aws

Guia do Desenvolvedor

AWS SDK de criptografia de banco de dados da

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

AWS SDK de criptografia de banco de dados da: Guia do Desenvolvedor

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

As marcas comerciais e imagens comerciais da Amazon não podem ser usadas no contexto de nenhum produto ou serviço que não seja da Amazon, nem de qualquer maneira que possa gerar confusão entre os clientes ou que deprecie ou desprestigie a Amazon. Todas as outras marcas comerciais que não pertencem à Amazon pertencem a seus respectivos proprietários, que podem ou não ser afiliados, patrocinados pela Amazon ou ter conexão com ela.

Table of Contents

O que é o SDK AWS de criptografia de banco de dados? ... 1
Desenvolvido em repositórios de código aberto ... 3
Suporte e manutenção ... 3
Enviar comentários ... 4
Conceitos ... 4
criptografia envelopada ... 5
Chave de dados ... 7
Chave de empacotamento ... 8
Tokens de autenticação ... 9
Ações criptográficas ... 9
Descrição do material ... 10
Contexto de criptografia ... 11
Gerenciador de material de criptografia ... 11
Criptografia simétrica e assimétrica ... 12
Confirmação de chave ... 12
Assinaturas digitais ... 13
Como funciona ... 15
Criptografar e assinar ... 16
Descriptografar e verificar ... 17
Pacotes de algoritmos compatíveis ... 18
Conjunto de algoritmos padrão ... 21
AES-GCM sem assinaturas digitais ECDSA ... 22
Interagindo com AWS KMS ... 24
Como configurar o SDK ... 26
Seleção de uma linguagem de programação ... 26
Seleção de chaves de encapsulamento ... 26
Criação de um filtro de descoberta ... 28
Trabalhar com bancos de dados multilocatários ... 30
Criação de beacons assinados ... 30
Repositórios de chaves ... 38
Principais conceitos e terminologia da loja ... 38
Implementação de permissões de privilégio mínimo ... 39
Crie um armazenamento de chaves ... 40
Configurar as principais ações do armazenamento ... 42

Configure suas principais ações de armazenamento ... 43
Criar chaves de ramificação ... 45
Alternar a chave de ramificação ativa ... 49
Tokens de autenticação ... 52
Como os tokens de autenticação funcionam ... 53
AWS KMS chaveiros ... 54
Permissões necessárias para tokens de autenticação do AWS KMS ... 55
Identificação AWS KMS keys em um AWS KMS chaveiro ... 56
Criando um AWS KMS chaveiro ... 57
Usando a multirregião AWS KMS keys ... 60
Usando um chaveiro AWS KMS Discovery ... 62
Usando um chaveiro de descoberta AWS KMS regional ... 65
AWS KMS Chaveiros hierárquicos ... 67
Como funciona ... 70
Pré-requisitos ... 72
Permissões obrigatórias ... 72
Escolha um cache ... 73
Criar um token de autenticação hierárquico ... 82
Uso do token de autenticação hierárquico para criptografia pesquisável ... 89
AWS KMS chaveiros ECDH ... 93
Permissões necessárias para AWS KMS chaveiros ECDH ... 94
Criando um AWS KMS chaveiro ECDH ... 94
Criando um AWS KMS chaveiro de descoberta ECDH ... 98
Tokens de autenticação AES Raw ... 101
Tokens de autenticação brutos do RSA ... 104
Chaveiros ECDH brutos ... 107
Criando um chaveiro ECDH bruto ... 108
Multitokens de autenticação ... 118
Criptografia pesquisável ... 122
Os beacons são adequados para meu conjunto de dados? ... 123
Cenário de criptografia pesquisável ... 126
Beacons ... 128
Beacons padrão ... 129
Beacons compostos ... 130
Planejar beacons ... 131
Considerações para bancos de dados multilocatários ... 133

Escolha de um tipo de beacon ... 133
Escolher um comprimento de beacon ... 140
Escolher um nome de beacon ... 147
Configurar beacons ... 148
Configurando beacons padrão ... 149
Configuração de beacons compostos ... 158
Exemplos de configuração ... 169
Uso de beacons ... 173
Consultar beacons ... 176
Criptografia pesquisável para bancos de dados multilocatários ... 178
Consultar beacons em um banco de dados multilocatário ... 181
Amazon DynamoDB ... 183
Criptografia do lado do cliente e do lado do servidor ... 184
Quais campos são criptografados e assinados? ... 186
Criptografar valores de atributos ... 187
Assinar o item ... 188
Criptografia pesquisável no DynamoDB ... 188
Configuração de índices secundários com beacons ... 189
Testando saídas de farol ... 190
Atualizar seu modelo de dados ... 196
Adicionar novos SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT atributos ENCRYPT_AND_SIGNSIGN_ONLY, e ... 198
Remover atributos existentes ... 199
Alterar um ENCRYPT_AND_SIGN atributo existente para SIGN_ONLY ou SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT ... 199
Alterar um existente SIGN_ONLY ou um SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT atributo para ENCRYPT_AND_SIGN ... 200
Adicionar um novo atributo DO_NOTHING ... 201
Alterar um atributo SIGN_ONLY existente para SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT ... 202
Alterar um atributo SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT existente para SIGN_ONLY ... 202
Linguagens de programação ... 203
Java ... 203
.NET ... 239
Rust ... 256

Legado 262
AWS Suporte à versão SDK de criptografia de banco de dados para DynamoDB 263
Como funciona 263
Conceitos 267
Provedor de materiais de criptografia 272
Linguagens de programação 303
Alterar seu modelo de dados 331
Solução de problemas 336
Renomeação do DynamoDB Encryption Client 340
Referência 342
Formato de descrição do material 342
AWS KMS Detalhes técnicos do chaveiro hierárquico 346
Histórico de documentos 348
..... cccli

O que é o SDK AWS de criptografia de banco de dados?

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados é um conjunto de bibliotecas de software que permitem incluir criptografia do lado do cliente no design do banco de dados. O SDK AWS de criptografia de banco de dados fornece soluções de criptografia em nível de registro. Especifique quais campos são criptografados e quais campos são incluídos nas assinaturas para garantir a autenticidade dos seus dados. Criptografar dados em trânsito e em repouso confidenciais ajuda você a garantir que os dados em texto simples não estejam disponíveis a terceiros, incluindo à AWS. O SDK de criptografia de banco de dados da AWS é fornecido gratuitamente sob a licença do Apache 2.0.

Este guia do desenvolvedor fornece uma visão geral conceitual do SDK de criptografia de AWS banco de dados, incluindo uma introdução à sua arquitetura, detalhes sobre como ele protege seus dados, como ele difere da criptografia do lado do servidor e orientação sobre como selecionar componentes essenciais para seu aplicativo para ajudá-lo a começar.

O SDK AWS de criptografia de banco de dados é compatível com o Amazon DynamoDB com criptografia em nível de atributo.

O SDK AWS de criptografia de banco de dados tem os seguintes benefícios:

Projetado especialmente para aplicativos de banco de dados

> Você não precisa ser um especialista em criptografia para usar o SDK de criptografia de AWS banco de dados. As implementações incluem métodos de ajuda que são projetados para trabalhar com os seus aplicativos atuais.
>
> Após criar e configurar os componentes necessários, o cliente de criptografia criptografa e assina de modo transparente os registros, quando você os adiciona a um banco de dados, e os verifica e os descriptografa quando você os recupera.

Inclui criptografia e assinatura seguras

O SDK AWS de criptografia de banco de dados inclui implementações seguras que criptografam os valores de campo em cada registro usando uma chave de criptografia de dados exclusiva e, em seguida, assinam o registro para protegê-lo contra alterações não autorizadas, como adicionar ou excluir campos ou trocar valores criptografados.

Usa materiais de criptografia de qualquer origem

O SDK AWS de criptografia de banco de dados usa chaveiros para gerar, criptografar e descriptografar a chave exclusiva de criptografia de dados que protege seu registro. Os tokens de autenticação determinam as chaves de empacotamento que criptografam essa chave de dados.

É possível usar chaves de empacotamento de qualquer fonte, incluindo serviços de criptografia, como AWS Key Management Service (AWS KMS) ou AWS CloudHSM. O SDK AWS de criptografia de banco de dados não requer um Conta da AWS ou nenhum AWS serviço.

Suporte para armazenamento em cache de materiais criptográficos

O chaveiro AWS KMS hierárquico é uma solução de armazenamento em cache de materiais criptográficos que reduz o número de AWS KMS chamadas usando chaves de ramificação AWS KMS protegidas persistentes em uma tabela do Amazon DynamoDB e, em seguida, armazenando localmente em cache materiais de chave de ramificação usados em operações de criptografia e descriptografia. Ele permite que você proteja seus materiais criptográficos sob uma chave KMS de criptografia simétrica sem ligar AWS KMS toda vez que você criptografa ou descriptografa um registro. O AWS KMS chaveiro hierárquico é uma boa opção para aplicativos que precisam minimizar as chamadas para. AWS KMS

Criptografia pesquisável

É possível criar bancos de dados capazes de pesquisar registros criptografados sem descriptografar o banco de dados inteiro. Dependendo do modelo de ameaça e dos requisitos de consulta, você pode usar criptografia pesquisável para realizar pesquisas de correspondência exata ou consultas complexas mais personalizadas em seu banco de dados criptografado.

Suporte para esquemas de banco de dados multilocatário

O SDK AWS de criptografia de banco de dados permite que você proteja os dados armazenados em bancos de dados com um esquema compartilhado, isolando cada inquilino com materiais de criptografia distintos. Se você tiver vários usuários executando operações de criptografia em seu banco de dados, use um dos AWS KMS chaveiros para fornecer a cada usuário uma

chave distinta para usar em suas operações criptográficas. Para obter mais informações, consulte Trabalhar com bancos de dados multilocatários.

Suporte para atualizações de esquemas simplificadas

Ao configurar o SDK AWS de criptografia de banco de dados, você fornece ações criptográficas que informam ao cliente quais campos criptografar e assinar, quais campos assinar (mas não criptografar) e quais ignorar. Depois de usar o SDK de criptografia de banco de dados da AWS para proteger seus registros, você ainda pode fazer alterações no seu modelo de dados. É possível atualizar ações criptográficas, como adicionar ou remover campos criptografados, em uma única implantação.

Desenvolvido em repositórios de código aberto

O SDK AWS de criptografia de banco de dados é desenvolvido em repositórios de código aberto no. GitHub É possível usar esses repositórios para visualizar o código, ler e enviar problemas e encontrar informações específicas para sua implementação.

O SDK AWS de criptografia de banco de dados para DynamoDB

- O repositório aws-database-encryption-sdk-dynamodb on GitHub oferece suporte às versões mais recentes do SDK de criptografia de AWS banco de dados para DynamoDB em Java, .NET e Rust.

  O SDK AWS de criptografia de banco de dados para DynamoDB é um produto da Dafny, uma linguagem com reconhecimento de verificação na qual você escreve especificações, o código para implementá-las e as provas para testá-las. O resultado é uma biblioteca que implementa os recursos do SDK de criptografia de banco de dados da AWS para DynamoDB em uma estrutura que garante a correção funcional.

Suporte e manutenção

O SDK AWS de criptografia de banco de dados usa a mesma política de manutenção que o AWS SDK e as ferramentas usam, incluindo suas fases de controle de versão e ciclo de vida. Como prática recomendada, você deve usar a versão mais recente do SDK de criptografia de banco de dados da AWS para sua linguagem de programação e atualizá-la à medida que novas versões forem lançadas.

Para obter mais informações, consulte a política de manutenção de ferramentas AWS SDKs AWS SDKs e ferramentas no Guia de referência de ferramentas.

Enviar comentários

Os seus comentários são bem-vindos. Se você tiver uma pergunta ou comentário, ou um problema a relatar, use os seguintes recursos.

Se você descobrir uma possível vulnerabilidade de segurança no SDK AWS de criptografia de banco de dados, notifique a AWS segurança. Não crie um GitHub problema público.

Para fornecer feedback sobre esta documentação, use o link de feedback em qualquer página.

AWS Conceitos do SDK de criptografia de banco de dados

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Este tópico explica os conceitos e a terminologia usados no SDK de criptografia de AWS banco de dados.

Para saber como os componentes do SDK de criptografia AWS de banco de dados interagem, consulteComo funciona o SDK AWS de criptografia de banco de dados.

Para saber mais sobre o SDK AWS de criptografia de banco de dados, consulte os tópicos a seguir.

- Saiba como o SDK AWS de criptografia de banco de dados usa criptografia de envelope para proteger seus dados.
- Saiba mais sobre os elementos da criptografia envelopada: as chaves de dados que protegem seus registros e as chaves de empacotamento que protegem suas chaves de dados.
- Saiba mais sobre os tokens de autenticação que determinam quais chaves de empacotamento você usa.
- Saiba mais sobre o contexto de criptografia que adiciona integridade ao seu processo de criptografia.
- Saiba mais sobre a descrição do material que os métodos de criptografia adicionam ao seu registro.
- Saiba mais sobre as ações criptográficas que informam ao SDK de criptografia de banco de dados da AWS quais campos criptografar e assinar.

Tópicos

- criptografia envelopada
- Chave de dados
- Chave de empacotamento
- Tokens de autenticação
- Ações criptográficas
- Descrição do material
- Contexto de criptografia
- Gerenciador de material de criptografia
- Criptografia simétrica e assimétrica
- Confirmação de chave
- Assinaturas digitais

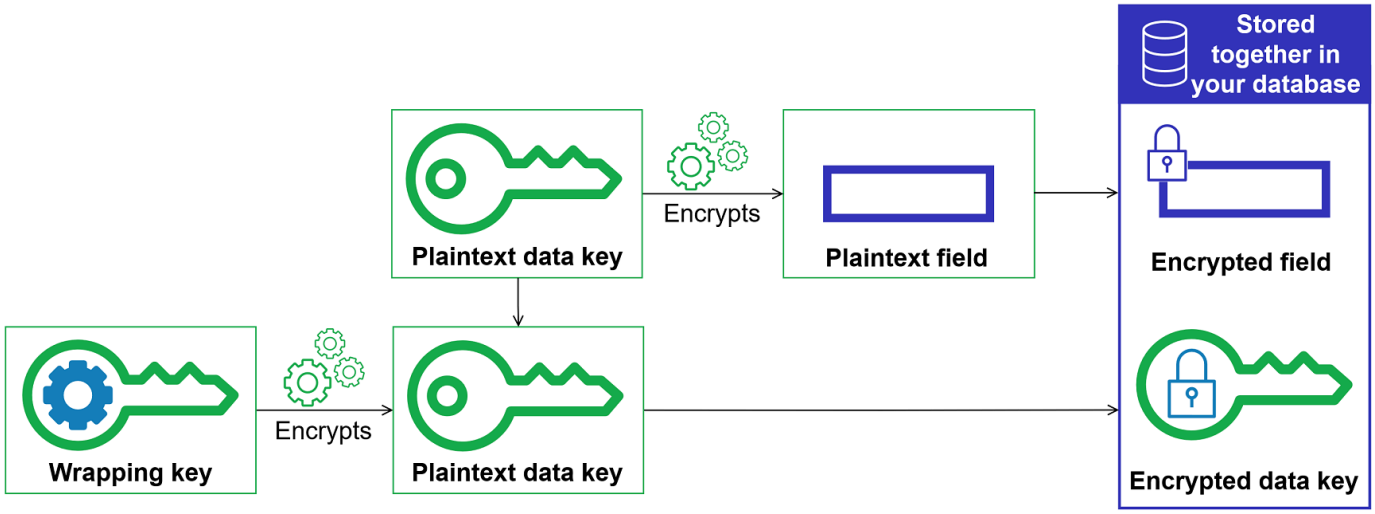
criptografia envelopada

A segurança dos dados criptografados depende em parte da proteção da chave de dados que pode descriptografá-los. Uma prática recomendada aceita para proteger a chave de dados é criptografá-la. Para fazer isso, você precisa de outra chave de criptografia, conhecida como chave de criptografia de chave ou chave de encapsulamento. Essa prática de uso de uma chave do KMS para criptografar chaves de dados é conhecida como criptografia envelopada.

Proteção de chaves de dados

O SDK AWS de criptografia de banco de dados criptografa cada campo com uma chave de dados exclusiva. Em seguida, ele criptografa cada chave de dados sob a chave de empacotamento especificada. Ele armazena as chaves de dados criptografadas na descrição do material.

Para especificar a chave de empacotamento, use um token de autenticação.



Criptografar os mesmos dados com várias chaves de empacotamento

É possível criptografar a chave de dados com várias chaves de empacotamento. Talvez você queira fornecer chaves de empacotamento distintas para usuários diferentes ou chaves de empacotamento de tipos variados ou em locais diferentes. Cada uma das chaves de encapsulamento criptografa a mesma chave de dados. O SDK AWS de criptografia de banco de dados armazena todas as chaves de dados criptografadas junto com os campos criptografados na descrição do material.

Para descriptografar os dados, você precisa fornecer pelo menos uma chave de empacotamento que possa descriptografar as chaves de dados criptografadas.

Combinação de pontos fortes de vários algoritmos

Para criptografar seus dados, por padrão, o SDK de criptografia de AWS banco de dados usa um conjunto de algoritmos com criptografia simétrica AES-GCM, uma função de derivação de chave (HKDF) baseada em HMAC e assinatura ECDSA. Para criptografar a chave de dados, você pode especificar um algoritmo de criptografia simétrico ou assimétrico apropriado à sua chave de encapsulamento.

Em geral, os algoritmos de criptografia de chaves simétricas são mais rápidos e produzem textos cifrados menores que a criptografia de chave pública ou assimétrica. No entanto, os algoritmos de chave pública fornecem separação inerente de funções. Para combinar os pontos fortes de cada um, você pode criptografar a chave de dados com a criptografia de chave pública.

Recomendamos usar um dos AWS KMS chaveiros sempre que possível. Ao usar o AWS KMS chaveiro, você pode escolher combinar os pontos fortes de vários algoritmos especificando um

AWS KMS key RSA assimétrico como sua chave de agrupamento. Também é possível usar uma chave do KMS de criptografia simétrica.

Chave de dados

Uma chave de dados é uma chave de criptografia que o SDK AWS de criptografia de banco de dados usa para criptografar os campos em um registro que estão marcados `ENCRYPT_AND_SIGN` nas ações criptográficas. Cada chave de dados é uma matriz de bytes que cumpre os requisitos para chaves criptográficas. O SDK AWS de criptografia de banco de dados usa uma chave de dados exclusiva para criptografar cada atributo.

Você não precisa especificar, gerar, implementar, estender, proteger nem usar chaves de dados. O SDK de criptografia de banco de dados da AWS faz esse trabalho para você quando você chama as operações de criptografia e descriptografia.

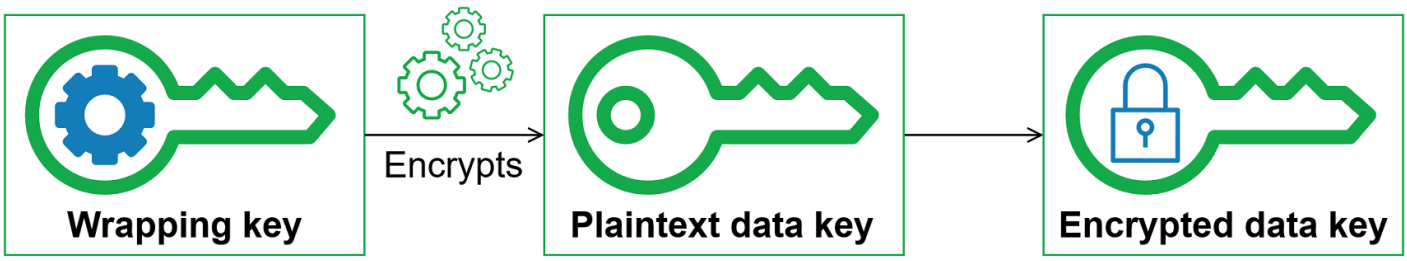
Para proteger suas chaves de dados, o SDK AWS de criptografia de banco de dados as criptografa sob uma ou mais chaves de criptografia de chave conhecidas como chaves de encapsulamento. Depois que o SDK AWS de criptografia de banco de dados usa suas chaves de dados em texto simples para criptografar seus dados, ele os remove da memória assim que possível. Em seguida, ele armazena as chaves de dados criptografadas na descrição do material. Para obter detalhes, consulte Como funciona o SDK AWS de criptografia de banco de dados.

> **Tip**
>
> No SDK AWS de criptografia de banco de dados, distinguimos as chaves de dados das chaves de criptografia de dados. Como prática recomendada, todos os conjuntos de algoritmos compatíveis devem usar uma função de derivação de chave. A função de derivação de chaves usa a chave de dados como entrada e retorna uma chave de criptografia de dados que é realmente usada para criptografar os registros. Por esse motivo, sempre dizemos que os dados são criptografados "sob" uma chave de dados em vez de "pela" chave de dados.

Cada chave de dados criptografada inclui metadados, incluindo o identificador da chave de encapsulamento que a criptografou. Esses metadados possibilitam que o SDK de criptografia AWS de banco de dados identifique chaves de encapsulamento válidas durante a descriptografia.

Chave de empacotamento

Uma chave de empacotamento é uma chave de criptografia que o SDK de criptografia de banco de dados da AWS usa para criptografar a chave de dados que criptografa seus registros. Cada chave de dados em texto simples pode ser criptografada sob uma ou mais chaves mestras. Você determina quais chaves de empacotamento são usadas para proteger seus dados ao configurar um token de autenticação.



O SDK AWS de criptografia de banco de dados oferece suporte a várias chaves de agrupamento comumente usadas, como AWS Key Management Service(AWS KMS) chaves KMS de criptografia simétrica (incluindo chaves multirregionais) e chaves RSA KMS assimétricas, AWS KMS chaves brutas AES-GCM (Advanced Encryption Standard/Galois Counter Mode) e chaves RSA brutas. Recomendamos utilizar chaves do KMS sempre que possível. Para decidir qual chave de empacotamento você deve usar, consulte Selecting wrapping keys.

Quando você usa a criptografia envelopada, você precisa proteger suas chaves de empacotamento contra acesso não autorizado. É possível fazer isso de uma das seguintes maneiras:

- Use um serviço projetado para essa finalidade, como o AWS Key Management Service (AWS KMS).
- Use um hardware security module (HSM - módulo de segurança de hardware), como os oferecidos pelo AWS CloudHSM.
- Use outras ferramentas e serviços de gerenciamento de chaves.

Se você não tem um sistema de gerenciamento de chaves, recomendamos AWS KMS. O SDK AWS de criptografia de banco de dados se integra AWS KMS para ajudar você a proteger e usar suas chaves de empacotamento.

Tokens de autenticação

Para especificar as chaves de empacotamento que você usa para criptografia e decodificação, use um token de autenticação. Você pode usar os chaveiros fornecidos pelo SDK do AWS Database Encryption ou criar suas próprias implementações.

Um token de autenticação gera, criptografa e descriptografa chaves de dados. Ele também gera as chaves MAC usadas para calcular os Códigos de Autenticação de Mensagens Baseados em Hash (HMACs) na assinatura. Ao definir um token de autenticação, você pode especificar as chaves de encapsulamento que criptografam suas chaves de dados. A maioria dos tokens de autenticação especificam pelo menos uma chave de encapsulamento ou um serviço que fornece e protege chaves de encapsulamento. Ao criptografar, o SDK AWS de criptografia de banco de dados usa todas as chaves de encapsulamento especificadas no chaveiro para criptografar a chave de dados. Para obter ajuda sobre como escolher e usar os chaveiros definidos pelo SDK do AWS Database Encryption, consulte Como usar chaveiros.

Ações criptográficas

As ações criptográficas informam ao criptografador quais ações devem ser executadas em cada campo em um registro.

Os valores das ações de atributo podem ser um destes:

- Criptografar e assinar: criptografa o campo. Inclua o campo criptografado na assinatura.
- Somente assinar: inclui o campo na assinatura.
- Assinar e incluir no contexto de criptografia — Inclua o campo no contexto de assinatura e criptografia.

  Por padrão, as chaves de partição e classificação são o único atributo incluído no contexto de criptografia. Você pode considerar definir campos adicionais para que o fornecedor da ID da chave de filial do seu AWS KMS chaveiro hierárquico possa identificar qual chave de ramificação é necessária para a descriptografia a partir `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` do contexto de criptografia. Para obter mais informações, consulte fornecedor de ID de chave de filial.

> **Note**
>
> Para usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica, você deve usar a versão 3.3 ou posterior do SDK de criptografia de AWS banco de dados.

Implante a nova versão para todos os leitores antes de atualizar seu modelo de dados para incluí-laSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT.

- Não fazer nada: não criptografa nem inclui o campo na assinatura.

Para qualquer campo que possa armazenar dados confidenciais, use Criptografar e assinar. Para valores de chave primária (por exemplo, uma chave de partição e uma chave de classificação em uma tabela do DynamoDB), use Somente assinar ou Assinar e incluir no contexto de criptografia. Se você especificar qualquer sinal e incluir atributos no contexto de criptografia, os atributos de partição e classificação também deverão ser Assinar e incluir no contexto de criptografia. Não é necessário especificar ações criptográficas para a descrição do material. O SDK AWS de criptografia de banco de dados assina automaticamente o campo em que a descrição do material está armazenada.

Escolha suas ações criptográficas com cuidado. Em caso de dúvida, use Criptografar e assinar. Depois de usar o SDK AWS de criptografia de banco de dados para proteger seus registros, você não pode alterar um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo existente `ENCRYPT_AND_SIGN` ou alterar a `DO_NOTHING` ação criptográfica atribuída a um campo existente`DO_NOTHING`. `SIGN_ONLY` No entanto, você ainda pode fazer outras alterações em seu modelo de dados. Por exemplo, você pode adicionar ou remover campos criptografados em uma única implantação.

Descrição do material

A descrição do material serve como cabeçalho para um registro criptografado. Quando você criptografa e assina campos com o SDK AWS de criptografia de banco de dados, o criptografador registra a descrição do material à medida que reúne os materiais criptográficos e armazena a descrição do material em um novo campo (`aws_dbe_head`) que o criptografador adiciona ao seu registro.

A descrição do material é uma estrutura de dados formatada portátil que contém cópias criptografadas das chaves de dados e outras informações, como algoritmos de criptografia, contexto de criptografia e instruções de criptografia e assinatura. O criptografador registra a descrição do material à medida que monta os materiais para criptografia e assinatura. Depois, quando precisar montar materiais de criptografia para verificar e descriptografar um campo, ele usará a descrição do material como guia.

O armazenamento de dados criptografados junto com o campo criptografado simplifica a operação e elimina a necessidade de armazenar e gerenciar chaves de dados criptografadas independentemente dos dados que elas criptografam.

Para obter informações técnicas sobre a descrição do material, consulte Formato de descrição do material.

Contexto de criptografia

Para melhorar a segurança de suas operações criptográficas, o SDK AWS de criptografia de banco de dados inclui um contexto de criptografia em todas as solicitações para criptografar e assinar um registro.

Um contexto de criptografia é um conjunto de pares de chave-valor que contêm dados autenticados adicionais arbitrários e não secretos. O SDK AWS de criptografia de banco de dados inclui o nome lógico do seu banco de dados e os valores da chave primária (por exemplo, uma chave de partição e uma chave de classificação em uma tabela do DynamoDB) no contexto de criptografia. Quando você criptografa e assina um campo, o contexto de criptografia é associado de maneira criptográfica aos registros criptografados de forma que o mesmo contexto de criptografia seja necessário para descriptografar os campos.

Se você usa um AWS KMS chaveiro, o SDK AWS de criptografia de banco de dados também usa o contexto de criptografia para fornecer dados autenticados adicionais (AAD) nas chamadas para as quais o chaveiro faz. AWS KMS

Sempre que você usar um algoritmo de criptografia com assinatura, o gerenciador de material de criptografia (CMM) adicionará um par de nome/valor ao contexto de criptografia consistindo em um nome reservado, `aws-crypto-public-key`, e um valor representando a chave de verificação pública. A chave de verificação pública é armazenada na descrição do material.

Gerenciador de material de criptografia

O gerenciador de material de criptografia (CMM) monta o material criptográfico usado para criptografar, descriptografar e assinar dados. Sempre que você usa o conjunto de algoritmos padrão, os materiais criptográficos incluem texto simples e chaves de dados criptografadas, chaves de assinatura simétricas e uma chave de assinatura assimétrica. Você nunca interage diretamente com o CMM. Os métodos de criptografia e descriptografia o processam para você.

Como o CMM atua como uma ligação entre o SDK de criptografia AWS de banco de dados e um chaveiro, é um ponto ideal para personalização e extensão, como suporte para aplicação de

políticas. É possível especificar explicitamente um CMM, mas isso não é obrigatório. Quando você especifica um token de autenticação, o SDK de criptografia de banco de dados da AWS cria um CMM padrão para você. O CMM padrão obtém o material de criptografia ou de descriptografia do token de autenticação que você especificar. Isso pode envolver uma chamada a um serviço criptográfico, como o AWS Key Management Service (AWS KMS).

Criptografia simétrica e assimétrica

A criptografia simétrica usa a mesma chave para criptografar e descriptografar dados.

A criptografia assimétrica usa um par de chaves de dados matematicamente relacionado. Uma chave no par criptografa os dados; somente a outra chave no par pode descriptografar os dados.

O SDK AWS de criptografia de banco de dados usa criptografia de envelope. Ele criptografa os dados com uma chave de dados simétrica. Ele criptografa a chave de dados simétrica com uma ou mais chaves de empacotamento simétricas ou assimétricas. Ele adiciona uma descrição do material ao registro que inclui pelo menos uma cópia criptografada da chave de dados.

Criptografar dados (criptografia simétrica)

> Para criptografar seus dados, o SDK AWS de criptografia de banco de dados usa uma chave de dados simétrica e um conjunto de algoritmos que inclui um algoritmo de criptografia simétrica. Para descriptografar os dados, o SDK de criptografia AWS de banco de dados usa a mesma chave de dados e o mesmo conjunto de algoritmos.

Criptografar chave de dados (criptografia simétrica ou assimétrica)

> O token de autenticação que você fornece para uma operação de criptografia e descriptografia determina como a chave de dados simétrica é criptografada e descriptografada. Você pode escolher um chaveiro que use criptografia simétrica, como um AWS KMS chaveiro com uma chave KMS de criptografia simétrica, ou um que use criptografia assimétrica, como um AWS KMS chaveiro com uma chave RSA KMS assimétrica.

Confirmação de chave

O SDK AWS de criptografia de banco de dados oferece suporte ao comprometimento de chaves (às vezes conhecido como robustez), uma propriedade de segurança que garante que cada texto cifrado possa ser descriptografado somente em um único texto simples. Para fazer isso, o compromisso da chave garante que somente a chave de dados que criptografou seu registro seja

usada para descriptografá-lo. O SDK AWS de criptografia de banco de dados inclui um compromisso fundamental para todas as operações de criptografia e descriptografia.

A maioria das cifras simétricas modernas (incluindo AES) criptografa texto sem formatação com uma única chave secreta, como a chave de dados exclusiva que o SDK de criptografia de AWS banco de dados usa para criptografar cada campo de texto sem formatação marcado em um registro. `ENCRYPT_AND_SIGN` Descriptografar esse registro com a mesma chave de dados retorna um texto sem formatação idêntico ao original. A decodificação com uma chave diferente geralmente falhará. Embora seja difícil, é tecnicamente possível decifrar um texto cifrado com duas chaves diferentes. Em casos raros, é possível encontrar uma chave que possa decifrar parcialmente o texto cifrado em um texto simples diferente, mas ainda inteligível.

O SDK AWS de criptografia de banco de dados sempre criptografa cada atributo em uma chave de dados exclusiva. Ele pode criptografar essa chave de dados em várias chaves de empacotamento, mas as chaves de empacotamento sempre criptografam a mesma chave de dados. No entanto, um registro criptografado sofisticado e criado manualmente pode, na verdade, conter chaves de dados diferentes, cada uma criptografada por uma chave de empacotamento diferente. Por exemplo, se um usuário descriptografar o registro criptografado, ele retornará 0x0 (falso), enquanto outro usuário descriptografando o mesmo registro criptografado obterá 0x1 (verdadeiro).

Para evitar esse cenário, o SDK AWS de criptografia de banco de dados inclui comprometimento de chave ao criptografar e descriptografar. O método de criptografia vincula criptograficamente a chave de dados exclusiva que produziu o texto cifrado ao compromisso da chave, um código de autenticação de mensagens por hash (HMAC) calculado sobre a descrição do material usando uma derivação da chave de dados. Em seguida, ele armazena o compromisso de chaves na descrição do material. Ao descriptografar um registro com comprometimento de chave, o SDK de criptografia AWS de banco de dados verifica se a chave de dados é a única chave para esse registro criptografado. Se a verificação da chave de dados falhar, a operação de descriptografia falhará.

