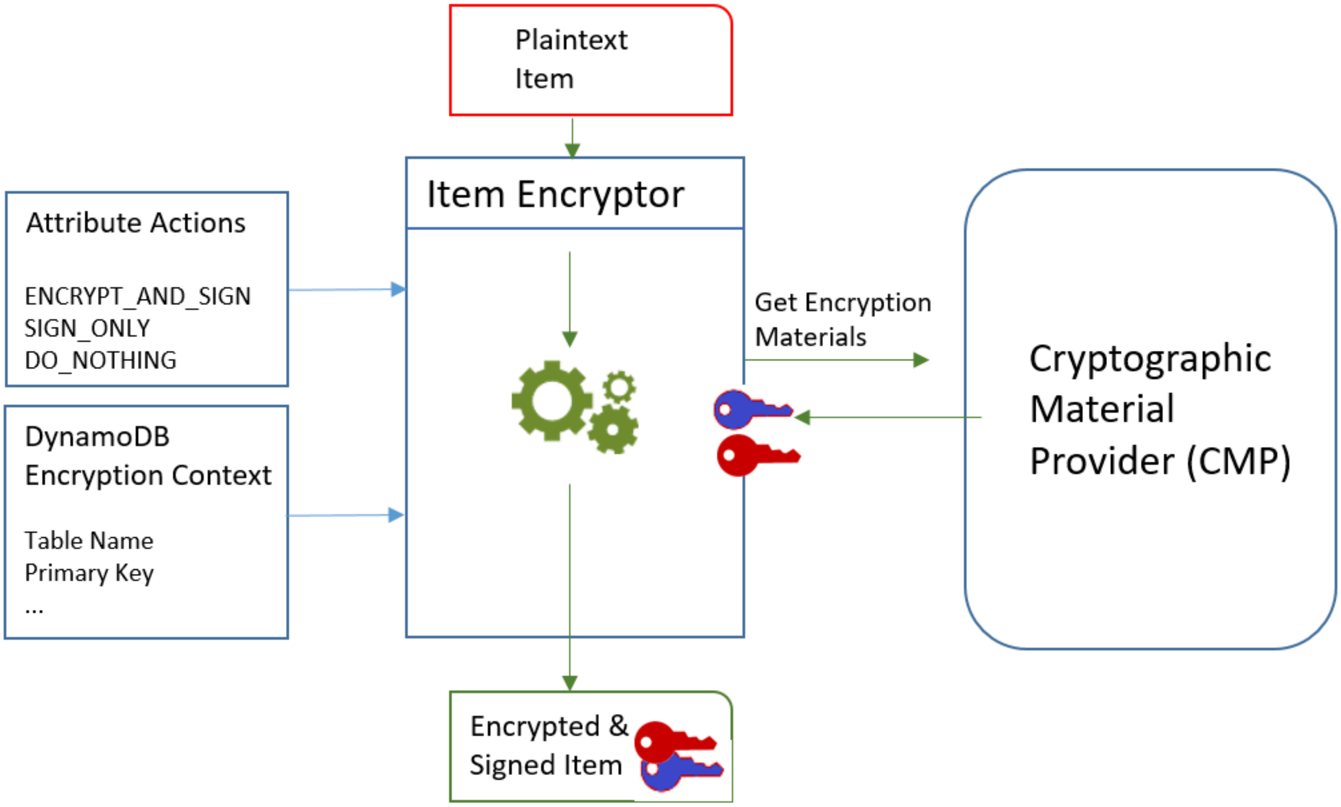
The DynamoDB Encryption Client is designed specifically to protect the data that you store in DynamoDB. The libraries include secure implementations that you can extend or use unchanged.

And, most elements are represented by abstract elements so you can create and use compatible custom components.

Encrypting and signing table items

At the core of the DynamoDB Encryption Client is an *item encryptor* that encrypts, signs, verifies, and decrypts table items. It takes in information about your table items and instructions about which items to encrypt and sign. It gets the encryption materials, and instructions on how to use them, from a cryptographic material provider that you select and configure.

The following diagram shows a high-level view of this process.



To encrypt and sign a table item, the DynamoDB Encryption Client needs:

- **Information about the table.** It gets information about the table from a DynamoDB encryption context that you supply. Some helpers get the required information from DynamoDB and create the DynamoDB encryption context for you.

Note

The *DynamoDB encryption context* in the DynamoDB Encryption Client is not related to the *encryption context* in AWS Key Management Service (AWS KMS) and the AWS Encryption SDK.

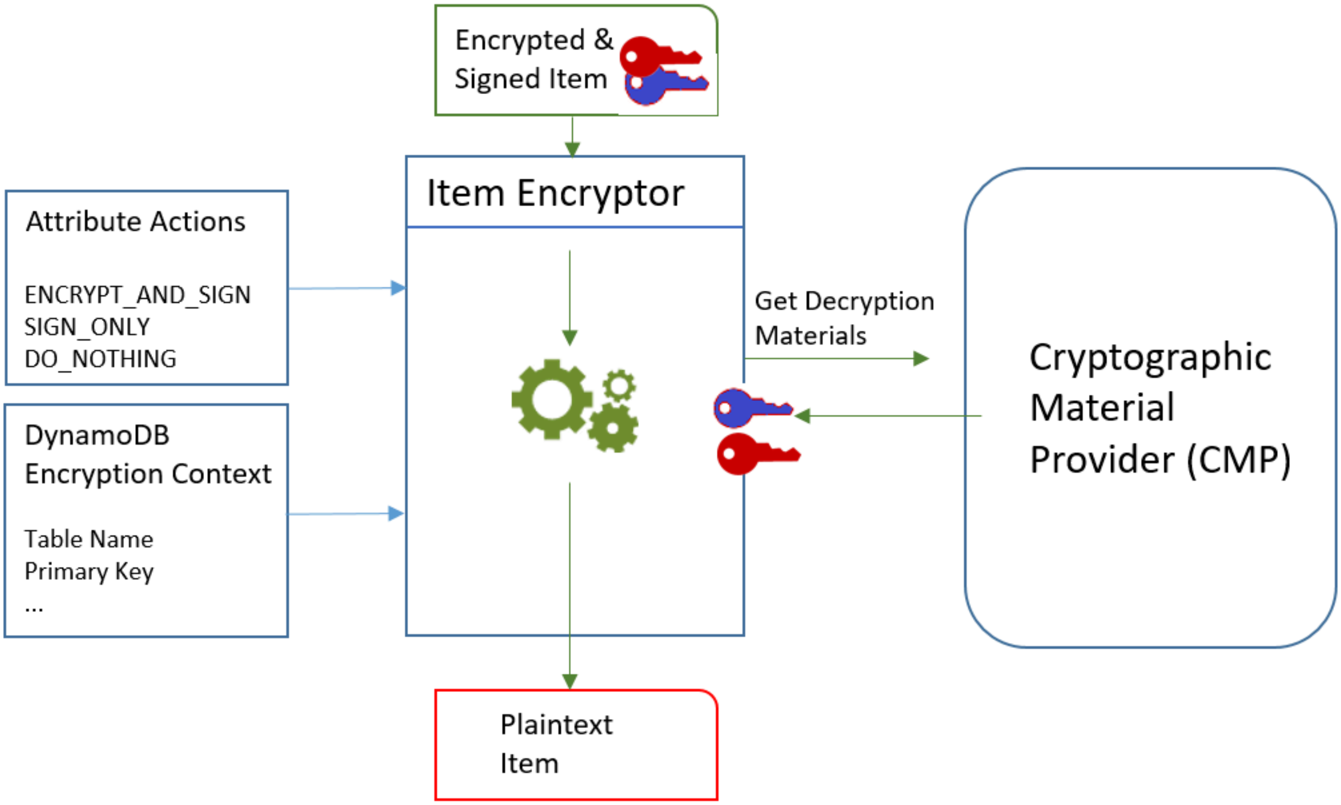
- **Which attributes to encrypt and sign.** It gets this information from the attribute actions that you supply.
- **Encryption materials, including encryption and signing keys.** It gets these from a cryptographic materials provider (CMP) that you select and configure.
- **Instructions for encrypting and signing the item**. The CMP adds instructions for using the encryption materials, including encryption and signing algorithms, to the actual material description.

The item encryptor uses all of these elements to encrypt and sign the item. The item encryptor also adds two attributes to the item: a material description attribute that contains the encryption and signing instructions (the actual material description), and an attribute that contains the signature. You can interact with the item encryptor directly, or use helper features that interact with the item encryptor for you to implement secure default behavior.

The result is a DynamoDB item containing encrypted and signed data.

Verifying and decrypting table items

These components also work together to verify and decrypt your item, as shown in the following diagram.



To verify and decrypt an item, the DynamoDB Encryption Client needs the same components, components with the same configuration, or components especially designed for decrypting the items, as follows:

- **Information about the table** from the DynamoDB encryption context.
- **Which attributes to verify and decrypt.** It gets these from the attribute actions.
- **Decryption materials, including verification and decryption keys**, from the cryptographic materials provider (CMP) that you select and configure.

  The encrypted item doesn't include any record of the CMP that was used to encrypt it. You must supply the same CMP, a CMP with the same configuration, or a CMP that is designed to decrypt items.
- **Information about how the item was encrypted and signed**, including the encryption and signing algorithms. The client gets these from the material description attribute in the item.

The item encryptor uses all of these elements to verify and decrypt the item. It also removes the material description and signature attributes. The result is a plaintext DynamoDB item.

Amazon DynamoDB Encryption Client concepts

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

This topic explains the concepts and terminology used in the Amazon DynamoDB Encryption Client.

To learn how the components of the DynamoDB Encryption Client interact, see How the DynamoDB Encryption Client works.

Topics

Cryptographic materials provider (CMP)

When implementing the DynamoDB Encryption Client, one of your first tasks is to select a cryptographic materials provider (CMP) (also known as an *encryption materials provider*). Your choice determines much of the rest of the implementation.

A *cryptographic materials provider* (CMP) collects, assembles, and returns the cryptographic materials that the item encryptor uses to encrypt and sign your table items. The CMP determines the encryption algorithms to use and how to generate and protect encryption and signing keys.

The CMP interacts with the item encryptor. The item encryptor requests encryption or decryption materials from the CMP, and the CMP returns them to the item encryptor. Then, the item encryptor uses the cryptographic materials to encrypt and sign, or verify and decrypt, the item.

You specify the CMP when you configure the client. You can create a compatible custom CMP, or use one of the many CMPs in the library. Most CMPs are available for multiple programming languages.

Item encryptors

The *item encryptor* is a lower-level component that performs cryptographic operations for the DynamoDB Encryption Client. It requests cryptographic materials from a cryptographic materials provider (CMP), then uses the materials that the CMP returns to encrypt and sign, or verify and decrypt, your table item.

You can interact with the item encryptor directly or use the helpers that your library provides. For example, the DynamoDB Encryption Client for Java includes an `AttributeEncryptor` helper class that you can use with the `DynamoDBMapper`, instead of interacting directly with the `DynamoDBEncryptor` item encryptor. The Python library includes `EncryptedTable`, `EncryptedClient`, and `EncryptedResource` helper classes that interact with the item encryptor for you.

Attribute actions

Attribute actions tell the item encryptor which actions to perform on each attribute of the item.

The attribute action values can be one of the following:

- **Encrypt and sign** – Encrypt the attribute value. Include the attribute (name and value) in the item signature.
- **Sign only** – Include the attribute in the item signature.
- **Do nothing** – Do not encrypt or sign the attribute.

For any attribute that can store sensitive data, use **Encrypt and sign**. For primary key attributes (partition key and sort key), use **Sign only**. The material description attribute and the signature attribute are not signed or encrypted. You don't need to specify attribute actions for these attributes.

Choose your attribute actions carefully. When in doubt, use **Encrypt and sign**. Once you have used the DynamoDB Encryption Client to protect your table items, you cannot change the action for an attribute without risking a signature validation error. For details, see Changing your data model.

Warning

Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

If the DynamoDB encryption context identifies your primary key attributes, the client will throw an error if you try to encrypt them.

The technique that you use to specify the attribute actions is different for each programming language. It might also be specific to helper classes that you use.

For details, see the documentation for your programming language.

- Python
- Java

Material description

The *material description* for an encrypted table item consists of information, such as encryption algorithms, about how the table item is encrypted and signed. The cryptographic materials provider (CMP) records the material description as it assembles the cryptographic materials for encryption and signing. Later, when it needs to assemble cryptographic materials to verify and decrypt the item, it uses the material description as its guide.

In the DynamoDB Encryption Client, the material description refers to three related elements:

Requested material description

Some cryptographic materials providers (CMPs) let you specify advanced options, such as an encryption algorithm. To indicate your choices, you add name-value pairs to the material description property of the DynamoDB encryption context in your request to encrypt a table item. This element is known as the *requested material description*. The valid values in the requested material description are defined by the CMP that you choose.

Note

Because the material description can override secure default values, we recommend that you omit the requested material description unless you have a compelling reason to use it.

Actual material description

The material description that the cryptographic materials providers (CMPs) return is known as the *actual material description*. It describes the actual values that the CMP used when it assembled the cryptographic materials. It usually consists of the requested material description, if any, with additions and changes.

Material description attribute

The client saves the actual material description in the *material description attribute* of the encrypted item. The material description attribute name is `amzn-ddb-map-desc` and its value is the actual material description. The client uses the values in the material description attribute to verify and decrypt the item.

DynamoDB encryption context

The *DynamoDB encryption context* supplies information about the table and item to the cryptographic materials provider (CMP). In advanced implementations, the DynamoDB encryption context can include a requested material description.

When you encrypt table items, the DynamoDB encryption context is cryptographically bound to the encrypted attribute values. When you decrypt, if the DynamoDB encryption context is not an exact, case-sensitive match for the DynamoDB encryption context that was used to encrypt, the decrypt operation fails. If you interact with the item encryptor directly, you must provide a DynamoDB encryption context when you call an encrypt or decrypt method. Most helpers create the DynamoDB encryption context for you.

Note

The *DynamoDB encryption context* in the DynamoDB Encryption Client is not related to the *encryption context* in AWS Key Management Service (AWS KMS) and the AWS Encryption SDK.