Assinaturas digitais

O SDK AWS de criptografia de banco de dados criptografa seus dados usando um algoritmo de criptografia autenticado, o AES-GCM, e o processo de descriptografia verifica a integridade e a autenticidade de uma mensagem criptografada sem usar uma assinatura digital. Mas como o AES-GCM usa chaves simétricas, qualquer pessoa que possa descriptografar a chave de dados usada para descriptografar o texto cifrado também pode criar manualmente um novo texto cifrado, causando uma possível preocupação de segurança. Por exemplo, se você usar um AWS KMS key

como chave de encapsulamento, um usuário com `kms:Decrypt` permissões poderá criar textos cifrados criptografados sem ligar. `kms:Encrypt`

Para evitar esse problema, o conjunto de algoritmos padrão adiciona uma assinatura do Algoritmo de assinatura digital de curva elíptica (ECDSA) aos registros criptografados. O conjunto de algoritmos padrão criptografa os campos em seu registro marcados com `ENCRYPT_AND_SIGN` usando um algoritmo de criptografia autenticado, o AES-GCM. Em seguida, ele calcula os Códigos de Autenticação de Mensagens Baseados em Hash (HMACs) e as assinaturas ECDSA assimétricas nos campos do seu registro marcados com, e. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` O processo de descriptografia usa as assinaturas para verificar se um usuário autorizado criptografou o registro.

Quando o conjunto de algoritmos padrão é usado, o SDK do AWS Database Encryption gera uma chave privada temporária e um par de chaves públicas para cada registro criptografado. O SDK AWS de criptografia de banco de dados armazena a chave pública na descrição do material e descarta a chave privada. Isso garante que ninguém possa criar outra assinatura que seja verificada com a chave pública. O algoritmo vincula a chave pública à chave de dados criptografada como dados autenticados adicionais na descrição do material, impedindo que usuários que só podem descriptografar campos alterem a chave pública ou afetem a verificação da assinatura.

O SDK AWS de criptografia de banco de dados sempre inclui a verificação HMAC. As assinaturas digitais ECDSA são habilitadas por padrão, mas não são obrigatórias. Se os usuários que criptografam dados e os usuários que decifram os dados forem igualmente confiáveis, considere usar um conjunto de algoritmos que não inclua assinaturas digitais para melhorar seu desempenho. Para obter mais informações sobre como selecionar conjuntos de algoritmos alternativos, consulte Escolha de um conjunto de algoritmos.

> **Note**
>
> Se um chaveiro não delimitar entre criptografadores e decodificadores, as assinaturas digitais não fornecem valor criptográfico.

AWS KMS os chaveiros, incluindo o AWS KMS chaveiro RSA assimétrico, podem delinear entre criptografadores e decodificadores com base nas políticas de chaves e nas políticas do IAM. AWS KMS

Devido à sua natureza criptográfica, os seguintes chaveiros não podem delimitar entre criptografadores e decodificadores:

- AWS KMS Chaveiro hierárquico
- AWS KMS Chaveiro ECDH
- Token de autenticação bruto do AES
- Token de autenticação bruto do RSA
- Chaveiro ECDH bruto

Como funciona o SDK AWS de criptografia de banco de dados

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados fornece bibliotecas de criptografia do lado do cliente projetadas especificamente para proteger os dados que você armazena nos bancos de dados. As bibliotecas incluem implementações seguras que você pode estender ou usar inalteradas. Para obter mais informações sobre como definir e usar componentes personalizados, consulte o GitHub repositório da implementação do seu banco de dados.

Os fluxos de trabalho desta seção explicam como o SDK de criptografia AWS de banco de dados criptografa, assina, descriptografa e verifica os dados em seu banco de dados. Esses fluxos de trabalho descrevem o processo básico usando elementos abstratos e os atributos padrão. Para obter detalhes sobre como o SDK AWS de criptografia de banco de dados funciona com sua implementação de banco de dados, consulte o tópico O que é criptografado para seu banco de dados.

O SDK AWS de criptografia de banco de dados usa criptografia de envelope para proteger seus dados. Cada mensagem é criptografada em uma chave de dados exclusiva. A chave de dados é usada para derivar uma chave de criptografia de dados exclusiva para cada campo marcado `ENCRYPT_AND_SIGN` em suas ações criptográficas. Em seguida, uma cópia da chave de dados é criptografada pelas chaves de empacotamento que você especificar. Para descriptografar o registro criptografado, o SDK de criptografia de AWS banco de dados usa as chaves de encapsulamento que você especifica para descriptografar pelo menos uma chave de dados criptografada. Em seguida, ele pode descriptografar o texto cifrado e retornar uma entrada de texto simples.

Para obter mais informações sobre os termos usados no SDK AWS de criptografia de banco de dados, consulteAWS Conceitos do SDK de criptografia de banco de dados.

Criptografar e assinar

Em essência, o SDK AWS de criptografia de banco de dados é um criptografador de registros que criptografa, assina, verifica e descriptografa os registros em seu banco de dados. Ele obtém informações sobre os registros e instruções sobre quais campos devem ser criptografados e assinados. Ele obtém os materiais de criptografia e as instruções sobre como usá-los de um gerenciador de material de criptografia configurado com a chave de empacotamento que você especifica.

O passo a passo a seguir descreve como o SDK de criptografia AWS de banco de dados criptografa e assina suas entradas de dados.

1. O gerenciador de materiais criptográficos fornece ao SDK AWS de criptografia de banco de dados chaves exclusivas de criptografia de dados: uma chave de dados em texto simples, uma cópia da chave de dados criptografada pela chave de encapsulamento especificada e uma chave MAC.

   > **Note**
   >
   > É possível criptografar a chave de dados em várias chaves de empacotamento. Cada uma das chaves de empacotamento criptografa uma cópia da chave de dados. O SDK AWS de criptografia de banco de dados armazena todas as chaves de dados criptografadas na descrição do material. O SDK de criptografia de banco de dados da AWS adiciona um novo campo (`aws_dbe_head`) ao registro que armazena a descrição do material.
   >
   > Uma chave MAC é derivada para cada cópia criptografada da chave de dados. As chaves MAC não são armazenadas na descrição do material. Em vez disso, o método de descriptografia usa as chaves de empacotamento para derivar as chaves MAC novamente.

2. O método de criptografia criptografa cada campo marcado como `ENCRYPT_AND_SIGN` nas ações criptográficas que você especificou.
3. O método de criptografia deriva `commitKey` da chave de dados e a usa para gerar um valor de comprometimento da chave e, em seguida, descarta a chave de dados.

4. Saiba mais sobre a descrição do material para o registro. A descrição do material contém as chaves de dados criptografadas e outras informações sobre o registro criptografado. Para obter uma lista completa das informações incluídas na descrição do material, consulte Formato de descrição do material.
5. O método de criptografia usa as chaves MAC retornadas na Etapa 1 para calcular os valores do Código de Autenticação de Mensagens Baseadas em Hash (HMAC) sobre a canonização da descrição do material, do contexto de criptografia e de cada campo marcado `ENCRYPT_AND_SIGN` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` nas ações `SIGN_ONLY` criptográficas. Os valores HMAC são armazenados em um novo campo (`aws_dbe_foot`) que o método de criptografia adiciona ao registro.
6. O método de criptografia calcula uma assinatura ECDSA com base na canonização da descrição do material, do contexto de criptografia e de cada campo marcado `ENCRYPT_AND_SIGNSIGN_ONLY`, ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` armazena as assinaturas ECDSA no campo. `aws_dbe_foot`

   > **Note**
   >
   > As assinaturas ECDSA são habilitadas por padrão, mas não são obrigatórias.

7. O método de criptografia armazena o registro criptografado e assinado em seu banco de dados

Descriptografar e verificar

1. O gerenciador de materiais criptográficos (CMM) fornece o método de decodificação com os materiais de decodificação armazenados na descrição do material, incluindo a chave de dados em texto simples e a chave MAC correspondente.

   - O CMM descriptografa a chave de dados criptografados com as chaves de empacotamento no token de autenticação especificado e gera a chave de dados de texto simples.
2. O método de decodificação compara e verifica o valor do comprometimento chave na descrição do material.
3. O método de decodificação verifica as assinaturas no campo de assinatura.

   Ele identifica quais campos estão marcados `ENCRYPT_AND_SIGN` ou a `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` partir da lista de campos não autenticados permitidos que você definiu. `SIGN_ONLY` O método de descriptografia usa a chave MAC

retornada na Etapa 1 para recalcular e comparar os valores HMAC dos campos marcados com, ou. `ENCRYPT_AND_SIGN` `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Em seguida, ele verifica as assinaturas ECDSA usando a chave pública armazenada no contexto de criptografia.

4. O método de descriptografia usa a chave de dados de texto simples para descriptografar cada valor marcado com `ENCRYPT_AND_SIGN`. Em seguida, o SDK AWS de criptografia de banco de dados descarta a chave de dados em texto simples.
5. O método de descriptografia retorna os registros de texto não criptografado.

Suítes de algoritmos compatíveis no SDK AWS de criptografia de banco de dados

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Um pacote de algoritmos é uma coleção de algoritmos criptográficos e de valores relacionados. Os sistemas criptográficos usam a implementação do algoritmo para gerar o texto cifrado.

O SDK AWS de criptografia de banco de dados usa um conjunto de algoritmos para criptografar e assinar os campos em seu banco de dados. Todos os pacotes de algoritmos compatíveis usam o algoritmo Advanced Encryption Standard (AES) com Galois/Counter Mode (GCM), conhecido como AES-GCM, para criptografar dados brutos. O SDK AWS de criptografia de banco de dados oferece suporte a chaves de criptografia de 256 bits. O tamanho da tag de autenticação é sempre 16 bytes.

AWS Suítes de algoritmos SDK de criptografia de banco de dados

Algoritmo	Algoritmo de criptografia	Tamanho da chave de dados (em bits)	Algoritmo de derivação de chave	Algoritmo de assinatura simétrica	Algoritmo de assinatura assimétrica	Compromisso com a chave
Padrão	AES-GCM	256	HKDF com SHA-512	HMAC-SHA-384	ECDSA com P-384 e SHA-384	HKDF com SHA-512

Algoritmo	Algoritmo de criptografia	Tamanho da chave de dados (em bits)	Algoritmo de derivação de chave	Algoritmo de assinatura simétrica	Algoritmo de assinatura assimétrica	Compromisso com a chave
AES-GCM sem assinaturas digitais ECDSA	AES-GCM	256	HKDF com SHA-512	HMAC-SHA-384	Nenhum	HKDF com SHA-512

Algoritmo de criptografia

O nome e o modo do algoritmo de criptografia utilizado. Os pacotes de algoritmos no SDK AWS de criptografia de banco de dados usam o algoritmo Advanced Encryption Standard (AES) com Galois/Counter Mode (GCM).

Tamanho da chave de dados

O tamanho da chave de dados em bits. O SDK AWS de criptografia de banco de dados é compatível com chaves de dados de 256 bits. A chave de dados é usada como entrada para uma função de derivação de extract-and-expand chave baseada em HMAC (HKDF). A saída da HKDF é usada como a chave de criptografia de dados no algoritmo de criptografia.

Algoritmo de derivação de chave

A função de derivação de extract-and-expand chave baseada em HMAC (HKDF) usada para derivar a chave de criptografia de dados. O SDK AWS de criptografia de banco de dados usa o HKDF definido na RFC 5869.

- A função hash usada é SHA-512
- Para a etapa de extração:
  - Nenhum sal é usado. De acordo com a RFC, o sal é definido como uma string de zeros.
  - O material de chaveamento de entrada é a chave de dados do chaveiro.
- Para a etapa de expansão:
  - A chave pseudoaleatória de entrada é a saída da etapa de extração.
  - O rótulo da chave são os bytes codificados em UTF-8 da string `DERIVEKEY` na ordem de bytes big endian.

- As informações da entrada são uma concatenação do ID do algoritmo seguido pelo rótulo de chave (nessa ordem).
- O comprimento do material de chaveamento de saída é o Tamanho da chave de dados. Essa saída é usada como a chave de criptografia de dados no algoritmo de criptografia.

Algoritmo de assinatura simétrica

O algoritmo HMAC (Código de Autenticação de Mensagem Baseado em Hash) usado para gerar uma assinatura simétrica. Todos os pacotes de algoritmos compatíveis incluem verificação HMAC.

O SDK AWS de criptografia de banco de dados serializa a descrição do material e todos os campos marcados com `ENCRYPT_AND_SIGN` `SIGN_ONLY`, ou. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Em seguida, ele usa o HMAC com um algoritmo de função hash criptográfica (SHA-384) para assinar a canonização.

A assinatura HMAC simétrica é armazenada em um novo campo (`aws_dbe_foot`) que o AWS Database Encryption SDK adiciona ao registro.

Algoritmo de assinatura assimétrica

O algoritmo de assinatura usado para gerar uma assinatura digital assimétrica.

O SDK AWS de criptografia de banco de dados serializa a descrição do material e todos os campos marcados com `ENCRYPT_AND_SIGN` `SIGN_ONLY`, ou. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Em seguida, ele usa o Algoritmo de Assinatura Digital de Curva Elíptica (ECDSA) com as seguintes especificações para assinar a canonização:

- A curva elíptica usada é a P-384, conforme definido no Padrão de Assinatura Digital (DSS) (FIPS PUB 186-4).
- A função hash usada é SHA-384.

A assinatura ECDSA assimétrica é armazenada com a assinatura HMAC simétrica no campo. `aws_dbe_foot`

As assinaturas digitais ECDSA são incluídas por padrão, mas não são obrigatórias.

Confirmação de chave

A função de derivação de extract-and-expand chave baseada em HMAC (HKDF) usada para derivar a chave de confirmação.

- A função hash usada é SHA-512

- Para a etapa de extração:
  - Nenhum sal é usado. De acordo com a RFC, o sal é definido como uma string de zeros.
  - O material de chaveamento de entrada é a chave de dados do chaveiro.
- Para a etapa de expansão:
  - A chave pseudoaleatória de entrada é a saída da etapa de extração.
  - As informações de entrada são os bytes codificados em UTF-8 da `COMMITKEY` string na ordem de bytes big endian.
  - O comprimento do material de chaveamento de saída é de 256 bits. Essa saída é usada como chave de confirmação.

A chave de confirmação calcula o comprometimento do registro, um hash distinto de código de autenticação de mensagens baseado em hash (HMAC) de 256 bits, sobre a descrição do material. Para obter uma explicação técnica sobre como adicionar comprometimento de chave a um conjunto de algoritmos, consulte Key Committing AEADs in Cryptology ePrint Archive.

Conjunto de algoritmos padrão

Por padrão, o SDK AWS de criptografia de banco de dados usa um conjunto de algoritmos com AES-GCM, uma função de derivação de extract-and-expand chave (HKDF) baseada em HMAC, verificação HMAC, assinaturas digitais ECDSA, comprometimento de chave e uma chave de criptografia de 256 bits.

O conjunto de algoritmos padrão inclui verificação HMAC (assinaturas simétricas) e assinaturas digitais ECDSA (assinaturas assimétricas). Essas assinaturas são armazenadas em um novo campo (`aws_dbe_foot`) que o SDK do AWS Database Encryption adiciona ao registro. As assinaturas digitais ECDSA são particularmente úteis quando a política de autorização permite que um conjunto de usuários criptografe dados e um conjunto diferente de usuários descriptografe dados.

O conjunto de algoritmos padrão também deriva de um compromisso chave — um hash HMAC que vincula a chave de dados ao registro. O valor de comprometimento da chave é um HMAC calculado a partir da descrição do material e da chave de confirmação. Em seguida, ele armazena o comprometimento de chaves na descrição do material. O comprometimento principal garante que cada texto cifrado seja decifrado em apenas um texto simples. Eles fazem isso validando a chave de dados usada como entrada para o algoritmo de criptografia. Ao criptografar, o conjunto de algoritmos obtém um HMAC de compromisso chave. Antes de decifrar, eles validam que a chave de dados produz o mesmo HMAC de comprometimento de chave. Caso contrário, a chamada decrypt falhará.

AES-GCM sem assinaturas digitais ECDSA

Embora o conjunto de algoritmos padrão provavelmente seja adequado para a maioria dos aplicativos, você pode escolher um conjunto alternativo de algoritmos. Por exemplo, alguns modelos de confiança seriam satisfeitos com um conjunto de algoritmos sem assinaturas digitais ECDSA. Use esse pacote somente quando os usuários que criptografam dados e os usuários que descriptografam dados forem igualmente confiáveis.

Todos os pacotes de algoritmos do AWS Database Encryption SDK incluem verificação HMAC (assinaturas simétricas). A única diferença é que o conjunto de algoritmos AES-GCM sem assinatura digital ECDSA carece da assinatura assimétrica que fornece uma camada adicional de autenticidade e não repúdio.

Por exemplo, se você tiver várias chaves de agrupamento em seu chaveiro,,`wrappingKeyA`, e `wrappingKeyBwrappingKeyC`, e você descriptografar um registro usando`wrappingKeyA`, a assinatura simétrica HMAC verifica se o registro foi criptografado por um usuário com acesso a. `wrappingKeyA` Se você usou o conjunto de algoritmos padrão, eles HMACs fornecem a mesma verificação e`wrappingKeyA`, além disso, usam a assinatura digital ECDSA para garantir que o registro foi criptografado por um usuário com permissões de criptografia para. `wrappingKeyA`

Para selecionar o conjunto de algoritmos AES-GCM sem assinaturas digitais, inclua o seguinte trecho em sua configuração de criptografia.

Java

O trecho a seguir especifica o conjunto de algoritmos AES-GCM sem assinaturas digitais ECDSA. Para obter mais informações, consulte the section called “Configuração de criptografia”.

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

C# / .NET

O trecho a seguir especifica o conjunto de algoritmos AES-GCM sem assinaturas digitais ECDSA. Para obter mais informações, consulte the section called “Configuração de criptografia”.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

O trecho a seguir especifica o conjunto de algoritmos AES-GCM sem assinaturas digitais ECDSA. Para obter mais informações, consulte the section called “Configuração de criptografia”.

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

Usando o SDK AWS de criptografia de banco de dados com AWS KMS

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Para usar o SDK AWS de criptografia de banco de dados, você precisa configurar um chaveiro e especificar uma ou mais chaves de encapsulamento. Se você não tiver uma infraestrutura de chaves, recomendamos usar o AWS Key Management Service (AWS KMS).

O SDK AWS de criptografia de banco de dados oferece suporte a dois tipos de AWS KMS chaveiros. O token de autenticação do AWS KMS tradicional usa o AWS KMS keys para gerar, criptografar e descriptografar chaves de dados. É possível usar criptografia simétrica (`SYMMETRIC_DEFAULT`) ou chaves RSA assimétricas do KMS. Como o SDK AWS de criptografia de banco de dados criptografa e assina cada registro com uma chave de dados exclusiva, o AWS KMS chaveiro deve exigir cada operação AWS KMS de criptografia e descriptografia. Para aplicativos que precisam minimizar o número de chamadas para AWS KMS, o SDK de criptografia de AWS banco de dados também oferece suporte ao AWS KMS chaveiro hierárquico. O chaveiro hierárquico é uma solução de armazenamento em cache de materiais criptográficos que reduz o número de AWS KMS chamadas usando chaves de ramificação AWS KMS protegidas persistentes em uma tabela do Amazon DynamoDB e, em seguida, armazenando localmente em cache materiais de chave de ramificação usados em operações de criptografia e descriptografia. Recomendamos usar os AWS KMS chaveiros sempre que possível.

Para interagir com AWS KMS, o SDK AWS de criptografia de banco de dados requer o AWS KMS módulo do AWS SDK para Java.

Para se preparar para usar o SDK AWS de criptografia de banco de dados com AWS KMS

1. Crie um Conta da AWS. Para saber como, consulte Como eu crio e ativo uma nova conta da Amazon Web Services? no Centro de AWS Conhecimento.
2. Crie uma criptografia AWS KMS key simétrica. Para obter ajuda, consulte Criação de chaves no Guia do desenvolvedor AWS Key Management Service .

> **Tip**
>
> Para usar o AWS KMS key programaticamente, você precisará do Amazon Resource Name (ARN) do. AWS KMS key Para ajudar a encontrar o ARN de uma AWS KMS key, consulte Encontrar o ID de chave e o ARN no Guia do desenvolvedor do AWS Key Management Service .

3. Gere um ID de chave de acesso e uma chave de acesso de segurança. Você pode usar o ID da chave de acesso e a chave de acesso secreta para um usuário do IAM ou AWS Security Token Service para criar uma nova sessão com credenciais de segurança temporárias que incluem um ID de chave de acesso, chave de acesso secreta e token de sessão. Como prática recomendada de segurança, recomendamos que você use credenciais temporárias em vez das credenciais de longo prazo associadas às suas contas de usuário do IAM ou AWS (raiz).

   Para criar um usuário do IAM com uma chave de acesso, consulte Criação de usuários do IAM no Guia do usuário do IAM.

   Para gerar mais informações sobre credenciais de segurança temporárias, consulte Solicitação de credenciais de segurança temporárias no Guia do usuário do IAM.
4. Defina suas AWS credenciais usando as instruções em AWS SDK para Javae o ID da chave de acesso e a chave de acesso secreta que você gerou na etapa 3. Se você gerou credenciais temporárias, também precisará especificar o token de sessão.

   Este procedimento AWS SDKs permite assinar solicitações AWS para você. As amostras de código no SDK AWS de criptografia de banco de dados que interagem com AWS KMS pressupõem que você tenha concluído essa etapa.

Configuração do SDK de criptografia de AWS banco de dados da

Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de AWS banco de dados da. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados da foi projetado para ser fácil de usar. Embora o SDK AWS de criptografia de banco de dados da tenha várias opções de configuração, os valores padrão são cuidadosamente escolhidos para serem práticos e seguros para a maioria dos aplicativos. No entanto, talvez seja necessário ajustar sua configuração para melhorar a performance ou incluir um atributo personalizado em seu design.

Tópicos

- Seleção de uma linguagem de programação
- Seleção de chaves de encapsulamento
- Criação de um filtro de descoberta
- Trabalhar com bancos de dados multilocatários
- Criação de beacons assinados

Seleção de uma linguagem de programação

O SDK AWS de criptografia de banco de dados para DynamoDB está disponível em várias linguagens de programação. As implementações de linguagem são projetadas para serem totalmente interoperáveis e oferecer os mesmos atributos, embora possam ser implementadas de maneiras diferentes. Normalmente, você usa a biblioteca compatível com sua aplicação.

Seleção de chaves de encapsulamento

O SDK AWS de criptografia de banco de dados da gera uma chave de dados simétrica exclusiva para criptografar cada campo. Não é necessário configurar, gerenciar ou usar as chaves de dados. O SDK AWS de criptografia de banco de dados da faz isso por você.

No entanto, você deve selecionar uma ou mais chaves de empacotamento para criptografar cada chave de dados. O SDK de criptografia de banco de dados da AWS é compatível com chaves do KMS de criptografia simétrica e chaves KMS RSA assimétricas AWS Key Management Service (AWS KMS). Ele também é compatível com chaves simétricas AES e chaves assimétricas RSA que você fornece em tamanhos diferentes. Você é responsável pela segurança e durabilidade de suas chaves de empacotamento, por isso recomendamos que use uma chave de criptografia em um módulo de segurança de hardware ou em um serviço de infraestrutura de chaves, como AWS KMS.

Para especificar suas chaves de empacotamento para criptografia e decodificação, use um token de autenticação. Dependendo do tipo de token de autenticação usado, é possível especificar uma chave de empacotamento ou várias chaves de empacotamento do mesmo tipo ou de tipos diferentes. Se você usar várias chaves de empacotamento para empacotar uma chave de dados, cada chave de empacotamento criptografará uma cópia da mesma chave de dados. As chaves de dados criptografadas (uma por chave de empacotamento) são armazenadas na descrição do material armazenada junto com o campo criptografado. Para descriptografar os dados, o SDK de criptografia de AWS banco de dados da deve primeiro usar uma de suas chaves de empacotamento para descriptografar uma chave de dados criptografada.

Recomendamos usar um dos tokens de autenticação AWS KMS do sempre que possível. O SDK AWS de criptografia de banco de dados da fornece o AWS KMS token de autenticação e o token de AWS KMS autenticação hierárquico do, o que reduz o número de chamadas ao. AWS KMS Para especificar uma AWS KMS key em um token de autenticação, use um identificador de AWS KMS chave do compatível. Se você usar o token de AWS KMS autenticação hierárquico do, deverá especificar o ARN da chave. Para obter detalhes sobre os identificadores de chave de uma AWS KMS chave, consulte Identificadores de chave no Guia do AWS Key Management Service desenvolvedor.

- Ao criptografar com um AWS KMS token de autenticação do, é possível especificar qualquer identificador de chave válido (ARN da chave, nome do alias ou ID da chave) para uma chave KMS de criptografia simétrica. Se você usar uma chave do RSA KMS assimétrica, deverá especificar o ARN da chave.

  Se você especificar um nome de alias ou ARN de alias para uma chave KMS ao criptografar, o SDK de criptografia de banco de dados da AWS salvará o ARN da chave atualmente associado a esse alias; ele não salvará o alias. As alterações no alias não afetam a chave do KMS usada para descriptografar suas chaves de dados.

- Por padrão, o token de autenticação do descriptografa registros no modo estrito (onde você especifica AWS KMS chaves do KMS específicas). Em um token de autenticação de descriptografia, você deve usar um ARN de chave para identificar AWS KMS keys .

  Quando você criptografa com um AWS KMS token de autenticação do, o SDK AWS de criptografia de banco de dados da armazena o ARN da chave da AWS KMS key na descrição do material com a chave de dados criptografada. Ao descriptografar no modo estrito, o SDK de criptografia de AWS banco de dados da verifica se o mesmo ARN da chave aparece no token de autenticação antes de tentar usar a chave de empacotamento para descriptografar a chave de dados criptografada. Se você usar um identificador de chave diferente, o SDK AWS de criptografia de banco de dados da não reconhecerá nem usará o AWS KMS key, mesmo que os identificadores se refiram à mesma chave.
- Ao descriptografar no modo de descoberta, não especifique chaves de empacotamento. Primeiro, o SDK AWS de criptografia de banco de dados da tenta descriptografar o registro com a chave ARN armazenada na descrição do material. Se isso não funcionar, o SDK de criptografia AWS de banco de dados da solicita que o faça AWS KMS a descriptografia do registro usando a chave do KMS que o criptografou, independentemente de quem é proprietário ou tem acesso a essa chave KMS.

Para especificar uma chave AES bruta ou um par de chaves RSA brutas como chave de empacotamento em um token de autenticação, você deve especificar um namespace e um nome. Ao descriptografar, você deve usar exatamente o mesmo namespace e nome para cada chave de empacotamento bruta que você usou ao criptografar. Se você usar um namespace ou nome diferente, o SDK de criptografia de AWS banco de dados da não reconhecerá nem usará a chave de empacotamento, mesmo que o material da chave seja o mesmo.

Criação de um filtro de descoberta

Ao descriptografar dados criptografados com chaves do KMS, é uma prática recomendada descriptografar no modo estrito, ou seja, limitar as chaves de empacotamento usadas somente às que você especificar. No entanto, se necessário, você também poderá descriptografar no modo de descoberta, onde você não especifica nenhuma chave de empacotamento. Nesse modo, o AWS KMS pode descriptografar a chave de dados criptografada usando a chave KMS que a criptografou, independentemente de quem possui ou tem acesso a essa chave KMS.

Se você for descriptografar no modo de descoberta, recomendamos que sempre use um filtro de descoberta, que limita as chaves do KMS que podem ser usadas às chaves em uma especificada e na partição. Conta da AWS O filtro de descoberta é opcional, mas é uma prática recomendada.

Use a tabela a seguir para determinar o valor da partição do seu filtro de descoberta.

Região	Partition
Regiões da AWS	`aws`
Regiões da China	`aws-cn`
AWS GovCloud (US) Regions	`aws-us-gov`

O exemplo a seguir mostra como criar um filtro de descoberta. Antes de usar o código, substitua os valores de exemplo por valores válidos para sua Conta da AWS e partição.

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()?;
```

Trabalhar com bancos de dados multilocatários

Com o SDK AWS de criptografia de banco de dados da, é possível configurar a criptografia do lado do cliente para bancos de dados com um esquema compartilhado, isolando cada locatário com materiais de criptografia distintos. Ao considerar um banco de dados multilocatário, reserve um tempo para analisar seus requisitos de segurança e como a multilocação pode afetá-los. Por exemplo, o uso de um banco de dados multilocatário pode afetar sua capacidade de combinar o SDK de criptografia de AWS banco de dados da com outra solução de criptografia do lado do servidor.

Se você tiver vários usuários executando operações de criptografia em seu banco de dados, poderá usar um dos tokens de autenticação do para AWS KMS fornecer a cada usuário uma chave distinta para usar em suas operações criptográficas. Gerenciar as chaves de dados para uma solução de criptografia do lado do cliente multilocatária pode ser complicado. Recomendamos organizar seus dados por locatário sempre que possível. Se o locatário for identificado pelos valores da chave primária (por exemplo, a chave de partição em uma tabela do Amazon DynamoDB), será mais fácil gerenciar suas chaves.

Você pode usar o token de autenticação AWS KMS do para isolar cada locatário com um token de autenticação do distinto AWS KMS e. AWS KMS keys Com base no volume de AWS KMS chamadas ao feitas por locatário, talvez você queira usar o token de autenticação AWS KMS hierárquico do para minimizar suas chamadas ao. AWS KMS O token de AWS KMS autenticação hierárquico é uma solução de armazenamento em cache de materiais criptográficos que reduz o número de AWS KMS chamadas do usando chaves de ramificação persistentes AWS KMS protegidas pelo em uma tabela do Amazon DynamoDB e, em seguida, armazenando localmente em cache materiais de chave de ramificação usados em operações de criptografia e descriptografia. Você deve usar o token de autenticação AWS KMS hierárquico do para implementar a criptografia pesquisável em seu banco de dados.

Criação de beacons assinados

O SDK AWS de criptografia de banco de dados da usa beacons padrão e beacons compostos para fornecer soluções de criptografia pesquisáveis que permitem pesquisar registros criptografados sem descriptografar todo o banco de dados consultado. No entanto, o SDK AWS de criptografia de banco de dados da também oferece suporte a beacons assinados que podem ser configurados inteiramente a partir de campos assinados de texto simples. Os beacons assinados são um tipo de beacon composto que indexa e executa consultas complexas em campos e. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

Por exemplo, se você tiver um banco de dados multilocatário, talvez queira criar um beacon assinado que permita consultar seu banco de dados em busca de registros criptografados pela chave de um locatário específico. Para obter mais informações, consulte Consultar beacons em um banco de dados multilocatário.

Você deve usar o token de autenticação AWS KMS hierárquico do para criar beacons assinados.

Para configurar um beacon assinado, forneça os valores a seguir.

Java

Configuração de farol composto

O exemplo a seguir define as listas de peças assinadas localmente na configuração do beacon assinado.

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

Definição da versão do Beacon

O exemplo a seguir define as listas de peças assinadas globalmente na versão beacon. Para obter mais informações sobre como definir a versão do beacon, consulte Usando 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

Veja o exemplo de código completo: BeaconConfig.cs

Configuração de beacon assinada

O exemplo a seguir define as listas de peças assinadas localmente na configuração do beacon assinado.

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

Definição da versão do Beacon

O exemplo a seguir define as listas de peças assinadas globalmente na versão beacon. Para obter mais informações sobre como definir a versão do beacon, consulte Usando 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
            }
        }
    }
};
```

Você pode definir suas peças assinadas em listas definidas local ou globalmente. Recomendamos definir suas peças assinadas em uma lista global na versão beacon sempre que possível. Ao definir peças assinadas globalmente, você pode definir cada peça uma vez e depois reutilizar as peças em várias configurações de faróis compostos. Se você pretende usar uma peça assinada apenas uma vez, você pode defini-la em uma lista local na configuração do beacon assinado. Você pode referenciar partes locais e globais na sua lista de construtores.

Se você definir suas listas de peças assinadas globalmente, deverá fornecer uma lista de peças do construtor que identifique todas as maneiras possíveis pelas quais o farol assinado pode montar os campos em sua configuração de farol.

> **Note**
>
> Para definir listas de peças assinadas globalmente, você deve usar a versão 3.2 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de definir qualquer parte nova globalmente.
> Você não pode atualizar as configurações de beacon existentes para definir listas de peças assinadas globalmente.

Nome do beacon

O nome que você usa ao consultar o beacon.

Um nome de beacon assinado não pode ser o mesmo nome de um campo não criptografado. Beacons diferentes não podem ter o mesmo nome.

Dividir caractere

O caractere usado para separar as partes que compõem seu beacon assinado.

O caractere dividido não pode aparecer nos valores de texto simples de nenhum dos campos a partir dos quais o beacon assinado foi construído.

Lista de partes assinadas

Identifica os campos assinados incluídos no beacon assinado.

Cada parte deve incluir um nome, uma fonte e um prefixo. A fonte é o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo `SIGN_ONLY` ou que a peça identifica. A fonte deve ser um nome de campo ou um índice referente ao valor de um campo aninhado. Se o nome da parte identificar a fonte, será possível omitir a fonte e o SDK de criptografia AWS de banco de dados da usará automaticamente o nome como fonte. Recomendamos especificar a fonte como nome da parte sempre que possível. O prefixo pode ser qualquer string, mas deve ser exclusivo. Duas partes assinadas em um beacon assinado não podem ter o mesmo prefixo. Recomendamos usar um valor curto que diferencie a parte de outras partes atendidas pelo beacon composto.

Recomendamos definir suas peças assinadas globalmente sempre que possível. Você pode considerar definir uma peça assinada localmente se pretende usá-la apenas em um farol composto. Uma parte definida localmente não pode ter o mesmo prefixo ou nome de uma parte definida globalmente.

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-" }
};
```

Lista de construtores (opcional)

Identifica os construtores que definem as diferentes maneiras pelas quais as partes assinadas podem ser montadas pelo beacon assinado.

Se você não especificar uma lista de construtores, o SDK de criptografia AWS de banco de dados da montará o beacon assinado com o construtor padrão a seguir.

- Todas as partes assinadas na ordem em que foram adicionadas à lista de partes assinadas
- Todas as partes são obrigatórias

Construtores

Cada construtor é uma lista ordenada de partes do construtor que define uma maneira pela qual o beacon assinado pode ser montado. As partes do construtor são unidas na ordem em que são adicionadas à lista, com cada parte separada pelo caractere de divisão especificado.

Cada parte do construtor nomeia uma parte assinada e define se essa parte é obrigatória ou opcional dentro do construtor. Por exemplo, se você quiser consultar um beacon assinado em `Field1`, `Field1.Field2` e `Field1.Field2.Field3`, marque `Field2` e `Field3` como opcionais e crie um construtor.

Cada construtor deve ter pelo menos uma parte obrigatória. Recomendamos tornar obrigatória a primeira parte de cada construtor para que você possa usar o operador `BEGINS_WITH` nas consultas.

Um construtor será bem-sucedido se todas as partes obrigatórias estiverem presentes no registro. Quando você grava um novo registro, o beacon assinado usa a lista de construtores para determinar se o beacon pode ser montado a partir dos valores fornecidos. Ele tenta montar o beacon na ordem em que os construtores foram adicionados à lista de construtores e usa o primeiro construtor bem-sucedido. Se nenhum construtor for bem-sucedido, o beacon não será gravado no registro.

Todos os leitores e gravadores devem especificar a mesma ordem de construtores para garantir que os resultados da consulta estejam corretos.

Use o procedimento a seguir para especificar sua própria lista de construtores.

1. Crie uma parte construtora para cada parte assinada para definir se essa parte é necessária ou não.

   O nome da parte do construtor deve ser o nome do campo assinado.

   O exemplo a seguir demonstra como criar partes do construtor para um campo assinado.

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
   ```

```
        .name("Field1")
        .required(true)
        .build();
```

C# / .NET

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

2. Crie um construtor para cada forma possível de montar o beacon assinado usando as partes do construtor que você criou na Etapa 1.

   Por exemplo, se quiser consultar `Field1.Field2.Field3` e`Field4.Field2.Field3`, você deverá criar dois construtores. `Field1` e `Field4` podem ser obrigatórios porque foram definidos em dois construtores separados.

   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. Crie uma lista de construtores que inclua todos os construtores que você criou na Etapa 2.

   Java

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

   C# / .NET

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

4. Especifique `constructorList` ao criar o beacon assinado.

Armazenamentos de chaves no SDK AWS de criptografia de banco de dados

No SDK AWS de criptografia de banco de dados, um armazenamento de chaves é uma tabela do Amazon DynamoDB que persiste os dados hierárquicos usados pelo chaveiro hierárquico.AWS KMS O armazenamento de chaves ajuda a reduzir o número de chamadas que você precisa fazer AWS KMS para realizar operações criptográficas com o chaveiro hierárquico.

O armazenamento de chaves persiste e gerencia as chaves de ramificação que o chaveiro hierárquico usa para realizar a criptografia de envelope e proteger as chaves de criptografia de dados. O armazenamento de chaves armazena a chave de ramificação ativa e todas as versões anteriores da chave de ramificação. A chave de ramificação ativa é a versão mais recente da chave de ramificação. O chaveiro hierárquico usa uma chave de criptografia de dados exclusiva para cada solicitação de criptografia e criptografa cada chave de criptografia de dados com uma chave de empacotamento exclusiva derivada da chave de ramificação ativa. O token de autenticação hierárquico depende da hierarquia estabelecida entre as chaves de ramificação ativas e suas chaves de agrupamento derivadas.

Principais conceitos e terminologia da loja

Armazenamento de chaves

A tabela do DynamoDB que persiste dados hierárquicos, como chaves de ramificação e chaves de beacon.

Chave raiz

Uma chave KMS de criptografia simétrica que gera e protege as chaves de ramificação e as chaves de beacon em seu armazenamento de chaves.

Chave de ramificação

Uma chave de dados que é reutilizada para derivar uma chave de empacotamento exclusiva para criptografia de envelopes. Você pode criar várias chaves de ramificação em um repositório de chaves, mas cada chave de ramificação só pode ter uma versão de chave de ramificação ativa por vez. A chave de ramificação ativa é a versão mais recente da chave de ramificação.

As chaves de ramificação são derivadas do AWS KMS keys uso da GenerateDataKeyWithoutPlaintext operação kms:.

Chave de encapsulamento

Uma chave de dados exclusiva usada para criptografar a chave de criptografia de dados usada nas operações de criptografia.

As chaves de empacotamento são derivadas das chaves de ramificação. Para obter mais informações sobre o processo de derivação de chaves, consulte Detalhes técnicos do AWS KMS chaveiro hierárquico.

Chave de criptografia de dados

Uma chave de dados usada em operações de criptografia. O chaveiro hierárquico usa uma chave de criptografia de dados exclusiva para cada solicitação de criptografia.

Chave de farol

Uma chave de dados usada para gerar beacons para criptografia pesquisável. Para obter mais informações, consulte Criptografia pesquisável.

Implementação de permissões de privilégio mínimo

Ao usar um armazenamento de chaves e AWS KMS chaveiros hierárquicos, recomendamos que você siga o princípio do menor privilégio definindo as seguintes funções:

Administrador do armazenamento de chaves

Os administradores do armazenamento de chaves são responsáveis por criar e gerenciar o armazenamento de chaves e as chaves de ramificação que ele persiste e protege. Os administradores do armazenamento de chaves devem ser os únicos usuários com permissões de gravação na tabela do Amazon DynamoDB que serve como seu armazenamento de chaves. Eles devem ser os únicos usuários com acesso a operações privilegiadas de administrador, como `CreateKey`e. `VersionKey` Você só pode realizar essas operações ao configurar estaticamente suas ações de armazenamento de chaves.

`CreateKey`é uma operação privilegiada que pode adicionar um novo ARN de chave KMS à sua lista de permissões de armazenamento de chaves. Essa chave KMS pode criar novas chaves de ramificação ativas. Recomendamos limitar o acesso a essa operação porque, depois que uma chave KMS é adicionada ao armazenamento de chaves da filial, ela não pode ser excluída.

Usuário da loja de chaves

Na maioria dos casos de uso, o usuário do armazenamento de chaves só interage com o armazenamento de chaves por meio do chaveiro hierárquico enquanto criptografa, descriptografa, assina e verifica dados. Como resultado, eles só precisam de permissões de leitura para a tabela do Amazon DynamoDB que serve como seu armazenamento de chaves. Os usuários do armazenamento de chaves só devem precisar acessar as operações de uso que possibilitam as operações criptográficas`GetActiveBranchKey`, como`GetBranchKeyVersion`, e. `GetBeaconKey` Eles não precisam de permissões para criar ou gerenciar as chaves de ramificação que usam.

Você pode realizar operações de uso quando suas ações de armazenamento de chaves são configuradas estaticamente ou quando estão configuradas para descoberta. Você não pode realizar operações de administrador (`CreateKeyeVersionKey`) quando suas ações de armazenamento de chaves estão configuradas para descoberta.

Se o administrador do armazenamento de chaves da filial tiver permitido várias chaves KMS no armazenamento de chaves da filial, recomendamos que os usuários do armazenamento de chaves configurem suas ações de armazenamento de chaves para descoberta, para que o chaveiro hierárquico possa usar várias chaves KMS.

Crie um armazenamento de chaves

Antes de criar chaves de ramificação ou usar um AWS KMS chaveiro hierárquico, você deve criar seu armazenamento de chaves, uma tabela do Amazon DynamoDB que gerencia e protege suas chaves de ramificação.

> **Important**
>
> Não exclua a tabela do DynamoDB que persiste suas chaves de ramificação. Se você excluir essa tabela, não conseguirá descriptografar nenhum dado criptografado usando o chaveiro hierárquico.

Siga os procedimentos de criar uma tabela no Amazon DynamoDB Developer Guide, usando os seguintes valores de string obrigatórios para a chave de partição e a chave de classificação.

	Chave de partição	Chave de classificação
Tabela base	`branch-key-id`	`type`

Nome do armazenamento de chaves lógicas

Ao nomear a tabela do DynamoDB que serve como seu armazenamento de chaves, é importante considerar cuidadosamente o nome lógico do armazenamento de chaves que você especificará ao configurar suas ações de armazenamento de chaves. O nome do armazenamento lógico de chaves atua como um identificador para seu armazenamento de chaves e não pode ser alterado depois de ser definido inicialmente pelo primeiro usuário. Você deve sempre especificar o mesmo nome lógico de armazenamento de chaves em suas ações de armazenamento de chaves.

Deve haver um one-to-one mapeamento entre o nome da tabela do DynamoDB e o nome do armazenamento de chaves lógicas. O nome do armazenamento lógico de chaves é vinculado criptograficamente a todos os dados armazenados na tabela para simplificar as operações de restauração do DynamoDB. Embora o nome do armazenamento de chaves lógicas possa ser diferente do nome da tabela do DynamoDB, é altamente recomendável especificar o nome da tabela do DynamoDB como o nome do armazenamento de chaves lógicas. Caso o nome da tabela mude após a restauração da tabela do DynamoDB a partir de um backup, o nome do armazenamento lógico de chaves pode ser mapeado para o novo nome da tabela do DynamoDB para garantir que o chaveiro hierárquico ainda possa acessar seu armazenamento de chaves.

Não inclua informações confidenciais ou sigilosas em seu nome lógico de armazenamento de chaves. O nome do armazenamento de chaves lógicas é exibido em texto simples em AWS KMS CloudTrail eventos como o. `tablename`

Próximas etapas

1. the section called “Configurar as principais ações do armazenamento”
2. the section called “Criar chaves de ramificação”
3. Crie um AWS KMS chaveiro hierárquico

Configurar as principais ações do armazenamento

As ações do armazenamento de chaves determinam quais operações seus usuários podem realizar e como seu AWS KMS chaveiro hierárquico usa as chaves KMS listadas como permitidas em seu armazenamento de chaves. O SDK AWS de criptografia de banco de dados é compatível com as seguintes configurações de ação de armazenamento de chaves.

Estático

Quando você configura estaticamente seu armazenamento de chaves, o armazenamento de chaves só pode usar a chave KMS associada ao ARN da chave KMS que você fornece `kmsConfiguration` ao configurar suas ações de armazenamento de chaves. Uma exceção é lançada se um ARN de chave KMS diferente for encontrado ao criar, versionar ou obter uma chave de ramificação.

Você pode especificar uma chave KMS multirregional na sua`kmsConfiguration`, mas todo o ARN da chave, incluindo a região, persiste nas chaves de ramificação derivadas da chave KMS. Você não pode especificar uma chave em uma região diferente. Você deve fornecer exatamente a mesma chave multirregional para que os valores correspondam.

Ao configurar estaticamente suas ações de armazenamento de chaves, você pode realizar operações de uso (`GetActiveBranchKey`,`GetBranchKeyVersion`,`GetBeaconKey`) e operações administrativas (`CreateKeyeVersionKey`). `CreateKey`é uma operação privilegiada que pode adicionar um novo ARN de chave KMS à sua lista de permissões de armazenamento de chaves. Essa chave KMS pode criar novas chaves de ramificação ativas. Recomendamos limitar o acesso a essa operação porque, depois que uma chave KMS é adicionada ao armazenamento de chaves, ela não pode ser excluída.

Descoberta

Quando você configura suas ações de armazenamento de chaves para descoberta, o armazenamento de chaves pode usar qualquer AWS KMS key ARN que esteja na lista de permissões em seu armazenamento de chaves. No entanto, uma exceção é lançada quando uma chave KMS multirregional é encontrada e a região no ARN da chave não corresponde à região do AWS KMS cliente que está sendo usada.

Ao configurar seu armazenamento de chaves para descoberta, você não pode realizar operações administrativas, como `CreateKey` `VersionKey` e. Você só pode realizar as operações de uso que permitem operações de criptografia, descriptografia, assinatura e verificação. Para obter mais informações, consulte the section called “Implementação de permissões de privilégio mínimo”.

Configure suas principais ações de armazenamento

Antes de configurar suas ações de armazenamento de chaves, verifique se os pré-requisitos a seguir foram atendidos.

- Determine quais operações você precisa realizar. Para obter mais informações, consulte the section called “Implementação de permissões de privilégio mínimo”.
- Escolha um nome de armazenamento de chaves lógicas

  Deve haver um one-to-one mapeamento entre o nome da tabela do DynamoDB e o nome do armazenamento de chaves lógicas. O nome do armazenamento lógico de chaves é vinculado criptograficamente a todos os dados armazenados na tabela para simplificar as operações de restauração do DynamoDB. Ele não pode ser alterado depois de definido inicialmente pelo primeiro usuário. Você deve sempre especificar o mesmo nome lógico de armazenamento de chaves em suas ações de armazenamento de chaves. Para obter mais informações, consulte logical key store name.

Configuração estática

O exemplo a seguir configura estaticamente as principais ações do armazenamento. Você deve especificar o nome da tabela do DynamoDB que serve como seu armazenamento de chaves, um nome lógico para o armazenamento de chaves e o ARN da chave KMS que identifica uma chave KMS de criptografia simétrica.

> **Note**
>
> Considere cuidadosamente o ARN da chave KMS que você especifica ao configurar estaticamente seu serviço de armazenamento de chaves. A `CreateKey` operação adiciona o ARN da chave KMS à sua lista de permissões do armazenamento de chaves da filial. Depois que uma chave KMS é adicionada ao armazenamento de chaves da filial, ela não pode ser excluída.

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)?;
```

Configuração do Discovery

O exemplo a seguir configura as principais ações de armazenamento para descoberta. Você deve especificar o nome da tabela do DynamoDB que serve como seu armazenamento de chaves e um nome lógico de armazenamento de chaves.

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()?;
```

Criar uma chave de ramificação ativa

Uma chave de ramificação é uma chave de dados derivada de uma AWS KMS key que o AWS KMS chaveiro hierárquico usa para reduzir o número de chamadas feitas. AWS KMS A chave

de ramificação ativa é a versão mais recente da chave de ramificação. O token de autenticação hierárquico gera uma chave de dados exclusiva para cada solicitação de criptografia e criptografa cada chave de dados com uma chave de encapsulamento exclusiva derivada da chave de ramificação ativa.

Para criar uma nova chave de ramificação ativa, você deve configurar estaticamente suas ações de armazenamento de chaves. `CreateKey`é uma operação privilegiada que adiciona o ARN da chave KMS especificado na configuração das ações do armazenamento de chaves à sua lista de permissões do armazenamento de chaves. Em seguida, a chave KMS é usada para gerar a nova chave de ramificação ativa. Recomendamos limitar o acesso a essa operação porque, depois que uma chave KMS é adicionada ao armazenamento de chaves, ela não pode ser excluída.

Recomendamos usar a `CreateKey` operação por meio da interface KeyStore Admin no plano de controle do seu aplicativo. Essa abordagem se alinha às melhores práticas para o gerenciamento de chaves.

Não crie chaves de ramificação no plano de dados. Essa prática pode resultar em:

- Chamadas desnecessárias para AWS KMS
- Várias chamadas simultâneas para ambientes de alta AWS KMS simultaneidade
- Várias TransactWriteItems chamadas para a tabela de apoio do DynamoDB.

A `CreateKey` operação inclui uma verificação de condição na `TransactWriteItems` chamada para evitar a substituição de chaves de ramificação existentes. No entanto, a criação de chaves no plano de dados ainda pode levar ao uso ineficiente de recursos e a possíveis problemas de desempenho.

Você pode colocar uma chave KMS na lista de permissões em seu armazenamento de chaves ou pode incluir várias chaves KMS na lista de permissões atualizando o ARN da chave KMS que você especificou na configuração de ações do armazenamento de chaves e chamando novamente. `CreateKey` Se você colocar várias chaves do KMS na lista de permissões, os usuários do armazenamento de chaves deverão configurar suas ações de armazenamento de chaves para descoberta, de forma que possam usar qualquer uma das chaves da lista de permissões ao qual tenham acesso. Para obter mais informações, consulte the section called “Configurar as principais ações do armazenamento”.

Permissões obrigatórias

Para criar chaves de ramificação, você precisa das ReEncrypt permissões kms: GenerateDataKeyWithoutPlaintext e kms: na chave KMS especificada nas ações do seu armazenamento de chaves.

Criar uma chave de ramificação

A operação a seguir cria uma nova chave de ramificação ativa usando a chave do KMS especificada na configuração de ações do armazenamento de chaves e adiciona a chave de ramificação ativa à tabela do DynamoDB que serve como armazenamento de chaves.

Ao chamar `CreateKey`, você pode optar por especificar os valores opcionais a seguir.

- `branchKeyIdentifier`: define um `branch-key-id` personalizado.

  Para criar um `branch-key-id` personalizado, você também deve incluir um contexto de criptografia adicional com o parâmetro `encryptionContext`.
- `encryptionContext`: define um conjunto opcional de pares de chave-valor não secretos que fornece dados autenticados adicionais (AAD) no contexto de criptografia incluído na chamada kms:. GenerateDataKeyWithoutPlaintext

  Esse contexto de criptografia adicional é exibido com o prefixo `aws-crypto-ec:`.

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();
```

Primeiro, a operação `CreateKey` gera os valores a seguir.

- Um Identificador Único Universal (UUID) versão 4 para o `branch-key-id` (a menos que você tenha especificado um `branch-key-id` personalizado).
- Um UUID da versão 4 para a versão da chave de ramificação
- Um `timestamp` no formato de data e hora ISO 8601 e em UTC (Tempo Universal Coordenado).

Em seguida, a `CreateKey` operação chama kms: GenerateDataKeyWithoutPlaintext usando a seguinte solicitação.

```
{
     "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**
>
> A operação `CreateKey` cria uma chave de ramificação ativa e uma chave de beacon, mesmo que você não tenha configurado seu banco de dados para criptografia pesquisável. Ambas as as as as chaves são armazenadas no armazenamento de chaves. Para obter mais informações, consulte Usar o token de autenticação hierárquico para criptografia pesquisável.

Em seguida, a `CreateKey` operação chama kms: ReEncrypt para criar um registro ativo para a chave de ramificação atualizando o contexto de criptografia.

Por último, a `CreateKey` operação chama ddb: TransactWriteItems para escrever um novo item que persistirá com a chave de ramificação na tabela que você criou na Etapa 2. O item tem os seguintes atributos:

```
{
      "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"
}
```

Alternar a chave de ramificação ativa

Só pode haver uma versão ativa para cada chave de ramificação por vez. Normalmente, cada versão ativa da chave de ramificação é usada para atender a várias solicitações. Porém, você controla até que ponto as chaves de ramificação ativas são reutilizadas e determina com que frequência a chave de ramificação ativa é alternada.

As chaves de ramificação não são usadas para criptografar chaves de dados em texto simples. Eles são usados para derivar as chaves de empacotamento exclusivas que criptografam chaves de dados de texto simples. O processo de derivação da chave de empacotamento produz uma chave de empacotamento exclusiva de 32 bytes com 28 bytes de randomização. Isso significa que uma chave de ramificação pode derivar mais de 79 octilhões, ou 2^{96}, chaves de empacotamento exclusivas antes que ocorra o desgaste criptográfico. Apesar desse risco de exaustão muito baixo, talvez seja necessário alternar suas chaves de ramificações ativas devido a regras comerciais ou contratuais ou regulamentações governamentais.

A versão ativa da chave de ramificação permanece ativa até que você a alterne. As versões anteriores da chave de ramificação ativa não serão usadas para realizar operações de criptografia e não podem ser usadas para derivar novas chaves de empacotamento, mas ainda podem ser consultadas e fornecer chaves de empacotamento para descriptografar as chaves de dados que eles criptografaram enquanto estavam ativos.

> **Warning**
>
> A exclusão de chaves de ramificação em ambientes de teste é irreversível. Você não pode recuperar chaves de ramificação excluídas. Quando você exclui e recria chaves de ramificação com a mesma ID em ambientes de teste, os seguintes problemas podem ocorrer:
>
> - Materiais de testes anteriores podem permanecer no cache
> - Alguns hosts ou threads de teste podem criptografar dados usando chaves de ramificação excluídas.
> - Os dados criptografados com ramificações excluídas não podem ser descriptografados
>
> Para evitar falhas de criptografia nos testes de integração:
>
> - Redefina a referência hierárquica do chaveiro antes de criar novas chaves de ramificação OU
> - Use uma chave de ramificação exclusiva IDs para cada teste

Permissões obrigatórias

Para girar as chaves de ramificação, você precisa das ReEncrypt permissões kms: GenerateDataKeyWithoutPlaintext e kms: na chave KMS especificada nas ações do seu armazenamento de chaves.

Alternar uma chave de ramificação ativa

Use a `VersionKey` operação para alternar sua chave de ramificação ativa. Quando você alterna a chave de ramificação ativa, uma nova chave de ramificação é criada para substituir a versão anterior. O `branch-key-id` não muda quando você alterna a chave de ramificação ativa. Você deve especificar o `branch-key-id` que identificará a chave de ramificação ativa atual quando você chamar `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?;
```

Tokens de autenticação

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados usa chaveiros para realizar a criptografia de envelopes. Tokens de autenticação geram, criptografam e descriptografam chaves de dados. Os tokens de autenticação determinam a origem das chaves de dados exclusivas que protegem cada registro criptografado, bem como as chaves de empacotamento que criptografam essa chave de dados. Você especifica um token de autenticação ao criptografar e especifica o mesmo ou outro token de autenticação ao descriptografar.

É possível usar cada token individualmente ou combiná-los em um multitoken de autenticação. Embora a maioria dos tokens de autenticação possa gerar, criptografar e descriptografar chaves de dados, você pode criar um que execute apenas uma operação, por exemplo, um token que gere apenas chaves de dados, e usá-lo em combinação com outros.

Recomendamos que você use um chaveiro que proteja suas chaves de agrupamento e execute operações criptográficas dentro de um limite seguro, como o AWS KMS chaveiro, que usa AWS KMS keys that never leave () sem criptografia. AWS Key Management ServiceAWS KMS Você também pode escrever um chaveiro que use chaves de agrupamento armazenadas em seus módulos de segurança de hardware (HSMs) ou protegidas por outros serviços de chave mestra.

O token de autenticação determina as chaves de encapsulamento que protegem as chaves de dados e, em última análise, os dados. Use as chaves de empacotamento mais seguras e práticas para sua tarefa. Sempre que possível, use chaves de empacotamento protegidas por um módulo de segurança de hardware (HSM) ou por uma infraestrutura de gerenciamento de chaves, como chaves do KMS em AWS Key Management Service(AWS KMS) ou chaves de criptografia em AWS CloudHSM.

O SDK AWS de criptografia de banco de dados fornece vários chaveiros e configurações de chaveiros, e você pode criar seus próprios chaveiros personalizados. Você também pode criar um multitoken de autenticação que inclua um ou mais tokens de autenticação do mesmo tipo ou de um tipo diferente.

Tópicos

- Como os tokens de autenticação funcionam
- AWS KMS chaveiros
- AWS KMS Chaveiros hierárquicos
- AWS KMS chaveiros ECDH
- Tokens de autenticação AES Raw
- Tokens de autenticação brutos do RSA
- Chaveiros ECDH brutos
- Multitokens de autenticação

Como os tokens de autenticação funcionam

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Quando você criptografa e assina um campo em seu banco de dados, o SDK de criptografia AWS de banco de dados solicita materiais de criptografia ao chaveiro. O token de autenticação retorna uma chave de dados de texto simples e uma cópia da chave que é criptografada por cada uma das chaves de empacotamento no token de autenticação e uma chave MAC associada à chave de dados. O SDK AWS de criptografia de banco de dados usa a chave de texto simples para criptografar os dados e, em seguida, remove a chave de dados de texto sem formatação da memória assim que possível. Em seguida, o SDK de criptografia de banco de dados da AWS adiciona uma descrição do material que inclui as chaves de dados criptografadas e outras informações, como instruções de criptografia e assinatura. O SDK AWS de criptografia de banco de dados usa a chave MAC para calcular códigos de autenticação de mensagens baseados em hash (HMACs) por meio da canonização da descrição do material e de todos os campos marcados com ou. `ENCRYPT_AND_SIGN` `SIGN_ONLY`

Ao descriptografar dados, você pode usar o mesmo token de autenticação usado para criptografar os dados ou um diferente. Para descriptografar os dados, um token de autenticação de decodificação deve ter acesso a pelo menos uma chave de empacotamento no token de autenticação de criptografia.

O SDK AWS de criptografia de banco de dados passa as chaves de dados criptografadas da descrição do material para o chaveiro e solicita que o chaveiro decifre qualquer uma delas. O token de autenticação usa suas chaves de empacotamento para descriptografar uma das chaves de dados criptografadas e retorna uma chave de dados de texto simples. O SDK de criptografia de banco de dados da AWS usa a chave de dados em texto simples para descriptografar os dados. Se nenhuma das chaves de empacotamento no token de autenticação puder descriptografar qualquer uma das chaves de dados criptografadas, a operação de descriptografia falhará.

Você pode usar um único token de autenticação ou também combinar tokens de autenticação do mesmo ou outro tipo em um multitoken de autenticação. Quando você criptografa dados, o multitoken de autenticação retorna uma cópia da chave de dados criptografada por todas as chaves de empacotamento em todos os tokens de autenticação que compreendem o multitoken de autenticação e uma chave MAC associada à chave de dados. É possível descriptografar os dados usando um token de autenticação com qualquer uma das chaves de empacotamento do multitoken de autenticação.

AWS KMS chaveiros

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Um AWS KMS chaveiro usa criptografia simétrica ou RSA assimétrica AWS KMS keyspara gerar, criptografar e descriptografar chaves de dados. AWS Key Management Service (AWS KMS) protege suas chaves KMS e executa operações criptográficas dentro do limite do FIPS. Recomendamos que você use um AWS KMS chaveiro ou um chaveiro com propriedades de segurança semelhantes, sempre que possível.

Você também pode usar uma chave KMS multirregional simétrica em um chaveiro. AWS KMS Para obter mais detalhes e exemplos de uso de várias regiões AWS KMS keys, consulteUsando a multirregião AWS KMS keys. Para obter mais informações sobre chaves multirregionais, consulte Usar chaves multirregionais no Guia do Desenvolvedor do AWS Key Management Service .

AWS KMS os chaveiros podem incluir dois tipos de chaves de embrulho:

- Chave geradora: gera uma chave de dados em texto simples e a criptografa. Um token de autenticação que criptografa dados deve ter uma chave geradora.

- Chaves adicionais: criptografa a chave de dados em texto simples gerada pela chave do gerador. AWS KMS os chaveiros podem ter zero ou mais chaves adicionais.

Você deve ter uma chave geradora para criptografar registros. Quando um AWS KMS chaveiro tem apenas uma AWS KMS chave, essa chave é usada para gerar e criptografar a chave de dados.

Como todos os chaveiros, os AWS KMS chaveiros podem ser usados de forma independente ou em um chaveiro múltiplo com outros chaveiros do mesmo tipo ou de um tipo diferente.

Tópicos

- Permissões necessárias para tokens de autenticação do AWS KMS
- Identificação AWS KMS keys em um AWS KMS chaveiro
- Criando um AWS KMS chaveiro
- Usando a multirregião AWS KMS keys
- Usando um chaveiro AWS KMS Discovery
- Usando um chaveiro de descoberta AWS KMS regional

Permissões necessárias para tokens de autenticação do AWS KMS

O SDK AWS de criptografia de banco de dados não exige um Conta da AWS e não depende de nenhum AWS service (Serviço da AWS). No entanto, para usar um AWS KMS chaveiro, você precisa de uma Conta da AWS e das seguintes permissões mínimas AWS KMS keys no seu chaveiro.

- Para criptografar com um AWS KMS chaveiro, você precisa da GenerateDataKey permissão kms: na chave do gerador. Você precisa da permissão KMS:Encrypt em todas as chaves adicionais no chaveiro. AWS KMS
- Para descriptografar com um AWS KMS chaveiro, você precisa da permissão kms:Decrypt em pelo menos uma chave no chaveiro. AWS KMS
- Para criptografar com um chaveiro múltiplo composto por AWS KMS chaveiros, você precisa da GenerateDataKey permissão kms: na chave do gerador no chaveiro do gerador. Você precisa da permissão KMS:Encrypt em todas as outras chaves em todos os outros chaveiros. AWS KMS
- Para criptografar com um AWS KMS chaveiro RSA assimétrico, você não precisa de kms: GenerateDataKey ou kms:Encrypt porque você deve especificar o material de chave pública que deseja usar para criptografia ao criar o chaveiro. Nenhuma AWS KMS chamada é feita ao

criptografar com este chaveiro. Para descriptografar com um AWS KMS chaveiro RSA assimétrico, você precisa da permissão KMS:Decrypt.

Para obter informações detalhadas sobre permissões para AWS KMS keys, consulte Autenticação e controle de acesso no Guia do AWS Key Management Service desenvolvedor.

Identificação AWS KMS keys em um AWS KMS chaveiro

Um AWS KMS chaveiro pode incluir um ou mais AWS KMS keys. Para especificar um AWS KMS key em um AWS KMS chaveiro, use um identificador de AWS KMS chave compatível. Os identificadores de chave que você pode usar para identificar um AWS KMS key em um chaveiro variam de acordo com a operação e a implementação da linguagem. Para obter detalhes sobre os identificadores de chave de uma AWS KMS key, consulteIdentificadores de chave no Guia do Desenvolvedor do AWS Key Management Service .

Como prática recomendada, use o identificador de chave mais específico que seja prático para sua tarefa.

- Para criptografar com um AWS KMS chaveiro, você pode usar um ID de chave, ARN de chave, nome de alias ou ARN de alias para criptografar dados.

  > **Note**
  >
  > Se você especificar um nome de alias ou um ARN de alias para uma chave do KMS em um token de autenticação de criptografia, a operação de criptografia salvará o ARN de chave atualmente associado ao alias nos metadados da chave de dados criptografada. Isso não salva o alias. As alterações no alias não afetam a chave do KMS usada para descriptografar suas chaves de dados criptografadas.

- Para decifrar com um AWS KMS chaveiro, você deve usar um ARN de chave para identificar. AWS KMS keys Para obter detalhes, consulte Seleção de chaves de encapsulamento.
- Em um token de autenticação usado para criptografia e descriptografia, você deve usar um ARN de chave para identificar AWS KMS keys.

Ao descriptografar, o SDK de criptografia AWS de banco de dados pesquisa no AWS KMS chaveiro uma AWS KMS key que possa descriptografar uma das chaves de dados criptografadas.

Especificamente, o SDK AWS de criptografia de banco de dados usa o seguinte padrão para cada chave de dados criptografada na descrição do material.

- O SDK AWS de criptografia de banco de dados obtém o ARN da chave que criptografou AWS KMS key a chave de dados a partir dos metadados da descrição do material.
- O SDK AWS de criptografia de banco de dados pesquisa no chaveiro de descriptografia por um ARN com AWS KMS key uma chave correspondente.
- Se encontrar um ARN AWS KMS key com uma chave correspondente no chaveiro, o SDK de criptografia de AWS banco de dados solicitará o uso da chave KMS AWS KMS para descriptografar a chave de dados criptografada.
- Caso contrário, ele passará para a próxima chave de dados criptografada, se houver.

Criando um AWS KMS chaveiro

Você pode configurar cada AWS KMS chaveiro com um único AWS KMS key ou vários AWS KMS keys iguais ou diferentes Contas da AWS e. Regiões da AWS O AWS KMS key deve ser uma chave de criptografia simétrica (`SYMMETRIC_DEFAULT`) ou uma chave RSA KMS assimétrica. Também é possível usar uma chave KMS multirregional criptografia simétrica. Você pode usar um ou mais AWS KMS chaveiros em um chaveiro múltiplo.

Você pode criar um AWS KMS chaveiro que criptografe e descriptografe dados, ou você pode criar AWS KMS chaveiros especificamente para criptografar ou descriptografar. Ao criar um AWS KMS chaveiro para criptografar dados, você deve especificar uma chave geradora, AWS KMS key que é usada para gerar uma chave de dados em texto simples e criptografá-la. A chave de dados não tem relação matemática com a chave KMS. Em seguida, se quiser, você pode especificar outras AWS KMS keys que criptografem a mesma chave de dados de texto sem formatação. Para descriptografar um campo criptografado protegido por esse chaveiro, o chaveiro de decodificação que você usa deve incluir pelo menos um dos definidos no chaveiro, ou não. AWS KMS keys AWS KMS keys(Um AWS KMS chaveiro sem AWS KMS keys é conhecido como chaveiro AWS KMS Discovery.)

Todas as chaves empacotadas em um token de autenticação de criptografia ou em vários tokens de autenticação devem ser capazes de criptografar a chave de dados. Se alguma chave de empacotamento falhar na criptografia, o método de criptografia falhará. Como resultado, o chamador deve ter as permissões necessárias para todas as chaves no token de autenticação. Se você usar um token de autenticação para criptografar dados, sozinho ou em um token de autenticação múltiplo, a operação de criptografia falhará.

Os exemplos a seguir usam o `CreateAwsKmsMrkMultiKeyring` método para criar um AWS KMS chaveiro com uma chave KMS de criptografia simétrica. O `CreateAwsKmsMrkMultiKeyring` método cria automaticamente o AWS KMS cliente e garante que o chaveiro manipule corretamente as chaves de região única e multirregião. Esses exemplos usam uma chave ARNs para identificar as chaves KMS. Para obter detalhes, consulte Identificação AWS KMS keys em um AWS KMS chaveiro

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?;
```

Os exemplos a seguir usam o `CreateAwsKmsRsaKeyring` método para criar um AWS KMS chaveiro com uma chave RSA KMS assimétrica. Para criar um AWS KMS chaveiro RSA assimétrico, forneça os seguintes valores.