The DynamoDB encryption context can include the following fields. All fields and values are optional.

- Table name
- Partition key name
- Sort key name
- Attribute name-value pairs
- Requested material description

Provider store

A *provider store* is a component that returns cryptographic materials providers (CMPs). The provider store can create the CMPs or get them from another source, such as another provider store. The provider store saves versions of the CMPs that it creates in persistent storage in which each stored CMP is identified by the material name of the requester and version number.

The Most Recent Provider in the DynamoDB Encryption Client gets its CMPs from a provider store, but you can use the provider store to supply CMPs to any component. Each Most Recent Provider is associated with one provider store, but a provider store can supply CMPs to many requesters across multiple hosts.

The provider store creates new versions of CMPs on demand, and returns new and existing versions. It also returns the latest version number for a given material name. This lets the requester know when the provider store has a new version of its CMP that it can request.

The DynamoDB Encryption Client includes a MetaStore, which is a provider store that creates Wrapped CMPs with keys that are stored in DynamoDB and encrypted by using an internal DynamoDB Encryption Client.

Learn more:

- Provider store: Java, Python
- MetaStore: Java, Python

Cryptographic materials provider

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

One of the most important decisions you make when using the DynamoDB Encryption Client is selecting a cryptographic materials provider (CMP). The CMP assembles and returns cryptographic materials to the item encryptor. It also determines how encryption and signing keys are generated, whether new key materials are generated for each item or are reused, and the encryption and signing algorithms that are used.

You can choose a CMP from the implementations provided in the DynamoDB Encryption Client libraries or build a compatible custom CMP. Your CMP choice might also depend on the programming language that you use.

This topic describes the most common CMPs and offers some advice to help you choose the best one for your application.

Direct KMS Materials Provider

The Direct KMS Materials Provider protects your table items under an AWS KMS key that never leaves AWS Key Management Service (AWS KMS) unencrypted. Your application doesn't have to generate or manage any cryptographic materials. Because it uses the AWS KMS key to generate unique encryption and signing keys for each item, this provider calls AWS KMS every time it encrypts or decrypts an item.

If you use AWS KMS and one AWS KMS call per transaction is practical for your application, this provider is a good choice.

For details, see Direct KMS Materials Provider.

Wrapped Materials Provider (Wrapped CMP)

The Wrapped Materials Provider (Wrapped CMP) lets you generate and manage your wrapping and signing keys outside of the DynamoDB Encryption Client.

The Wrapped CMP generates a unique encryption key for each item. Then it uses wrapping (or unwrapping) and signing keys that you supply. As such, you determine how the wrapping and signing keys are generated and whether they are unique to each item or are reused. The Wrapped CMP is a secure alternative to the Direct KMS Provider for applications that don't use AWS KMS and can safely manage cryptographic materials.

For details, see Wrapped Materials Provider.

Most Recent Provider

The *Most Recent Provider* is a cryptographic materials provider (CMP) that is designed to work with a provider store. It gets CMPs from the provider store, and gets the cryptographic materials that it returns from the CMPs. The Most Recent Provider typically uses each CMP to satisfy multiple requests for cryptographic materials, but you can use the features of the provider store to control the extent to which materials are reused, determine how often its CMP is rotated, and even change the type of CMP that is used without changing the Most Recent Provider.

You can use the Most Recent Provider with any compatible provider store. The DynamoDB Encryption Client includes a MetaStore, which is a provider store that returns Wrapped CMPs.

The Most Recent Provider is a good choice for applications that need to minimize calls to their cryptographic source, and applications that can reuse some cryptographic materials without violating their security requirements. For example, it allows you to protect your cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) without calling AWS KMS every time you encrypt or decrypt an item.

For details, see Most Recent Provider.

Static Materials Provider

The Static Materials Provider is designed for testing, proof-of-concept demonstrations, and legacy compatibility. It doesn't generate any unique cryptographic materials for each item. It returns the same encryption and signing keys that you supply, and those keys are used directly to encrypt, decrypt, and sign your table items.

> **Note**
>
> The Asymmetric Static Provider in the Java library is not a static provider. It just supplies alternate constructors for the Wrapped CMP. It is safe for production use, but you should use the Wrapped CMP directly whenever possible.

Topics

- Direct KMS Materials Provider
- Wrapped Materials Provider
- Most Recent Provider
- Static Materials Provider

Direct KMS Materials Provider

> **Note**
>
> Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The *Direct KMS Materials Provider* (Direct KMS Provider) protects your table items under an AWS KMS key that never leaves AWS Key Management Service (AWS KMS) unencrypted. This cryptographic materials provider returns a unique encryption key and signing key for every table item. To do so, it calls AWS KMS every time you encrypt or decrypt an item.

If you're processing DynamoDB items at a high frequency and large scale, you might exceed the AWS KMS requests-per-second limits, causing processing delays. If you need to exceed a limit, create a case in the AWS Support Center. You might also consider using a cryptographic materials provider with limited key reuse, such as the Most Recent Provider.

To use the Direct KMS Provider, the caller must have an AWS account, at least one AWS KMS key, and permission to call the GenerateDataKey and Decrypt operations on the AWS KMS key. The AWS KMS key must be a symmetric encryption key; the DynamoDB Encryption Client does not support asymmetric encryption. If you are using a DynamoDB global table, you might want to specify an AWS KMS multi-Region key. For details, see How to use it.

> **Note**
>
> When you use the Direct KMS Provider, the names and values of your primary key attributes appear in plaintext in the AWS KMS encryption context and AWS CloudTrail logs

of related AWS KMS operations. However, the DynamoDB Encryption Client never exposes the plaintext of any encrypted attribute values.

The Direct KMS Provider is one of several cryptographic materials providers (CMPs) that the DynamoDB Encryption Client supports. For information about the other CMPs, see Cryptographic materials provider.

For example code, see:

- Java: AwsKmsEncryptedItem
- Python: aws-kms-encrypted-table, aws-kms-encrypted-item

Topics

- How to use it
- How it works

How to use it

To create a Direct KMS Provider, use the key ID parameter to specify a symmetric encryption KMS key in your account. The value of the key ID parameter can be the key ID, key ARN, alias name, or alias ARN of the AWS KMS key. For details about the key identifiers, see Key identifiers in the *AWS Key Management Service Developer Guide*.

The Direct KMS Provider requires a symmetric encryption KMS key. You cannot use an asymmetric KMS key. However, you can use a multi-Region KMS key, a KMS key with imported key material, or a KMS key in a custom key store. You must have kms:GenerateDataKey and kms:Decrypt permission on the KMS key. As such, you must use a customer managed key, not an AWS managed or AWS owned KMS key.

The DynamoDB Encryption Client for Python determines the Region for calling AWS KMS from the Region in the key ID parameter value, if it includes one. Otherwise, it uses the Region in the AWS KMS client, if you specify one, or the Region that you configure in the AWS SDK for Python (Boto3). For information about Region selection in Python, see Configuration in the AWS SDK for Python (Boto3) API Reference.

The DynamoDB Encryption Client for Java determines the Region for calling AWS KMS from the Region in the AWS KMS client, if the client you specify includes a Region. Otherwise, it uses the

Region that you configure in the AWS SDK for Java. For information about Region selection in the AWS SDK for Java, see AWS Region selection in the AWS SDK for Java Developer Guide.

Java

```
// Replace the example key ARN and Region with valid values for your application
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Python

The following example uses the key ARN to specify the AWS KMS key. If your key identifier doesn't include an AWS Region, the DynamoDB Encryption Client gets the Region from the configured Botocore session, if there is one, or from Boto defaults.

```
# Replace the example key ID with a valid value
kms_key = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key)
```

If you are using Amazon DynamoDB global tables, we recommend that you encrypt your data under an AWS KMS multi-Region key. Multi-Region keys are AWS KMS keys in different AWS Regions that can be used interchangeably because they have the same key ID and key material. For details, see Using multi-Region keys in the *AWS Key Management Service Developer Guide*.

Note

If you are using the global tables version 2017.11.29, you must set attribute actions so the reserved replication fields are not encrypted or signed. For details, see Issues with older version global tables.

To use a multi-Region key with the DynamoDB Encryption Client, create a multi-Region key and replicate it into the Regions in which your application runs. Then configure the Direct KMS Provider to use the multi-Region key in the Region in which the DynamoDB Encryption Client calls AWS KMS.

The following example configures the DynamoDB Encryption Client to encrypt data in the US East (N. Virginia) (us-east-1) Region and decrypt it in the US West (Oregon) (us-west-2) Region using a multi-Region key.

Java

In this example, the DynamoDB Encryption Client gets the Region for calling AWS KMS from the Region in the AWS KMS client. The `keyArn` value identifies a multi-Region key in the same Region.

```
// Encrypt in us-east-1

// Replace the example key ARN and Region with valid values for your application
final String usEastKey = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-east-1'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usEastKey);
```

```
// Decrypt in us-west-2

// Replace the example key ARN and Region with valid values for your application
final String usWestKey = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usWestKey);
```

Python

In this example, the DynamoDB Encryption Client gets the Region for calling AWS KMS from the Region in the key ARN.

```
# Encrypt in us-east-1

# Replace the example key ID with a valid value
```

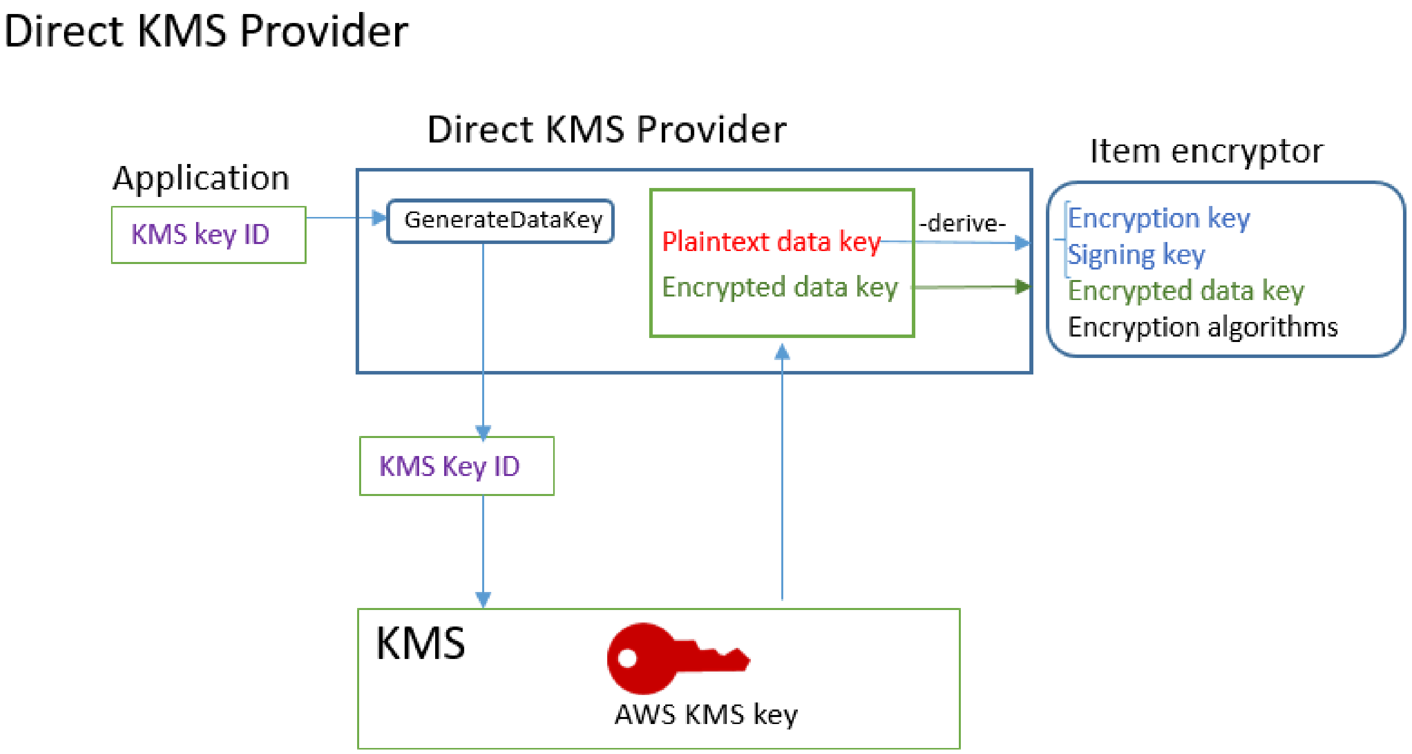
```
us_east_key = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_east_key)
```