- `kmsClient`: criar um novo AWS KMS cliente
- `kmsKeyID`: o ARN da chave que identifica sua chave RSA KMS assimétrica
- `publicKey`: a ByteBuffer de um arquivo PEM codificado em UTF-8 que representa a chave pública da chave para a qual você passou `kmsKeyID`
- `encryptionAlgorithm`: o algoritmo de criptografia deve ser `RSAES_OAEP_SHA_256` ou `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?;
```

Usando a multirregião AWS KMS keys

Você pode usar a multirregião AWS KMS keys como chaves de encapsulamento no SDK de criptografia de AWS banco de dados. Se você criptografar com uma chave multirregional em uma Região da AWS, poderá descriptografar usando uma chave multirregional relacionada em outra. Região da AWS

As chaves KMS multirregionais são um conjunto de AWS KMS keys chaves diferentes Regiões da AWS que têm o mesmo material de chave e ID de chave. É possível usar essas chaves relacionadas como se fossem a mesma chave em regiões diferentes. As chaves multirregionais oferecem suporte a cenários comuns de recuperação de desastres e backup que exigem criptografia em uma região e descriptografia em uma região diferente sem fazer uma chamada entre regiões para. AWS KMS Para obter mais informações sobre chaves multirregionais, consulte Usar chaves multirregionais no Guia do Desenvolvedor do AWS Key Management Service .

Para oferecer suporte a chaves multirregionais, o SDK AWS de criptografia de banco de dados inclui AWS KMS multi-Region-aware chaveiros. O método `CreateAwsKmsMrkMultiKeyring` oferece suporte a chaves de região única e de várias regiões.

- Para chaves de região única, o multi-Region-aware símbolo se comporta exatamente como o chaveiro de região única. AWS KMS Ele tenta descriptografar o texto cifrado somente com a chave de região única que criptografou os dados. Para simplificar sua experiência com o AWS KMS chaveiro, recomendamos usar o `CreateAwsKmsMrkMultiKeyring` método sempre que você usar uma chave KMS de criptografia simétrica.
- Para chaves multirregionais, o multi-Region-aware símbolo tenta descriptografar o texto cifrado com a mesma chave multirregional que criptografou os dados ou com a chave multirregional relacionada na região especificada.

Nos multi-Region-aware chaveiros que usam mais de uma chave KMS, você pode especificar várias chaves de região única e multirregião. No entanto, é possível especificar somente uma chave de cada conjunto de chaves de várias regiões relacionadas. Se você especificar mais de um identificador de chave com o mesmo ID de chave, a chamada do construtor falhará.

Os exemplos a seguir criam um AWS KMS chaveiro com uma chave KMS multirregional. Os exemplos especificam uma chave multirregional como chave geradora e uma chave de região única como chave secundária.

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?;
```

Ao usar AWS KMS chaveiros multirregionais, você pode descriptografar texto cifrado no modo estrito ou no modo de descoberta. Para descriptografar o texto cifrado no modo estrito, instancie o símbolo multi-Region-aware com o ARN da chave multirregional relacionada na região em que você está descriptografando o texto cifrado. Se você especificar o ARN da chave de uma chave multirregional relacionada em uma região diferente (por exemplo, a região em que o registro foi criptografado), o multi-Region-aware símbolo fará uma chamada entre regiões para isso. AWS KMS key

Ao descriptografar no modo estrito, o multi-Region-aware símbolo requer uma chave ARN. Ele aceita somente um ARN de chave de cada conjunto de chaves de várias regiões relacionadas.

Também é possível descriptografar no modo de descoberta com chaves do AWS KMS multirregionais. Ao descriptografar no modo de descoberta, você não especifica nenhuma AWS KMS keys. (Para obter informações sobre chaveiros de AWS KMS descoberta de uma única região, consulteUsando um chaveiro AWS KMS Discovery.)

Se você criptografou com uma chave multirregional, o multi-Region-aware símbolo no modo de descoberta tentará descriptografar usando uma chave multirregional relacionada na região local. Se não existir nenhuma, a chamada falhará. No modo de descoberta, o SDK AWS de criptografia de banco de dados não tentará fazer uma chamada entre regiões para a chave multirregional usada para criptografia.

Usando um chaveiro AWS KMS Discovery

Ao descriptografar, é uma prática recomendada especificar as chaves de encapsulamento que o SDK de criptografia de AWS banco de dados pode usar. Para seguir essa prática recomendada, use um chaveiro de AWS KMS decodificação que limite as chaves de AWS KMS encapsulamento às que você especificar. No entanto, você também pode criar um chaveiro de AWS KMS descoberta, ou seja, um AWS KMS chaveiro que não especifique nenhuma chave de agrupamento.

O SDK AWS de criptografia de banco de dados fornece um chaveiro de AWS KMS descoberta padrão e um chaveiro de descoberta para AWS KMS chaves multirregionais. Para obter informações sobre como usar chaves de várias regiões com o SDK de criptografia de banco de dados da AWS , consulte Usando a multirregião AWS KMS keys.

Como ele não especifica nenhuma chave de empacotamento, um token de autenticação de descoberta não pode criptografar dados. Se você usar um token de autenticação para criptografar dados, sozinho ou em um token de autenticação múltiplo, a operação de criptografia falhará.

Ao descriptografar, um chaveiro de descoberta permite que o SDK de criptografia AWS de banco de dados solicite AWS KMS a decodificação de qualquer chave de dados criptografada usando AWS KMS key aquela que a criptografou, independentemente de quem a possui ou tem acesso a ela. AWS KMS key A chamada será bem-sucedida somente quando o chamador tiver a permissão `kms:Decrypt` na AWS KMS key.

> **Important**
>
> Se você incluir um chaveiro de AWS KMS descoberta em um chaveiro de descriptografia múltiplo, o chaveiro de descoberta substituirá todas as restrições de chave KMS especificadas por outros chaveiros no chaveiro múltiplo. O token de autenticação múltiplo se comporta como o token de autenticação menos restritivo. Se você usar um token de autenticação de descoberta para criptografar dados, sozinho ou em um token de autenticação múltiplo, a operação de criptografia falhará

O SDK AWS de criptografia de banco de dados fornece um chaveiro de AWS KMS descoberta para sua conveniência. No entanto, recomendamos que você use um token de autenticação mais limitado sempre que possível pelas razões a seguir.

- Autenticidade — Um chaveiro de AWS KMS descoberta pode usar qualquer AWS KMS key chave usada para criptografar uma chave de dados na descrição do material, desde que o chamador tenha permissão para usá-la para descriptografar. AWS KMS key Isso pode não ser o AWS KMS key que o chamador pretende usar. Por exemplo, uma das chaves de dados criptografadas pode ter sido criptografada de forma menos segura AWS KMS key que qualquer pessoa possa usar.
- Latência e desempenho — Um chaveiro de AWS KMS descoberta pode ser visivelmente mais lento do que outros chaveiros porque o SDK de criptografia de AWS banco de dados tenta descriptografar todas as chaves de dados criptografadas, incluindo aquelas criptografadas por AWS KMS keys outras regiões Contas da AWS e AWS KMS keys que o chamador não tem permissão para usar para descriptografia.

Se você usa um chaveiro de descoberta, recomendamos que você use um filtro de descoberta para limitar as chaves KMS que podem ser usadas para aquelas em partições Contas da AWS e partições

especificadas. Para obter ajuda para encontrar seu ID de conta e partição, consulte Seus Conta da AWS identificadores e formato ARN no. Referência geral da AWS

Os exemplos de código a seguir instanciam um chaveiro de AWS KMS descoberta com um filtro de descoberta que limita as chaves KMS que o SDK de criptografia AWS de banco de dados pode usar às da partição e da `aws` conta de exemplo. `111122223333`

Antes de usar esse código, substitua os valores de exemplo Conta da AWS e de partição por valores válidos para sua partição Conta da AWS e. Se as chaves do KMS estiverem em regiões da China, use o valor de partição `aws-cn`. Se as chaves do KMS estiverem em AWS GovCloud (US) Regions, use o valor de partição `aws-us-gov`. Para todas as outras Regiões da AWS, use o valor de partição `aws`.

Java

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

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?;
```

Usando um chaveiro de descoberta AWS KMS regional

Um chaveiro de descoberta AWS KMS regional é um chaveiro que não especifica as chaves ARNs KMS. Em vez disso, ele permite que o SDK AWS de criptografia de banco de dados seja descriptografado usando somente as chaves KMS em particular. Regiões da AWS

Ao descriptografar com um chaveiro de descoberta AWS KMS regional, o SDK de criptografia de AWS banco de dados descriptografa qualquer chave de dados criptografada que tenha sido criptografada de acordo com um no especificado. AWS KMS key Região da AWS Para ter sucesso, o chamador deve ter `kms:Decrypt` permissão em pelo menos um dos AWS KMS keys itens especificados Região da AWS que criptografou uma chave de dados.

Como outros tokens de autenticação de descoberta, o token de autenticação de descoberta regional não tem efeito na criptografia. Ele funciona somente ao descriptografar campos criptografados. Se você usar um token de autenticação de descoberta regional em um token de autenticação múltiplo usado para criptografar e descriptografar, ele só será efetivo durante a descriptografia. Se você usar um token de autenticação de descoberta multirregional para criptografar dados, sozinho ou em um token de autenticação com vários tokens de autenticação, a operação de criptografia falhará.

> **Important**
>
> Se você incluir um chaveiro de descoberta AWS KMS regional em um chaveiro de descriptografia múltiplo, o chaveiro de descoberta regional substituirá todas as restrições de chave KMS especificadas por outros chaveiros no chaveiro múltiplo. O token de autenticação múltiplo se comporta como o token de autenticação menos restritivo. Um token de autenticação de descoberta do AWS KMS não tem efeito na criptografia quando usado sozinho ou em um multitoken de autenticação.

O chaveiro de descoberta regional no SDK AWS de criptografia de banco de dados tenta descriptografar somente com chaves KMS na região especificada. Ao usar um chaveiro de descoberta, você configura a região no AWS KMS cliente. Essas implementações do SDK de criptografia de AWS banco de dados não filtram as chaves do KMS por região, mas AWS KMS falharão na solicitação de descriptografia das chaves do KMS fora da região especificada.

Se você usa um chaveiro de descoberta, recomendamos usar um filtro de descoberta para limitar as chaves KMS usadas na descriptografia às chaves especificadas e nas partições. Contas da AWS

Por exemplo, o código a seguir cria um chaveiro de descoberta AWS KMS regional com um filtro de descoberta. Esse chaveiro limita o SDK AWS de criptografia de banco de dados às chaves KMS na conta 111122223333 na região Oeste dos EUA (Oregon) (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 Chaveiros hierárquicos

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

> **Note**
>
> Em 24 de julho de 2023, as chaves de ramificação criadas durante a versão prévia para desenvolvedores não são compatíveis. Crie novas chaves de ramificação para continuar usando o armazenamento de chaves que você criou durante a versão prévia para desenvolvedores.

Com o AWS KMS chaveiro hierárquico, você pode proteger seus materiais criptográficos com uma chave KMS de criptografia simétrica sem ligar AWS KMS toda vez que criptografar ou descriptografar um registro. É uma boa opção para aplicativos que precisam minimizar as chamadas e aplicativos que podem reutilizar alguns materiais criptográficos sem violar seus requisitos de segurança. AWS KMS

O chaveiro hierárquico é uma solução de armazenamento em cache de materiais criptográficos que reduz o número de AWS KMS chamadas usando chaves de ramificação AWS KMS protegidas persistentes em uma tabela do Amazon DynamoDB e, em seguida, armazenando localmente em cache materiais de chave de ramificação usados em operações de criptografia e descriptografia. A tabela do DynamoDB serve como o armazenamento de chaves que gerencia e protege as chaves de ramificação. Ele armazena a chave de ramificação ativa e todas as versões anteriores da chave de ramificação. A chave de ramificação ativa é a versão mais recente da chave de ramificação. O chaveiro hierárquico usa uma chave de criptografia de dados exclusiva para cada solicitação de criptografia e criptografa cada chave de criptografia de dados com uma chave de empacotamento exclusiva derivada da chave de ramificação ativa. O token de autenticação hierárquico depende da hierarquia estabelecida entre as chaves de ramificação ativas e suas chaves de agrupamento derivadas.

O token de autenticação hierárquico normalmente usa cada versão da chave de ramificação para atender a várias solicitações. Porém, você controla até que ponto as chaves de ramificação ativas são reutilizadas e determina com que frequência a chave de ramificação ativa é alternada. A versão ativa da chave de ramificação permanece ativa até que você a alterne. As versões anteriores da chave de ramificação ativa não serão usadas para realizar operações de criptografia, mas ainda podem ser consultadas e usadas em operações de descriptografia.

Quando você instancia o token de autenticação hierárquico, ele cria um cache local. Você especifica um limite de cache que define o tempo máximo em que os materiais da chave de ramificação são armazenados no cache local antes de expirarem e serem despejados do cache. O chaveiro hierárquico faz uma AWS KMS chamada para descriptografar a chave de ramificação e montar os

materiais da chave de ramificação na primeira vez em que a é especificado em uma `branch-key-id` operação. Em seguida, os materiais da chave de ramificação são armazenados no cache local e reutilizados para todas as operações de criptografia e descriptografia que especificam `branch-key-id` até que o limite do cache expire. Armazenar materiais de chave de filial no cache local reduz AWS KMS as chamadas. Por exemplo, considere um limite de cache de 15 minutos. Se você realizar 10.000 operações de criptografia dentro desse limite de cache, o AWS KMS chaveiro tradicional precisaria fazer 10.000 AWS KMS chamadas para satisfazer 10.000 operações de criptografia. Se você tiver um ativo`branch-key-id`, o chaveiro hierárquico só precisará fazer uma AWS KMS chamada para satisfazer 10.000 operações de criptografia.

O cache local separa os materiais de criptografia dos materiais de decodificação. Os materiais de criptografia são reunidos a partir da chave de ramificação ativa e reutilizados em todas as operações de criptografia até que o limite de cache expire. Os materiais de descriptografia são reunidos a partir do ID e da versão da chave de ramificação identificados nos metadados do campo criptografado e são reutilizados para todas as operações de descriptografia relacionadas ao ID e à versão da chave de ramificação até que o limite de cache expire. O cache local pode armazenar várias versões da mesma chave de ramificação ao mesmo tempo. Quando o cache local é configurado para usar umbranch key ID supplier, ele também pode armazenar materiais de chave de ramificação de várias chaves de ramificação ativas ao mesmo tempo.

> **Note**
>
> Todas as menções ao chaveiro hierárquico no SDK de criptografia de AWS banco de dados se referem ao chaveiro hierárquico. AWS KMS

Tópicos

- Como funciona
- Pré-requisitos
- Permissões obrigatórias
- Escolha um cache
- Criar um token de autenticação hierárquico
- Uso do token de autenticação hierárquico para criptografia pesquisável

Como funciona

As instruções a seguir descrevem como o token de autenticação hierárquico reúne materiais de criptografia e descriptografia e as diferentes chamadas que o token de autenticação faz para operações de criptografia e descriptografia. Para obter detalhes técnicos sobre a derivação da chave de empacotamento e os processos de criptografia da chave de dados em texto simples, consulte Detalhes técnicos do token de autenticação hierárquico do AWS KMS.

Criptografar e assinar

O passo a passo a seguir descreve como o token de autenticação hierárquico reúne materiais de criptografia e obtém uma chave de empacotamento exclusiva.

1. O método de criptografia solicita materiais de criptografia ao token de autenticação hierárquico. O chaveiro gera uma chave de dados em texto simples e, em seguida, verifica se há materiais de chave de ramificação válidos no cache local para gerar a chave de empacotamento. Se houver materiais de chave de filial válidos, o chaveiro prossegue para a Etapa 4.
2. Se não houver materiais de chave de ramificação válidos, o chaveiro hierárquico consulta o armazenamento de chaves em busca da chave de ramificação ativa.

    a. O armazenamento de chaves faz chamadas AWS KMS para descriptografar a chave de ramificação ativa e retorna a chave de ramificação ativa em texto simples. Os dados que identificam a chave de ramificação ativa são serializados para fornecer dados autenticados adicionais (AAD) na chamada de descriptografia para o AWS KMS.

    b. O armazenamento de chaves retorna a chave de ramificação em texto simples e os dados que a identificam, como a versão da chave de ramificação.
3. O token de autenticação hierárquico reúne materiais de chave de ramificação (a chave de ramificação em texto simples e a versão da chave de ramificação) e armazena uma cópia deles no cache local.
4. O token de autenticação hierárquico deriva uma chave de empacotamento exclusiva da chave de ramificação de texto simples e um sal aleatório de 16 bytes. Ele usa a chave de empacotamento derivada para criptografar uma cópia da chave de dados em texto simples.

O método de criptografia usa os materiais de criptografia para criptografar e assinar o registro. Para obter mais informações sobre como os registros são criptografados e assinados no SDK de criptografia de banco de dados da AWS , consulte Criptografar e assinar.

Descriptografar e verificar

O passo a passo a seguir descreve como o token de autenticação hierárquico reúne materiais de decodificação e decifra a chave de dados criptografada.

1. O método de descriptografia identifica a chave de dados criptografada no campo de descrição do material do registro criptografado e a passa para o token de autenticação hierárquico.
2. O token de autenticação hierárquico desserializa os dados que identificam a chave de dados criptografada, incluindo a versão da chave de ramificação, o sal de 16 bytes e outras informações que descrevem como a chave de dados foi criptografada.

   Para obter mais informações, consulte AWS KMS Detalhes técnicos do chaveiro hierárquico.
3. O token de autenticação hierárquico verifica se há materiais de chave de ramificação válidos no cache local que correspondam à versão da chave de ramificação identificada na Etapa 2. Se houver materiais de chave de ramificação válidos, o token de autenticação prosseguirá para a Etapa 6 .
4. Se não houver materiais de chave de ramificação válidos, o chaveiro hierárquico consulta o armazenamento de chaves em busca da chave de ramificação que corresponde à versão da chave de ramificação identificada na Etapa 2.

   a. O armazenamento de chaves faz chamadas AWS KMS para descriptografar a chave de ramificação e retorna a chave de ramificação ativa em texto simples. Os dados que identificam a chave de ramificação ativa são serializados para fornecer dados autenticados adicionais (AAD) na chamada de descriptografia para o AWS KMS.

   b. O armazenamento de chaves retorna a chave de ramificação em texto simples e os dados que a identificam, como a versão da chave de ramificação.
5. O token de autenticação hierárquico reúne materiais de chave de ramificação (a chave de ramificação em texto simples e a versão da chave de ramificação) e armazena uma cópia deles no cache local.
6. O token de autenticação hierárquico usa os materiais de chave de ramificação montados e o sal de 16 bytes identificado na Etapa 2 para reproduzir a chave de empacotamento exclusiva que criptografou a chave de dados.
7. O token de autenticação hierárquico usa a chave de empacotamento reproduzida para descriptografar a chave de dados e retorna a chave de dados em texto simples.

O método de decodificação usa os materiais de decodificação e a chave de dados de texto simples para descriptografar e verificar o registro. Para obter mais informações sobre como os registros

são descriptografados e verificados no SDK de criptografia de AWS banco de dados, consulte Descriptografar e verificar.

Pré-requisitos

Antes de criar e usar um chaveiro hierárquico, verifique se os seguintes pré-requisitos foram atendidos.

- Você, ou o administrador do armazenamento de chaves, criou um armazenamento de chaves e criou pelo menos uma chave de ramificação ativa.
- Você configurou suas principais ações de armazenamento.

> **Note**
>
> A forma como você configura suas ações de armazenamento de chaves determina quais operações você pode realizar e quais chaves KMS o chaveiro hierárquico pode usar. Para obter mais informações, consulte Principais ações do armazenamento.

- Você tem as AWS KMS permissões necessárias para acessar e usar as chaves de armazenamento de chaves e de ramificação. Para obter mais informações, consulte the section called “Permissões obrigatórias”.
- Você analisou os tipos de cache compatíveis e configurou o tipo de cache que melhor atende às suas necessidades. Para ter mais informações, consulte the section called “Escolha um cache”

Permissões obrigatórias

O SDK AWS de criptografia de banco de dados não exige um Conta da AWS e não depende de nenhum AWS service (Serviço da AWS). No entanto, para usar um chaveiro hierárquico, você precisa de uma Conta da AWS e das seguintes permissões mínimas sobre a (s) criptografia AWS KMS key(s) simétrica (s) em seu armazenamento de chaves.

- Para criptografar e descriptografar dados com o chaveiro hierárquico, você precisa do KMS:Decrypt.
- Para criar e girar chaves de ramificação, você precisa de kms: GenerateDataKeyWithoutPlaintext e kms:. ReEncrypt

Para obter mais informações sobre como controlar o acesso às chaves da filial e ao armazenamento de chaves, consultethe section called “Implementação de permissões de privilégio mínimo”.

Escolha um cache

O chaveiro hierárquico reduz o número de chamadas feitas ao AWS KMS armazenar em cache localmente os materiais de chave de ramificação usados nas operações de criptografia e descriptografia. Antes de criar seu chaveiro hierárquico, você precisa decidir que tipo de cache deseja usar. Você pode usar o cache padrão ou personalizar o cache para melhor atender às suas necessidades.

O chaveiro hierárquico suporta os seguintes tipos de cache:

- the section called “Cache padrão”
- the section called “MultiThreaded cache”
- the section called “StormTracking cache”
- the section called “Cache compartilhado”

Cache padrão

Para a maioria dos usuários, o cache Default atende aos requisitos de segmentação. O cache Default foi projetado para oferecer suporte a ambientes com muitos threads. Quando uma entrada de materiais de chave de ramificação expira, o cache padrão impede que vários segmentos sejam chamados, AWS KMS notificando um segmento de que a entrada de materiais de chave de ramificação expirará com 10 segundos de antecedência. Isso garante que somente um thread envie uma solicitação AWS KMS para atualizar o cache.

O padrão e StormTracking os caches oferecem suporte ao mesmo modelo de segmentação, mas você só precisa especificar a capacidade de entrada para usar o cache padrão. Para personalizações de cache mais granulares, use o. the section called “StormTracking cache”

A menos que você queira personalizar o número de entradas de materiais de chave de ramificação que podem ser armazenadas no cache local, você não precisa especificar um tipo de cache ao criar o chaveiro hierárquico. Se você não especificar um tipo de cache, o chaveiro hierárquico usa o tipo de cache padrão e define a capacidade de entrada como 1000.

Para personalizar o cache padrão, especifique os seguintes valores:

- Capacidade de entrada: limita o número de entradas de materiais de chave da ramificação que podem ser armazenadas no cache local.

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

O MultiThreaded cache é seguro para uso em ambientes com vários processos, mas não fornece nenhuma funcionalidade para minimizar as chamadas do Amazon AWS KMS DynamoDB. Como resultado, quando uma entrada de materiais de chave de ramificação expirar, todos os tópicos serão notificados ao mesmo tempo. Isso pode resultar em várias AWS KMS chamadas para atualizar o cache.

Para usar o MultiThreaded cache, especifique os seguintes valores:

- Capacidade de entrada: limita o número de entradas de materiais de chave da ramificação que podem ser armazenadas no cache local.
- Tamanho de entrada de limpeza de tail: define o número de entradas a serem limpas se a capacidade de entrada for atingida.

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

O StormTracking cache foi projetado para suportar ambientes altamente multisegmentados. Quando uma entrada de materiais de chave de ramificação expira, o StormTracking cache impede que vários segmentos sejam chamados AWS KMS notificando um segmento de que a entrada de materiais de chave de ramificação expirará com antecedência. Isso garante que somente um thread envie uma solicitação AWS KMS para atualizar o cache.

Para usar o StormTracking cache, especifique os seguintes valores:

- Capacidade de entrada: limita o número de entradas de materiais de chave da ramificação que podem ser armazenadas no cache local.

Valor padrão: 1000 entradas

- Tamanho de entrada de limpeza de tail: define o número de entradas de materiais de chave da ramificação a serem limpas por vez.

  Valor padrão: 1 entrada

- Período de carência: define o número de segundos antes da expiração em que é feita uma tentativa de atualizar os materiais de chave da ramificação.

  Valor padrão: 10 segundos

- Intervalo de carência: define o número de segundos entre as tentativas de atualizar os materiais de chave da ramificação.

  Valor padrão: 1 segundo

- Fan out: define o número de tentativas simultâneas que podem ser feitas para atualizar os materiais de chave da ramificação.

  Valor padrão: 20 tentativas

- Tempo de ativação (TTL) em trânsito: define o número de segundos até que uma tentativa de atualizar os materiais de chave de ramificação atinja o tempo limite. Sempre que o cache retorna `NoSuchEntry` em resposta a `GetCacheEntry`, essa chave de ramificação é considerada em trânsito até que a mesma chave seja gravada com uma entrada `PutCache`.

  Valor padrão: 10 segundos

- Sleep: define o número de segundos que um thread deve ficar em repouso se `fanOut` for excedido.

  Valor padrão: 20 milissegundos

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()?)
```

Cache compartilhado

Por padrão, o chaveiro hierárquico cria um novo cache local toda vez que você instancia o chaveiro. No entanto, o cache compartilhado pode ajudar a conservar memória, permitindo que você compartilhe um cache em vários chaveiros hierárquicos. Em vez de criar um novo cache de materiais criptográficos para cada chaveiro hierárquico que você instancia, o cache compartilhado armazena somente um cache na memória, que pode ser usado por todos os chaveiros hierárquicos que fazem referência a ele. O cache compartilhado ajuda a otimizar o uso da memória, evitando a duplicação

de materiais criptográficos nos chaveiros. Em vez disso, os chaveiros hierárquicos podem acessar o mesmo cache subjacente, reduzindo o consumo geral de memória.

Ao criar seu cache compartilhado, você ainda define o tipo de cache. Você pode especificar um the section called “Cache padrão”the section called “MultiThreaded cache”, ou the section called “StormTracking cache” como o tipo de cache ou substituir qualquer cache personalizado compatível.

Partições

Vários chaveiros hierárquicos podem usar um único cache compartilhado. Ao criar um chaveiro hierárquico com um cache compartilhado, você pode definir uma ID de partição opcional. O ID da partição distingue qual chaveiro hierárquico está sendo gravado no cache. Se dois chaveiros hierárquicos fizerem referência ao mesmo ID de partição e ID de chave de ramificaçãological key store name, os dois chaveiros compartilharão as mesmas entradas de cache no cache. Se você criar dois chaveiros hierárquicos com o mesmo cache compartilhado, mas com uma partição diferente IDs, cada chaveiro acessará somente as entradas do cache de sua própria partição designada no cache compartilhado. As partições atuam como divisões lógicas dentro do cache compartilhado, permitindo que cada chaveiro hierárquico opere de forma independente em sua própria partição designada, sem interferir com os dados armazenados na outra partição.

Se você pretende reutilizar ou compartilhar as entradas de cache em uma partição, você deve definir seu próprio ID de partição. Quando você passa a ID da partição para seu chaveiro hierárquico, o chaveiro pode reutilizar as entradas de cache que já estão presentes no cache compartilhado, em vez de precisar recuperar e reautorizar os materiais da chave de ramificação novamente. Se você não especificar uma ID de partição, uma ID de partição exclusiva será automaticamente atribuída ao chaveiro toda vez que você instanciar o chaveiro hierárquico.

Os procedimentos a seguir demonstram como criar um cache compartilhado com o tipo de cache padrão e passá-lo para um chaveiro hierárquico.

1. Crie um `CryptographicMaterialsCache` (CMC) usando a 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. Crie um `CacheType` objeto para o cache compartilhado.

   Passe o `sharedCryptographicMaterialsCache` que você criou na Etapa 1 para o novo `CacheType` objeto.

   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. Passe o `sharedCache` objeto da Etapa 2 para seu chaveiro hierárquico.

   Ao criar um chaveiro hierárquico com um cache compartilhado, você pode, opcionalmente, definir um `partitionID` para compartilhar entradas de cache em vários chaveiros hierárquicos. Se você não especificar uma ID de partição, o chaveiro hierárquico atribuirá automaticamente ao chaveiro uma ID de partição exclusiva.

> **Note**
>
> Seus chaveiros hierárquicos compartilharão as mesmas entradas de cache em um cache compartilhado se você criar dois ou mais chaveiros que façam referência ao mesmo ID de partição e ID de chave de logical key store name ramificação. Se você não quiser que vários chaveiros compartilhem as mesmas entradas de cache, use uma ID de partição exclusiva para cada chaveiro hierárquico.

O exemplo a seguir cria um chaveiro hierárquico com um branch key ID supplier limite de cache de 600 segundos. Para obter mais informações sobre os valores definidos na seguinte configuração de chaveiro hierárquico, consulte. the section called “Criar um token de autenticação hierárquico”

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?;
```

Criar um token de autenticação hierárquico

Para criar um chaveiro hierárquico, você deve fornecer os seguintes valores:

- Um nome de armazenamento de chaves

  O nome da tabela do DynamoDB que você, ou o administrador do armazenamento de chaves, criou para servir como seu armazenamento de chaves.

-

  Um tempo de vida do cache (TTL)

  A quantidade de tempo, em segundos, em que uma entrada de materiais de chave de ramificação no cache local pode ser usada antes de expirar. O limite de cache TTL determina a frequência com que o cliente liga AWS KMS para autorizar o uso das chaves de ramificação. Este valor deve ser maior que zero. Depois que o limite de cache TTL expirar, a entrada nunca será atendida e será removida do cache local.

- Um identificador de chave de ramificação

Você pode configurar estaticamente o `branch-key-id` que identifica uma única chave de ramificação ativa em seu armazenamento de chaves ou fornecer um fornecedor de ID de chave de filial.

O fornecedor da ID da chave de filial usa os campos armazenados no contexto de criptografia para determinar qual chave de ramificação é necessária para descriptografar um registro. Por padrão, somente as chaves de partição e classificação são incluídas no contexto de criptografia. No entanto, você pode usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica para incluir campos adicionais no contexto de criptografia.

É altamente recomendável usar um fornecedor de ID de chave de filial para bancos de dados de vários locatários em que cada inquilino tenha sua própria chave de filial. Você pode usar o fornecedor da ID da chave da filial para criar um nome amigável para a chave da filial, IDs a fim de facilitar o reconhecimento da ID correta da chave da filial para um inquilino específico. Por exemplo, o nome amigável permite que você se refira a uma chave de ramificação como `tenant1` em vez de `b3f61619-4d35-48ad-a275-050f87e15122`.

Para operações de descriptografia, você pode configurar estaticamente um único token de autenticação hierárquico para restringir a descriptografia a um único locatário ou usar o fornecedor da ID da chave da ramificação para identificar qual locatário é responsável por descriptografar um registro.

- (Opcional) Um cache

  Se você quiser personalizar o tipo de cache ou o número de entradas de materiais de chave de ramificação que podem ser armazenadas no cache local, especifique o tipo de cache e a capacidade de entrada ao inicializar o token de autenticação.

  O chaveiro hierárquico suporta os seguintes tipos de cache: Padrão, MultiThreaded StormTracking, e Compartilhado. Para obter mais informações e exemplos que demonstram como definir cada tipo de cache, consultethe section called “Escolha um cache”.

  Se você não especificar um cache, o token de autenticação hierárquico usará automaticamente o tipo de cache Default e definirá a capacidade de entrada como 1000.

- (Opcional) Uma ID de partição

Se você especificar othe section called “Cache compartilhado”, você pode, opcionalmente, definir uma ID de partição. O ID da partição distingue qual chaveiro hierárquico está sendo gravado no cache. Se você pretende reutilizar ou compartilhar as entradas de cache em uma partição, você deve definir seu próprio ID de partição. Você pode especificar qualquer string para o ID da partição. Se você não especificar uma ID de partição, uma ID de partição exclusiva será automaticamente atribuída ao chaveiro na criação.

Para obter mais informações, consulte Partitions.

> **Note**
>
> Seus chaveiros hierárquicos compartilharão as mesmas entradas de cache em um cache compartilhado se você criar dois ou mais chaveiros que façam referência ao mesmo ID de partição e ID de chave de logical key store name ramificação. Se você não quiser que vários chaveiros compartilhem as mesmas entradas de cache, use uma ID de partição exclusiva para cada chaveiro hierárquico.

- (Opcional) Uma lista de Tokens de Concessão

  Se você controlar o acesso à chave do KMS no token de autenticação hierárquico com concessões, deverá fornecer todos os tokens de concessão necessários ao inicializar o token de autenticação.

Crie um chaveiro hierárquico com uma ID de chave de ramificação estática

Os exemplos a seguir demonstram como criar um chaveiro hierárquico com um ID de chave de ramificação estáticothe section called “Cache padrão”, o e um TTL de limite de cache de 600 segundos.

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?;
```

Crie um chaveiro hierárquico com um fornecedor de ID de chave de filial

Os procedimentos a seguir demonstram como criar um chaveiro hierárquico com um fornecedor de ID de chave de filial.