```
# Decrypt in us-west-2

# Replace the example key ID with a valid value
us_west_key = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_west_key)
```

How it works

The Direct KMS Provider returns encryption and signing keys that are protected by an AWS KMS key that you specify, as shown in the following diagram.



- To generate encryption materials, the Direct KMS Provider asks AWS KMS to generate a unique data key for each item using an AWS KMS key that you specify. It derives encryption and signing keys for the item from the plaintext copy of the data key, and then returns the encryption and signing keys, along with the encrypted data key, which is stored in the material description attribute of the item.

The item encryptor uses the encryption and signing keys and removes them from memory as soon as possible. Only the encrypted copy of the data key from which they were derived is saved in the encrypted item.

- To generate decryption materials, the Direct KMS Provider asks AWS KMS to decrypt the encrypted data key. Then, it derives verification and signing keys from the plaintext data key, and returns them to the item encryptor.

  The item encryptor verifies the item and, if verification succeeds, decrypts the encrypted values. Then, it removes the keys from memory as soon as possible.

Get encryption materials

This section describes in detail the inputs, outputs, and processing of the Direct KMS Provider when it receives a request for encryption materials from the item encryptor.

Input (from the application)

- The key ID of an AWS KMS key.

Input (from the item encryptor)

- DynamoDB encryption context

Output (to the item encryptor)

- Encryption key (plaintext)
- Signing key
- In actual material description: These values are saved in the material description attribute that the client adds to the item.
  - amzn-ddb-env-key: Base64-encoded data key encrypted by the AWS KMS key
  - amzn-ddb-env-alg: Encryption algorithm, by default AES/256
  - amzn-ddb-sig-alg: Signing algorithm, by default, HmacSHA256/256
  - amzn-ddb-wrap-alg: kms

Processing

1. The Direct KMS Provider sends AWS KMS a request to use the specified AWS KMS key to generate a unique data key for the item. The operation returns a plaintext key and a copy that is encrypted under the AWS KMS key. This is known as the *initial key material*.

   The request includes the following values in plaintext in AWS KMS encryption context. These non-secret values are cryptographically bound to the encrypted object, so the same encryption context is required on decrypt. You can use these values to identify the call to AWS KMS in AWS CloudTrail logs.

   - amzn-ddb-env-alg – Encryption algorithm, by default AES/256
   - amzn-ddb-sig-alg – Signing algorithm, by default HmacSHA256/256
   - (Optional) aws-kms-table – *`table name`*
   - (Optional) *`partition key name`* – *`partition key value`* (binary values are Base64-encoded)
   - (Optional) *`sort key name`* – *`sort key value`* (binary values are Base64-encoded)

   The Direct KMS Provider gets the values for the AWS KMS encryption context from the DynamoDB encryption context for the item. If the DynamoDB encryption context doesn't include a value, such as the table name, that name-value pair is omitted from the AWS KMS encryption context.
2. The Direct KMS Provider derives a symmetric encryption key and a signing key from the data key. By default, it uses Secure Hash Algorithm (SHA) 256 and RFC5869 HMAC-based Key Derivation Function to derive a 256-bit AES symmetric encryption key and a 256-bit HMAC-SHA-256 signing key.
3. The Direct KMS Provider returns the output to the item encryptor.
4. The item encryptor uses the encryption key to encrypt the specified attributes and the signing key to sign them, using the algorithms specified in the actual material description. It removes the plaintext keys from memory as soon as possible.

Get decryption materials

This section describes in detail the inputs, outputs, and processing of the Direct KMS Provider when it receives a request for decryption materials from the item encryptor.

Input (from the application)

- The key ID of an AWS KMS key.

The value of the key ID can be the key ID, key ARN, alias name or alias ARN of the AWS KMS key. Any values that aren't included in the key ID, such as the Region, must be available in the AWS named profile. The key ARN provides all of the values that AWS KMS needs.

Input (from the item encryptor)

- A copy of the DynamoDB encryption context that contains the contents of the material description attribute.

Output (to the item encryptor)

- Encryption key (plaintext)
- Signing key

Processing

1. The Direct KMS Provider gets the encrypted data key from the material description attribute in the encrypted item.
2. It asks AWS KMS to use the specified AWS KMS key to decrypt the encrypted data key. The operation returns a plaintext key.

   This request must use the same AWS KMS encryption context that was used to generate and encrypt the data key.

   - aws-kms-table – *`table name`*
   - *`partition key name`* – *`partition key value`* (binary values are Base64-encoded)
   - (Optional) *`sort key name`* – *`sort key value`* (binary values are Base64-encoded)
   - amzn-ddb-env-alg – Encryption algorithm, by default AES/256
   - amzn-ddb-sig-alg – Signing algorithm, by default HmacSHA256/256
3. The Direct KMS Provider uses Secure Hash Algorithm (SHA) 256 and RFC5869 HMAC-based Key Derivation Function to derive a 256-bit AES symmetric encryption key and a 256-bit HMAC-SHA-256 signing key from the data key.
4. The Direct KMS Provider returns the output to the item encryptor.
5. The item encryptor uses the signing key to verify the item. If it succeeds, it uses the symmetric encryption key to decrypt the encrypted attribute values. These operations use the encryption

and signing algorithms specified in the actual material description. The item encryptor removes the plaintext keys from memory as soon as possible.

Wrapped Materials Provider

> **Note**
>
> Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The *Wrapped Materials Provider* (Wrapped CMP) lets you use wrapping and signing keys from any source with the DynamoDB Encryption Client. The Wrapped CMP does not depend on any AWS service. However, you must generate and manage your wrapping and signing keys outside of the client, including providing the correct keys to verify and decrypt the item.

The Wrapped CMP generates a unique item encryption key for each item. It wraps the item encryption key with the wrapping key that you provide and saves the wrapped item encryption key in the material description attribute of the item. Because you supply the wrapping and signing keys, you determine how the wrapping and signing keys are generated and whether they are unique to each item or are reused.

The Wrapped CMP is a secure implementation and a good choice for applications that can manage cryptographic materials.

The Wrapped CMP is one of several cryptographic materials providers (CMPs) that the DynamoDB Encryption Client supports. For information about the other CMPs, see Cryptographic materials provider.

For example code, see:

- Java: AsymmetricEncryptedItem
- Python: wrapped-rsa-encrypted-table, wrapped-symmetric-encrypted-table

Topics

- How to use it
- How it works

How to use it

To create a Wrapped CMP, specify a wrapping key (required on encrypt), an unwrapping key (required on decrypt), and a signing key. You must supply keys when you encrypt and decrypt items.

The wrapping, unwrapping, and signing keys can be symmetric keys or asymmetric key pairs.

Java

```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

final WrappedMaterialsProvider cmp =
    new WrappedMaterialsProvider(wrappingKeys.getPublic(),
                                 wrappingKeys.getPrivate(),
                                 signingKeys);
```

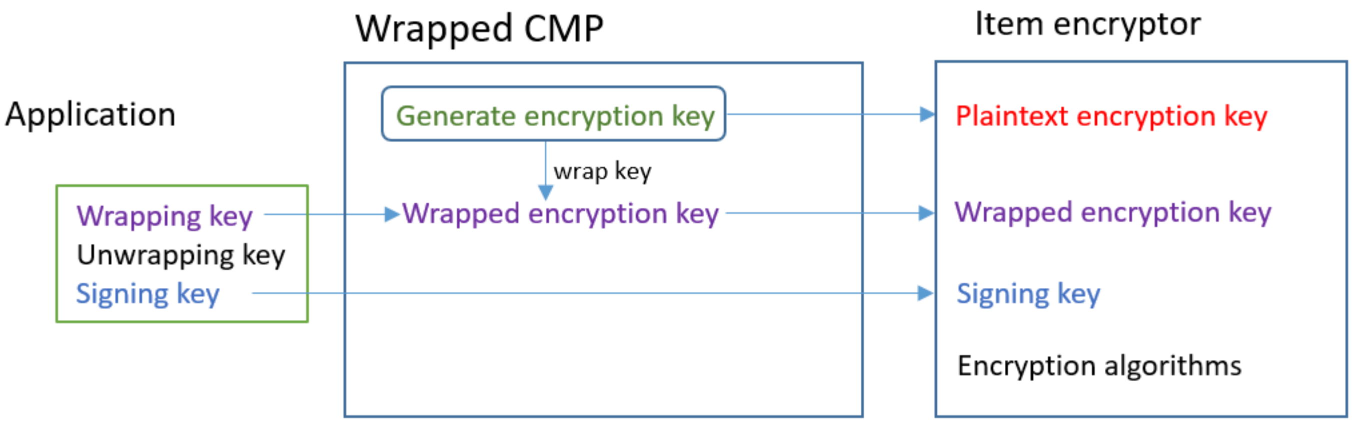
Python

```
# This example uses symmetric wrapping and signing keys
wrapping_key = ...
signing_key  = ...

wrapped_cmp = WrappedCryptographicMaterialsProvider(
    wrapping_key=wrapping_key,
    unwrapping_key=wrapping_key,
    signing_key=signing_key
)
```

How it works

The Wrapped CMP generates a new item encryption key for every item. It uses the wrapping, unwrapping, and signing keys that you provide, as shown in the following diagram.



Get encryption materials

This section describes in detail the inputs, outputs, and processing of the Wrapped Materials Provider (Wrapped CMP) when it receives a request for encryption materials.

Input (from application)

- Wrapping key: An Advanced Encryption Standard (AES) symmetric key, or an RSA public key. Required if any attribute values are encrypted. Otherwise, it is optional and ignored.
- Unwrapping key: Optional and ignored.
- Signing key

Input (from the item encryptor)

- DynamoDB encryption context

Output (to the item encryptor):

- Plaintext item encryption key
- Signing key (unchanged)
- Actual material description: These values are saved in the material description attribute that the client adds to the item.
  - `amzn-ddb-env-key`: Base64-encoded wrapped item encryption key
  - `amzn-ddb-env-alg`: Encryption algorithm used to encrypt the item. The default is AES-256-CBC.

- `amzn-ddb-wrap-alg`: The wrapping algorithm that the Wrapped CMP used to wrap the item encryption key. If the wrapping key is an AES key, the key is wrapped using unpadded `AES-Keywrap` as defined in RFC 3394. If the wrapping key is an RSA key, the key is encrypted by using RSA OAEP with MGF1 padding.

Processing

When you encrypt an item, you pass in a wrapping key and a signing key. An unwrapping key is optional and ignored.

1. The Wrapped CMP generates a unique symmetric item encryption key for the table item.
2. It uses the wrapping key that you specify to wrap the item encryption key. Then, it removes it from memory as soon as possible.
3. It returns the plaintext item encryption key, the signing key that you supplied, and an actual material description that includes the wrapped item encryption key, and the encryption and wrapping algorithms.
4. The item encryptor uses the plaintext encryption key to encrypt the item. It uses the signing key that you supplied to sign the item. Then, it removes the plaintext keys from memory as soon as possible. It copies the fields in the actual material description, including the wrapped encryption key (`amzn-ddb-env-key`), to the material description attribute of the item.

Get decryption materials

This section describes in detail the inputs, outputs, and processing of the Wrapped Materials Provider (Wrapped CMP) when it receives a request for decryption materials.

Input (from application)

- Wrapping key: Optional and ignored.
- Unwrapping key: The same Advanced Encryption Standard (AES) symmetric key or RSA private key that corresponds to the RSA public key used to encrypt. Required if any attribute values are encrypted. Otherwise, it is optional and ignored.
- Signing key

Input (from the item encryptor)

- A copy of the DynamoDB encryption context that contains the contents of the material description attribute.

Output (to the item encryptor)

- Plaintext item encryption key
- Signing key (unchanged)

Processing

When you decrypt an item, you pass in an unwrapping key and a signing key. A wrapping key is optional and ignored.

1. The Wrapped CMP gets the wrapped item encryption key from the material description attribute of the item.
2. It uses the unwrapping key and algorithm to unwrap the item encryption key.
3. It returns the plaintext item encryption key, the signing key, and encryption and signing algorithms to the item encryptor.
4. The item encryptor uses the signing key to verify the item. If it succeeds, it uses the item encryption key to decrypt the item. Then, it removes the plaintext keys from memory as soon as possible.

Most Recent Provider

> **Note**
>
> Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The *Most Recent Provider* is a cryptographic materials provider (CMP) that is designed to work with a provider store. It gets CMPs from the provider store, and gets the cryptographic materials that it returns from the CMPs. It typically uses each CMP to satisfy multiple requests for cryptographic materials. But you can use the features of its provider store to control the extent to which materials are reused, determine how often its CMP is rotated, and even change the type of CMP that it uses without changing the Most Recent Provider.

> ⓘ **Note**
>
> The code associated with the `MostRecentProvider` symbol for the Most Recent Provider might store cryptographic materials in memory for the lifetime of the process. It might allow a caller to use keys that they're no longer authorized to use.
> The `MostRecentProvider` symbol is deprecated in older supported versions of the DynamoDB Encryption Client and removed from version 2.0.0. It is replaced by the `CachingMostRecentProvider` symbol. For details, see Updates to the Most Recent Provider.

The Most Recent Provider is a good choice for applications that need to minimize calls to the provider store and its cryptographic source, and applications that can reuse some cryptographic materials without violating their security requirements. For example, It allows you to protect your cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) without calling AWS KMS every time you encrypt or decrypt an item.

The provider store that you choose determines the type of CMPs that the Most Recent Provider uses and how often it gets a new CMP. You can use any compatible provider store with the Most Recent Provider, including custom provider stores that you design.

The DynamoDB Encryption Client includes a *MetaStore* that creates and returns Wrapped Materials Providers (Wrapped CMPs). The MetaStore saves multiple versions of the Wrapped CMPs that it generates in an internal DynamoDB table and protects them with client-side encryption by an internal instance of the DynamoDB Encryption Client.

You can configure the MetaStore to use any type of internal CMP to protect the materials in the table, including a Direct KMS Provider that generates cryptographic materials protected by your AWS KMS key, a Wrapped CMP that uses wrapping and signing keys that you supply, or a compatible custom CMP that you design.

For example code, see:

- Java: MostRecentEncryptedItem
- Python: most_recent_provider_encrypted_table

Topics

- How to use it
- How it works
- Updates to the Most Recent Provider

How to use it

To create a Most Recent Provider, you need to create and configure a provider store, and then create a Most Recent Provider that uses the provider store.

The following examples show how to create a Most Recent Provider that uses a MetaStore and protects the versions in its internal DynamoDB table with cryptographic materials from a Direct KMS Provider. These examples use the `CachingMostRecentProvider` symbol.

Each Most Recent Provider has a name that identifies its CMPs in the MetaStore table, a time-to-live (TTL) setting, and a cache size setting that determines how many entries the cache can hold. These examples set the cache size to 1000 entries and a TTL of 60 seconds.

Java

```
// Set the name for MetaStore's internal table
final String keyTableName = 'metaStoreTable'

// Set the Region and AWS KMS key
final String region = 'us-west-2'
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

// Set the TTL and cache size
final long ttlInMillis = 60000;
final long cacheSize = 1000;

// Name that identifies the MetaStore's CMPs in the provider store
final String materialName = 'testMRP'

// Create an internal DynamoDB client for the MetaStore
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

// Create an internal Direct KMS Provider for the MetaStore
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider kmsProv = new DirectKmsMaterialProvider(kms,
 keyArn);

// Create an item encryptor for the MetaStore,
// including the Direct KMS Provider
final DynamoDBEncryptor keyEncryptor = DynamoDBEncryptor.getInstance(kmsProv);

// Create the MetaStore
final MetaStore metaStore = new MetaStore(ddb, keyTableName, keyEncryptor);

//Create the Most Recent Provider
final CachingMostRecentProvider cmp = new CachingMostRecentProvider(metaStore,
 materialName, ttlInMillis, cacheSize);
```

Python