1. Crie um fornecedor de ID de chave de filial

   O exemplo a seguir cria nomes amigáveis para as duas chaves de ramificação criadas na Etapa 1 e chama `CreateDynamoDbEncryptionBranchKeyIdSupplier` a criação de um fornecedor de ID de chave de filial com o cliente AWS Database Encryption SDK for DynamoDB.

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. Criar um token de autenticação hierárquico

Os exemplos a seguir inicializam um chaveiro hierárquico com o fornecedor de ID de chave de filial criado na Etapa 1, um limite de cache TLL de 600 segundos e um tamanho máximo de cache de 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?;
```

Uso do token de autenticação hierárquico para criptografia pesquisável

A criptografia pesquisável permite pesquisar registros criptografados sem descriptografar todo o banco de dados. Isso é feito indexando o valor de texto simples de um campo criptografado com um beacon. Para implementar a criptografia pesquisável, você deve usar um token de autenticação hierárquico.

A operação `CreateKey` de armazenamento de chaves gera uma chave de ramificação e uma chave de beacon. A chave de ramificação é usada em operações de criptografia e descriptografia de registros. A chave do beacon é usada para gerar beacons.

A chave de ramificação e a chave de beacon são protegidas pelo mesmo AWS KMS key que você especifica ao criar seu serviço de armazenamento de chaves. Depois que a `CreateKey` operação chama AWS KMS para gerar a chave de ramificação, ela chama kms: GenerateDataKeyWithoutPlaintext uma segunda vez para gerar a chave de beacon usando a seguinte solicitação.

```
{
   "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"
}
```

Depois de gerar as duas chaves, a `CreateKey` operação chama ddb: TransactWriteItems para escrever dois novos itens que manterão a chave de ramificação e a chave de beacon em seu armazenamento de chaves de ramificação.

Quando você configura um beacon padrão, o SDK do AWS Database Encryption consulta a chave do beacon no armazenamento de chaves. Em seguida, ele usa uma função de derivação de extract-and-expand chave baseada em HMAC (HKDF) para combinar a chave do farol com o nome do farol padrão para criar a chave HMAC para um determinado farol.

Ao contrário das chaves de ramificação, há apenas uma versão de chave de beacon `branch-key-id` em um armazenamento de chaves. A chave do beacon nunca é alternada.

Definição da fonte de chave de beacon

Ao definir a versão do beacon para seus beacons padrão e compostos, você deve identificar a chave do beacon e definir um limite de tempo de vida do cache (TTL) para os materiais da chave do beacon. Os materiais das chaves do beacon são armazenados em um cache local separado das chaves da ramificação. O trecho a seguir demonstra como definir o `keySource` para um banco de dados de locatário único. Identifique sua chave de beacon pelo `branch-key-id` que ela está associada.

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()?,
```

```
))
```

Definição da fonte do beacon em um banco de dados multilocatário

Se você tiver um banco de dados multilocatário, deverá especificar os valores a seguir ao configurar o `keySource`.

- keyFieldName

  Define o nome do campo que armazena o `branch-key-id` associado à chave de beacon usada para gerar beacons para um determinado locatário. O `keyFieldName` pode ser qualquer string, mas deve ser exclusiva para todos os outros campos do banco de dados. Quando você grava novos registros em seu banco de dados, a chave `branch-key-id` que identifica a chave de beacon usada para gerar quaisquer beacons para esse registro é armazenada nesse campo. Você deve incluir esse campo em suas consultas de beacon e identificar os materiais de chave de beacon apropriados necessários para recalcular o beacon. Para obter mais informações, consulte Consultar beacons em um banco de dados multilocatário.

- cacheTTL

  A quantidade de tempo, em segundos, em que uma entrada de materiais de chave de beacon no cache local pode ser usada antes de expirar. Esse valor deve ser maior que zero. Quando o limite de cache TTL expira, a entrada é removida do cache local.

- (Opcional) Um cache

  Se você quiser personalizar o tipo de cache ou o número de entradas de materiais de chave de ramificação que podem ser armazenadas no cache local, especifique o tipo de cache e a capacidade de entrada ao inicializar o token de autenticação.

  O chaveiro hierárquico suporta os seguintes tipos de cache: Padrão, MultiThreaded StormTracking, e Compartilhado. Para obter mais informações e exemplos que demonstram como definir cada tipo de cache, consultethe section called “Escolha um cache”.

  Se você não especificar um cache, o token de autenticação hierárquico usará automaticamente o tipo de cache Default e definirá a capacidade de entrada como 1000.

O exemplo a seguir cria um chaveiro hierárquico com um fornecedor de ID de chave de filial, um limite de cache (TLL) de 600 segundos e uma capacidade de entrada de 1.000.

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 chaveiros ECDH

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

> **Important**
>
> O chaveiro AWS KMS ECDH só está disponível na versão 1.5.0 ou posterior da Material Providers Library.

Um chaveiro AWS KMS ECDH usa um acordo de chave assimétrica AWS KMS keyspara derivar uma chave de embalagem simétrica compartilhada entre duas partes. Primeiro, o chaveiro usa o algoritmo de acordo de chaves Elliptic Curve Diffie-Hellman (ECDH) para derivar um segredo compartilhado da chave privada no par de chaves KMS do remetente e da chave pública do destinatário. Em seguida, o chaveiro usa o segredo compartilhado para derivar a chave de empacotamento compartilhada que protege suas chaves de criptografia de dados. A função de derivação de chave que o SDK AWS de criptografia de banco de dados usa (`KDF_CTR_HMAC_SHA384`) para derivar a chave de encapsulamento compartilhada está em conformidade com as recomendações do NIST para derivação de chaves.

A função de derivação de chave retorna 64 bytes de material de chaveamento. Para garantir que ambas as partes usem o material de codificação correto, o SDK AWS de criptografia de banco de dados usa os primeiros 32 bytes como chave de compromisso e os últimos 32 bytes como chave de empacotamento compartilhada. Na descriptografia, se o chaveiro não puder reproduzir a mesma chave de compromisso e chave de empacotamento compartilhada armazenadas no campo de descrição do material do registro criptografado, a operação falhará. Por exemplo, se você criptografar um registro com um chaveiro configurado com a chave privada de Alice e a chave pública de Bob, um chaveiro configurado com a chave privada de Bob e a chave pública de Alice reproduzirá a mesma chave de compromisso e chave de encapsulamento compartilhada e poderá descriptografar

o registro. Se a chave pública de Bob não for de um par de chaves KMS, Bob poderá criar um chaveiro ECDH bruto para descriptografar o registro.

O chaveiro AWS KMS ECDH criptografa registros com uma chave simétrica usando o AES-GCM. A chave de dados é então criptografada em envelope com a chave de empacotamento compartilhada derivada usando o AES-GCM. Cada chaveiro AWS KMS ECDH pode ter apenas uma chave de embrulho compartilhada, mas você pode incluir vários chaveiros AWS KMS ECDH, sozinhos ou com outros chaveiros, em um chaveiro múltiplo.

Tópicos

- Permissões necessárias para AWS KMS chaveiros ECDH
- Criando um AWS KMS chaveiro ECDH
- Criando um AWS KMS chaveiro de descoberta ECDH

Permissões necessárias para AWS KMS chaveiros ECDH

O SDK AWS de criptografia de banco de dados não exige uma AWS conta e não depende de nenhum AWS serviço. No entanto, para usar um chaveiro AWS KMS ECDH, você precisa de uma AWS conta e das seguintes permissões mínimas AWS KMS keys no seu chaveiro. As permissões variam de acordo com o esquema de contrato de chaves que você usa.

- Para criptografar e descriptografar registros usando o esquema de contrato de *`KmsPrivateKeyToStaticPublicKey`* chave, você precisa de kms: GetPublicKey e kms: DeriveSharedSecret no par de chaves KMS assimétrico do remetente. Se você fornecer diretamente a chave pública codificada em DER do remetente ao instanciar seu chaveiro, precisará apenas da DeriveSharedSecret permissão kms: no par de chaves KMS assimétrico do remetente.
- Para descriptografar registros usando o esquema de contrato de `KmsPublicKeyDiscovery` chaves, você precisa das GetPublicKey permissões kms: DeriveSharedSecret e kms: no par de chaves assimétrico KMS especificado.

Criando um AWS KMS chaveiro ECDH

Para criar um chaveiro AWS KMS ECDH que criptografe e descriptografe dados, você deve usar o esquema de contrato de chave. `KmsPrivateKeyToStaticPublicKey` Para inicializar um chaveiro

AWS KMS ECDH com o esquema de contrato de `KmsPrivateKeyToStaticPublicKey` chaves, forneça os seguintes valores:

- ID do remetente AWS KMS key

  Deve identificar um par de chaves KMS de curva elíptica (ECC) assimétrica recomendado pelo NIST com um valor de. `KeyUsage` `KEY_AGREEMENT` A chave privada do remetente é usada para derivar o segredo compartilhado.

- (Opcional) Chave pública do remetente

  Deve ser uma chave pública X.509 codificada por DER, também conhecida como `SubjectPublicKeyInfo` (SPKI), conforme definido na RFC 5280.

  A AWS KMS GetPublicKeyoperação retorna a chave pública de um par de chaves KMS assimétrico no formato codificado em DER exigido.

  Para reduzir o número de AWS KMS chamadas que seu chaveiro faz, você pode fornecer diretamente a chave pública do remetente. Se nenhum valor for fornecido para a chave pública do remetente, o chaveiro liga AWS KMS para recuperar a chave pública do remetente.

- Chave pública do destinatário

  Você deve fornecer a chave pública X.509 codificada em DER do destinatário, também conhecida como `SubjectPublicKeyInfo` (SPKI), conforme definido na RFC 5280.

  A AWS KMS GetPublicKeyoperação retorna a chave pública de um par de chaves KMS assimétrico no formato codificado em DER exigido.

- Especificação da curva

  Identifica a especificação da curva elíptica nos pares de chaves especificados. Os pares de chaves do remetente e do destinatário devem ter a mesma especificação de curva.

  Valores válidos: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

- (Opcional) Uma lista de Tokens de Concessão

  Se você controlar o acesso à chave KMS em seu chaveiro AWS KMS ECDH com concessões, deverá fornecer todos os tokens de concessão necessários ao inicializar o chaveiro.

C# / .NET

O exemplo a seguir cria um chaveiro AWS KMS ECDH com a chave KMS do remetente, a chave pública do remetente e a chave pública do destinatário. Este exemplo usa o `senderPublicKey` parâmetro opcional para fornecer a chave pública do remetente. Se você não fornecer a chave pública do remetente, o chaveiro liga AWS KMS para recuperar a chave pública do remetente. Os pares de chaves do remetente e do destinatário estão na `ECC_NIST_P256` curva.

```
// 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

O exemplo a seguir cria um chaveiro AWS KMS ECDH com a chave KMS do remetente, a chave pública do remetente e a chave pública do destinatário. Este exemplo usa o `senderPublicKey` parâmetro opcional para fornecer a chave pública do remetente. Se você não fornecer a chave

pública do remetente, o chaveiro liga AWS KMS para recuperar a chave pública do remetente. Os pares de chaves do remetente e do destinatário estão na `ECC_NIST_P256` curva.

```
// 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

O exemplo a seguir cria um chaveiro AWS KMS ECDH com a chave KMS do remetente, a chave pública do remetente e a chave pública do destinatário. Este exemplo usa o `sender_public_key` parâmetro opcional para fornecer a chave pública do remetente. Se você não fornecer a chave pública do remetente, o chaveiro liga AWS KMS para recuperar a chave pública do remetente.

```
// 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?;
```

Criando um AWS KMS chaveiro de descoberta ECDH

Ao descriptografar, é uma prática recomendada especificar as chaves que o SDK de criptografia de AWS banco de dados pode usar. Para seguir essa prática recomendada, use um chaveiro AWS KMS ECDH com o esquema de contrato de `KmsPrivateKeyToStaticPublicKey` chaves. No entanto, você também pode criar um chaveiro de descoberta AWS KMS ECDH, ou seja, um chaveiro AWS KMS ECDH que pode descriptografar qualquer registro em que a chave pública do par de chaves KMS especificado corresponda à chave pública do destinatário armazenada no campo de descrição do material do registro criptografado.

> **Important**
>
> Ao descriptografar registros usando o esquema de contrato de `KmsPublicKeyDiscovery` chave, você aceita todas as chaves públicas, independentemente de quem as possua.

Para inicializar um chaveiro AWS KMS ECDH com o esquema de contrato de `KmsPublicKeyDiscovery` chaves, forneça os seguintes valores:

- AWS KMS key ID do destinatário

  Deve identificar um par de chaves KMS de curva elíptica (ECC) assimétrica recomendado pelo NIST com um valor de. `KeyUsage` `KEY_AGREEMENT`
- Especificação da curva

  Identifica a especificação da curva elíptica no par de chaves KMS do destinatário.

  Valores válidos: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`
- (Opcional) Uma lista de Tokens de Concessão

  Se você controlar o acesso à chave KMS em seu chaveiro AWS KMS ECDH com concessões, deverá fornecer todos os tokens de concessão necessários ao inicializar o chaveiro.

C# / .NET

O exemplo a seguir cria um chaveiro de descoberta AWS KMS ECDH com um par de chaves KMS na curva. `ECC_NIST_P256` Você deve ter as DeriveSharedSecret permissões kms: GetPublicKey e kms: no par de chaves KMS especificado. Esse chaveiro pode descriptografar qualquer registro em que a chave pública do par de chaves KMS especificado corresponda à chave pública do destinatário armazenada no campo de descrição do material do registro criptografado.

```
// 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

O exemplo a seguir cria um chaveiro de descoberta AWS KMS ECDH com um par de chaves KMS na curva. ECC_NIST_P256 Você deve ter as DeriveSharedSecret permissões kms: GetPublicKey e kms: no par de chaves KMS especificado. Esse chaveiro pode descriptografar qualquer registro em que a chave pública do par de chaves KMS especificado corresponda à chave pública do destinatário armazenada no campo de descrição do material do registro criptografado.

```
// 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?;
```

Tokens de autenticação AES Raw

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados permite que você use uma chave simétrica AES que você fornece como uma chave de empacotamento que protege sua chave de dados. Você precisa gerar, armazenar e proteger o material de chaves, de preferência em um módulo de segurança de hardware (HSM) ou em um sistema de gerenciamento de chaves. Use um token de autenticação AES bruto quando precisar fornecer a chave de empacotamento e criptografar as chaves de dados local ou offline.

O token de autenticação bruto do AES usa o algoritmo AES-GCM e uma chave de empacotamento que você especifica como uma matriz de bytes para criptografar chaves de dados. É possível especificar somente uma chave de empacotamento em cada token de autenticação bruto do AES, mas você pode incluir vários tokens de autenticação brutos do AES, sozinhos ou com outros tokens de autenticação, em um multitoken de autenticação.

Nomes e namespaces de chaves

Para identificar a chave AES em um token de autenticação, o token de autenticação bruto do AES usa um namespace de chave e um nome de chave fornecidos por você. Esses valores não são secretos. Eles aparecem em texto simples na descrição do material que o SDK do AWS Database Encryption adiciona ao registro. Recomendamos usar um namespace de chave em seu HSM ou sistema de gerenciamento de chaves e um nome de chave que identifique a chave AES nesse sistema.

> **Note**
>
> O namespace e o nome da chave são equivalentes aos campos ID do provedor (ou provedor) e ID da chave no `JceMasterKey`.

Se você construir tokens de autenticação diferentes para criptografar e descriptografar um determinado campo, o namespace e os valores do nome são essenciais. Se o namespace e o nome da chave no token de autenticação de decodificação não corresponderem exatamente e com distinção entre maiúsculas e minúsculas ao namespace e ao nome da chave no token de autenticação de criptografia, o token de autenticação de decodificação não será usado, mesmo que os bytes do material da chave sejam idênticos.

Por exemplo, é possível definir um token de autenticação AES bruto com namespace `HSM_01` e nome de chave `AES_256_012`. Em seguida, você usa esse token de autenticação para criptografar alguns dados. Para descriptografar esses dados, construa um token de autenticação bruto do AES bruto com o mesmo namespace de chave, nome de chave e material de chave.

O exemplo a seguir mostra como criar um token de autenticação bruto do AES. A variável `AESWrappingKey` representa o material principal que você fornece.

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?;
```

Tokens de autenticação brutos do RSA

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O token de autenticação bruto do RSA realiza a criptografia e a descriptografia assimétricas das chaves de dados na memória local com chaves de empacotamento pública e privada fornecidas. Você precisa gerar, armazenar e proteger a chave privada, de preferência em um módulo de segurança de hardware (HSM) ou com o sistema de gerenciamento de chaves. A função de criptografia criptografa a chave de dados com chave pública do RSA. A função de descriptografia descriptografa a chave de dados usando a chave privada. É possível selecionar entre os vários modos de padding do RSA.

Um token de autenticação bruto do RSA que criptografa e descriptografa deve incluir uma chave pública e um par de chaves privadas assimétricas. No entanto, é possível criptografar dados com um token de autenticação bruto do RSA que tenha apenas uma chave pública e descriptografar dados com um token de autenticação bruto do RSA que tenha apenas uma chave privada. É possível incluir qualquer token de autenticação bruto do RSA em um multitoken de autenticação. Se você configurar um token de autenticação bruto do RSA com uma chave pública e privada, certifique-se de que eles façam parte do mesmo par de chaves.

O chaveiro RSA bruto é equivalente e interopera com o JceMasterKeyno AWS Encryption SDK for Java quando é usado com chaves de criptografia assimétrica RSA.

> **Note**
>
> O token de autenticação RSA bruto não oferece suporte a chaves do KMS assimétricas. Para usar chaves RSA KMS assimétricas, construa um token de autenticação do AWS KMS.

Namespaces e nomes

Para identificar a chave RSA em um token de autenticação, o token de autenticação bruto do RSA usa um namespace de chave e um nome de chave fornecidos por você. Esses valores não são secretos. Eles aparecem em texto simples na descrição do material que o SDK do AWS Database Encryption adiciona ao registro. Recomendamos usar um namespace de chave e um nome de

chave que identifique o par de chaves RSA (ou a sua chave privada) no HSM ou no sistema de gerenciamento de chaves..

> **Note**
>
> O namespace e o nome da chave são equivalentes aos campos ID do provedor (ou provedor) e ID da chave no `JceMasterKey`.

Se você construir tokens de autenticação diferentes para criptografar e descriptografar um determinado registro, o namespace e os valores do nome são essenciais. Se o namespace e o nome da chave no token de autenticação de decodificação não corresponderem exatamente e com distinção entre maiúsculas e minúsculas ao namespace e ao nome da chave no token de autenticação de criptografia, o token de autenticação de decodificação não será usado, mesmo que as chaves sejam do mesmo par de chaves.

O namespace da chave e o nome da chave do material da chave nos tokens de autenticação de criptografia e decodificação devem ser os mesmos, independentemente de o token de autenticação conter a chave pública RSA, a chave privada RSA ou ambas as chaves no par de chaves. Por exemplo, suponha que você criptografe dados com um token de autenticação RSA bruto para uma chave pública RSA com o namespace de chave `HSM_01` e nome de chave `RSA_2048_06`. Para descriptografar esses dados, construa um token de autenticação RSA bruto com a chave privada (ou par de chaves) e o mesmo namespace e nome de chave.

Modo de preenchimento

Você deve especificar um modo de preenchimento para tokens de autenticação RSA brutos usados para criptografia e descriptografia, ou usar atributos de sua implementação de linguagem que o especifiquem para você.

O AWS Encryption SDK suporta os seguintes modos de preenchimento, sujeitos às restrições de cada idioma. Recomendamos um modo de preenchimento OAEP, particularmente OAEP com SHA-256 e com preenchimento SHA-256. MGF1 O modo PKCS1de preenchimento é suportado somente para compatibilidade com versões anteriores.

- OAEP com SHA-1 e com preenchimento SHA-1 MGF1
- OAEP com SHA-256 e com preenchimento SHA-256 MGF1
- OAEP com SHA-384 e com preenchimento SHA-384 MGF1

- OAEP com SHA-512 e com preenchimento SHA-512 MGF1
- PKCS1 Preenchimento v1.5

O exemplo Java a seguir mostra como criar um chaveiro RSA bruto com a chave pública e privada de um par de chaves RSA e o OAEP com SHA-256 e com o modo de preenchimento SHA-256. MGF1 As variáveis `RSAPublicKey` e `RSAPrivateKey` representam o material principal que você fornece.

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?;
```

Chaveiros ECDH brutos

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Important

O chaveiro ECDH bruto só está disponível na versão 1.5.0 da Material Providers Library.

O chaveiro ECDH bruto usa os pares de chaves públicas-privadas de curva elíptica que você fornece para derivar uma chave de empacotamento compartilhada entre duas partes. Primeiro, o chaveiro obtém um segredo compartilhado usando a chave privada do remetente, a chave pública do destinatário e o algoritmo de acordo de chave Elliptic Curve Diffie-Hellman (ECDH). Em seguida, o chaveiro usa o segredo compartilhado para derivar a chave de empacotamento compartilhada que protege suas chaves de criptografia de dados. A função de derivação de chave que o SDK

AWS de criptografia de banco de dados usa (`KDF_CTR_HMAC_SHA384`) para derivar a chave de encapsulamento compartilhada está em conformidade com as recomendações do NIST para derivação de chaves.

A função de derivação de chave retorna 64 bytes de material de chaveamento. Para garantir que ambas as partes usem o material de codificação correto, o SDK AWS de criptografia de banco de dados usa os primeiros 32 bytes como chave de compromisso e os últimos 32 bytes como chave de empacotamento compartilhada. Na descriptografia, se o chaveiro não puder reproduzir a mesma chave de compromisso e chave de empacotamento compartilhada armazenadas no campo de descrição do material do registro criptografado, a operação falhará. Por exemplo, se você criptografar um registro com um chaveiro configurado com a chave privada de Alice e a chave pública de Bob, um chaveiro configurado com a chave privada de Bob e a chave pública de Alice reproduzirá a mesma chave de compromisso e chave de encapsulamento compartilhada e poderá descriptografar o registro. Se a chave pública de Bob for de um AWS KMS key par, Bob poderá criar um chaveiro AWS KMS ECDH para decifrar o registro.

O chaveiro Raw ECDH criptografa registros com uma chave simétrica usando o AES-GCM. A chave de dados é então criptografada em envelope com a chave de empacotamento compartilhada derivada usando o AES-GCM. Cada chaveiro Raw ECDH pode ter apenas uma chave de embrulho compartilhada, mas você pode incluir vários chaveiros Raw ECDH, sozinhos ou com outros chaveiros, em um chaveiro múltiplo.

Você é responsável por gerar, armazenar e proteger suas chaves privadas, preferencialmente em um módulo de segurança de hardware (HSM) ou sistema de gerenciamento de chaves. Os pares de chaves do remetente e do destinatário devem estar na mesma curva elíptica. O SDK AWS de criptografia de banco de dados é compatível com as seguintes especificações de curva elíptica:

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

Criando um chaveiro ECDH bruto

O chaveiro Raw ECDH suporta três esquemas de contrato principais:`RawPrivateKeyToStaticPublicKey`, e. `EphemeralPrivateKeyToStaticPublicKey` `PublicKeyDiscovery` O esquema de contrato de chave selecionado determina quais operações criptográficas você pode realizar e como os materiais de chaveamento são montados.

Tópicos

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

Use o esquema de contrato de `RawPrivateKeyToStaticPublicKey` chave para configurar estaticamente a chave privada do remetente e a chave pública do destinatário no chaveiro. Esse esquema de contrato de chave pode criptografar e descriptografar registros.

Para inicializar um chaveiro ECDH bruto com o esquema de contrato de `RawPrivateKeyToStaticPublicKey` chave, forneça os seguintes valores:

- Chave privada do remetente

  Você deve fornecer a chave privada codificada por PEM do remetente (PrivateKeyInfo estruturas PKCS #8), conforme definido na RFC 5958.

- Chave pública do destinatário

  Você deve fornecer a chave pública X.509 codificada em DER do destinatário, também conhecida como `SubjectPublicKeyInfo` (SPKI), conforme definido na RFC 5280.

  Você pode especificar a chave pública de um contrato de chave assimétrica (par de chaves KMS) ou a chave pública de um par de chaves gerado fora do. AWS

- Especificação da curva

  Identifica a especificação da curva elíptica nos pares de chaves especificados. Os pares de chaves do remetente e do destinatário devem ter a mesma especificação de curva.

  Valores válidos: `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

O exemplo Java a seguir usa o esquema de contrato de `RawPrivateKeyToStaticPublicKey` chave para configurar estaticamente a chave privada do remetente e a chave pública do destinatário. Ambos os pares de chaves estão na `ECC_NIST_P256` curva.

```
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

O exemplo de Python a seguir usa o esquema de contrato de `raw_ecdh_static_configuration` chave para configurar estaticamente a chave privada do remetente e a chave pública do destinatário. Ambos os pares de chaves devem estar na mesma curva.

```
// 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

Os chaveiros configurados com o esquema de contrato de `EphemeralPrivateKeyToStaticPublicKey` chaves criam um novo par de chaves localmente e derivam uma chave de empacotamento compartilhada exclusiva para cada chamada criptografada.

Esse esquema de contrato de chave só pode criptografar registros. Para descriptografar registros criptografados com o esquema de contrato de `EphemeralPrivateKeyToStaticPublicKey` chave, você deve usar um esquema de contrato de chave de descoberta configurado com a mesma chave pública do destinatário. Para descriptografar, você pode usar um chaveiro ECDH bruto com o algoritmo de acordo de chave ou, se a `PublicKeyDiscovery`chave pública do destinatário for de um par de chaves KMS de acordo de chave assimétrico, você pode AWS KMS usar um chaveiro ECDH com o esquema de contrato de chave. KmsPublicKeyDiscovery

Para inicializar um chaveiro ECDH bruto com o esquema de contrato de `EphemeralPrivateKeyToStaticPublicKey` chave, forneça os seguintes valores:

- Chave pública do destinatário

  Você deve fornecer a chave pública X.509 codificada em DER do destinatário, também conhecida como `SubjectPublicKeyInfo` (SPKI), conforme definido na RFC 5280.

  Você pode especificar a chave pública de um contrato de chave assimétrica (par de chaves KMS) ou a chave pública de um par de chaves gerado fora do. AWS

- Especificação da curva

  Identifica a especificação da curva elíptica na chave pública especificada.

  Ao criptografar, o chaveiro cria um novo par de chaves na curva especificada e usa a nova chave privada e a chave pública especificada para derivar uma chave de empacotamento compartilhada.

  Valores válidos: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `EphemeralPrivateKeyToStaticPublicKey` chaves. Ao criptografar, o chaveiro criará um novo par de chaves localmente na curva especificada`ECC_NIST_P256`.

```
// 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

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `EphemeralPrivateKeyToStaticPublicKey` chaves. Ao criptografar, o chaveiro criará um novo par de chaves localmente na curva especificada`ECC_NIST_P256`.

```
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

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `ephemeral_raw_ecdh_static_configuration` chaves. Ao criptografar, o chaveiro criará um novo par de chaves localmente na curva especificada.

```
// 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

Ao descriptografar, é uma prática recomendada especificar as chaves de encapsulamento que o SDK de criptografia de AWS banco de dados pode usar. Para seguir essa prática recomendada, use um chaveiro ECDH que especifique a chave privada do remetente e a chave pública do destinatário. No entanto, você também pode criar um chaveiro de descoberta de ECDH bruto, ou seja, um chaveiro ECDH bruto que pode descriptografar qualquer registro em que a chave pública da chave especificada corresponda à chave pública do destinatário armazenada no campo de descrição do material do registro criptografado. Esse esquema de contrato de chave só pode descriptografar registros.

> **Important**
>
> Ao descriptografar registros usando o esquema de contrato de `PublicKeyDiscovery` chave, você aceita todas as chaves públicas, independentemente de quem as possua.

Para inicializar um chaveiro ECDH bruto com o esquema de contrato de `PublicKeyDiscovery` chave, forneça os seguintes valores:

- Chave privada estática do destinatário

  Você deve fornecer a chave privada codificada por PEM do destinatário (PrivateKeyInfo estruturas PKCS #8), conforme definido na RFC 5958.

- Especificação da curva

  Identifica a especificação da curva elíptica na chave privada especificada. Os pares de chaves do remetente e do destinatário devem ter a mesma especificação de curva.

  Valores válidos: `ECC_NIST_P256`, `ECC_NIS_P384`, `ECC_NIST_P512`

C# / .NET

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `PublicKeyDiscovery` chaves. Esse chaveiro pode descriptografar qualquer registro em que a chave pública da chave privada especificada corresponda à chave pública do destinatário armazenada no campo de descrição do material do registro criptografado.

```
// 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

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `PublicKeyDiscovery` chaves. Esse chaveiro pode descriptografar qualquer registro em que a chave pública da chave privada especificada corresponda à chave pública do destinatário armazenada no campo de descrição do material do registro criptografado.

```
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

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `discovery_raw_ecdh_static_configuration` chaves. Esse chaveiro pode descriptografar qualquer mensagem em que a chave pública da chave privada especificada corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

```
// 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?;
```

Multitokens de autenticação

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

É possível combinar tokens de autenticação em um multitoken de autenticação. Um multitoken de autenticação é um token que consiste em um ou mais tokens de autenticação individuais do mesmo ou de outro tipo. O efeito é como se estivesse usando vários tokens de autenticação em uma série. Quando você usa um multitoken de autenticação para criptografar dados, qualquer uma das chaves de empacotamento em qualquer um de seus tokens de autenticação pode descriptografar esses dados.

Ao criar um multitoken de autenticação para criptografar dados, é possível designar um dos tokens de autenticação como o token de autenticação gerador. Todos os outros tokens de autenticação são conhecidos como tokens de autenticação filho. O token de autenticação gerador cria e criptografa a chave de dados em texto simples. Depois, todas as chaves de empacotamento em todos os tokens filho criptografam a mesma chave de dados em texto simples. O multitoken de autenticação retorna a chave em texto simples e uma chave de dados criptografada para cada chave de empacotamento do multitoken de autenticação. Se o chaveiro do gerador for um chaveiro KMS, a chave do gerador no AWS KMS chaveiro gera e criptografa a chave de texto simples. Em seguida, todas as chaves adicionais AWS KMS keys no AWS KMS chaveiro e todas as chaves de embrulho em todos os chaveiros secundários do chaveiro múltiplo criptografam a mesma chave de texto sem formatação.

Ao descriptografar, o SDK de criptografia AWS de banco de dados usa os chaveiros para tentar descriptografar uma das chaves de dados criptografadas. Os tokens de autenticação são chamados na ordem em que são especificados no multitoken de autenticação. O processamento para assim que qualquer chave em qualquer token de autenticação pode descriptografar uma chave de dados criptografada.

Para criar um multitoken de autenticação, primeiro instancie os tokens de autenticação filho. Neste exemplo, usamos um AWS KMS chaveiro e um chaveiro AES bruto, mas você pode combinar qualquer chaveiro compatível em um chaveiro múltiplo.