```
# Designate an AWS KMS key
kms_key_id = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

# Set the name for MetaStore's internal table
meta_table_name = 'metaStoreTable'

# Name that identifies the MetaStore's CMPs in the provider store
material_name = 'testMRP'

# Create an internal DynamoDB table resource for the MetaStore
meta_table = boto3.resource('dynamodb').Table(meta_table_name)

# Create an internal Direct KMS Provider for the MetaStore
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)

# Create the MetaStore with the Direct KMS Provider
meta_store = MetaStore(
    table=meta_table,
    materials_provider=kms_cmp
)

# Create a Most Recent Provider using the MetaStore
#    Sets the TTL (in seconds) and cache size (# entries)
most_recent_cmp = MostRecentProvider(
    provider_store=meta_store,
    material_name=material_name,
    version_ttl=60.0,
    cache_size=1000
)
```

How it works

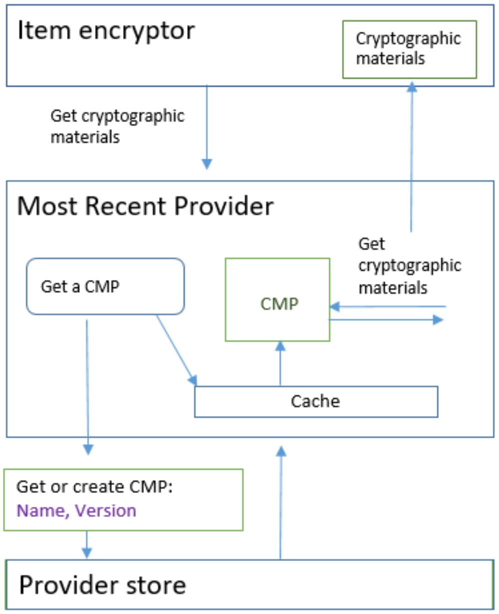
The Most Recent Provider gets CMPs from a provider store. Then, it uses the CMP to generate the cryptographic materials that it returns to the item encryptor.

About the Most Recent Provider

The Most Recent Provider gets a cryptographic materials provider (CMP) from a provider store. Then, it uses the CMP to generate the cryptographic materials it returns. Each Most Recent Provider is associated with one provider store, but a provider store can supply CMPs to multiple providers across multiple hosts.

The Most Recent Provider can work with any compatible CMP from any provider store. It requests encryption or decryption materials from the CMP and returns the output to the item encryptor. It does not perform any cryptographic operations.

To request a CMP from its provider store, the Most Recent Provider supplies its material name and the version of an existing CMP it wants to use. For encryption materials, the Most Recent Provider always requests the maximum ("most recent") version. For decryption materials, it requests the version of the CMP that was used to create the encryption materials, as shown in the following diagram.



The Most Recent Provider saves versions of the CMPs that the provider store returns in a local Least Recently Used (LRU) cache in memory. The cache enables the Most Recent Provider to get the CMPs that it needs without calling the provider store for every item. You can clear the cache on demand.

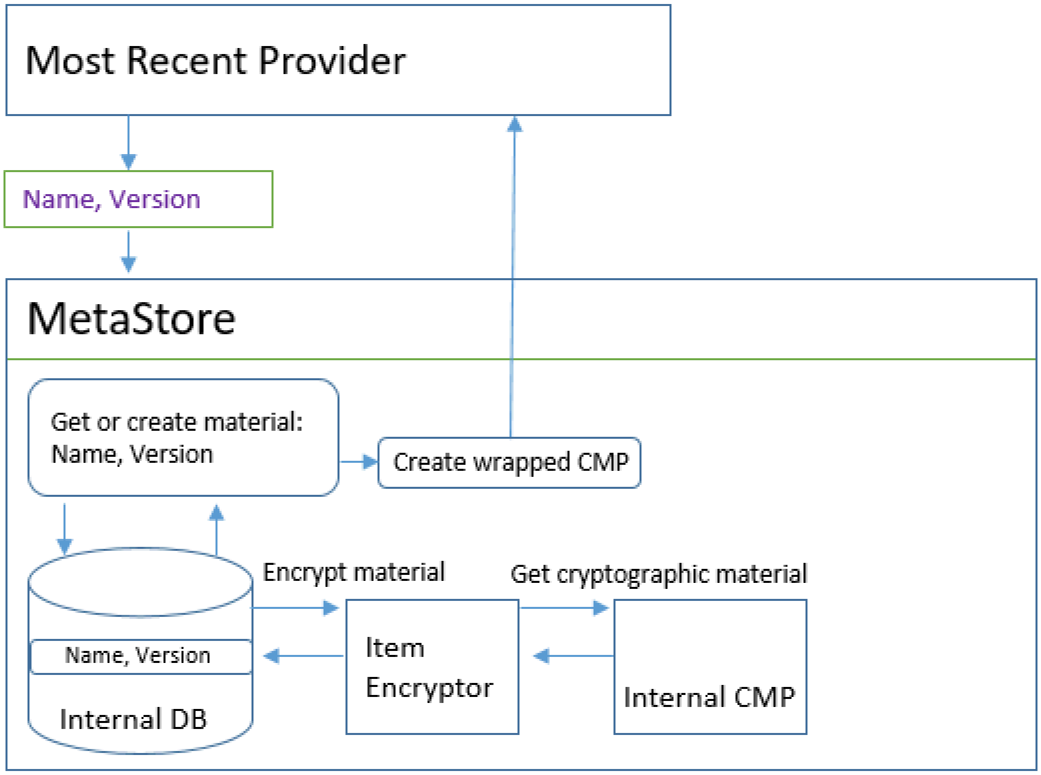
The Most Recent Provider uses a configurable time-to-live value that you can adjust based on the characteristics of your application.

About the MetaStore

You can use a Most Recent Provider with any provider store, including a compatible custom provider store. The DynamoDB Encryption Client includes a MetaStore, a secure implementation that you can configure and customize.

A *MetaStore* is a provider store that creates and returns Wrapped CMPs that are configured with the wrapping key, unwrapping key, and signing key that Wrapped CMPs require. A MetaStore is a secure option for a Most Recent Provider because Wrapped CMPs always generate unique item encryption keys for every item. Only the wrapping key that protects the item encryption key and the signing keys are reused.

The following diagram shows the components of the MetaStore and how it interacts with the Most Recent Provider.



The MetaStore generates the Wrapped CMPs, and then stores them (in encrypted form) in an internal DynamoDB table. The partition key is the name of the Most Recent Provider material; the sort key its version number. The materials in the table are protected by an internal DynamoDB Encryption Client, including an item encryptor and internal cryptographic materials provider (CMP).

You can use any type of internal CMP in your MetaStore, including the a Direct KMS Provider, a Wrapped CMP with cryptographic materials that you provide, or a compatible custom CMP. If the internal CMP in your MetaStore is a Direct KMS Provider, your reusable wrapping and signing keys are protected under a AWS KMS key in AWS Key Management Service (AWS KMS). The MetaStore calls AWS KMS every time it adds a new CMP version to its internal table or gets a CMP version from its internal table.

Setting a time-to-live value

You can set a time-to-live (TTL) value for each Most Recent Provider that you create. In general, use the lowest TTL value that is practical for your application.

The use of the TTL value is changed in the `CachingMostRecentProvider` symbol for the Most Recent Provider.

> **Note**
>
> The `MostRecentProvider` symbol for the Most Recent Provider is deprecated in older supported versions of the DynamoDB Encryption Client and removed from version 2.0.0. It is replaced by the `CachingMostRecentProvider` symbol. We recommend that you update your code as soon as possible. For details, see Updates to the Most Recent Provider.

`CachingMostRecentProvider`

The `CachingMostRecentProvider` uses the TTL value in two different ways.

- The TTL determines how often the Most Recent Provider checks the provider store for a new version of the CMP. If a new version is available, the Most Recent Provider replaces its CMP and refreshes its cryptographic materials. Otherwise, it continues to use its current CMP and cryptographic materials.
- The TTL determines how long CMPs in the cache can be used. Before it uses a cached CMP for encryption, the Most Recent Provider evaluates its time in the cache. If the CMP cache time exceeds the TTL, the CMP is evicted from the cache and the Most Recent Provider gets a new, latest-version CMP from its provider store.

`MostRecentProvider`

> In the `MostRecentProvider`, the TTL determines how often the Most Recent Provider checks the provider store for a new version of the CMP. If a new version is available, the Most Recent Provider replaces its CMP and refreshes its cryptographic materials. Otherwise, it continues to use its current CMP and cryptographic materials.

The TTL does not determine how often a new CMP version is created. You create new CMP versions by rotating the cryptographic materials.

An ideal TTL value varies with the application and its latency and availability goals. A lower TTL improves your security profile by reducing the time that cryptographic materials are stored in memory. Also, a lower TTL refreshes critical information more frequently. For example, if your internal CMP is a Direct KMS Provider, it verifies more frequently that the caller is still authorized to use an AWS KMS key.

However, if the TTL is too brief, the frequent calls to the provider store can increase your costs and cause your provider store to throttle requests from your application and other applications that share your service account. You might also benefit from coordinating the TTL with the rate at which you rotate cryptographic materials.

During testing, vary the TTL and cache size under different work loads until you find a configuration that works for your application and your security and performance standards.

Rotating cryptographic materials

When a Most Recent Provider needs encryption materials, it always uses the most recent version of its CMP that it knows about. The frequency that it checks for a newer version is determined by the time-to-live (TTL) value that you set when you configure the Most Recent Provider.

When the TTL expires, the Most Recent Provider checks the provider store for newer version of the CMP. If one is available, the Most Recent Provider get it and replaces the CMP in its cache. It uses this CMP and its cryptographic materials until it discovers that provider store has a newer version.

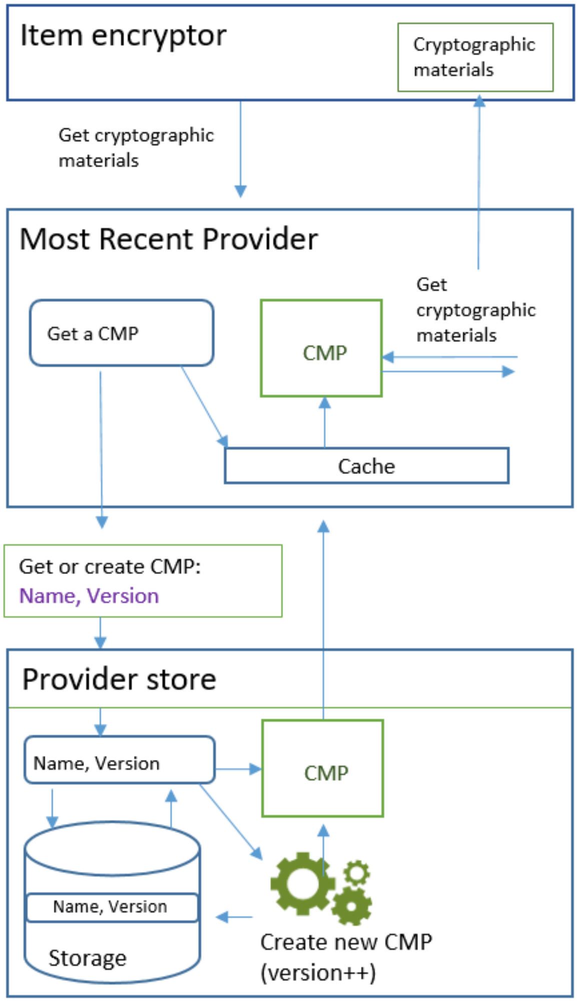
To tell the provider store to create a new version of a CMP for a Most Recent Provider, call the provider store's Create New Provider operation with the material name of the Most Recent Provider. The provider store creates a new CMP and saves an encrypted copy in its internal storage with a greater version number. (It also returns a CMP, but you can discard it.) As a result, the next time the Most Recent Provider queries the provider store for the maximum version number of its

CMPs, it gets the new greater version number, and uses it in subsequent requests to the store to see if a new version of the CMP has been created.

You can schedule your Create New Provider calls based on time, the number of items or attributes processed, or any other metric that makes sense for your application.

Get encryption materials

The Most Recent Provider uses the following process, shown in this diagram, to get the encryption materials that it returns to the item encryptor. The output depends on the type of CMP that the provider store returns. The Most Recent Provider can use any compatible provider store, including the MetaStore that is included in the DynamoDB Encryption Client.



When you create a Most Recent Provider by using the `CachingMostRecentProvider` symbol, you specify a provider store, a name for the Most Recent Provider, and a time-to-live (TTL) value. You can also optionally specify a cache size, which determines the maximum number of cryptographic materials that can exist in the cache.

When the item encryptor asks the Most Recent Provider for encryption materials, the Most Recent Provider begins by searching its cache for the latest version of its CMP.

- If it finds the latest version CMP in its cache and the CMP has not exceeded the TTL value, the Most Recent Provider uses the CMP to generate encryption materials. Then, it returns the encryption materials to the item encryptor. This operation does not require a call to the provider store.
- If the latest version of the CMP is not in its cache, or if it is in the cache but has exceeded its TTL value, the Most Recent Provider requests a CMP from its provider store. The request includes the Most Recent Provider material name and the maximum version number that it knows.
    1. The provider store returns a CMP from its persistent storage. If the provider store is a MetaStore, it gets an encrypted Wrapped CMP from its internal DynamoDB table by using the Most Recent Provider material name as the partition key and the version number as the sort key. The MetaStore uses its internal item encryptor and internal CMP to decrypt the Wrapped CMP. Then, it returns the plaintext CMP to the Most Recent Provider . If the internal CMP is a Direct KMS Provider, this step includes a call to the AWS Key Management Service (AWS KMS).
    2. The CMP adds the `amzn-ddb-meta-id` field to the actual material description. Its value is the material name and version of the CMP in its internal table. The provider store returns the CMP to the Most Recent Provider.
    3. The Most Recent Provider caches the CMP in memory.
    4. The Most Recent Provider uses the CMP to generate encryption materials. Then, it returns the encryption materials to the item encryptor.

Get decryption materials

When the item encryptor asks the Most Recent Provider for decryption materials, the Most Recent Provider uses the following process to get and return them.

1. The Most Recent Provider asks the provider store for the version number of the cryptographic materials that were used to encrypt the item. It passes in the actual material description from the material description attribute of the item.
2. The provider store gets the encrypting CMP version number from the `amzn-ddb-meta-id` field in the actual material description and returns it to the Most Recent Provider.
3. The Most Recent Provider searches its cache for the version of CMP that was used to encrypt and sign the item.

- If it finds the matching version of the CMP is in its cache and the CMP has not exceeded the time-to-live (TTL) value, the Most Recent Provider uses the CMP to generate decryption materials. Then, it returns the decryption materials to the item encryptor. This operation does not require a call to the provider store or any other CMP.
- If the matching version of the CMP is not in its cache, or if the cached AWS KMS key has exceeded its TTL value, the Most Recent Provider requests a CMP from its provider store. It sends its material name and the encrypting CMP version number in the request.
  1. The provider store searches its persistent storage for the CMP by using the Most Recent Provider name as the partition key and the version number as the sort key.
     - If the name and version number are not in its persistent storage, the provider store throws an exception. If the provider store was used to generate the CMP, the CMP should be stored in its persistent storage, unless it was intentionally deleted.
     - If the CMP with the matching name and version number are in the provider store's persistent storage, the provider store returns the specified CMP to the Most Recent Provider.

       If the provider store is a MetaStore, it gets the encrypted CMP from its DynamoDB table. Then, it uses cryptographic materials from its internal CMP to decrypt the encrypted CMP before it returns the CMP to Most Recent Provider. If the internal CMP is a Direct KMS Provider, this step includes a call to the AWS Key Management Service (AWS KMS).
  2. The Most Recent Provider caches the CMP in memory.
  3. The Most Recent Provider uses the CMP to generate decryption materials. Then, it returns the decryption materials to the item encryptor.

Updates to the Most Recent Provider

The symbol for the Most Recent Provider is changed from `MostRecentProvider` to `CachingMostRecentProvider`.

> **Note**
>
> The `MostRecentProvider` symbol, which represents the Most Recent Provider, is deprecated in version 1.15 of the DynamoDB Encryption Client for Java and version 1.3 of the DynamoDB Encryption Client for Python and removed from versions 2.0.0 of the DynamoDB Encryption Client in both language implementations. Instead, use the `CachingMostRecentProvider`.

The `CachingMostRecentProvider` implements the following changes:

- The `CachingMostRecentProvider` periodically removes cryptographic materials from memory when their time in memory exceeds the configured time-to-live (TTL) value.

  The `MostRecentProvider` might store cryptographic materials in memory for the lifetime of the process. As a result, the Most Recent Provider might not be aware of authorization changes. It might use encryption keys after the caller's permissions to use them are revoked.

  If you can't update to this new version, you can get a similar effect by periodically calling the `clear()` method on the cache. This method manually flushes the cache contents and requires the Most Recent Provider to request a new CMP and new cryptographic materials.
- The `CachingMostRecentProvider` also includes a cache size setting that gives you more control over the cache.

To update to the `CachingMostRecentProvider`, you have to change the symbol name in your code. In all other respects, the `CachingMostRecentProvider` is fully backwards compatible with the `MostRecentProvider`. You don't need to re-encrypt any table items.

However, the `CachingMostRecentProvider` generates more calls to the underlying key infrastructure. It calls the provider store at least once in each time-to-live (TTL) interval. Applications with numerous active CMPs (due to frequent rotation) or applications with large fleets are most likely to be sensitive to this change.

Before releasing your updated code, test it thoroughly to ensure that the more frequent calls don't impair your application or cause throttling by services on which your provider depends, such as AWS Key Management Service (AWS KMS) or Amazon DynamoDB. To mitigate any performance problems, adjust the cache size and the time-to-live of the `CachingMostRecentProvider` based on the performance characteristics you observe. For guidance, see Setting a time-to-live value.

Static Materials Provider

> **Note**
>
> Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The *Static Materials Provider* (Static CMP) is a very simple cryptographic materials provider (CMP) that is intended for testing, proof-of-concept demonstrations, and legacy compatibility.

To use the Static CMP to encrypt a table item, you supply an Advanced Encryption Standard (AES) symmetric encryption key and a signing key or key pair. You must supply the same keys to decrypt the encrypted item. The Static CMP does not perform any cryptographic operations. Instead, it passes the encryption keys that you supply to the item encryptor unchanged. The item encryptor encrypts the items directly under the encryption key. Then, it uses the signing key directly to sign them.

Because the Static CMP does not generate any unique cryptographic materials, all table items that you process are encrypted with the same encryption key and signed by the same signing key. When you use the same key to encrypt the attributes values in numerous items or use the same key or key pair to sign all items, you risk exceeding the cryptographic limits of the keys.

> **Note**
>
> The Asymmetric Static Provider in the Java library is not a static provider. It just supplies alternate constructors for the Wrapped CMP. It's safe for production use, but you should use the Wrapped CMP directly whenever possible.

The Static CMP is one of several cryptographic materials providers (CMPs) that the DynamoDB Encryption Client supports. For information about the other CMPs, see Cryptographic materials provider.

For example code, see:

- Java: SymmetricEncryptedItem

Topics

- How to use it
- How it works

How to use it

To create a static provider, supply an encryption key or key pair and a signing key or key pair. You need to provide key material to encrypt and decrypt table items.

Java