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?;
```

Em seguida, crie o multitoken de autenticação e especifique seu token gerador, se houver. Neste exemplo, criamos um chaveiro múltiplo no qual o chaveiro é o AWS KMS chaveiro do gerador e o chaveiro AES é o chaveiro infantil.

Java

O `CreateMultiKeyringInput` construtor Java permite definir um gerador de chaveiros e um chaveiro secundário. O objeto `createMultiKeyringInput` resultante é imutável.

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

C# / .NET

O construtor.NET `CreateMultiKeyringInput` permite definir um token de autenticação gerador e tokens de autenticação secundários. O objeto `CreateMultiKeyringInput` resultante é imutável.

```
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?;
```

Agora, é possível usar o multitoken de autenticação para criptografar e descriptografar dados.

Criptografia pesquisável

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

A criptografia pesquisável permite pesquisar registros criptografados sem descriptografar todo o banco de dados. Isso é feito usando beacons, que criam um mapa entre o valor de texto simples gravado em um campo e o valor criptografado que está realmente armazenado em seu banco de dados. O SDK AWS de criptografia de banco de dados armazena o beacon em um novo campo que ele adiciona ao registro. Dependendo do tipo de beacon que você usa, você pode realizar pesquisas de correspondência exata ou consultas complexas mais personalizadas em seus dados criptografados.

> **Note**
>
> A criptografia pesquisável no SDK AWS de criptografia de banco de dados difere da criptografia simétrica pesquisável definida em pesquisas acadêmicas, como a criptografia simétrica pesquisável.

Um beacon é uma tag truncada do código de autenticação de mensagens por hash (HMAC) que cria um mapa entre o texto simples e os valores criptografados de um campo. Quando você grava um novo valor em um campo criptografado configurado para criptografia pesquisável, o SDK de criptografia de AWS banco de dados calcula um HMAC sobre o valor de texto sem formatação. Essa saída de HMAC é uma correspondência de um para um (1:1) para o valor de texto sem formatação desse campo. A saída de HMAC é truncada para que vários valores de texto simples distintos sejam mapeados para a mesma etiqueta de HMAC truncada. Esses falsos positivos limitam a capacidade de um usuário não autorizado de identificar informações diferenciadas sobre o valor do texto sem formatação. Quando você consulta um beacon, o SDK de criptografia de banco de dados da AWS filtra automaticamente esses falsos positivos e retorna o resultado da sua consulta em texto simples.

O número médio de falsos positivos gerados para cada beacon é determinado pelo comprimento do beacon restante após o truncamento. Para obter ajuda na determinação do comprimento adequado do beacon para sua implementação, consulte Determinação do comprimento do beacon.

> **Note**
>
> A criptografia pesquisável foi projetada para ser implementada em bancos de dados novos e não preenchidos. Qualquer beacon configurado em um banco de dados existente mapeará somente os novos registros enviados para o banco de dados, não há como um beacon mapear os dados existentes.

Tópicos

- Os beacons são adequados para meu conjunto de dados?
- Cenário de criptografia pesquisável

Os beacons são adequados para meu conjunto de dados?

Usar beacons para realizar consultas de dados criptografados reduz o desempenho de custos associados aos banco de dados de criptografia do lado do cliente. Quando você usa beacons, há uma compensação inerente entre a eficiência de suas consultas e a quantidade de informações reveladas sobre a distribuição dos dados. O beacon não altera o estado criptografado do campo. Quando você criptografa e assina um campo com o SDK AWS de criptografia de banco de dados, o valor em texto simples do campo nunca é exposto ao banco de dados. O banco de dados armazena o valor aleatório e criptografado do campo.

Os beacons são armazenados junto com os campos criptografados a partir dos quais são calculados. Isso significa que, mesmo que um usuário não autorizado não consiga visualizar os valores de texto simples de um campo criptografado, ele poderá realizar análises estatísticas nos beacons para saber mais sobre a distribuição do seu conjunto de dados e, em casos extremos, identificar os valores de texto simples para os quais um beacon mapeia. A maneira como você configura seus beacons pode mitigar esses riscos. Em particular, escolher o comprimento correto do beacon pode ajudá-lo a preservar a confidencialidade do seu conjunto de dados.

Segurança versus desempenho

- Quanto menor o comprimento do beacon, mais segurança é preservada.

- Quanto maior o comprimento do beacon, mais desempenho é preservado.

A criptografia pesquisável pode não ser capaz de fornecer os níveis desejados de desempenho e segurança para todos os conjuntos de dados. Analise seu modelo de ameaça, requisitos de segurança e necessidades de desempenho antes de configurar qualquer beacon.

Considere os seguintes requisitos de exclusividade do conjunto de dados ao determinar se a criptografia pesquisável é adequada para seu conjunto de dados.

Distribuição

A quantidade de segurança preservada por um beacon depende da distribuição do seu conjunto de dados. Quando você configura um campo criptografado para criptografia pesquisável, o SDK AWS de criptografia de banco de dados calcula um HMAC sobre os valores de texto simples gravados nesse campo. Todos os beacons calculados para um determinado campo são calculados usando a mesma chave, com exceção dos bancos de dados multilocatários que usam uma chave distinta para cada locatário. Isso significa que, se o mesmo valor de texto simples for gravado no campo várias vezes, a mesma tag HMAC será criada para cada instância desse valor de texto sem formatação.

Você deve evitar construir beacons a partir de campos que contenham valores muito comuns. Por exemplo, considere um banco de dados que armazena o endereço de cada residente do estado de Illinois. Se você construir um beacon a partir do `City` campo criptografado, o beacon calculado sobre "Chicago" estará sobre-representado devido à grande porcentagem da população de Illinois que vive em Chicago. Mesmo que um usuário não autorizado possa ler apenas os valores criptografados e os valores do beacon, ele poderá identificar quais registros contêm dados para residentes de Chicago se o beacon preservar essa distribuição. Para minimizar a quantidade de informações distintivas reveladas sobre sua distribuição, você deve truncar suficientemente o beacon. O comprimento do beacon necessário para ocultar essa distribuição desigual tem custos de desempenho significativos que podem não atender às necessidades do seu aplicativo.

Você deve analisar cuidadosamente a distribuição do seu conjunto de dados para determinar o quanto seus beacons precisam ser truncados. O comprimento do beacon restante após o truncamento se correlaciona diretamente com a quantidade de informações estatísticas que podem ser identificadas sobre sua distribuição. Talvez seja necessário escolher comprimentos de beacon mais curtos para minimizar suficientemente a quantidade de informações distintivas reveladas sobre seu conjunto de dados.

Em casos extremos, você não pode calcular o comprimento do beacon para um conjunto de dados distribuído de forma desigual que equilibre efetivamente o desempenho e a segurança. Por exemplo, você não deve construir um beacon a partir de um campo que armazena o resultado de um exame médico para uma doença rara. Como se espera que os resultados de `NEGATIVE` sejam significativamente mais prevalentes no conjunto de dados, os resultados de `POSITIVE` podem ser facilmente identificados pela raridade. É muito difícil ocultar a distribuição quando o campo tem apenas dois valores possíveis. Se você usar um comprimento de beacon curto o suficiente para ocultar a distribuição, todos os valores de texto simples serão mapeados para a mesma tag HMAC. Se você usar um comprimento de beacon maior, é óbvio quais beacons são mapeados para valores `POSITIVE` de texto simples.

Correlação

É altamente recomendável que você evite construir beacons distintos a partir de campos com valores correlacionados. Os beacons construídos a partir de campos correlacionados exigem comprimentos de beacon mais curtos para minimizar suficientemente a quantidade de informações reveladas sobre a distribuição de cada conjunto de dados a um usuário não autorizado. Você deve analisar cuidadosamente o seu conjunto de dados, incluindo a sua entropia e distribuição conjunta de valores correlacionados, para determinar o quanto seus beacons precisam ser truncados. Se o comprimento do beacon resultante não atender às suas necessidades de desempenho, os beacons podem não ser adequados para seu conjunto de dados.

Por exemplo, você não deve construir dois beacons separados de campos `City` e `ZIPCode` porque o CEP provavelmente estará associado a apenas uma cidade. Normalmente, os falsos positivos gerados por um beacon limitam a capacidade de um usuário não autorizado de identificar informações diferenciadas sobre seu conjunto de dados. Mas a correlação entre os campos `City` e `ZIPCode` significa que um usuário não autorizado pode identificar facilmente quais resultados são falsos positivos e distinguir os diferentes CEPs.

Você deve evitar construir beacons a partir de campos que contenham os mesmos valores de texto simples. Por exemplo, você não deve construir um beacon a partir dos campos `mobilePhone` e `preferredPhone`, pois eles provavelmente têm os mesmos valores. Se você criar beacons distintos dos dois campos, o SDK de criptografia AWS de banco de dados criará os beacons para cada campo em chaves diferentes. Isso resulta em duas tags HMAC diferentes para o mesmo valor de texto simples. É improvável que os dois beacons distintos tenham os mesmos falsos positivos e um usuário não autorizado poderá distinguir números de telefone diferentes.

Mesmo que seu conjunto de dados contenha campos correlacionados ou tenha uma distribuição desigual, você poderá construir beacons que preservem a confidencialidade do seu conjunto de dados usando beacons menores. No entanto, o comprimento do beacon não garante que cada valor exclusivo em seu conjunto de dados produza vários falsos positivos que minimizem efetivamente a quantidade de informações distintivas reveladas sobre seu conjunto de dados. O comprimento do beacon estima apenas o número médio de falsos positivos produzidos. Quanto mais desigualmente distribuído seu conjunto de dados, menos efetivo é o comprimento do beacon na determinação do número médio de falsos positivos produzidos.

Considere cuidadosamente a distribuição dos campos a partir dos quais você constrói os beacons e considere o quanto você precisará truncar o comprimento do beacon para atender aos seus requisitos de segurança. Os tópicos a seguir neste capítulo pressupõem que seus beacons estejam distribuídos uniformemente e não contenham dados correlacionados.

Cenário de criptografia pesquisável

O exemplo a seguir demonstra uma solução de criptografia simples que pode ser pesquisada. No aplicativo, os campos de exemplo usados neste exemplo podem não atender às recomendações de exclusividade de distribuição e correlação para beacons. É possível usar esse exemplo para referência ao ler sobre os conceitos de criptografia pesquisável neste capítulo.

Considere um banco de dados chamado `Employees` que monitora os dados dos funcionários de uma empresa. Cada registro no banco de dados contém campos chamados EmployeeID LastName,, FirstName, e Address. Cada campo no banco de dados `Employees` é identificado pela chave primária `EmployeeID`.

Veja a seguir um exemplo de um registro de texto sem formatação no banco de dados.

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

Se você marcou os campos `LastName` e `FirstName` como `ENCRYPT_AND_SIGN` em suas ações criptográficas, os valores nesses campos são criptografados localmente antes de serem carregados no banco de dados. Os dados criptografados enviados são totalmente aleatórios, o banco de dados não reconhece esses dados como protegidos. Ele apenas detecta entradas de dados típicas. Isso significa que o registro que está realmente armazenado no banco de dados pode ter a seguinte aparência.

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

Se você precisar consultar o banco de dados para obter correspondências exatas no `LastName` campo, configure um farol padrão chamado LastNamepara mapear os valores de texto simples gravados no `LastName` campo para os valores criptografados armazenados no banco de dados.

Esse farol calcula a HMACs partir dos valores de texto simples no campo. `LastName` Cada saída HMAC é truncada para que não seja mais uma correspondência exata para o valor do texto sem formatação. Por exemplo, o hash completo e o hash truncado para `Jones` podem ter a seguinte aparência.

Hash completo

```
2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f
```

Hash truncado

```
b35099d408c833
```

Depois que o beacon padrão for configurado, você poderá realizar pesquisas de igualdade no campo `LastName`. Por exemplo, se você quiser pesquisar`Jones`, use o LastNamebeacon para realizar a consulta a seguir.

```
LastName = Jones
```

O SDK AWS de criptografia de banco de dados filtra automaticamente os falsos positivos e retorna o resultado em texto simples da sua consulta.

Beacons

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Um beacon é uma tag truncada do código de autenticação de mensagens por hash (HMAC) que cria um mapa entre o valor de texto simples e os valores criptografados que estão realmente armazenados no banco de dados. O beacon não altera o estado criptografado do campo. O beacon calcula um HMAC sobre o valor de texto simples do campo e o armazena junto com o valor criptografado. Essa saída de HMAC é uma correspondência de um para um (1:1) para o valor de texto sem formatação desse campo. A saída de HMAC é truncada para que vários valores de texto simples distintos sejam mapeados para a mesma etiqueta de HMAC truncada. Esses falsos positivos limitam a capacidade de um usuário não autorizado de identificar informações diferenciadas sobre o valor do texto sem formatação.

Os beacons só podem ser construídos a partir de campos `ENCRYPT_AND_SIGN` marcados ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` em suas ações criptográficas. `SIGN_ONLY` O beacon em si não está assinado nem criptografado. Você não pode construir um beacon com campos marcados com `DO_NOTHING`.

O tipo de beacon que você configura determina o tipo de consultas que você é capaz de realizar. Há dois tipos de beacons que oferecem suporte à criptografia pesquisável. Os beacons padrão realizam pesquisas de igualdade. Os beacons compostos combinam cadeias de texto simples literais e beacons padrão para realizar operações complexas de banco de dados. Depois de configurar os beacons, você deve configurar um índice secundário para cada beacon antes de poder pesquisar nos campos criptografados. Para obter mais informações, consulte Configuração de índices secundários com beacons.

Tópicos

- Beacons padrão
- Beacons compostos

Beacons padrão

Os beacons padrão são a maneira mais simples de implementar criptografia pesquisável em seu banco de dados. Eles só podem realizar pesquisas de igualdade para um único campo criptografado ou virtual. Para saber mais sobre a configuração de beacons padrão, consulte Configuração de beacons padrão.

O campo a partir do qual um beacon padrão é construído é chamado de fonte do beacon. Ele identifica a localização dos dados que o beacon precisa mapear. A fonte do beacon pode ser um campo criptografado ou um campo virtual. A fonte do beacon em cada beacon padrão deve ser exclusiva. Você não pode configurar dois beacons com a mesma fonte de beacon.

Os beacons padrão podem ser usados para realizar pesquisas de igualdade para um campo criptografado ou virtual. Ou, eles podem ser usados para construir beacons compostos para realizar operações de banco de dados mais complexas. Para ajudá-lo a organizar e gerenciar beacons padrão, o SDK de criptografia de AWS banco de dados fornece os seguintes estilos de beacon opcionais que definem o uso pretendido de um beacon padrão. Para obter mais informações, consulte Definindo estilos de beacon.

Você pode criar um farol padrão que realiza pesquisas de igualdade para um único campo criptografado ou pode criar um farol padrão que realiza pesquisas de igualdade na concatenação de vários camposENCRYPT_AND_SIGN,SIGN_ONLY, e SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT criando um campo virtual.

Campos virtuais

> Um campo virtual é um campo conceitual construído a partir de um ou mais campos de origem. A criação de um campo virtual não grava um novo campo em seu registro. O campo virtual não é armazenado explicitamente em seu banco de dados. Ele é usado na configuração de beacon padrão para fornecer instruções ao beacon sobre como identificar um segmento específico de um campo ou concatenar vários campos em um registro para realizar uma consulta específica. Um campo virtual exige pelo menos um campo criptografado.

> **Note**
>
> O exemplo a seguir demonstra os tipos de transformações e consultas que você pode realizar com um campo virtual. No aplicativo, os campos de exemplo usados neste exemplo podem não atender às recomendações de exclusividade de distribuição e correlação para beacons.

Por exemplo, se você quiser realizar pesquisas de igualdade na concatenação dos campos `FirstName` e `LastName`, você pode criar um dos campos virtuais a seguir.

- Um campo `NameTag` virtual, construído a partir da primeira letra do campo `FirstName`, seguida pelo campo `LastName`, tudo em minúsculas. Esse campo virtual permite que você consulte `NameTag=mjones`.
- Um campo `LastFirst` virtual, que é construído a partir do campo `LastName`, seguido pelo campo `FirstName`. Esse campo virtual permite que você consulte `LastFirst=JonesMary`.

Ou, se você quiser realizar pesquisas de igualdade em um segmento específico de um campo criptografado, crie um campo virtual que identifique o segmento que você deseja consultar.

Por exemplo, se você quiser consultar um campo `IPAddress` criptografado usando os três primeiros segmentos do endereço IP, crie o seguinte campo virtual.

- Um campo `IPSegment` virtual, construído a partir de `Segments('.', 0, 3)`. Esse campo virtual permite que você consulte `IPSegment=192.0.2`. A consulta retorna todos os registros com um valor `IPAddress` que começa com "192.0.2".

Os campos virtuais devem ser exclusivos. Dois campos virtuais não podem ser construídos exatamente a partir dos mesmos campos de origem.

Para obter ajuda na configuração de campos virtuais e os beacons que os usam, consulte Criar um campo virtual.

Beacons compostos

Os beacons compostos criam índices que melhoram o desempenho das consultas e permitem que você execute operações de banco de dados mais complexas. É possível usar beacons compostos para combinar cadeias de texto simples literais e beacons padrão para realizar consultas complexas em registros criptografados, como consultar dois tipos de registro diferentes de um único índice ou

consultar uma combinação de campos com uma chave de classificação. Para obter mais exemplos de soluções de beacon composto, consulte Escolher um tipo de beacon.

Os faróis compostos podem ser construídos a partir de faróis padrão ou uma combinação de faróis padrão e campos assinados. Eles são construídos a partir de uma lista de partes. Todos os beacons compostos devem incluir uma lista de partes criptografadas que identifique os campos `ENCRYPT_AND_SIGN` incluídos no beacon. Cada campo `ENCRYPT_AND_SIGN` deve ser identificado por um beacon padrão. Os faróis compostos mais complexos também podem incluir uma lista de partes assinadas que identificam o texto simples `SIGN_ONLY` ou os `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campos incluídos no farol e uma lista de partes do construtor que identificam todas as maneiras possíveis pelas quais o farol composto pode montar os campos.

> **Note**
>
> O SDK AWS de criptografia de banco de dados também oferece suporte a beacons assinados que podem ser configurados inteiramente a partir de texto simples `SIGN_ONLY` e campos. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` Os beacons assinados são um tipo de farol composto que indexa e executa consultas complexas em campos assinados, mas não criptografados. Para obter mais informações, consulte Criação de beacons assinados.

Para obter ajuda com a configuração de beacons compostos, consulte Configurar beacons compostos.

O tipo de beacon que você configura determina o beacon composto determina os tipos de consultas que você pode realizar. Por exemplo, você pode tornar algumas partes criptografadas e assinadas opcionais para permitir mais flexibilidade em suas consultas. Para obter mais informações sobre os tipos de consultas que os beacons compostos podem realizar, consulte Consultar beacons.

Planejar beacons

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Os beacons são projetados para serem implementados em bancos de dados novos e vazios. Qualquer beacon configurado em um banco de dados existente mapeará somente novos registros gravados no banco de dados. Os beacons são calculados a partir do valor de texto simples de um campo, uma vez que o campo é criptografado, não há como o beacon mapear os dados existentes. Depois de gravar novos registros com o beacon, não será possível atualizar a configuração do beacon. No entanto, é possível adicionar novos beacons aos novos campos que você adiciona ao seu registro.

Para implementar a criptografia pesquisável, você deve usar o token de autenticação hierárquico do AWS KMS para gerar, criptografar e descriptografar as chaves de dados usadas para proteger seus registros. Para obter mais informações, consulte Uso do token de autenticação hierárquico para criptografia pesquisável.

Antes de configurar beacons para criptografia pesquisável, você precisa analisar seus requisitos de criptografia, padrões de acesso ao banco de dados e modelo de ameaça para determinar a melhor solução para seu banco de dados.

O tipo de beacon que você configura determina o tipo de consultas que é possível realizar. O comprimento do beacon que você especifica na configuração padrão do beacon determina o número esperado de falsos positivos produzidos para um determinado beacon. É altamente recomendável identificar e planejar os tipos de consultas que você precisa realizar antes de configurar os beacons. Depois de usar um beacon, a configuração não poderá ser atualizada.

É altamente recomendável que você revise e conclua as tarefas a seguir antes de configurar qualquer beacon.

- Determine se os beacons são adequados para seu conjunto de dados
- Escolha um tipo de beacon
- Escolher um comprimento de beacon
- Escolha um nome de beacon

Lembre-se dos requisitos de exclusividade de beacon a seguir ao planejar a solução de criptografia pesquisável para seu banco de dados.

- Cada beacon padrão deve ter uma fonte de beacon exclusiva

  Vários beacons padrão não podem ser construídos a partir do mesmo campo criptografado ou virtual.

No entanto, um único beacon padrão pode ser usado para construir vários beacons compostos.

- Evite criar um campo virtual com campos de origem que se sobreponham aos beacons padrão existentes

  Construir um beacon padrão a partir de um campo virtual que contém um campo de origem usado para criar outro beacon padrão pode reduzir a segurança de ambos os beacons.

  Para obter mais informações, consulte Considerações de segurança para campos virtuais.

Considerações para bancos de dados multilocatários

Para consultar beacons configurados em um banco de dados multilocatário, você deve incluir o campo que armazena o `branch-key-id` associado ao locatário que criptografou o registro em sua consulta. Você define esse campo ao definir a fonte da chave do beacon. Para que a consulta seja bem-sucedida, o valor nesse campo deve identificar os materiais de chave de beacon apropriados necessários para recalcular o beacon.

Antes de configurar seus beacons, você deve decidir como planeja incluir `branch-key-id` em suas consultas. Para obter mais informações sobre as diferentes maneiras de incluir `branch-key-id` em suas consultas, consulte Consultar beacons em um banco de dados multilocatário.

Escolha de um tipo de beacon

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Com a criptografia pesquisável, você pode pesquisar registros criptografados mapeando os valores de texto simples em um campo criptografado com um beacon. O tipo de beacon que você configura determina o tipo de consultas que você pode realizar.

É altamente recomendável identificar e planejar os tipos de consultas que você precisa realizar antes de configurar os beacons. Depois de configurar os beacons, você deve configurar um índice secundário para cada beacon antes de poder pesquisar nos campos criptografados. Para obter mais informações, consulte Configuração de índices secundários com beacons.

Os beacons criam um mapa entre o valor de texto simples gravado em um campo e o valor criptografado que está realmente armazenado em seu banco de dados. Não é possível comparar os valores de dois beacons padrão, mesmo que eles contenham o mesmo texto simples subjacente. Os dois beacons padrão produzirão duas etiquetas de HMAC diferentes para os mesmos valores de texto simples. Como resultado, os beacons padrão não podem realizar as consultas a seguir.

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

Você só pode realizar as consultas acima se comparar as partes assinadas dos beacons compostos, com exceção do `CONTAINS` operador, que pode ser usado com beacons compostos para identificar o valor total de um campo criptografado ou assinado que o beacon montado contém. Ao comparar partes assinadas, você pode, opcionalmente, incluir o prefixo de uma parte criptografada, mas não pode incluir o valor criptografado de um campo. Para obter mais informações sobre os tipos de consultas que os beacons padrão e compostos podem realizar, consulte Consultar beacons.

Considere as seguintes soluções de criptografia pesquisáveis ao analisar seus padrões de acesso ao banco de dados. Os exemplos a seguir definem qual beacon configurar para atender aos diferentes requisitos de criptografia e consulta.

Beacons padrão

Os beacons padrão só podem realizar pesquisas de igualdade. É possível usar beacons padrão para realizar as consultas a seguir.

Consultar um único campo criptografado

Se você quiser identificar registros que contenham um valor específico para um campo criptografado, crie um beacon padrão.

Exemplos

Para o exemplo a seguir, considere um banco de dados chamado `UnitInspection` que monitora os dados de inspeção de uma instalação de produção. Cada registro no banco de dados contém campos chamados `work_id`, `inspection_date`, `inspector_id_last4` e `unit`. O ID completo do inspetor é um número entre 0 e 99.999.999. No entanto, para garantir que o

conjunto de dados seja distribuído uniformemente, ele armazena apenas os últimos quatro dígitos `inspector_id_last4` da ID do inspetor. Cada campo no banco de dados é identificado pela chave primária `work_id`. Os campos `inspector_id_last4` e `unit` são marcados `ENCRYPT_AND_SIGN` nas ações criptográficas.

Veja a seguir um exemplo de uma entrada de texto sem formatação no banco de dados `UnitInspection`.

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

Consultar um único campo criptografado em um registro

> Se o campo `inspector_id_last4` precisar ser criptografado, mas você ainda precisar consultá-lo para obter correspondências exatas, construa um beacon padrão a partir do campo `inspector_id_last4`. Em seguida, use o beacon padrão para criar um índice secundário. É possível usar esse índice secundário para consultar o campo `inspector_id_last4` criptografado.

Para obter ajuda sobre a configuração de beacons padrão, consulte Configuração de beacons padrão.

Consultar um campo virtual

Um campo virtual é um campo conceitual construído a partir de um ou mais campos de origem. Se você quiser realizar pesquisas de igualdade para um segmento específico de um campo criptografado ou realizar pesquisas de igualdade na concatenação de vários campos, construa um beacon padrão a partir de um campo virtual. Todos os campos virtuais devem incluir pelo menos um campo de origem criptografado.

Exemplos

Os exemplos a seguir criam campos virtuais para o banco de dados `Employees`. Veja a seguir um exemplo de um registro de texto sem formatação no banco de dados `Employees`.