```
// To encrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// To decrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Verification key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);
```

Python

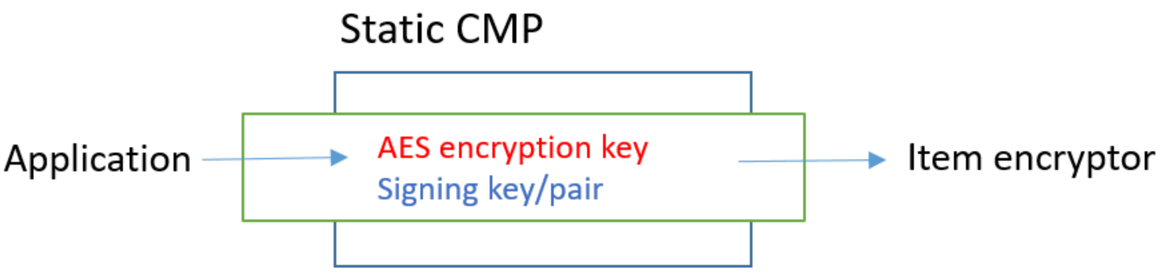
```
# You can provide encryption materials, decryption materials, or both
encrypt_keys = EncryptionMaterials(
    encryption_key = ...,
    signing_key = ...
)

decrypt_keys = DecryptionMaterials(
    decryption_key = ...,
    verification_key = ...
)

static_cmp = StaticCryptographicMaterialsProvider(
    encryption_materials=encrypt_keys
    decryption_materials=decrypt_keys
)
```

How it works

The Static Provider passes the encryption and signing keys that you supply to the item encryptor, where they are used directly to encrypt and sign your table items. Unless you supply different keys for each item, the same keys are used for every item.



Get encryption materials

This section describes in detail the inputs, outputs, and processing of the Static Materials Provider (Static CMP) when it receives a request for encryption materials.

Input (from the application)

- An encryption key – This must be a symmetric key, such as an Advanced Encryption Standard (AES) key.
- A signing key – This can be a symmetric key or an asymmetric key pair.

Input (from the item encryptor)

- DynamoDB encryption context

Output (to the item encryptor)

- The encryption key passed as input.
- The signing key passed as input.
- Actual material description: The requested material description, if any, unchanged.

Get decryption materials

This section describes in detail the inputs, outputs, and processing of the Static Materials Provider (Static CMP) when it receives a request for decryption materials.

Although it includes separate methods for getting encryption materials and getting decryption materials, the behavior is the same.

Input (from the application)

- An encryption key – This must be a symmetric key, such as an Advanced Encryption Standard (AES) key.
- A signing key – This can be a symmetric key or an asymmetric key pair.

Input (from the item encryptor)

- DynamoDB encryption context (not used)

Output (to the item encryptor)

- The encryption key passed as input.
- The signing key passed as input.

Amazon DynamoDB Encryption Client available programming languages

> **Note**
>
> Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The Amazon DynamoDB Encryption Client is available for the following programming languages. The language-specific libraries vary, but the resulting implementations are interoperable. For example, you can encrypt (and sign) an item with the Java client and decrypt the item with the Python client.

For more information, see the corresponding topic.

Topics

- Amazon DynamoDB Encryption Client for Java
- DynamoDB Encryption Client for Python

Amazon DynamoDB Encryption Client for Java

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

This topic explains how to install and use the Amazon DynamoDB Encryption Client for Java. For details about programming with the DynamoDB Encryption Client, see the Java examples, the examples in the aws-dynamodb-encryption-java repository on GitHub, and the Javadoc for the DynamoDB Encryption Client.

Note

Versions 1.*x*.*x* of the DynamoDB Encryption Client for Java are in end-of-support phase effective July 2022. Upgrade to a newer version as soon as possible.

Topics

Prerequisites

Before you install the Amazon DynamoDB Encryption Client for Java, be sure you have the following prerequisites.

A Java development environment

You will need Java 8 or later. On the Oracle website, go to Java SE Downloads, and then download and install the Java SE Development Kit (JDK).

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

AWS SDK for Java

The DynamoDB Encryption Client requires the DynamoDB module of the AWS SDK for Java even if your application doesn't interact with DynamoDB. You can install the entire SDK or just this module. If you are using Maven, add `aws-java-sdk-dynamodb` to your `pom.xml` file.

For more information about installing and configuring the AWS SDK for Java, see AWS SDK for Java.

Installation

You can install the Amazon DynamoDB Encryption Client for Java in the following ways.

Manually

To install the Amazon DynamoDB Encryption Client for Java, clone or download the aws-dynamodb-encryption-java GitHub repository.

Using Apache Maven

The Amazon DynamoDB Encryption Client for Java is available through Apache Maven with the following dependency definition.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

After you install the SDK, get started by looking at the example code in this guide and the DynamoDB Encryption Client Javadoc on GitHub.

Using the DynamoDB Encryption Client for Java

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption

Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

This topic explains some of the features of the DynamoDB Encryption Client in Java that might not be found in other programming language implementations.

For details about programming with the DynamoDB Encryption Client, see the Java examples, the examples in the `aws-dynamodb-encryption-java repository` on GitHub, and the Javadoc for the DynamoDB Encryption Client.

Topics

- Item encryptors: AttributeEncryptor and DynamoDBEncryptor
- Configuring save behavior
- Attribute actions in Java
- Overriding table names

Item encryptors: AttributeEncryptor and DynamoDBEncryptor

The DynamoDB Encryption Client in Java has two item encryptors: the lower-level DynamoDBEncryptor and the AttributeEncryptor.

The `AttributeEncryptor` is a helper class that helps you use the DynamoDBMapper in the AWS SDK for Java with the `DynamoDB Encryptor` in the DynamoDB Encryption Client. When you use the `AttributeEncryptor` with the `DynamoDBMapper`, it transparently encrypts and signs your items when you save them. It also transparently verifies and decrypts your items when you load them.

Configuring save behavior

You can use the `AttributeEncryptor` and `DynamoDBMapper` to add or replace table items with attributes that are signed only or encrypted and signed. For these tasks, we recommend that you configure it to use the `PUT` save behavior, as shown in the following example. Otherwise, you might not be able to decrypt your data.

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
```

```
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

If you use the default save behavior, which updates only the attributes that are modeled in the table item, attributes that are not modeled are not included in the signature, and are not changed by table writes. As a result, on later reads of all attributes, the signature will not validate, because it doesn't include un-modeled attributes.

You can also use the `CLOBBER` save behavior. This behavior is identical to the `PUT` save behavior except that it disables optimistic locking and overwrites the item in the table.

To prevent signature errors, the DynamoDB Encryption Client throws a runtime exception if an `AttributeEncryptor` is used with a `DynamoDBMapper` that is not configured with a save behavior of `CLOBBER` or `PUT`.

To see this code used in an example, see Using the DynamoDBMapper and the AwsKmsEncryptedObject.java example in the `aws-dynamodb-encryption-java` repository in GitHub.

Attribute actions in Java

Attribute actions determine which attribute values are encrypted and signed, which are only signed, and which are ignored. The method you use to specify attribute actions depends on whether you use the `DynamoDBMapper` and `AttributeEncryptor`, or the lower-level DynamoDBEncryptor.

> ⚠ **Important**
>
> After you use your attribute actions to encrypt your table items, adding or removing attributes from your data model might cause a signature validation error that prevents you from decrypting your data. For a detailed explanation, see Changing your data model.

Attribute actions for the DynamoDBMapper

When you use the `DynamoDBMapper` and `AttributeEncryptor`, you use annotations to specify the attribute actions. The DynamoDB Encryption Client uses the standard DynamoDB attribute annotations that define the attribute type to determine how to protect an attribute. By default, all attributes are encrypted and signed except for primary keys, which are signed but not encrypted.

Note

Do not encrypt the value of attributes with the @DynamoDBVersionAttribute annotation, although you can (and should) sign them. Otherwise, conditions that use its value will have unintended effects.

```
// Attributes are encrypted and signed
@DynamoDBAttribute(attributeName="Description")

// Partition keys are signed but not encrypted
@DynamoDBHashKey(attributeName="Title")

// Sort keys are signed but not encrypted
@DynamoDBRangeKey(attributeName="Author")
```

To specify exceptions, use the encryption annotations defined in the DynamoDB Encryption Client for Java. If you specify them at the class level, they become the default value for the class.

```
// Sign only
@DoNotEncrypt

// Do nothing; not encrypted or signed
@DoNotTouch
```

For example, these annotations sign but do not encrypt the `PublicationYear` attribute, and do not encrypt or sign the `ISBN` attribute value.

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

Attribute actions for the DynamoDBEncryptor

To specify attribute actions when you use the DynamoDBEncryptor directly, create a `HashMap` object in which the name-value pairs represent attribute names and the specified actions.

The valid values are for the attribute actions are defined in the `EncryptionFlags` enumerated type. You can use `ENCRYPT` and `SIGN` together, use `SIGN` alone, or omit both. However, if you use `ENCRYPT` alone, the DynamoDB Encryption Client throws an error. You cannot encrypt an attribute that you don't sign.