```
{
```

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

Consultar um segmento de um campo criptografado

Neste exemplo, o campo `SSN` é criptografado.

Se você quiser consultar o campo `SSN` usando os últimos quatro dígitos de um número de previdência social, crie um campo virtual que identifique o segmento que você planeja consultar.

Um campo `Last4SSN` virtual, construído a partir de `Suffix(4)` permite que você faça consultas `Last4SSN=0000`. Use esse campo virtual para construir um beacon padrão. Em seguida, use o beacon padrão para criar um índice secundário. É possível usar esse índice secundário para fazer consultas no campo virtual. Essa consulta retorna todos os registros com um valor `SSN` que termina com os últimos quatro dígitos que você especificou.

Consultar a concatenação de vários campos

> **Note**
>
> O exemplo a seguir demonstra os tipos de transformações e consultas que você pode realizar com um campo virtual. No aplicativo, os campos de exemplo usados neste exemplo podem não atender às recomendações de exclusividade de distribuição e correlação para beacons.

Se você quiser realizar pesquisas de igualdade em uma concatenação dos campos `FirstName` e `LastName`, é possível criar um campo virtual `NameTag`, construído a partir da primeira letra do campo `FirstName`, seguida pelo campo `LastName`, tudo em minúsculas. Use esse campo virtual para construir um beacon padrão. Em seguida, use o beacon padrão para criar um índice secundário. É possível usar esse índice secundário para fazer consultas `NameTag=mjones` no campo virtual.

Pelo menos um dos campos de origem deve ser criptografado. `FirstName` ou `LastName` podem ser criptografados, ou ambos podem ser criptografados. Todos os campos de origem de texto simples devem ser marcados como `SIGN_ONLY` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` em suas ações criptográficas.

Para obter ajuda na configuração de campos virtuais e os beacons que os usam, consulte Criar um campo virtual.

Beacons compostos

Os beacons compostos criam um índice a partir de cadeias de texto simples literais e beacons padrão para realizar operações complexas de banco de dados. É possível usar beacons compostos para realizar as consultas a seguir.

Consulte uma combinação de campos criptografados em um único índice

Se você precisar consultar uma combinação de campos criptografados em um único índice, crie um beacon composto que combine os beacons padrão individuais construídos para cada campo criptografado para formar um único índice.

Depois de configurar o beacon composto, é possível criar um índice secundário que especifica o beacon composto como a chave de partição para realizar consultas de correspondência exata ou usar uma chave de classificação para realizar consultas mais complexas. Os índices secundários que especificam o beacon composto como chave de classificação podem realizar consultas de correspondência exata e consultas complexas mais personalizadas.

Exemplos

Para os exemplos a seguir, considere um banco de dados chamado `UnitInspection` que monitora os dados de inspeção de uma instalação de produção. Cada registro no banco de dados contém campos chamados `work_id`, `inspection_date`, `inspector_id_last4` e `unit`. O ID completo do inspetor é um número entre 0 e 99.999.999. No entanto, para garantir que o conjunto de dados seja distribuído uniformemente, ele armazena apenas os últimos quatro dígitos `inspector_id_last4` da ID do inspetor. Cada campo no banco de dados é identificado pela chave primária `work_id`. Os campos `inspector_id_last4` e `unit` são marcados `ENCRYPT_AND_SIGN` nas ações criptográficas.

Veja a seguir um exemplo de uma entrada de texto sem formatação no banco de dados `UnitInspection`.

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

Realizar pesquisas de igualdade em uma combinação de campos criptografados

Se você quiser consultar o banco de dados `UnitInspection` para obter correspondências exatas em `inspector_id_last4.unit`, primeiro crie beacons padrão distintos para os campos `inspector_id_last4` e `unit`. Em seguida, crie um beacon composto a partir dos dois beacons padrão.

Depois de configurar o beacon composto, crie um índice secundário que especifica o beacon composto como a chave de partição. Use esse índice secundário para consultar as correspondências exatas em `inspector_id_last4.unit`. Por exemplo, você pode consultar esse beacon para encontrar uma lista das inspeções que um inspetor realizou para uma determinada unidade.

Execute consultas complexas em uma combinação de campos criptografados

Se você quiser consultar o banco de dados `UnitInspection` em `inspector_id_last4` e `inspector_id_last4.unit`, primeiro crie beacons padrão distintos para os campos `inspector_id_last4` e `unit`. Em seguida, crie um beacon composto a partir dos dois beacons padrão.

Depois de configurar o beacon composto, crie um índice secundário que especifica o beacon composto como a chave de classificação. Use esse índice secundário para consultar o banco de dados `UnitInspection` em busca de entradas que começam com um determinado inspetor ou consultar o banco de dados para obter uma lista de todas as unidades dentro de um intervalo de ID de unidade específico que foram inspecionadas por um determinado inspetor. Também é possível realizar pesquisas de correspondência exata em `inspector_id_last4.unit`.

Para obter ajuda com a configuração de beacons compostos, consulte Configurar beacons compostos.

Consulte uma combinação de campos de texto simples criptografados em um único índice

Se você precisar consultar uma combinação de campos criptografados de texto simples em um único índice, crie um beacon composto que combine os beacons padrão individuais e campos de texto simples para formar um único índice. Os campos de texto simples usados para construir o farol composto devem estar marcados `SIGN_ONLY` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` em suas ações criptográficas.

Depois de configurar o beacon composto, é possível criar um índice secundário que especifica o beacon composto como a chave de partição para realizar consultas de correspondência exata ou usar uma chave de classificação para realizar consultas mais complexas. Os índices secundários que especificam o beacon composto como chave de classificação podem realizar consultas de correspondência exata e consultas complexas mais personalizadas.

Exemplos

Para os exemplos a seguir, considere um banco de dados chamado `UnitInspection` que monitora os dados de inspeção de uma instalação de produção. Cada registro no banco de dados contém campos chamados `work_id`, `inspection_date`, `inspector_id_last4` e `unit`. O ID completo do inspetor é um número entre 0 e 99.999.999. No entanto, para garantir que o conjunto de dados seja distribuído uniformemente, ele armazena apenas os últimos quatro dígitos `inspector_id_last4` da ID do inspetor. Cada campo no banco de dados é identificado pela chave primária `work_id`. Os campos `inspector_id_last4` e `unit` são marcados `ENCRYPT_AND_SIGN` nas ações criptográficas.

Veja a seguir um exemplo de uma entrada de texto sem formatação no banco de dados `UnitInspection`.

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

Realizar pesquisas de igualdade em uma combinação de campos

Se você quiser consultar o banco de dados `UnitInspection` para inspeções conduzidas por um inspetor específico em uma data específica, primeiro crie um beacon padrão para

o campo `inspector_id_last4`. O campo `inspector_id_last4` é marcado como `ENCRYPT_AND_SIGN` nas ações criptográficas. Todas as partes criptografadas exigem seu próprio beacon padrão. O campo `inspection_date` está marcado `SIGN_ONLY` e não requer um beacon padrão. Em seguida, crie um beacon composto a partir do campo `inspection_date` e do beacon `inspector_id_last4` padrão.

Depois de configurar o beacon composto, crie um índice secundário que especifica o beacon composto como a chave de partição. Use esse índice secundário para consultar os bancos de dados em busca de registros com correspondências exatas com um determinado inspetor e data de inspeção. Por exemplo, você pode consultar o banco de dados para obter uma lista de todas as inspeções realizadas pelo inspetor cujo ID termina em `8744` em uma data específica.

Execute consultas complexas em uma combinação de campos

Se você quiser consultar o banco de dados para inspeções conduzidas dentro de um intervalo `inspection_date`, ou consultar o banco de dados para inspeções conduzidas em uma determinada `inspection_date` restringida por `inspector_id_last4` ou `inspector_id_last4.unit`, primeiro crie beacons padrão distintos para os campos `inspector_id_last4` e `unit`. Em seguida, crie um beacon composto a partir do campo `inspection_date` de texto simples e dos dois beacons padrão.

Depois de configurar o beacon composto, crie um índice secundário que especifica o beacon composto como a chave de classificação. Use esse índice secundário para realizar consultas para inspeções realizadas em datas específicas por um inspetor específico. Por exemplo, você pode consultar o banco de dados para obter uma lista de todas as unidades inspecionadas na mesma data. Ou você pode consultar o banco de dados para obter uma lista de todas as inspeções realizadas em uma unidade específica entre um determinado intervalo de datas de inspeção.

Para obter ajuda com a configuração de beacons compostos, consulte Configurar beacons compostos.

Escolher um comprimento de beacon

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Quando você grava um novo valor em um campo criptografado configurado para criptografia pesquisável, o SDK de criptografia de AWS banco de dados calcula um HMAC sobre o valor de texto sem formatação. Essa saída de HMAC é uma correspondência de um para um (1:1) para o valor de texto sem formatação desse campo. A saída de HMAC é truncada para que vários valores de texto simples distintos sejam mapeados para a mesma etiqueta de HMAC truncada. Essas colisões, ou falsos positivos, limitam a capacidade de um usuário não autorizado de identificar informações diferenciadas sobre o valor do texto sem formatação.

O número médio de falsos positivos gerados para cada beacon é determinado pelo comprimento do beacon restante após o truncamento. Você só precisa definir o comprimento do beacon ao configurar os beacons padrão. Os beacons compostos usam os comprimentos dos beacons padrão a partir dos quais são construídos.

O beacon não altera o estado criptografado do campo. No entanto, quando você usa beacons, há uma compensação inerente entre a eficiência de suas consultas e a quantidade de informações reveladas sobre a distribuição dos dados.

O objetivo da criptografia pesquisável é reduzir os custos de desempenho associados aos bancos de dados criptografados do lado do cliente usando beacons para realizar consultas em dados criptografados. Os beacons são armazenados junto com os campos criptografados a partir dos quais são calculados. Isso significa que eles podem revelar informações diferenciadas sobre a distribuição do seu conjunto de dados. Em casos extremos, um usuário não autorizado pode analisar as informações reveladas sobre sua distribuição e usá-las para identificar o valor em texto simples de um campo. Escolher o comprimento certo do beacon pode ajudar a mitigar esses riscos e preservar a confidencialidade de sua distribuição.

Analise seu modelo de ameaça para determinar o nível de segurança de que você precisa. Por exemplo, quanto mais pessoas tiverem acesso ao seu banco de dados, mas não devem ter acesso aos dados em texto simples, mais você pode querer proteger a confidencialidade da distribuição do conjunto de dados. Para aumentar a confidencialidade, um beacon precisa gerar mais falsos positivos. O aumento da confidencialidade resulta na redução do desempenho das consultas.

Segurança versus desempenho

- Um comprimento de beacon muito longo produz poucos falsos positivos e pode revelar informações diferenciadas sobre a distribuição do seu conjunto de dados.
- Um comprimento de beacon muito curto produz muitos falsos positivos e aumenta o custo de desempenho das consultas porque exige uma varredura mais ampla do banco de dados.

Ao determinar o comprimento adequado do beacon para sua solução, você deve encontrar um comprimento que preserve adequadamente a segurança de seus dados sem afetar o desempenho de suas consultas mais do que o absolutamente necessário. A quantidade de segurança preservada por um beacon depende da distribuição do seu conjunto de dados e da correlação dos campos a partir dos quais seus beacons são construídos. Os tópicos a seguir pressupõem que seus beacons estejam distribuídos uniformemente e não contenham dados correlacionados.

Tópicos

- Cálculo do tamanho do beacon
- Exemplo

Cálculo do tamanho do beacon

O comprimento do beacon é definido em bits e se refere ao número de bits da tag HMAC que são mantidos após o truncamento. O comprimento recomendado do beacon varia de acordo com a distribuição do conjunto de dados, a presença de valores correlacionados e seus requisitos específicos de segurança e desempenho. Se seu conjunto de dados estiver distribuído uniformemente, você poderá usar as equações e procedimentos a seguir para ajudar a identificar o melhor comprimento de beacon para sua implementação. Essas equações estimam apenas o número médio de falsos positivos que o beacon produzirá, mas não garantem que cada valor exclusivo em seu conjunto de dados produza um número específico de falsos positivos.

> **Note**
>
> A eficácia dessas equações depende da distribuição do seu conjunto de dados. Se seu conjunto de dados não estiver distribuído uniformemente, consulte. Os beacons são adequados para meu conjunto de dados?
> Em geral, quanto mais longe seu conjunto de dados estiver de uma distribuição uniforme, mais você precisará reduzir o comprimento do beacon.

1.

   Estimar a população

   A população é o número esperado de valores exclusivos no campo a partir do qual seu beacon padrão é construído, não é o número total esperado de valores armazenados no campo. Por exemplo, considere um `Room` campo criptografado que identifica o local das reuniões dos

funcionários. Espera-se que o `Room` campo armazene 100.000 valores totais, mas existem apenas 50 salas diferentes que os funcionários podem reservar para reuniões. Isso significa que a população é 50 porque há apenas 50 valores exclusivos possíveis que podem ser armazenados no `Room` campo.

> **Note**
>
> Se seu beacon padrão for construído a partir de um campo virtual, a população usada para calcular o comprimento do beacon é o número de combinações exclusivas criadas pelo campo virtual.

Ao estimar sua população, não se esqueça de considerar o crescimento projetado do conjunto de dados. Depois de gravar novos registros com o beacon, não será possível atualizar o comprimento do beacon. Analise seu modelo de ameaças e todas as soluções de banco de dados existentes para criar uma estimativa do número de valores exclusivos que você espera que esse campo armazene nos próximos cinco anos.

A sua população não precisa ser precisa. Primeiro, identifique o número de valores exclusivos em seu banco de dados atual ou estime o número de valores exclusivos que você espera armazenar no primeiro ano. Em seguida, use as perguntas a seguir para ajudá-lo a determinar o crescimento projetado de valores exclusivos nos próximos cinco anos.

- Você espera que os valores exclusivos se multipliquem por 10?
- Você espera que os valores exclusivos se multipliquem por 100?
- Você espera que os valores exclusivos se multipliquem por 1000?

A diferença entre 50.000 e 60.000 valores exclusivos não é significativa e ambos resultarão no mesmo comprimento de beacon recomendado. No entanto, a diferença entre 50.000 e 500.000 valores exclusivos afetará significativamente o comprimento recomendado do beacon.

Considere analisar os dados públicos com base na frequência de tipos de dados comuns, como códigos postais ou sobrenomes. Por exemplo, existem 41.707 CEPs nos Estados Unidos. A população que você usa deve ser proporcional ao seu próprio banco de dados. Se o `ZIPCode` campo em seu banco de dados incluir dados de todos os Estados Unidos, você poderá definir sua população como 41.707, mesmo que o `ZIPCode` campo não tenha atualmente 41.707 valores exclusivos. Se o `ZIPCode` campo em seu banco de dados incluir apenas dados de um

único estado e sempre incluirá dados de um único estado, você poderá definir sua população como o número total de CEPs nesse estado, em vez de 41.704.

2. Calcule a faixa recomendada para o número esperado de colisões

   Para determinar o comprimento adequado do beacon para um determinado campo, você deve primeiro identificar um intervalo apropriado para o número esperado de colisões. O número esperado de colisões representa o número médio esperado de valores de texto simples exclusivos que são mapeados para uma tag HMAC específica. O número esperado de falsos positivos para um valor de texto simples exclusivo é um a menos do que o número esperado de colisões.

   Recomendamos que o número esperado de colisões seja maior ou igual a dois e menor que a raiz quadrada da sua população. As equações a seguir só funcionam se sua população tiver 16 ou mais valores exclusivos.

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

   Se o número de colisões for menor que dois, o beacon produzirá poucos falsos positivos. Recomendamos dois como o número mínimo de colisões esperadas, pois isso significa que, em média, cada valor exclusivo no campo gerará pelo menos um falso positivo ao ser mapeado para outro valor exclusivo.

3. Calcule o intervalo recomendado para comprimentos de beacon

   Depois de identificar o número mínimo e máximo de colisões esperadas, use a equação a seguir para identificar uma faixa de comprimentos de beacon apropriados.

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

   Primeiro, resolva o comprimento do beacon em que o número de colisões esperadas é igual a dois (o número mínimo recomendado de colisões esperadas).

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

   Em seguida, calcule o comprimento do beacon em que o número esperado de colisões é igual à raiz quadrada da sua população (o número máximo recomendado de colisões esperadas).

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

Recomendamos arredondar a saída produzida por essa equação para o menor comprimento do beacon. Por exemplo, se a equação produzir um comprimento de beacon de 15,6, recomendamos arredondar esse valor para 15 bits em vez de arredondar para 16 bits.

4. Escolher um comprimento de beacon

   Essas equações identificam apenas uma faixa recomendada de comprimentos de beacon para seu campo. Recomendamos usar um comprimento de beacon menor para preservar a segurança do seu conjunto de dados sempre que possível. No entanto, o comprimento do beacon que você realmente usa é determinado pelo seu modelo de ameaça. Considere seus requisitos de desempenho ao analisar seu modelo de ameaça para determinar o melhor comprimento do beacon para seu campo.

   Usar um comprimento de beacon menor reduz o desempenho da consulta, enquanto usar um comprimento de beacon maior diminui a segurança. Em geral, se seu conjunto de dados estiver distribuído de forma desigual ou se você construir beacons distintos a partir de campos correlacionados, precisará usar beacons menores para minimizar a quantidade de informações reveladas sobre a distribuição de seus conjuntos de dados.

   Se você analisar seu modelo de ameaça e decidir que qualquer informação distintiva revelada sobre a distribuição de um campo não representa uma ameaça à sua segurança geral, você pode optar por usar um comprimento de beacon maior do que o intervalo recomendado calculado. Por exemplo, se você calculou o intervalo recomendado de comprimentos de beacon para um campo como 9 a 16 bits, você pode optar por usar um comprimento de beacon de 24 bits para evitar qualquer perda de desempenho.

   Escolha o comprimento do beacon com cuidado. Depois de gravar novos registros com o beacon, não será possível atualizar o comprimento do beacon.

Exemplo

Considere um banco de dados que marcou o `unit` campo como `ENCRYPT_AND_SIGN` nas ações criptográficas. Para configurar um beacon padrão para o campo `unit`, precisamos determinar o número esperado de falsos positivos e o comprimento do beacon para o campo `unit`.

1. Estimar a população

   Depois de analisar nosso modelo de ameaças e a solução atual de banco de dados, esperamos que o campo `unit` eventualmente tenha 100.000 valores exclusivos.

Isso significa que População = 100.000.

2. Calcule a faixa recomendada para o número esperado de colisões.

   Neste exemplo, o número esperado de colisões deve estar entre 2 e 316.

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

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

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

3. Calcule o intervalo recomendado para o comprimento do beacon.

   Neste exemplo, o comprimento do beacon deve estar entre 9 e 16 bits.

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

   a. Calcule o comprimento do beacon em que o número esperado de colisões é igual ao mínimo identificado na Etapa 2.

      $$2 = 100{,}000 * 2^{-(\text{beacon length})}$$

      Comprimento do beacon = 15,6 ou 15 bits

   b. Calcule o comprimento do beacon em que o número esperado de colisões é igual ao máximo identificado na Etapa 2.

      $$316 = 100{,}000 * 2^{-(\text{beacon length})}$$

      Comprimento do beacon = 8,3 ou 8 bits

4. Determine o comprimento do beacon adequado aos seus requisitos de segurança e desempenho.

   Para cada bit abaixo de 15, o custo de desempenho e a segurança dobram.

   - 16 bits
     - Em média, cada valor exclusivo será mapeado para 1,5 outras unidades.

    - Segurança: dois registros com a mesma tag HMAC truncada têm 66% de probabilidade de ter o mesmo valor em texto simples.
    - Desempenho: uma consulta recuperará 15 registros para cada 10 registros que você realmente solicitou.
- 14 bits
    - Em média, cada valor exclusivo será mapeado para 6,1 outras unidades.
    - Segurança: dois registros com a mesma tag HMAC truncada têm 33% de probabilidade de ter o mesmo valor em texto simples.
    - Desempenho: uma consulta recuperará 30 registros para cada 10 registros que você realmente solicitou.

Escolher um nome de beacon

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Cada beacon é identificado por um nome de beacon exclusivo. Depois que um beacon é configurado, o nome do beacon é o nome que você usa ao consultar um campo criptografado. O nome de um beacon pode ter o mesmo nome de um campo criptografado ou campo virtual, mas não pode ser igual ao nome de um campo não criptografado. Dois beacons diferentes não podem ter o mesmo nome de beacon.

Para obter exemplos que demonstram como nomear e configurar beacons, consulte Configurar beacons.

Nomear o beacon padrão

Ao nomear beacons padrão, é altamente recomendável que o nome do seu beacon seja resolvido para a fonte do beacon sempre que possível. Isso significa que o nome do beacon e o nome do campo criptografado ou virtual a partir do qual seu beacon padrão é construído são os mesmos. Por exemplo, se você estiver criando um beacon padrão para um campo criptografado chamado `LastName`, o nome do seu beacon também deve ser `LastName`.

Quando o nome do beacon é o mesmo da fonte do beacon, você pode omitir a fonte do beacon da sua configuração e o SDK do AWS Database Encryption usará automaticamente o nome do beacon como fonte do beacon.

Configurar beacons

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Há dois tipos de beacons que oferecem suporte à criptografia pesquisável. Os beacons padrão realizam pesquisas de igualdade. Eles são a maneira mais simples de implementar criptografia pesquisável em seu banco de dados. Os beacons compostos combinam cadeias de texto simples literais e beacons padrão para realizar consultas mais complexas.

Os beacons são projetados para serem implementados em bancos de dados novos e vazios. Qualquer beacon configurado em um banco de dados existente mapeará somente novos registros gravados no banco de dados. Os beacons são calculados a partir do valor de texto simples de um campo, uma vez que o campo é criptografado, não há como o beacon mapear os dados existentes. Depois de gravar novos registros com o beacon, não será possível atualizar a configuração do beacon. No entanto, é possível adicionar novos beacons aos novos campos que você adiciona ao seu registro.

Depois de determinar seus padrões de acesso, a configuração dos beacons deve ser a segunda etapa na implementação do banco de dados. Depois de configurar todos os seus beacons, você precisa criar um chaveiro AWS KMS hierárquico, definir a versão do beacon, configurar um índice secundário para cada beacon, definir suas ações criptográficas e configurar seu banco de dados e o cliente Database Encryption SDK. AWS Para ter mais informações, consulte Usar beacons.

Para facilitar a definição da versão do beacon, recomendamos criar listas para beacons padrão e compostos. Adicione cada beacon que você criar à respectiva lista de beacons padrão ou compostos à medida que você os configura.

Tópicos

- Configurando beacons padrão
- Configuração de beacons compostos

- Exemplos de configuração

Configurando beacons padrão

Os beacons padrão são a maneira mais simples de implementar criptografia pesquisável em seu banco de dados. Eles só podem realizar pesquisas de igualdade para um único campo criptografado ou virtual.

Sintaxe de exemplo de configuração

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()?,
```

Para configurar um beacon padrão, forneça os valores a seguir.

Nome do beacon

O nome que você usa ao consultar um campo criptografado.

O nome de um beacon pode ter o mesmo nome de um campo criptografado ou campo virtual, mas não pode ser igual ao nome de um campo não criptografado. É altamente recomendável usar o nome do campo criptografado ou do campo virtual a partir do qual seu beacon padrão é construído sempre que possível. Dois beacons diferentes não podem ter o mesmo nome de beacon. Para obter ajuda para determinar o melhor nome de beacon para sua implementação, consulte Escolher um nome de beacon.

Comprimento do beacon

O número de bits do valor de hash do beacon que são mantidos após o truncamento.

O comprimento do beacon determina o número médio de falsos positivos produzidos por um determinado beacon. Para obter mais informações e ajudar a determinar o comprimento adequado do beacon para sua implementação, consulte Determinar o comprimento do beacon.

Fonte do beacon (opcional)

O campo a partir do qual um beacon padrão é construído.

A fonte do beacon deve ser um nome de campo ou um índice referente ao valor de um campo aninhado. Quando o nome do beacon é o mesmo da fonte do beacon, você pode omitir a fonte do beacon da sua configuração e o SDK do AWS Database Encryption usará automaticamente o nome do beacon como fonte do beacon.

Criação de um campo virtual

Para criar um campo virtual, você deve fornecer um nome para o campo virtual e uma lista dos campos de origem. A ordem em que você adiciona campos de origem à lista de partes virtuais determina a ordem em que eles são concatenados para criar o campo virtual. O exemplo a seguir concatena dois campos de origem em sua totalidade para criar um campo virtual.

> **Note**
>
> Recomendamos verificar se seus campos virtuais produzem o resultado esperado antes de preencher seu banco de dados. Para obter mais informações, consulte Teste de saídas de beacon.

Java

Veja o exemplo de código completo: 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

Veja o exemplo de código completo: VirtualBeaconSearchableEncryptionExample.cs

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

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

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

Rust

Veja o exemplo de código completo: 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];
```

Para criar um campo virtual com um segmento específico de um campo de origem, você deve definir essa transformação antes de adicionar o campo de origem à sua lista de partes virtuais.

Considerações de segurança para campos virtuais

Os beacons não alteram o estado criptografado do campo. No entanto, quando você usa beacons, há uma compensação inerente entre a eficiência de suas consultas e a quantidade de informações reveladas sobre a distribuição dos dados. A forma como você configura seu beacon determina o nível de segurança que é preservado por esse beacon.

Evite criar um campo virtual com campos de origem que se sobreponham aos beacons padrão existentes. A criação de campos virtuais que incluam um campo de origem que já tenha sido usado para criar um beacon padrão pode reduzir o nível de segurança de ambos os beacons. A extensão da redução da segurança depende do nível de entropia adicionado pelos campos de origem adicionais. O nível de entropia é determinado pela distribuição de valores exclusivos no campo de origem adicional e pelo número de bits que o campo de origem adicional contribui para o tamanho geral do campo virtual.

É possível usar a população e o comprimento do beacon para determinar se os campos de origem de um campo virtual preservam a segurança do seu conjunto de dados. A população é o número esperado de valores exclusivos em um campo. A sua população não precisa ser precisa. Para obter ajuda para estimar a população de um campo, consulte Estimar a população.

Considere o exemplo a seguir ao analisar a segurança de seus campos virtuais.

- Beacon1 é construído a partir de `FieldA`. `FieldA` tem uma população maior que $2^{(\text{comprimento do Beacon1})}$.
- Beacon2 é construído a partir de `VirtualField`, que é construído a partir de `FieldA`, `FieldB`, `FieldC` e `FieldD`. Juntos, `FieldB`, `FieldC` e `FieldD` têm uma população maior que 2^N

O Beacon2 preserva a segurança do Beacon1 e do Beacon2 se as seguintes afirmações forem verdadeiras:

```
N ≥ (Beacon1 length)/2
```

and

```
N ≥ (Beacon2 length)/2
```

Definindo estilos de farol

Os beacons padrão podem ser usados para realizar pesquisas de igualdade para um campo criptografado ou virtual. Ou, eles podem ser usados para construir beacons compostos para realizar operações de banco de dados mais complexas. Para ajudá-lo a organizar e gerenciar beacons padrão, o SDK de criptografia de AWS banco de dados fornece os seguintes estilos de beacon opcionais que definem o uso pretendido de um beacon padrão.

> **Note**
>
> Para definir estilos de beacon, você deve usar a versão 3.2 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão em todos os leitores antes de adicionar estilos de beacon às suas configurações de beacon.

PartOnly

Um farol padrão definido como só `PartOnly` pode ser usado para definir uma parte criptografada de um farol composto. Você não pode consultar diretamente um `PartOnly` farol padrão.

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

Por padrão, cada farol padrão gera uma chave HMAC exclusiva para o cálculo do farol. Como resultado, você não pode realizar uma pesquisa de igualdade nos campos criptografados a partir de dois beacons padrão separados. Um farol padrão definido como `Shared` usa a chave HMAC de outro farol padrão para seus cálculos.

Por exemplo, se você precisar comparar `beacon1` campos com `beacon2` campos, defina `beacon2` como um `Shared` farol que usa a chave HMAC de `beacon1` para seus cálculos.

> **Note**
>
> Considere suas necessidades de segurança e desempenho antes de configurar qualquer `Shared` beacon. `Shared`os beacons podem aumentar a quantidade de informações estatísticas que podem ser identificadas sobre a distribuição do seu conjunto de dados. Por exemplo, eles podem revelar quais campos compartilhados contêm o mesmo valor em texto simples.

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

Por padrão, se o valor de um campo for um conjunto, o SDK do AWS Database Encryption calcula um único beacon padrão para o conjunto. Como resultado, você não pode realizar a consulta `CONTAINS(a, :value)` onde `a` está um campo criptografado. Um farol padrão definido como `AsSet` calcula valores individuais de farol padrão para cada elemento individual do conjunto e armazena o valor do farol no item como um conjunto. Isso permite que o SDK AWS de criptografia de banco de dados realize a consulta`CONTAINS(a, :value)`.

Para definir um farol `AsSet` padrão, os elementos no conjunto devem ser da mesma população para que todos possam usar o mesmo comprimento de farol. O conjunto de faróis pode ter menos

elementos do que o conjunto de texto simples se houver colisões ao calcular os valores dos faróis.

> **Note**
>
> Considere suas necessidades de segurança e desempenho antes de configurar qualquer `AsSet` beacon. `AsSet`os beacons podem aumentar a quantidade de informações estatísticas que podem ser identificadas sobre a distribuição do seu conjunto de dados. Por exemplo, eles podem revelar o tamanho do conjunto de texto simples.

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

Um farol padrão definido como `SharedSet` combina as `AsSet` funções `Shared` e para que você possa realizar pesquisas de igualdade nos valores criptografados de um conjunto e campo. Isso permite que o SDK AWS de criptografia de banco de dados realize a consulta `CONTAINS(a, b)` onde *a* está um conjunto criptografado e *b* um campo criptografado.

> **Note**
>
> Considere suas necessidades de segurança e desempenho antes de configurar qualquer `Shared` beacon. `SharedSet`os beacons podem aumentar a quantidade de informações estatísticas que podem ser identificadas sobre a distribuição do seu conjunto de dados. Por exemplo, eles podem revelar o tamanho do conjunto de texto sem formatação ou quais campos compartilhados contêm o mesmo valor em texto sem formatação.

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()?
```

Configuração de beacons compostos

Os beacons compostos para combinar cadeias de texto simples literais e beacons padrão para realizar operações de banco de dados complexas, como consultar dois tipos de registro diferentes de um único índice ou consultar uma combinação de campos com uma chave de classificação. Faróis compostos podem ser construídos a partir de`ENCRYPT_AND_SIGN`,`SIGN_ONLY`, e `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campos. Você deve criar um beacon padrão para cada campo criptografado incluído no beacon composto.

> **Note**
>
> Recomendamos verificar se seus beacons compostos produzem o resultado esperado antes de preencher seu banco de dados. Para obter mais informações, consulte Teste de saídas de beacon.

Sintaxe de exemplo de configuração

Java

Configuração de farol composto

O exemplo a seguir define listas de peças criptografadas e assinadas localmente na configuração do beacon composto.

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

Definição da versão do Beacon

O exemplo a seguir define listas de peças criptografadas e assinadas globalmente na versão beacon. Para obter mais informações sobre como definir a versão do beacon, consulte Usando 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

Veja o exemplo de código completo: BeaconConfig.cs

Configuração de farol composto

O exemplo a seguir define listas de peças criptografadas e assinadas localmente na configuração do beacon composto.

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

Definição da versão do Beacon

O exemplo a seguir define listas de peças criptografadas e assinadas globalmente na versão beacon. Para obter mais informações sobre como definir a versão do beacon, consulte Usando 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

Veja o exemplo de código completo: beacon_config.rs

Configuração de farol composto

O exemplo a seguir define listas de peças criptografadas e assinadas localmente na configuração do beacon composto.

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

Definição da versão do Beacon

O exemplo a seguir define listas de peças criptografadas e assinadas globalmente na versão beacon. Para obter mais informações sobre como definir a versão do beacon, consulte Usando 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];
```

Você pode definir suas partes criptografadas e assinadas em listas definidas local ou globalmente. Recomendamos definir suas partes criptografadas e assinadas em uma lista global na versão do beacon sempre que possível. Ao definir peças criptografadas e assinadas globalmente, você pode definir cada peça uma vez e depois reutilizá-las em várias configurações de faróis compostos. Se você pretende usar uma peça criptografada ou assinada apenas uma vez, você pode defini-la em

uma lista local na configuração do beacon composto. Você pode referenciar partes locais e globais na sua lista de construtores.

Se você definir suas listas de peças criptografadas e assinadas globalmente, deverá fornecer uma lista de peças do construtor que identifique todas as formas possíveis pelas quais o farol composto pode montar os campos em sua configuração de farol composto.

> **Note**
>
> Para definir listas de peças criptografadas e assinadas globalmente, você deve usar a versão 3.2 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de definir qualquer nova parte globalmente.
> Você não pode atualizar as configurações de beacon existentes para definir listas de peças criptografadas e assinadas globalmente.

Para configurar um beacon composto, forneça os valores a seguir.

Nome do beacon

O nome que você usa ao consultar um campo criptografado.

O nome de um beacon pode ter o mesmo nome de um campo criptografado ou campo virtual, mas não pode ser igual ao nome de um campo não criptografado. Beacons diferentes não podem ter o mesmo nome. Para obter ajuda para determinar o melhor nome de beacon para sua implementação, consulte Escolher um nome de beacon.

Dividir caractere

O personagem usado para separar as partes que compõem seu beacon composto.

O caractere dividido não pode aparecer nos valores de texto simples de nenhum dos campos a partir dos quais o beacon composto foi construído.

Lista de peças criptografadas

Identifica os campos `ENCRYPT_AND_SIGN` incluídos no beacon composto.

Cada parte deve incluir um nome e um prefixo. O nome da parte deve ser o nome do beacon padrão construído a partir do campo criptografado. O prefixo pode ser qualquer string, mas deve ser exclusivo. Uma parte criptografada não pode ter o mesmo prefixo de uma parte assinada.

Recomendamos usar um valor curto que diferencie a parte de outras partes atendidas pelo beacon composto.

Recomendamos definir suas partes criptografadas globalmente sempre que possível. Você pode considerar definir uma parte criptografada localmente se pretende usá-la apenas em um farol composto. Uma peça criptografada definida localmente não pode ter o mesmo prefixo ou nome de uma peça criptografada definida globalmente.

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()?
];
```

Lista de peças assinadas

Identifica os campos assinados incluídos no farol composto.

> **Note**
>
> As peças assinadas são opcionais. Você pode configurar um farol composto que não faça referência a nenhuma peça assinada.

Cada parte deve incluir um nome, uma fonte e um prefixo. A fonte é o `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo `SIGN_ONLY` ou que a peça identifica. A fonte deve ser um nome de campo ou um índice referente ao valor de um campo aninhado. Se o nome da peça identificar a fonte, você poderá omitir a fonte e o SDK do AWS Database Encryption usará automaticamente o nome como fonte. Recomendamos especificar a fonte como nome da parte sempre que possível. O prefixo pode ser qualquer string, mas deve ser exclusivo. Uma parte criptografada assinada não pode ter o mesmo prefixo de uma parte criptografada. Recomendamos usar um valor curto que diferencie a parte de outras partes atendidas pelo beacon composto.

Recomendamos definir suas peças assinadas globalmente sempre que possível. Você pode considerar definir uma peça assinada localmente se pretende usá-la apenas em um farol composto. Uma peça assinada definida localmente não pode ter o mesmo prefixo ou nome de uma peça assinada definida globalmente.

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()?,
];
```

Lista de construtores

Identifica os construtores que definem as diferentes maneiras pelas quais as partes criptografadas e assinadas podem ser montadas pelo beacon composto. Você pode referenciar partes locais e globais na sua lista de construtores.

Se você construir seu farol composto a partir de partes criptografadas e assinadas globalmente definidas, deverá fornecer uma lista de construtores.

Se você não usar nenhuma peça criptografada ou assinada globalmente definida para construir seu farol composto, a lista de construtores é opcional. Se você não especificar uma lista de construtores, o SDK do AWS Database Encryption monta o beacon composto com o construtor padrão a seguir.

- Todas as partes assinadas na ordem em que foram adicionadas à lista de partes assinadas
- Todas as partes criptografadas na ordem em que foram adicionadas à lista de partes criptografadas
- Todas as partes são obrigatórias

Construtores

Cada construtor é uma lista ordenada de partes do construtor que define uma maneira pela qual o beacon composto pode ser montado. As partes do construtor são unidas na ordem em que são adicionadas à lista, com cada parte separada pelo caractere de divisão especificado.

Cada parte do construtor nomeia uma parte criptografada ou assinada e define se essa parte é obrigatória ou opcional dentro do construtor. Por exemplo, se você quiser consultar um beacon composto em `Field1`, `Field1.Field2` e `Field1.Field2.Field3`, marque `Field2` e `Field3` como opcional e crie um construtor.

Cada construtor deve ter pelo menos uma parte obrigatória. Recomendamos tornar obrigatória a primeira parte de cada construtor para que você possa usar o operador `BEGINS_WITH` nas consultas.

Um construtor será bem-sucedido se todas as partes obrigatórias estiverem presentes no registro. Quando você grava um novo registro, o beacon composto usa a lista de construtores para determinar se o beacon pode ser montado a partir dos valores fornecidos. Ele tenta montar o beacon na ordem em que os construtores foram adicionados à lista de construtores e usa o primeiro construtor bem-sucedido. Se nenhum construtor for bem-sucedido, o beacon não será gravado no registro.

Todos os leitores e gravadores devem especificar a mesma ordem de construtores para garantir que os resultados da consulta estejam corretos.

Use o procedimento a seguir para especificar sua própria lista de construtores.

1. Crie uma parte construtora para cada parte criptografada e assinada para definir se essa parte é necessária ou não.

   O nome da parte do construtor deve ser o nome do beacon padrão ou do campo assinado que ele representa.

   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. Crie um construtor para cada forma possível de montar o beacon composto usando as partes do construtor que você criou na Etapa 1.

   Por exemplo, se quiser consultar `Field1.Field2.Field3` e`Field4.Field2.Field3`, você deverá criar dois construtores. `Field1` e `Field4` podem ser obrigatórios porque foram definidos em dois construtores separados.

   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. Crie uma lista de construtores que inclua todos os construtores que você criou na Etapa 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. Especifique `constructorList` quando você cria seu farol composto.

Exemplos de configuração

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Os exemplos a seguir demonstram como configurar beacons padrão e compostos. As configurações a seguir não fornecem comprimentos de beacon. Para obter ajuda na determinação do comprimento adequado do beacon para sua configuração, consulte Escolher um comprimento do beacon.

Para ver exemplos completos de código que demonstram como configurar e usar beacons, consulte os exemplos de criptografia pesquisável em Java, .NET e Rust no repositório -dynamodb em. aws-database-encryption-sdk GitHub

Tópicos

- Beacons padrão
- Beacons compostos

Beacons padrão

Se você quiser consultar o campo `inspector_id_last4` para obter correspondências exatas, crie um beacon padrão usando a configuração a seguir.

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];
```

Beacons compostos

Se você quiser consultar o banco de dados `UnitInspection` em `inspector_id_last4` e `inspector_id_last4.unit`, crie um beacon composto com a configuração a seguir. Este beacon composto requer apenas partes criptografadas.