```
ENCRYPT
SIGN
```

> **Warning**
>
> Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

If you specify a primary key in the encryption context and then specify `ENCRYPT` in the attribute action for either primary key attribute, the DynamoDB Encryption Client throws an exception.

For example, the following Java code creates an `actions` HashMap that encrypts and signs all attributes in the `record` item. The exceptions are the partition key and sort key attributes, which are signed but not encrypted, and the `test` attribute, which is not signed or encrypted.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // no break; falls through to next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Don't encrypt or sign
      break;
    default:
      // Encrypt and sign everything else
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

Then, when you call the encryptRecord method of the `DynamoDBEncryptor`, specify the map as the value of the `attributeFlags` parameter. For example, this call to `encryptRecord` uses the `actions` map.

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Overriding table names

In the DynamoDB Encryption Client, the name of the DynamoDB table is an element of the DynamoDB encryption context that is passed to the encryption and decryption methods. When you encrypt or sign table items, the DynamoDB encryption context, including the table name, is cryptographically bound to the ciphertext. If the DynamoDB encryption context that is passed to the decrypt method doesn't match the DynamoDB encryption context that was passed to the encrypt method, the decrypt operation fails.

Occasionally, the name of a table changes, such as when you back up a table or perform a point-in-time recovery. When you decrypt or verify the signature of these items, you must pass in the same DynamoDB encryption context that was used to encrypt and sign the items, including the original table name. The current table name is not needed.

When you use the `DynamoDBEncryptor`, you assemble the DynamoDB encryption context manually. However, if you are using the `DynamoDBMapper`, the `AttributeEncryptor` creates the DynamoDB encryption context for you, including the current table name. To tell the `AttributeEncryptor` to create an encryption context with a different table name, use the `EncryptionContextOverrideOperator`.

For example, the following code creates instances of the cryptographic materials provider (CMP) and the `DynamoDBEncryptor`. Then it calls the `setEncryptionContextOverrideOperator` method of the `DynamoDBEncryptor`. It uses the `overrideEncryptionContextTableName` operator, which overrides one table name. When it is configured this way, the `AttributeEncryptor` creates a DynamoDB encryption context that includes `newTableName` in place of `oldTableName`. For a complete example, see EncryptionContextOverridesWithDynamoDBMapper.java.

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

When you call the load method of the `DynamoDBMapper`, which decrypts and verifies the item, you specify the original table name.

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

You can also use the `overrideEncryptionContextTableNameUsingMap` operator, which overrides multiple table names.

The table name override operators are typically used when decrypting data and verifying signatures. However, you can use them to set the table name in the DynamoDB encryption context to a different value when encrypting and signing.

Do not use the table name override operators if you are using the `DynamoDBEncryptor`. Instead, create an encryption context with the original table name and submit it to the decryption method.

Example code for the DynamoDB Encryption Client for Java

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The following examples show you how to use the DynamoDB Encryption Client for Java to protect DynamoDB table items in your application. You can find more examples (and contribute your own) in the examples directory of the aws-dynamodb-encryption-java repository on GitHub.

Topics

- Using the DynamoDBEncryptor

- Using the DynamoDBMapper

Using the DynamoDBEncryptor

This example shows how to use the lower-level DynamoDBEncryptor with the Direct KMS Provider. The Direct KMS Provider generates and protects its cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) that you specify.

You can use any compatible cryptographic materials provider (CMP) with the `DynamoDBEncryptor`, and you can use the Direct KMS Provider with the `DynamoDBMapper` and AttributeEncryptor.

See the complete code sample: AwsKmsEncryptedItem.java

Step 1: Create the Direct KMS Provider

Create an instance of the AWS KMS client with the specified region. Then, use the client instance to create an instance of the Direct KMS Provider with your preferred AWS KMS key.

This example uses the Amazon Resource Name (ARN) to identify the AWS KMS key, but you can use any valid key identifier.

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

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Step 2: Create an item

This example defines a `record` HashMap that represents a sample table item.

```
final String partitionKeyName = "partition_attribute";
final String sortKeyName = "sort_attribute";

final Map<String, AttributeValue> record = new HashMap<>();
record.put(partitionKeyName, new AttributeValue().withS("value1"));
record.put(sortKeyName, new AttributeValue().withN("55"));
record.put("example", new AttributeValue().withS("data"));
record.put("numbers", new AttributeValue().withN("99"));
record.put("binary", new AttributeValue().withB(ByteBuffer.wrap(new byte[]{0x00,
 0x01, 0x02})));
record.put("test", new AttributeValue().withS("test-value"));
```

Step 3: Create a DynamoDBEncryptor

Create an instance of the `DynamoDBEncryptor` with the Direct KMS Provider.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

Step 4: Create a DynamoDB encryption context

The DynamoDB encryption context contains information about the table structure and how it is encrypted and signed. If you use the `DynamoDBMapper`, the `AttributeEncryptor` creates the encryption context for you.

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

Step 5: Create the attribute actions object

Attribute actions determine which attributes of the item are encrypted and signed, which are only signed, and which are not encrypted or signed.

In Java, to specify attribute actions, you create a HashMap of attribute name and `EncryptionFlags` value pairs.

For example, the following Java code creates an `actions` HashMap that encrypts and signs all attributes in the `record` item, except for the partition key and sort key attributes, which are signed, but not encrypted, and the `test` attribute, which is not signed or encrypted.

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();
```

```
for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // fall through to the next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Neither encrypted nor signed
      break;
    default:
      // Encrypt and sign all other attributes
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

Step 6: Encrypt and sign the item

To encrypt and sign the table item, call the `encryptRecord` method on the instance of the `DynamoDBEncryptor`. Specify the table item (`record`), the attribute actions (`actions`), and the encryption context (`encryptionContext`).

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Step 7: Put the item in the DynamoDB table

Finally, put the encrypted and signed item in the DynamoDB table.

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

Using the DynamoDBMapper

The following example shows you how to use the DynamoDB mapper helper class with the Direct KMS Provider. The Direct KMS Provider generates and protects its cryptographic materials under an AWS KMS key in AWS Key Management Service (AWS KMS) that you specify.

You can use any compatible cryptographic materials provider (CMP) with the `DynamoDBMapper`, and you can use the Direct KMS Provider with the lower-level `DynamoDBEncryptor`.

See the complete code sample: [AwsKmsEncryptedObject.java]

Step 1: Create the Direct KMS Provider

Create an instance of the AWS KMS client with the specified region. Then, use the client instance to create an instance of the Direct KMS Provider with your preferred AWS KMS key.

This example uses the Amazon Resource Name (ARN) to identify the AWS KMS key, but you can use any valid key identifier.

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

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Step 2: Create the DynamoDB Encryptor and DynamoDBMapper

Use the Direct KMS Provider that you created in the previous step to create an instance of the DynamoDB Encryptor. You need to instantiate the lower-level DynamoDB Encryptor to use the DynamoDB Mapper.

Next, create an instance of your DynamoDB database and a mapper configuration, and use them to create an instance of the DynamoDB Mapper.

⚠ **Important**

When using the `DynamoDBMapper` to add or edit signed (or encrypted and signed) items, configure it to use a save behavior, such as PUT, that includes all attributes, as shown in the following example. Otherwise, you might not be able to decrypt your data.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp)
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

Step 3: Define your DynamoDB table

Next, define your DynamoDB table. Use annotations to specify the attribute actions. This example creates a DynamoDB table, `ExampleTable`, and a `DataPoJo` class that represents table items.

In this sample table, the primary key attributes will be signed but not encrypted. This applies to the `partition_attribute`, which is annotated with `@DynamoDBHashKey`, and the `sort_attribute`, which is annotated with `@DynamoDBRangeKey`.

Attributes that are annotated with `@DynamoDBAttribute`, such as `some numbers`, will be encrypted and signed. The exceptions are attributes that use the `@DoNotEncrypt` (sign only) or `@DoNotTouch` (do not encrypt or sign) encryption annotations defined by the DynamoDB Encryption Client. For example, because the `leave me` attribute has a `@DoNotTouch` annotation, it will not be encrypted or signed.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String example;
  private long someNumbers;
  private byte[] someBinary;
  private String leaveMe;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }
  @DynamoDBAttribute(attributeName = "example")
  public String getExample() {
    return example;
  }

  public void setExample(String example) {
    this.example = example;
  }

  @DynamoDBAttribute(attributeName = "some numbers")
  public long getSomeNumbers() {
    return someNumbers;
  }

  public void setSomeNumbers(long someNumbers) {
    this.someNumbers = someNumbers;
  }

  @DynamoDBAttribute(attributeName = "and some binary")
  public byte[] getSomeBinary() {
    return someBinary;
  }

  public void setSomeBinary(byte[] someBinary) {
    this.someBinary = someBinary;
  }

  @DynamoDBAttribute(attributeName = "leave me")
  @DoNotTouch
  public String getLeaveMe() {
    return leaveMe;
  }

  public void setLeaveMe(String leaveMe) {
    this.leaveMe = leaveMe;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ", sortAttribute="
        + sortAttribute + ", example=" + example + ", someNumbers=" + someNumbers
        + ", someBinary=" + Arrays.toString(someBinary) + ", leaveMe=" + leaveMe +
 "]";
```

```
    }
}
```

Step 4: Encrypt and save a table item

Now, when you create a table item and use the DynamoDB Mapper to save it, the item is automatically encrypted and signed before it is added to the table.

This example defines a table item called `record`. Before it is saved in the table, its attributes are encrypted and signed based on the annotations in the `DataPoJo` class. In this case, all attributes except for `PartitionAttribute`, `SortAttribute`, and `LeaveMe` are encrypted and signed. `PartitionAttribute` and `SortAttributes` are only signed. The `LeaveMe` attribute is not encrypted or signed.

To encrypt and sign the `record` item, and then add it to the `ExampleTable`, call the `save` method of the `DynamoDBMapper` class. Because your DynamoDB Mapper is configured to use the PUT save behavior, the item replaces any item with the same primary keys, instead of updating it. This ensures that the signatures match and you can decrypt the item when you get it from the table.

```
DataPoJo record = new DataPoJo();
record.setPartitionAttribute("is this");
record.setSortAttribute(55);
record.setExample("data");
record.setSomeNumbers(99);
record.setSomeBinary(new byte[]{0x00, 0x01, 0x02});
record.setLeaveMe("alone");

mapper.save(record);
```

DynamoDB Encryption Client for Python

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

This topic explains how to install and use the DynamoDB Encryption Client for Python. You can find the code in the aws-dynamodb-encryption-python repository on GitHub, including complete and tested sample code to help you get started.

Note

Versions 1.*x.x* and 2.*x.x* of the DynamoDB Encryption Client for Python are in end-of-support phase effective July 2022. Upgrade to a newer version as soon as possible.

Topics

Prerequisites

Before you install the Amazon DynamoDB Encryption Client for Python, be sure you have the following prerequisites.

A supported version of Python

Python 3.8 or later is required by the Amazon DynamoDB Encryption Client for Python versions 3.3.0 and later. To download Python, see Python downloads.

Earlier versions of the Amazon DynamoDB Encryption Client for Python support Python 2.7 and Python 3.4 and later, but we recommend that you use the latest version of the DynamoDB Encryption Client.

The pip installation tool for Python

Python 3.6 and later include **pip**, although you might want to upgrade it. For more information about upgrading or installing pip, see Installation in the **pip** documentation.

Installation

Use **pip** to install the Amazon DynamoDB Encryption Client for Python, as shown in the following examples.

To install the latest version

```
pip install dynamodb-encryption-sdk
```

For more details about using **pip** to install and upgrade packages, see Installing Packages.

The DynamoDB Encryption Client requires the cryptography library on all platforms. All versions of **pip** install and build the **cryptography** library on Windows. **pip** 8.1 and later installs and builds **cryptography** on Linux. If you are using an earlier version of **pip** and your Linux environment doesn't have the tools needed to build the **cryptography** library, you need to install them. For more information, see Building cryptography on Linux.

You can get the latest development version of the DynamoDB Encryption Client from the aws-dynamodb-encryption-python repository on GitHub.

After you install the DynamoDB Encryption Client, get started by looking at the example Python code in this guide.

Using the DynamoDB Encryption Client for Python

> **Note**
>
> Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

This topic explains some of the features of the DynamoDB Encryption Client for Python that might not be found in other programming language implementations. These features are designed to make it easier to use the DynamoDB Encryption Client in the most secure way. Unless you have an unusual use case, we recommend that you use them.

For details about programming with the DynamoDB Encryption Client, see the Python examples in this guide, the examples in the aws-dynamodb-encryption-python repository on GitHub, and the Python documentation for the DynamoDB Encryption Client.

Topics

- Client helper classes
- TableInfo class
- Attribute actions in Python

Client helper classes

The DynamoDB Encryption Client for Python includes several client helper classes that mirror the Boto 3 classes for DynamoDB. These helper classes are designed to make it easier to add encryption and signing to your existing DynamoDB application and avoid the most common problems, as follows:

- Prevent you from encrypting the primary key in your item, either by adding an override action for the primary key to the AttributeActions object, or by throwing an exception if your `AttributeActions` object explicitly tells the client to encrypt the primary key. If the default action in your `AttributeActions` object is `DO_NOTHING`, the client helper classes use that action for the primary key. Otherwise, they use `SIGN_ONLY`.
- Create a TableInfo object and populate the DynamoDB encryption context based on a call to DynamoDB. This helps to ensure that your DynamoDB encryption context is accurate and the client can identify the primary key.
- Support methods, such as `put_item` and `get_item`, that transparently encrypt and decrypt your table items when you write to or read from a DynamoDB table. Only the `update_item` method is unsupported.

You can use the client helper classes instead of interacting directly with the lower-level item encryptor. Use these classes unless you need to set advanced options in the item encryptor.

The client helper classes include:

- EncryptedTable for applications that use the Table resource in DynamoDB to process one table at a time.
- EncryptedResource for applications that use the Service Resource class in DynamoDB for batch processing.
- EncryptedClient for applications that use the lower-level client in DynamoDB.

To use the client helper classes, the caller must have permission to call the DynamoDB DescribeTable operation on the target table.

TableInfo class

The TableInfo class is a helper class that represents a DynamoDB table, complete with fields for its primary key and secondary indexes. It helps you to get accurate, real-time information about the table.

If you use a client helper class, it creates and uses a `TableInfo` object for you. Otherwise, you can create one explicitly. For an example, see Use the item encryptor.

When you call the `refresh_indexed_attributes` method on a `TableInfo` object, it populates the property values of the object by calling the DynamoDB DescribeTable operation. Querying the table is much more reliable than hard-coding index names. The `TableInfo` class also includes an `encryption_context_values` property that provides the required values for the DynamoDB encryption context.

To use the `refresh_indexed_attributes` method, the caller must have permission to call the DynamoDB DescribeTable operation on the target table.

Attribute actions in Python

Attribute actions tell the item encryptor which actions to perform on each attribute of the item. To specify attribute actions in Python, create an `AttributeActions` object with a default action and any exceptions for particular attributes. The valid values are defined in the `CryptoAction` enumerated type.

> ⚠ **Important**
>
> After you use your attribute actions to encrypt your table items, adding or removing attributes from your data model might cause a signature validation error that prevents you from decrypting your data. For a detailed explanation, see Changing your data model.

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

For example, this `AttributeActions` object establishes `ENCRYPT_AND_SIGN` as the default for all attributes, and specifies exceptions for the `ISBN` and `PublicationYear` attributes.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

If you use a client helper class, you don't need to specify an attribute action for the primary key attributes. The client helper classes prevent you from encrypting your primary key.

If you do not use a client helper class and the default action is `ENCRYPT_AND_SIGN`, you must specify an action for the primary key. The recommended action for primary keys is `SIGN_ONLY`. To make this easy, use the `set_index_keys` method, which uses SIGN_ONLY for primary keys, or DO_NOTHING, when that is the default action.

Warning

Do not encrypt the primary key attributes. They must remain in plaintext so DynamoDB can find the item without running a full table scan.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Example code for the DynamoDB Encryption Client for Python

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

The following examples show you how to use the DynamoDB Encryption Client for Python to protect DynamoDB data in your application. You can find more examples (and contribute your own) in the examples directory of the aws-dynamodb-encryption-python repository on GitHub.

Topics

- Use the EncryptedTable client helper class
- Use the item encryptor

Use the EncryptedTable client helper class

The following example shows you how to use the Direct KMS Provider with the `EncryptedTable` client helper class. This example uses the same cryptographic materials provider as the Use the item encryptor example that follows. However, it uses the `EncryptedTable` class instead of interacting directly with the lower-level item encryptor.

By comparing these examples, you can see the work that the client helper class does for you. This includes creating the DynamoDB encryption context and making sure the primary key attributes are always signed, but never encrypted. To create the encryption context and discover the primary key, the client helper classes call the DynamoDB DescribeTable operation. To run this code, you must have permission to call this operation.

See the complete code sample: aws_kms_encrypted_table.py

Step 1: Create the table

Start by creating an instance of a standard DynamoDB table with the table name.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Step 2: Create a cryptographic materials provider

Create an instance of the cryptographic materials provider (CMP) that you selected.

This example uses the Direct KMS Provider, but you can use any compatible CMP. To create a Direct KMS Provider, specify an AWS KMS key. This example uses the Amazon Resource Name (ARN) of the AWS KMS key, but you can use any valid key identifier.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Step 3: Create the attribute actions object

Attribute actions tell the item encryptor which actions to perform on each attribute of the item. The `AttributeActions` object in this example encrypts and signs all items except for the `test` attribute, which is ignored.

Do not specify attribute actions for the primary key attributes when you use a client helper class. The `EncryptedTable` class signs, but never encrypts, the primary key attributes.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

Step 4: Create the encrypted table

Create the encrypted table using the standard table, the Direct KMS Provider, and the attribute actions. This step completes the configuration.

```
encrypted_table = EncryptedTable(
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

Step 5: Put the plaintext item in the table

When you call the `put_item` method on the `encrypted_table`, your table items are transparently encrypted, signed, and added to your DynamoDB table.

First, define the table item.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Then, put it in the table.

```
encrypted_table.put_item(Item=plaintext_item)
```

To get the item from the DynamoDB table in its encrypted form, call the `get_item` method on the `table` object. To get the decrypted item, call the `get_item` method on the `encrypted_table` object.

Use the item encryptor

This example shows you how to interact directly with the item encryptor in the DynamoDB Encryption Client when encrypting table items, instead of using the client helper classes that interact with the item encryptor for you.

When you use this technique, you create the DynamoDB encryption context and configuration object (`CryptoConfig`) manually. Also, you encrypt the items in one call and put them in your DynamoDB table in a separate call. This allows you to customize your `put_item` calls and use the DynamoDB Encryption Client to encrypt and sign structured data that is never sent to DynamoDB.

This example uses the Direct KMS Provider, but you can use any compatible CMP.

See the complete code sample: aws_kms_encrypted_item.py

Step 1: Create the table

Start by creating an instance of a standard DynamoDB table resource with the table name.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Step 2: Create a cryptographic materials provider

Create an instance of the cryptographic materials provider (CMP) that you selected.

This example uses the Direct KMS Provider, but you can use any compatible CMP. To create a Direct KMS Provider, specify an AWS KMS key. This example uses the Amazon Resource Name (ARN) of the AWS KMS key, but you can use any valid key identifier.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Step 3: Use the TableInfo helper class

To get information about the table from DynamoDB, create an instance of the TableInfo helper class. When you work directly with the item encryptor, you need to create a `TableInfo` instance and call its methods. The client helper classes do this for you.

The `refresh_indexed_attributes` method of `TableInfo` uses the DescribeTable DynamoDB operation to get real-time, accurate information about the table. This includes its primary key and its local and global secondary indexes. The caller needs to have permission to call `DescribeTable`.

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

Step 4: Create the DynamoDB encryption context

The DynamoDB encryption context contains information about the table structure and how it is encrypted and signed. This example creates a DynamoDB encryption context explicitly, because it interacts with the item encryptor. The client helper classes create the DynamoDB encryption context for you.

To get the partition key and sort key, you can use the properties of the TableInfo helper class.

```
index_key = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
}

encryption_context = EncryptionContext(
    table_name=table_name,
    partition_key_name=table_info.primary_index.partition,
    sort_key_name=table_info.primary_index.sort,
    attributes=dict_to_ddb(index_key)
)
```

Step 5: Create the attribute actions object

Attribute actions tell the item encryptor which actions to perform on each attribute of the item. The `AttributeActions` object in this example encrypts and signs all items except for the primary key attributes, which are signed, but not encrypted, and the `test` attribute, which is ignored.

When you interact directly with the item encryptor and your default action is `ENCRYPT_AND_SIGN`, you must specify an alternative action for the primary key. You can use the `set_index_keys` method, which uses `SIGN_ONLY` for the primary key, or it uses `DO_NOTHING` if it's the default action.

To specify the primary key, this example uses the index keys in the [TableInfo] object, which is populated by a call to DynamoDB. This technique is safer than hard-coding primary key names.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Step 6: Create the configuration for the item

To configure the DynamoDB Encryption Client, use the objects that you just created in a [CryptoConfig] configuration for the table item. The client helper classes create the CryptoConfig for you.

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

Step 7: Encrypt the item

This step encrypts and signs the item, but it doesn't put it in the DynamoDB table.

When you use a client helper class, your items are transparently encrypted and signed, and then added to your DynamoDB table when you call the `put_item` method of the helper class. When you use the item encryptor directly, the encrypt and put actions are independent.

First, create a plaintext item.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Then, encrypt and sign it. The `encrypt_python_item` method requires the `CryptoConfig` configuration object.

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

Step 8: Put the item in the table

This step puts the encrypted and signed item in the DynamoDB table.

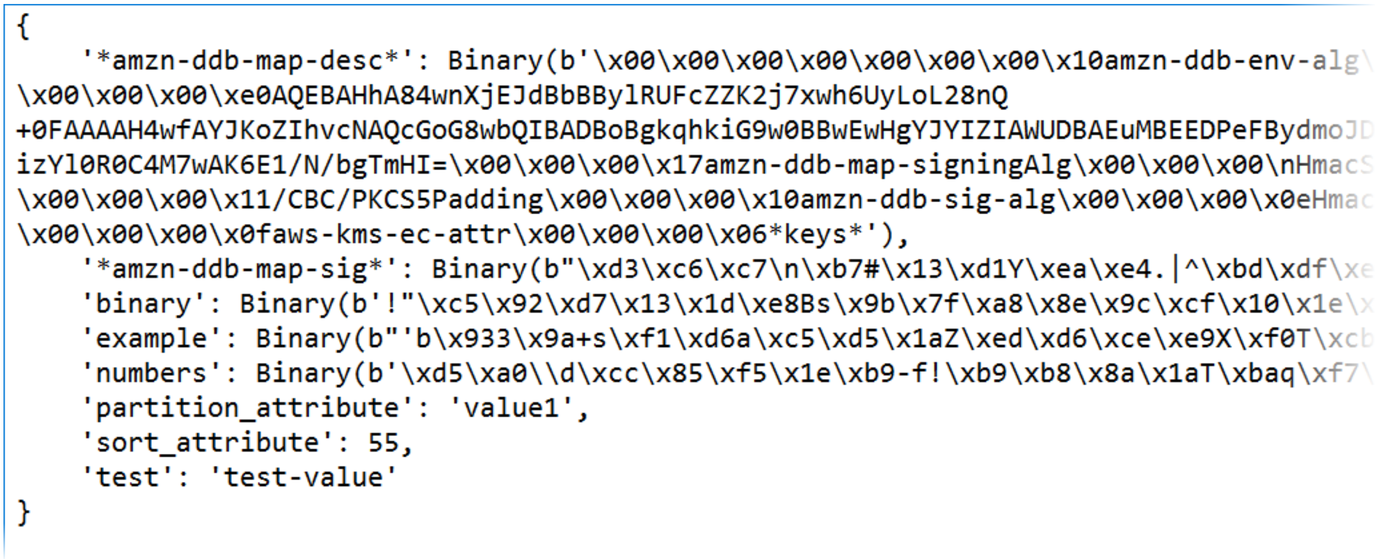
```
table.put_item(Item=encrypted_item)
```

To view the encrypted item, call the `get_item` method on the original `table` object, instead of the `encrypted_table` object. It gets the item from the DynamoDB table without verifying and decrypting it.

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

The following image shows part of an example encrypted and signed table item.

The encrypted attribute values are binary data. The names and values of the primary key attributes (`partition_attribute` and `sort_attribute`) and the `test` attribute remain in plaintext. The output also shows the attribute that contains the signature (`*amzn-ddb-map-sig*`) and the materials description attribute (`*amzn-ddb-map-desc*`).



```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

Changing your data model

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

Every time you encrypt or decrypt an item, you need to provide attribute actions that tell the DynamoDB Encryption Client which attributes to encrypt and sign, which attributes to sign (but not encrypt), and which to ignore. Attribute actions are not saved in the encrypted item and the DynamoDB Encryption Client does not update your attribute actions automatically.

Important

The DynamoDB Encryption Client does not support the encryption of existing, unencrypted DynamoDB table data.

Whenever you change your data model, that is, when you add or remove attributes from your table items, you risk an error. If the attribute actions that you specify do not account for all attributes in the item, the item might not be encrypted and signed the way that you intend. More importantly, if the attribute actions that you provide when decrypting an item differ from the attribute actions that you provided when encrypting the item, the signature verification might fail.

For example, if the attribute actions used to encrypt the item tell it to sign the `test` attribute, the signature in the item will include the `test` attribute. But if the attribute actions used to decrypt the item do not account for the `test` attribute, the verification will fail because the client will try to verify a signature that does not include the `test` attribute.

This is a particular problem when multiple applications read and write the same DynamoDB items because the DynamoDB Encryption Client must calculate the same signature for items in all applications. It's also a problem for any distributed application because changes in attribute actions must propagate to all hosts. Even if your DynamoDB tables are accessed by one host in one process, establishing a best practice process will help prevent errors if the project ever becomes more complex.

To avoid signature validation errors that prevent you from reading your table items, use the following guidance.

- Adding an attribute — If the new attribute changes your attribute actions, fully deploy the attribute action change before including the new attribute in an item.
- Removing an attribute — If you stop using an attribute in your items, do not change your attribute actions.
- Changing the action — After you have used an attribute actions configuration to encrypt your table items, you cannot safely change the default action or the action for an existing attribute without re-encrypting every item in your table.

Signature validation errors can be extremely difficult to resolve, so the best approach is to prevent them.

Topics

Adding an attribute

When you add a new attribute to table items, you might need to change your attribute actions. To prevent signature validation errors, we recommend that you implement this change in a two-stage process. Verify that the first stage is complete before starting the second stage.

1. Change the attribute actions in all applications that read or write to the table. Deploy these changes and confirm that the update has been propagated to all destination hosts.
2. Write values to the new attribute in your table items.

This two-stage approach ensures that all applications and hosts have the same attribute actions, and will calculate the same signature, before any encounter the new attribute. This is important even when the action for the attribute is *Do nothing* (don't encrypt or sign), because the default for some encryptors is to encrypt and sign.

The following examples show the code for the first stage in this process. They add a new item attribute, `link`, which stores a link to another table item. Because this link must remain in plain text, the example assigns it the sign-only action. After fully deploying this change and then verifying that all applications and hosts have the new attribute actions, you can begin to use the `link` attribute in your table items.

Java DynamoDB Mapper

When using the `DynamoDB Mapper` and `AttributeEncryptor`, by default, all attributes are encrypted and signed except for primary keys, which are signed but not encrypted. To specify a sign-only action, use the `@DoNotEncrypt` annotation.

This example uses the `@DoNotEncrypt` annotation for the new `link` attribute.

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String link;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "link")
  @DoNotEncrypt
  public String getLink() {
    return link;
  }

  public void setLink(String link) {
    this.link = link;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ",
        sortAttribute=" + sortAttribute + ",
        link=" + link + "]";
  }
}
```

Java DynamoDB encryptor

In the lower-level DynamoDB encryptor, you must set actions for each attribute. This example uses a switch statement where the default is `encryptAndSign` and exceptions are specified for the partition key, sort key, and the new `link` attribute. In this example, if the link attribute code was not fully deployed before it was used, the link attribute would be encrypted and signed by some applications, but only signed by others.

```
for (final String attributeName : record.keySet()) {
    switch (attributeName) {
        case partitionKeyName:
            // fall through to the next case
        case sortKeyName:
            // partition and sort keys must be signed, but not encrypted
            actions.put(attributeName, signOnly);
            break;
        case "link":
            // only signed
            actions.put(attributeName, signOnly);
            break;
        default:
            // Encrypt and sign all other attributes
            actions.put(attributeName, encryptAndSign);
            break;
    }
}
```

Python

In the DynamoDB Encryption Client for Python, you can specify a default action for all attributes and then specify exceptions.

If you use a Python client helper class, you don't need to specify an attribute action for the primary key attributes. The client helper classes prevent you from encrypting your primary key. However, if you are not using a client helper class, you must set the SIGN_ONLY action on your partition key and sort key. If you accidentally encrypt your partition or sort key, you won't be able to recover your data without a full table scan.

This example specifies an exception for the new `link` attribute, which gets the `SIGN_ONLY` action.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

Removing an attribute

If you no longer need an attribute in items that have been encrypted with the DynamoDB Encryption Client, you can stop using the attribute. However, do not delete or change the action for that attribute. If you do, and then encounter an item with that attribute, the signature calculated for the item will not match the original signature, and the signature validation will fail.

Although you might be tempted to remove all traces of the attribute from your code, add a comment that the item is no longer used instead of deleting it. Even if you do a full table scan to delete all instances of the attribute, an encrypted item with that attribute might be cached or in process somewhere in your configuration.

Troubleshooting issues in your DynamoDB Encryption Client application

Note

Our client-side encryption library was renamed to AWS Database Encryption SDK. The following topic provides information on versions 1.*x*—2.*x* of the DynamoDB Encryption Client for Java and versions 1.*x*—3.*x* of the DynamoDB Encryption Client for Python. For more information, see AWS Database Encryption SDK for DynamoDB version support.

This section describes problems that you might encounter when using the DynamoDB Encryption Client and offers suggestions for resolving them.

To provide feedback on the DynamoDB Encryption Client, file an issue in the aws-dynamodb-encryption-java or aws-dynamodb-encryption-python GitHub repository.

To provide feedback on this documentation, use the feedback link on any page.

Topics

- Access denied
- Signature verification fails
- Issues with older version global tables
- Poor performance of the Most Recent Provider

Access denied

Problem: Your application is denied access to a resource that it needs.

Suggestion: Learn about the required permissions and add them to the security context in which your application runs.

Details

To run an application that uses the a DynamoDB Encryption Client library, the caller must have permission to use its components. Otherwise, they will be denied access to the required elements.

- The DynamoDB Encryption Client does not require an Amazon Web Services (AWS) account or depend on any AWS service. However, if your application uses AWS, you need an AWS account and users who have permission to use the account.
- The DynamoDB Encryption Client does not require Amazon DynamoDB. However, If the application that uses the client creates DynamoDB tables, puts items into a table, or gets items from a table, the caller must have permission to use the required DynamoDB operations in your AWS account. For details, see the access control topics in the *Amazon DynamoDB Developer Guide*.
- If your application uses a client helper class in the DynamoDB Encryption Client for Python, the caller must have permission to call the DynamoDB DescribeTable operation.
- The DynamoDB Encryption Client does not require AWS Key Management Service (AWS KMS). However, if your application uses a Direct KMS Materials Provider, or it uses a Most Recent

Provider with a provider store that uses AWS KMS, the caller must have permission to use the AWS KMS GenerateDataKey and Decrypt operations.

Signature verification fails

Problem: An item cannot be decrypted because signature verification fails. The item also might not be encrypted and signed as you intend.

Suggestion: Be sure that the attribute actions that you provide account for all attributes in the item. When decrypting an item, be sure to provide attribute actions that match the actions used to encrypt the item.

Details

The attribute actions that you provide tell the DynamoDB Encryption Client which attributes to encrypt and sign, which attributes to sign (but not encrypt), and which to ignore.

If the attribute actions that you specify do not account for all attributes in the item, the item might not be encrypted and signed the way that you intend. If the attribute actions that you provide when decrypting an item differ from the attribute actions that you provided when encrypting the item, the signature verification might fail. This is a particular problem for distributed applications in which new attribute actions might not have propagated to all hosts.

Signature validation errors are difficult to resolve. For help preventing them, take extra precautions when changing your data model. For details, see Changing your data model.

Issues with older version global tables

Problem: Items in an older version Amazon DynamoDB global table cannot be decrypted because signature verification fails.

Suggestion: Set attribute actions so the reserved replication fields are not encrypted or signed.

Details

You can use the DynamoDB Encryption Client with DynamoDB global tables. We recommend that you use global tables with a multi-Region KMS key and replicate the KMS key into all AWS Regions where the global table is replicated.

Beginning with global tables version 2019.11.21, you can use global tables with the DynamoDB Encryption Client without any special configuration. However, if you use global tables version 2017.11.29, you must ensure that reserved replication fields are not encrypted or signed.

If you are using the global tables version 2017.11.29, you must set the attribute actions for the following attributes to `DO_NOTHING` in Java or `@DoNotTouch` in Python.

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

If you are using any other version of global tables, no action is required.

Poor performance of the Most Recent Provider

Problem: Your application is less responsive, especially after updating to a newer version of the DynamoDB Encryption Client.

Suggestion: Adjust the time-to-live value and cache size.

Details

The Most Recent Provider is designed to improve the performance of applications that use the DynamoDB Encryption Client by allowing limited reuse of cryptographic materials. When you configure the Most Recent Provider for your application, you have to balance improved performance with the security concerns that arise from caching and reuse.

In newer versions of the DynamoDB Encryption Client, the time-to-live (TTL) value determines how long cached cryptographic material providers (CMPs) can be used. The TTL also determines how often the Most Recent Provider checks for a new version of the CMP.

If your TTL is too long, your application might violate your business rules or security standards. If your TTL is too brief, frequent calls to the provider store can cause your provider store to throttle requests from your application and other applications that share your service account. To resolve this issue, adjust the TTL and cache size to a value that meets your latency and availability goals and conforms to your security standards. For details, see Setting a time-to-live value.

Amazon DynamoDB Encryption Client rename

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

On June 9, 2023, our client-side encryption library was renamed to AWS Database Encryption SDK. The AWS Database Encryption SDK is compatible with Amazon DynamoDB. It can decrypt and read items encrypted by the legacy DynamoDB Encryption Client. For more information on the legacy DynamoDB Encryption Client versions, see AWS Database Encryption SDK for DynamoDB version support.

The AWS Database Encryption SDK provides version 3.*x* of the Java client-side encryption library for DynamoDB, which is a major rewrite of the DynamoDB Encryption Client for Java. It includes many updates, such as a new structured data format, improved multitenancy support, seamless schema changes, and searchable encryption support.

To learn more about the new features introduced with the AWS Database Encryption SDK, see the following topics.

Searchable encryption

You can design databases that can search encrypted records without decrypting the entire database. Depending on your threat model and query requirements, you can use searchable encryption to perform exact match searches or more customized complex queries on your encrypted records.

Keyrings

The AWS Database Encryption SDK uses *keyrings* to perform envelope encryption. Keyrings generate, encrypt, and decrypt the data keys that protect your records. The AWS Database Encryption SDK supports AWS KMS keyrings that use symmetric encryption or asymmetric RSA AWS KMS keys to protect your data keys, and AWS KMS Hierarchical keyrings that enable you to protect your cryptographic materials under a symmetric encryption KMS key without calling AWS KMS every time you encrypt or decrypt a record. You can also specify your own key material with Raw AES keyrings and Raw RSA keyrings.

Seamless schema changes

When you configure the AWS Database Encryption SDK, you provide cryptographic actions that tell the client which fields to encrypt and sign, which fields to sign (but not encrypt), and which to ignore. After you have used the AWS Database Encryption SDK to protect your records, you can still make changes to your data model. You can update your cryptographic actions, such as adding or removing encrypted fields, in a single deployment.

Configure existing DynamoDB tables for client-side encryption

Legacy versions of the DynamoDB Encryption Client were designed to be implemented in new, unpopulated tables. With the AWS Database Encryption SDK for DynamoDB, you can migrate your existing Amazon DynamoDB tables to version 3.*x* of the Java client-side encryption library for DynamoDB.

Reference

Our client-side encryption library was renamed to the AWS Database Encryption SDK. This developer guide still provides information on the DynamoDB Encryption Client.

The following topics provide technical details for the AWS Database Encryption SDK.

Material description format

The material description serves as the header for an encrypted record. When you encrypt and sign fields with the AWS Database Encryption SDK, the encryptor records the material description as it assembles the cryptographic materials and stores the material description in a new field (`aws_dbe_head`) that the encryptor adds to your record. The material description is a portable formatted data structure that contains the encrypted data key and information about how the record was encrypted and signed. The following table describes the values that form the material description. The bytes are appended in the order shown.

Value	Length in bytes
Version	1
Signatures Enabled	1
Record ID	32
Encrypt Legend	Variable
Encryption Context Length	2
???	Variable
Encrypted Data Key Count	1
Encrypted Data Keys	Variable
Record Commitment	1

Version

The version of this `aws_dbe_head` field's format.

Signatures Enabled

Encodes whether ECDSA digital signatures are enabled for this record.

Byte value	Meaning
`0x01`	ECDSA digital signatures enabled (default)
`0x00`	ECDSA digital signatures disabled

Record ID

A randomly generated 256-bit value that identifies the record. The Record ID:

- Uniquely identifies the encrypted record.
- Binds the material description to the encrypted record.

Encrypt Legend

A serialized description of which authenticated fields were encrypted. The Encrypt Legend is used to determine what fields the decryption method should attempt to decrypt.

Byte value	Meaning
`0x65`	`ENCRYPT_AND_SIGN`
`0x73`	`SIGN_ONLY`

The Encrypt Legend is serialized as follows:

1. Lexicographically by the byte sequence that represents their canonical path.
2. For each field, in order, append one of the byte values specified above to indicate whether that field should be encrypted.

Encryption Context Length

The length of the encryption context. It is a 2-byte value interpreted as a 16-bit unsigned integer. The maximum length is 65,535 bytes.

Encryption Context

A set of name-value pairs that contain arbitrary, non-secret additional authenticated data.

When ECDSA digital signatures are enabled, the encryption context contains the key-value pair `{"aws-crypto-footer-ecdsa-key": Qtxt}`. `Qtxt` represents the elliptic curve point `Q` compressed according to SEC 1 version 2.0 and then base64-encoded.

Encrypted Data Key Count

The number of encrypted data keys. It is a 1-byte value interpreted as a 8-bit unsigned integer that specifies the number of encrypted data keys. The maximum number of encrypted data keys in each record is 255.

Encrypted Data Keys

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

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

Encrypted Data Key Structure

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

Key Provider ID Length

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

Key Provider ID

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

Key Provider Information Length

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

Key Provider Information

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

When you are using an AWS KMS keyring, this value contains the Amazon Resource Name (ARN) of the AWS KMS key.

Encrypted Data Key Length

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

Encrypted Data Key

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

Record Commitment

A distinct 256-bit Hash-Based Message Authentication Code (HMAC) hash calculated over all preceding material description bytes using the commit key.

AWS KMS Hierarchical keyring technical details

The AWS KMS Hierarchical keyring uses a unqiue data key to encrypt each field and encrypts each data key with a unique wrapping key derived from an active branch key. It uses a key derivation in counter mode with a pseudorandom function with HMAC SHA-256 to derive the 32 byte wrapping key with the following inputs.

- A 16 byte random salt
- The active branch key
- The UTF-8 encoded value for the key provider identifier "aws-kms-hierarchy"

The Hierarchical keyring uses the derived wrapping key to encrypt a copy of the plaintext data key using AES-GCM-256 with a 16 byte authentication tag and the following inputs.

- The derived wrapping key is used as the AES-GCM cipher key
- The data key is used as the AES-GCM message
- A 12 byte random initialization vector (IV) is used as the AES-GCM IV
- Additional authenticated data (AAD) containing the following serialized values.

Value	Length in bytes	Interpreted as
"aws-kms-hierarchy"	17	UTF-8 encoded
The branch key identifier	Variable	UTF-8 encoded
The branch key version	16	UTF-8 encoded
Encryption context	Variable	UTF-8 encoded key value pairs

Document history for the AWS Database Encryption SDK Developer Guide

The following table describes significant changes to this documentation. In addition to these major changes, we also update the documentation frequently to improve the descriptions and examples, and to address the feedback that you send to us. To be notified about significant changes, subscribe to the RSS feed.

Change	Description	Date
New feature	Added documentation for the AWS KMS ECDH keyring and Raw ECDH keyring.	June 17, 2024
General Availability (GA) release	Introducing support for the .NET client-side encryption library for DynamoDB.	January 17, 2024
General Availability (GA) release	Updated documentation for the GA release of version 3.*x* of the Java client-side encryption library for DynamoDB. **Warning** Branch keys created during the developer preview release are no longer supported.	July 24, 2023
Rebrand of DynamoDB Encryption Client	The client-side encryption library is renamed to AWS Database Encryption SDK.	June 9, 2023

Preview release	Added and updated documentation for version 3.*x* of the Java client-side encryption library for DynamoDB, which includes a new structured data format, improved multitenancy support, seamless schema changes, and searchable encryption support.	June 9, 2023
Documentation change	Replace the AWS Key Management Service term *customer master key (CMK)* with *AWS KMS key* and *KMS key*.	August 30, 2021
New feature	Added support for AWS Key Management Service (AWS KMS) multi-Region keys. Multi-Region keys are AWS KMS keys in different AWS Regions that can be used interchangeably because they have the same key ID and key material.	June 8, 2021
New example	Added example of using the DynamoDBMapper in Java.	September 6, 2018
Python support	Added support for Python, in addition to Java.	May 2, 2018
Initial release	Initial release of this documentation.	May 2, 2018