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()?];
```

Uso de beacons

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Os beacons permitem pesquisar registros criptografados sem descriptografar todo o banco de dados que está sendo consultado. Os beacons são projetados para serem implementados em bancos de

dados novos e vazios. Qualquer beacon configurado em um banco de dados existente mapeará somente novos registros gravados no banco de dados. Os beacons são calculados a partir do valor de texto simples de um campo, uma vez que o campo é criptografado, não há como o beacon mapear os dados existentes. Depois de gravar novos registros com o beacon, não será possível atualizar a configuração do beacon. No entanto, é possível adicionar novos beacons aos novos campos que você adiciona ao seu registro.

Depois de configurar seus beacons, você deve concluir as etapas a seguir antes de começar a preencher seu banco de dados e realizar consultas em seus beacons.

1. Crie um AWS KMS chaveiro hierárquico

   Para usar a criptografia pesquisável, você deve usar o token de autenticação hierárquico do AWS KMS para gerar, criptografar e descriptografar as chaves de dados usadas para proteger seus registros.

   Depois de configurar seus beacons, reúna os pré-requisitos do token de autenticação hierárquico e crie seu token de autenticação hierárquico.

   Para obter mais detalhes sobre por que o token de autenticação hierárquico é necessário, consulte Uso do token de autenticação hierárquico para criptografia pesquisável.

2.

   Definir a versão do beacon

   Especifique sua`keyStore`,`keySource`, uma lista de todos os beacons padrão que você configurou, uma lista de todos os beacons compostos que você configurou, uma lista de peças criptografadas, uma lista de peças assinadas e uma versão do beacon. Você deve especificar 1 para a versão do beacon. Para obter orientação sobre como definir `keySource`, consulte Definição da fonte de chave de beacon.

   O exemplo de Java a seguir define a versão do beacon para um único banco de dados de locatário. Para obter ajuda na definição da versão do beacon para um banco de dados multilocatário, consulte Criptografia pesquisável para bancos de dados multilocatários.

   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. Configurar índices secundários

   Depois de configurar os beacons, você deve configurar um índice secundário que reflete cada beacon antes de poder pesquisar nos campos criptografados. Para obter mais informações, consulte Configuração de índices secundários com beacons.

4. Definir ações criptográficas

   Todos os campos usados para construir um beacon padrão devem ser marcados com `ENCRYPT_AND_SIGN`. Todos os outros campos usados para construir beacons devem ser marcados `SIGN_ONLY` com ou. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

5. Configurar um cliente SDK AWS de criptografia de banco de dados

   Para configurar um cliente SDK AWS de criptografia de banco de dados que proteja os itens da tabela em sua tabela do DynamoDB, consulte Biblioteca de criptografia Java do lado do cliente para o DynamoDB.

Consultar beacons

O tipo de beacon que você configura determina o tipo de consultas que você é capaz de realizar. Os beacons padrão usam expressões de filtro para realizar pesquisas de igualdade. Os beacons compostos combinam cadeias de texto simples literais e beacons padrão para realizar consultas complexas. Ao consultar dados criptografados, você pesquisa pelo nome do beacon.

Não é possível comparar os valores de dois beacons padrão, mesmo que eles contenham o mesmo texto simples subjacente. Os dois beacons padrão produzirão duas etiquetas de HMAC diferentes para os mesmos valores de texto simples. Como resultado, os beacons padrão não podem realizar as consultas a seguir.

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

Os beacons compostos podem realizar as consultas a seguir.

- BEGINS_WITH(*a*), onde *a* reflete o valor total do campo com o qual o beacon composto montado começa. Não é possível usar o operador BEGINS_WITH para identificar um valor que comece com uma substring específica. No entanto, é possível usar BEGINS_WITH(*S_*), em que *S_* reflete o prefixo de uma parte com a qual o beacon composto montado começa.
- CONTAINS(*a*), em que *a* reflete o valor total de um campo que o beacon composto montado contém. Não é possível usar o operador CONTAINS para identificar um registro que contenha uma substring específica ou um valor dentro de um conjunto.

  Por exemplo, não é possível realizar uma consulta CONTAINS(*path*, "*a*" em que *a* reflita o valor em um conjunto.
- É possível comparar partes assinadas de beacons compostos. Ao comparar partes assinadas, é possível, opcionalmente, acrescentar o prefixo de uma parte criptografada a uma ou mais partes assinadas, mas não é possível incluir o valor de um campo criptografado em nenhuma consulta.

  Por exemplo, é possível comparar partes assinadas e consultar em *signedField1* = *signedField2* ou*value* IN (*signedField1*, *signedField2*, ...).

  Também é possível comparar partes assinadas e o prefixo de uma parte criptografada por meio de consulta em *signedField1.A_* = *signedField2.B_*.
- *field* BETWEEN *a* AND *b*, em que *a* e *b* são partes assinadas. Opcionalmente, é possível acrescentar o prefixo de uma parte criptografada a uma ou mais partes assinadas, mas não é possível incluir o valor de um campo criptografado em nenhuma consulta.

Você deve incluir o prefixo de cada parte incluída em uma consulta em um beacon composto. Por exemplo, se você construiu um beacon composto, `compoundBeacon`, a partir de dois campos `encryptedField` e `signedField`, deverá incluir os prefixos configurados para essas duas partes ao consultar o beacon.

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

Criptografia pesquisável para bancos de dados multilocatários

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Para implementar a criptografia pesquisável em seu banco de dados, você deve usar um token de autenticação hierárquico do AWS KMS. O AWS KMS chaveiro hierárquico gera, criptografa e descriptografa as chaves de dados usadas para proteger seus registros. Ele também cria a chave do beacon usada para gerar beacons. Ao usar o AWS KMS chaveiro hierárquico com bancos de dados multilocatários, há uma chave de ramificação e uma chave de beacon distintas para cada inquilino. Para consultar dados criptografados em um banco de dados multilocatário, você deve identificar os materiais de chave do beacon usados para gerar o beacon que você está consultando. Para obter mais informações, consulte the section called “Uso do token de autenticação hierárquico para criptografia pesquisável”.

Ao definir a versão do beacon para um banco de dados multilocatário, especifique uma lista de todos os beacons padrão que você configurou, uma lista de todos os beacons compostos que você configurou, uma versão do beacon e uma `keySource`. Você deve definir sua fonte de chave de beacon como `MultiKeyStore` e incluir um `keyFieldName` de tempo de vida útil para o cache de chave de beacon local e tamanho máximo de cache para o cache de chave de beacon local.

Se você configurou algum beacon assinado, ele deve ser incluído no seu `compoundBeaconList`. Os beacons assinados são um tipo de farol composto que indexa e executa consultas complexas em campos e. `SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

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

O `keyFieldName` define o nome do campo que armazena o `branch-key-id` associado à chave de beacon usada para gerar beacons para um determinado locatário.

Quando você grava novos registros em seu banco de dados, a chave `branch-key-id` que identifica a chave de beacon usada para gerar quaisquer beacons para esse registro é armazenada nesse campo.

Por padrão, `keyField` é um campo conceitual que não está explicitamente armazenado em seu banco de dados. O SDK AWS de criptografia de banco de dados identifica a chave `branch-key-id` de dados criptografada na descrição do material e armazena o valor no conceito `keyField` para você referenciar em seus beacons compostos e beacons assinados. Como a descrição do material é assinada, o `keyField` conceitual é considerado uma parte assinada.

Você também pode incluir o `keyField` em suas ações criptográficas como um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` campo `SIGN_ONLY` ou para armazenar explicitamente o campo em seu banco de dados. Se você fizer isso, deverá incluir manualmente o `branch-key-id` no `keyField` sempre que gravar um registro no seu banco de dados.

Consultar beacons em um banco de dados multilocatário

Para consultar um beacon, você deve incluir o `keyField` em sua consulta para identificar os materiais de chave de beacon apropriados necessários para recalcular o beacon. Você deve especificar o `branch-key-id` associado à chave de beacon usada para gerar os beacons para um registro. Você não pode especificar o nome amigável que identifica o nome de um locatário `branch-key-id` no fornecedor da ID da chave da ramificação. É possível incluir o `keyField` em suas consultas das maneiras a seguir.

Beacons compostos

Quer você armazene explicitamente o `keyField` em seus registros ou não, você poderá inclur `keyField` diretamente em seus beacons compostos como uma parte assinada. A parte `keyField` assinada deve ser necessária.

Por exemplo, se você quiser construir um beacon composto, `compoundBeacon`, a partir de dois campos, `encryptedField` `esignedField`, você também deverá incluir `keyField` como uma parte assinada. Isso permite executar a consulta a seguir em `compoundBeacon`.

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

Beacons assinados

O SDK AWS de criptografia de banco de dados usa beacons padrão e compostos para fornecer soluções de criptografia pesquisáveis. Esses beacons devem incluir pelo menos um campo criptografado. No entanto, o SDK AWS de criptografia de banco de dados também oferece suporte a beacons assinados que podem ser configurados inteiramente a partir de texto simples `SIGN_ONLY` e campos. `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`

Os beacons assinados podem ser construídos a partir de uma única parte. Independentemente de você armazenar explicitamente `keyField` em seus registros ou não, você pode construir um beacon assinado a partir do `keyField` e usá-lo para criar consultas compostas que combinam uma consulta no beacon `keyField` assinado com uma consulta em um de seus outros beacons. Por exemplo, você poderia realizar a consulta a seguir.

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

Para obter ajuda sobre a configuração de beacons assinados, consulte Criação de beacons assinados

Consulte diretamente no **`keyField`**

Se você especificou o `keyField` em suas ações criptográficas e armazenou explicitamente o campo em seu registro, você pode criar uma consulta composta que combina uma consulta no seu beacon com uma consulta no `keyField`. É possível optar por consultar diretamente no `keyField` se desejar consultar um beacon padrão. Por exemplo, você poderia realizar a consulta a seguir.

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

AWS SDK de criptografia de banco de dados para DynamoDB

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O AWS Database Encryption SDK for DynamoDB é uma biblioteca de software que permite incluir criptografia do lado do cliente em seu design do Amazon DynamoDB. O SDK AWS de criptografia de banco de dados para DynamoDB fornece criptografia em nível de atributo e permite que você especifique quais itens criptografar e quais itens incluir nas assinaturas para garantir a autenticidade de seus dados. Criptografar dados em trânsito e em repouso confidenciais ajuda você a garantir que os dados em texto simples não estejam disponíveis a terceiros, incluindo à AWS.

> **Note**
>
> O SDK AWS de criptografia de banco de dados não é compatível com partiQL.

No DynamoDB, uma tabela é uma coleção de itens. Cada item é uma coleção de atributos. Cada atributo tem um nome e um valor. O SDK AWS de criptografia de banco de dados para DynamoDB criptografa os valores dos atributos. Em seguida, ele calcula uma assinatura sobre os atributos. É possível especificar quais valores de atributo criptografar e quais incluir na assinatura das ações criptográficas.

Os tópicos deste capítulo fornecem uma visão geral do SDK de criptografia de AWS banco de dados para DynamoDB, incluindo quais campos são criptografados, orientações sobre instalação e configuração do cliente e exemplos de Java para ajudar você a começar.

Tópicos

- Criptografia do lado do cliente e do lado do servidor
- Quais campos são criptografados e assinados?
- Criptografia pesquisável no DynamoDB
- Atualizar seu modelo de dados

- AWS SDK de criptografia de banco de dados para linguagens de programação disponíveis do DynamoDB
- Cliente legado de criptografia do DynamoDB

Criptografia do lado do cliente e do lado do servidor

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados para DynamoDB oferece suporte à criptografia do lado do cliente, na qual você criptografa os dados da tabela antes de enviá-los para o banco de dados. Contudo, o DynamoDB fornece um atributo de criptografia em repouso que criptografa a tabela de forma transparente quando ela é mantida no disco e a descriptografa quando você a acessa.

As ferramentas escolhidas dependem da confidencialidade dos seus dados e dos requisitos de segurança do seu aplicativo. Você pode usar o AWS Database Encryption SDK para DynamoDB e a criptografia em repouso. Quando você envia itens criptografados e assinados ao DynamoDB, o DynamoDB não os reconhece como itens protegidos. Ele apenas detecta itens típicos da tabela com valores de atributo binários.

Criptografia do lado do servidor em repouso

O DynamoDB é compatível com a criptografia em repouso, um atributo de criptografia do lado do servidor no qual o DynamoDB criptografa suas tabelas de forma transparente quando elas são mantidas no disco e as descriptografa quando você acessa os dados da tabela.

Quando você usa um AWS SDK para interagir com o DynamoDB, por padrão, seus dados são criptografados em trânsito por uma conexão HTTPS, descriptografados no endpoint do DynamoDB e depois criptografados novamente antes de serem armazenados no DynamoDB.

- Criptografia por padrão. O DynamoDB criptografa e descriptografa de forma transparente todas as tabelas quando elas são gravadas. Não há nenhuma opção para habilitar ou desabilitar a criptografia em repouso.
- O DynamoDB cria e gerencia as chaves de criptografia.A chave exclusiva de cada tabela é protegida por uma AWS KMS key que nunca deixa o AWS Key Management Service (AWS KMS)

descriptografado. Por padrão, o DynamoDB usa uma Chave pertencente à AWS na conta do serviço do DynamoDB, mas é possível escolher uma Chave gerenciada pela AWS ou uma chave gerenciada pelo cliente na conta para proteger algumas ou todas as suas tabelas.

- Todos os dados da tabela são criptografados no disco. Quando uma tabela criptografada é salva no disco, o DynamoDB criptografa todos os dados da tabela, incluindo a chave primária e os índices secundários locais e globais. Se sua tabela tem uma chave de classificação, algumas dessas chaves que marcam os limites de intervalo são armazenadas em textos simples nos metadados da tabela.
- Os objetos relacionados a tabelas também são criptografados. A criptografia em repouso protege os streams do DynamoDB, as tabelas globais e os backups sempre que eles estão gravados em mídia durável.
- Os itens são descriptografados quando você os acessa. Quando você acessa a tabela, o DynamoDB descriptografa a parte da tabela que inclui o item de destino e retorna o item em texto sem formatação para você.

AWS SDK de criptografia de banco de dados para DynamoDB

A criptografia do lado do cliente fornece end-to-end proteção para seus dados, em trânsito e em repouso, desde a origem até o armazenamento no DynamoDB. Seus dados em texto simples nunca são expostos a terceiros, inclusive. AWS Você pode usar o AWS Database Encryption SDK for DynamoDB com novas tabelas do DynamoDB ou migrar suas tabelas existentes do Amazon DynamoDB para a versão mais recente do Database Encryption SDK for DynamoDB. AWS

- Seus dados são protegidos em trânsito e em repouso. Nunca é exposto a terceiros, inclusive AWS.
- É possível assinar os itens da tabela. É possível direcionar o SDK de criptografia de banco de dados da AWS para calcular uma assinatura em todo ou em parte de um item da tabela, incluindo os atributos de chave primária e o nome da tabela. Essa assinatura permite que você detecte alterações não autorizadas no item como um todo, incluindo a adição ou a exclusão de atributos ou a troca de valores de atributos.
- Você determina como seus dados são protegidos selecionando um token de autenticação. O token de autenticação determina as chaves de empacotamento que protegem as chaves de dados e, em última análise, os dados. Use as chaves de encapsulamento mais seguras e práticas para sua tarefa.
- O SDK AWS de criptografia de banco de dados para DynamoDB não criptografa a tabela inteira. Você escolhe quais atributos são criptografados em seus itens. O SDK AWS de criptografia de

banco de dados para DynamoDB não criptografa um item inteiro. Ele não criptografa nomes de atributo ou nomes e valores dos atributos da chave primária (chave de partição e de classificação).

AWS Encryption SDK

Se você estiver criptografando dados armazenados no DynamoDB, recomendamos o SDK de criptografia de banco de dados para AWS o DynamoDB.

O AWS Encryption SDK é uma biblioteca de criptografia do lado do cliente que ajuda você a criptografar e descriptografar dados genéricos. Embora possa proteger qualquer tipo de dado, ele não foi projetado para trabalhar com dados estruturados, como registros de banco de dados. Ao contrário do SDK AWS de criptografia de banco de dados para DynamoDB, AWS Encryption SDK ele não pode fornecer verificação de integridade em nível de item e não tem lógica para reconhecer atributos ou impedir a criptografia de chaves primárias.

Se você usar o AWS Encryption SDK para criptografar qualquer elemento da sua tabela, lembre-se de que ele não é compatível com o SDK de criptografia de AWS banco de dados para DynamoDB. Não é possível criptografar com uma biblioteca e descriptografar com uma diferente.

Quais campos são criptografados e assinados?

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

O SDK AWS de criptografia de banco de dados para DynamoDB é uma biblioteca de criptografia do lado do cliente projetada especialmente para aplicativos do Amazon DynamoDB. O Amazon DynamoDB armazena dados em tabelas, que são uma coleção de itens. Cada item é uma coleção de atributos. Cada atributo tem um nome e um valor. O SDK AWS de criptografia de banco de dados para DynamoDB criptografa os valores dos atributos. Em seguida, ele calcula uma assinatura sobre os atributos. Você pode especificar quais valores de atributo criptografar e quais incluir na assinatura.

A criptografia protege a confidencialidade do valor do atributo. A assinatura fornece a integridade de todos os atributos assinados e a relação entre ele, além de fornecer a autenticação. Ele permite que você detecte alterações não autorizadas no item como um todo, incluindo a adição ou a exclusão de atributos, ou a substituição de um valor criptografado por outro.

Em um item criptografado, alguns dados permanecem em texto simples, incluindo o nome da tabela, todos os nomes de atributos, valores de atributos que você não criptografa e os nomes e valores dos atributos de chave primária (chave de partição e chave de classificação). Não armazene dados confidenciais nesses campos.

Para obter mais informações sobre como o SDK AWS de criptografia de banco de dados para DynamoDB funciona, consulte. Como funciona o SDK AWS de criptografia de banco de dados

> **Note**
>
> Todas as menções de ações de atributos nos tópicos do AWS Database Encryption SDK for DynamoDB se referem a ações criptográficas.

Tópicos

- Criptografar valores de atributos
- Assinar o item

Criptografar valores de atributos

O SDK AWS de criptografia de banco de dados para DynamoDB criptografa os valores (mas não o nome ou o tipo do atributo) dos atributos que você especifica. Para determinar quais valores de atributos são criptografados, use as ações de atributos.

Por exemplo, esse item inclui atributos `example` e `test`.

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

Se você criptografar o atributo `example`, mas não o `test`, os resultados serão semelhantes aos seguintes. O valor do atributo `example` criptografado são dados binários, em vez de uma 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'
...
```

Os atributos de chave primária (chave de partição e chave de classificação) de cada item devem permanecer em texto sem formatação porque o DynamoDB os usa para encontrar o item na tabela. Eles devem ser assinados, mas não criptografados.

O SDK AWS de criptografia de banco de dados para DynamoDB identifica os atributos da chave primária para você e garante que seus valores sejam assinados, mas não criptografados. E, se você identificar a chave primária e tentar criptografá-la, o cliente gerará uma exceção.

O cliente armazena a descrição do material em um novo atributo (`aws_dbe_head`) que ele adiciona ao item. A descrição do material descreve como o item foi criptografado e assinado. O cliente usa essas informações para verificar e descriptografar o item. O campo que armazena a descrição do material não é criptografado.

Assinar o item

Depois de criptografar os valores dos atributos especificados, o SDK de criptografia de AWS banco de dados para DynamoDB calcula códigos de autenticação de mensagens baseados em hash (HMACs) e uma assinatura digital por meio da canonização da descrição do material, do contexto de criptografia e de cada campo marcado ou nas ações do atributo. `ENCRYPT_AND_SIGN`, `SIGN_ONLY`, `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` As assinaturas ECDSA são habilitadas por padrão, mas não são obrigatórias. O cliente armazena as assinaturas HMACs e em um novo atributo (`aws_dbe_foot`) que ele adiciona ao item.

Criptografia pesquisável no DynamoDB

Para configurar suas tabelas do Amazon DynamoDB para criptografia pesquisável, você deve usar o token de autenticação hierárquico do AWS KMS para gerar, criptografar e descriptografar as chaves de dados usadas para proteger seus itens. Você também deve incluir o `SearchConfig` na configuração de criptografia da tabela.

> **Note**
>
> Se você estiver usando a biblioteca de criptografia Java do lado do cliente para o DynamoDB, deverá usar o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB para criptografar, assinar, verificar e descriptografar os itens da tabela. O DynamoDB Enhanced Client e o `DynamoDBItemEncryptor` de nível inferior não oferecem suporte à criptografia pesquisável.

Tópicos

- Configuração de índices secundários com beacons
- Testando saídas de farol

Configuração de índices secundários com beacons

Depois de configurar os beacons, você deve configurar um índice secundário que reflete cada beacon antes de poder pesquisar nos atributos criptografados.

Quando você configura um beacon padrão ou composto, o SDK de criptografia AWS de banco de dados adiciona o `aws_dbe_b_` prefixo ao nome do beacon para que o servidor possa identificar facilmente os beacons. Por exemplo, se você nomear um beacon composto, `compoundBeacon`, o nome completo do beacon será `aws_dbe_b_compoundBeacon`. Se você quiser configurar índices secundários que incluam um beacon padrão ou composto, deverá incluir o prefixo `aws_dbe_b_` ao identificar o nome do beacon.

Partição e chaves de classificação

Não é possível criptografar valores de chave primária. Suas chaves de partição e classificação devem ser assinadas. Os valores de chave primária não podem ser um beacon padrão ou composto.

Seus valores de chave primária devem ser`SIGN_ONLY`, a menos que você especifique algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também devem ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Os valores de chave primária podem ser beacons assinados. Se você configurou beacons assinados distintos para cada um dos seus valores de chave primária, deverá o nome do atributo que identifica o valor da chave primária como o nome do beacon assinado. No entanto, o SDK AWS de criptografia de banco de dados não adiciona o `aws_dbe_b_` prefixo aos beacons assinados. Mesmo que você tenha configurado beacons assinados distintos para os valores de chave primária, você só precisará especificar os nomes dos atributos para os valores de chave primária ao configurar um índice secundário.

Índices secundários locais

A chave de classificação para um índice secundário local pode ser um beacon.

Se você especificar um beacon para a chave de classificação, o tipo deverá ser String. Se você especificar um beacon padrão ou composto para a chave de classificação, ele deverá incluir o

prefixo `aws_dbe_b_` ao especificar o nome do beacon. Se você especificar um beacon assinado, especifique o nome do beacon sem nenhum prefixo.

Índices secundários globais

As chaves de classificação e de partição de um índice secundário global podem ser beacons.

Se você especificar um beacon para a partição ou chave de classificação, o tipo deverá ser String. Se você especificar um beacon padrão ou composto para a chave de classificação, ele deverá incluir o prefixo `aws_dbe_b_` ao especificar o nome do beacon. Se você especificar um beacon assinado, especifique o nome do beacon sem nenhum prefixo.

Projeções de atributo

Uma projeção é o conjunto de atributos que é copiado de uma tabela para um índice secundário. A chave de partição e a chave de classificação da tabela são sempre projetadas no índice; é possível projetar outros atributos para suportar os requisitos de consulta da sua aplicação. O DynamoDB fornece três opções diferentes para projeções de atributo: `KEYS_ONLY`, `INCLUDE` e `ALL`.

Se você usar a projeção do atributo INCLUDE para pesquisar em um beacon, deverá especificar os nomes de todos os atributos a partir dos quais o beacon é construído e o nome do beacon com o prefixo `aws_dbe_b_`. Por exemplo, se você configurou um beacon composto, `compoundBeacon`, `defield1`, `field2` `efield3`, deverá especificar `aws_dbe_b_compoundBeacon`, `field1`, `field2` e `field3` na projeção.

Um índice secundário global só pode usar os atributos explicitamente especificados na projeção, mas um índice secundário local pode usar qualquer atributo.

Testando saídas de farol

Se você configurou beacons compostos ou construiu seus beacons usando campos virtuais, recomendamos verificar se esses beacons produzem a saída esperada antes de preencher sua tabela do DynamoDB.

O SDK AWS de criptografia de banco de dados fornece o `DynamoDbEncryptionTransforms` serviço para ajudá-lo a solucionar problemas de campo virtual e saídas de beacon composto.

Testando campos virtuais

O snippet a seguir cria itens de teste, define o `DynamoDbEncryptionTransforms` serviço com a configuração de criptografia de tabela do DynamoDB e demonstra como usá-lo para verificar se o campo virtual produz `ResolveAttributes` a saída esperada.

Java

Veja a amostra de código completa: 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

Veja o exemplo de código completo: 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

Veja o exemplo de código completo: 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");
```

Testando faróis compostos

O trecho a seguir cria um item de teste, define o `DynamoDbEncryptionTransforms` serviço com a configuração de criptografia de tabela do DynamoDB e demonstra como usá-lo para verificar se o beacon composto produz `ResolveAttributes` a saída esperada.

Java

Veja a amostra de código completa: 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

Veja o exemplo de código completo: 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

Veja o exemplo de código completo: 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
```

Atualizar seu modelo de dados

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Ao configurar o SDK AWS de criptografia de banco de dados para o DynamoDB, você fornece ações de atributos. Na criptografia, o SDK AWS de criptografia de banco de dados usa as ações de atributos para identificar quais atributos criptografar e assinar, quais atributos assinar (mas não criptografar) e quais ignorar. Os atributos não assinados permitidos informam ao cliente quais atributos foram excluídos das assinaturas. Na descriptografia, o SDK de criptografia AWS de banco de dados usa os atributos não assinados permitidos que você definiu para identificar quais atributos não estão incluídos nas assinaturas. As ações de atributo não são salvas no item criptografado e o SDK AWS de criptografia de banco de dados não atualiza suas ações de atributo automaticamente.

Escolha suas ações de atributos com cuidado. Em caso de dúvida, use Criptografar e assinar. Depois de usar o SDK AWS de criptografia de banco de dados para proteger seus itens, você não pode alterar um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo ou existente `ENCRYPT_AND_SIGN` para`DO_NOTHING`. `SIGN_ONLY` No entanto, é possível fazer as alterações a seguir.

- Adicionar novos `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributos `ENCRYPT_AND_SIGN``SIGN_ONLY`, e
- Remover atributos existentes
- Alterar um `ENCRYPT_AND_SIGN` atributo existente para `SIGN_ONLY` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- Alterar um existente `SIGN_ONLY` ou um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo para `ENCRYPT_AND_SIGN`
- Adicionar um novo atributo `DO_NOTHING`
- Alterar um atributo `SIGN_ONLY` existente para `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`
- Alterar um atributo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` existente para `SIGN_ONLY`

Considerações sobre a criptografia pesquisável

Antes de atualizar o modelo de dados, considere cuidadosamente como as atualizações podem afetar os beacons que você construiu a partir dos atributos. Depois de gravar novos registros com o beacon, não será possível atualizar a configuração do beacon. Não é possível atualizar as ações de atributos associadas aos atributos que você usou para construir beacons. Se você remover um atributo existente e o beacon associado, não poderá consultar registros existentes usando esse beacon. É possível criar novos beacons para novos campos adicionados ao registro, mas não é possível atualizar os beacons existentes para incluir o novo campo.

Considerações sobre atributos **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`**

Por padrão, as chaves de partição e classificação são o único atributo incluído no contexto de criptografia. Você pode considerar definir campos adicionais para que o fornecedor da ID da chave de filial do seu AWS KMS chaveiro hierárquico possa identificar qual chave de ramificação é necessária para a descriptografia a partir `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` do contexto de criptografia. Para obter mais informações, consulte fornecedor de ID de chave de filial. Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> Para usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica, você deve usar a versão 3.3 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de atualizar seu modelo de dados para incluí-la`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Adicionar novos **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`** atributos **`ENCRYPT_AND_SIGN`**`SIGN_ONLY`, e

Para adicionar um novo`ENCRYPT_AND_SIGN`,`SIGN_ONLY`, ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, defina o novo atributo em suas ações de atributo.

Você não pode remover um `DO_NOTHING` atributo existente e adicioná-lo novamente como um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo `ENCRYPT_AND_SIGN`SIGN_ONLY, ou.

Usar uma classe de dados anotada

Se você definiu as ações de atributo com um `TableSchema`, adicione o novo atributo à sua classe de dados anotada com a anotação . Se você não especificar uma anotação de ação de atributo para o novo atributo, o cliente criptografará e assinará o novo atributo por padrão (a menos que o atributo faça parte da chave primária). Se quiser assinar apenas o novo atributo, você deve adicionar o novo atributo com a `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` anotação `@DynamoDBEncryptionSignOnly` ou.

Usar um objeto de modelo

Se você definiu manualmente suas ações de atributo, adicione o novo atributo às ações de atributo em seu modelo de objeto e especifique`ENCRYPT_AND_SIGN`,`SIGN_ONLY`, ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` como a ação do atributo.

Remover atributos existentes

Se você decidir que não precisa mais de um atributo, pode parar de gravar dados nesse atributo ou removê-lo formalmente das ações de atributo. Quando você para de gravar novos dados em um atributo, o atributo ainda aparece nas ações de atributo. Isso pode ser útil se você precisar começar a usar o atributo novamente no futuro. A remoção formal do atributo das ações do atributo não o remove do conjunto de dados. O conjunto de dados ainda conterá itens que incluem esse atributo.

Para remover formalmente um `DO_NOTHING` atributo existente`ENCRYPT_AND_SIGN`,`SIGN_ONLY`, ou`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, atualize suas ações de atributo.

Se você remover um atributo `DO_NOTHING`, não deverá remover esse atributo dos atributos não assinados permitidos. Mesmo que você não esteja mais gravando novos valores nesse atributo, o cliente ainda precisa saber que o atributo não está assinado para ler os itens existentes que contêm o atributo.

Usar uma classe de dados anotada

Se você definiu as ações de atributo com um `TableSchema`, remova o novo atributo da classe de dados anotada.

Usar um objeto de modelo

Se você definiu manualmente as ações de atributo, remova o atributo das ações de atributo em seu modelo de objeto.

Alterar um **`ENCRYPT_AND_SIGN`** atributo existente para **`SIGN_ONLY`** ou **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`**

Para alterar um `ENCRYPT_AND_SIGN` atributo existente para `SIGN_ONLY` ou`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, você deve atualizar suas ações de atributo. Depois de implantar a atualização, o cliente poderá verificar e descriptografar valores existentes gravados no atributo, mas só assinará novos valores gravados no atributo.

> **Note**
>
> Considere cuidadosamente seus requisitos de segurança antes de alterar um `ENCRYPT_AND_SIGN` atributo existente para `SIGN_ONLY` ou`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Qualquer atributo que possa armazenar dados confidenciais deve ser criptografado.

Usar uma classe de dados anotada

Se você definiu suas ações de atributo com um`TableSchema`, atualize o atributo existente para incluir a `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` anotação `@DynamoDBEncryptionSignOnly` ou em sua classe de dados anotada.

Usar um objeto de modelo

Se você definiu manualmente suas ações de atributo, atualize a ação de atributo associada ao atributo existente de `ENCRYPT_AND_SIGN` para `SIGN_ONLY` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` em seu modelo de objeto.

Alterar um existente **`SIGN_ONLY`** ou um **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`** atributo para **`ENCRYPT_AND_SIGN`**

Para alterar um atributo existente `SIGN_ONLY` ou um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo para`ENCRYPT_AND_SIGN`, você deve atualizar suas ações de atributo. Depois de implantar a atualização, o cliente poderá verificar os valores existentes gravados no atributo, mas só criptografará e assinará novos valores gravados no atributo.

Usar uma classe de dados anotada

Se você definiu suas ações de atributo com um`TableSchema`, remova a `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` anotação `@DynamoDBEncryptionSignOnly` ou do atributo existente.

Usar um objeto de modelo

Se você definiu manualmente suas ações de atributo, atualize a ação de atributo associada ao atributo de `SIGN_ONLY` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` para `ENCRYPT_AND_SIGN` em seu modelo de objeto.

Adicionar um novo atributo **`DO_NOTHING`**

Para reduzir o risco de erro ao adicionar um novo atributo `DO_NOTHING`, recomendamos especificar um prefixo distinto ao nomear os atributos `DO_NOTHING` e, em seguida, usar esse prefixo para definir os atributos não assinados permitidos.

Você não pode remover um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo ou existente `ENCRYPT_AND_SIGN` da sua classe de dados anotada e depois adicionar o atributo novamente como um `DO_NOTHING` atributo. `SIGN_ONLY` Só é possível adicionar atributos `DO_NOTHING` totalmente novos.

As etapas que você executa para adicionar um novo atributo `DO_NOTHING` dependem de você ter definido os atributos não assinados permitidos explicitamente em uma lista ou com um prefixo.

Uso de um prefixo de atributos não assinados permitido

Se você definiu suas ações de atributo com um `TableSchema`, adicione o novo atributo `DO_NOTHING` à sua classe de dados anotada com a anotação `@DynamoDBEncryptionDoNothing`. Se você definiu manualmente as ações de atributo, atualize as ações de atributo para incluir o novo atributo. Certifique-se de configurar explicitamente o novo atributo com a ação do atributo `DO_NOTHING`. Você deve incluir o mesmo prefixo distinto no nome do novo atributo.

Uso de uma lista de atributos não assinados permitido

1. Adicione o novo atributo `DO_NOTHING` à sua lista de atributos não assinados permitidos e implante a lista atualizada.
2. Implante a alteração da Etapa 1.

   Não é possível passar para a Etapa 3 até que a alteração tenha se propagado para todos os hosts que precisam ler esses dados.
3. Adicione o novo atributo `DO_NOTHING` às ações de atributo.

   a. Se você definiu suas ações de atributo com um `TableSchema`, adicione o novo atributo `DO_NOTHING` à sua classe de dados anotada com a anotação `@DynamoDBEncryptionDoNothing`.

b. Se você definiu manualmente as ações de atributo, atualize as ações de atributo para incluir o novo atributo. Certifique-se de configurar explicitamente o novo atributo com a ação do atributo `DO_NOTHING`.

4. Implante a alteração da Etapa 3.

Alterar um atributo **`SIGN_ONLY`** existente para **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`**

Para alterar um atributo `SIGN_ONLY` existente para `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, você deve atualizar suas ações de atributo. Depois de implantar a atualização, o cliente poderá verificar os valores existentes gravados no atributo e continuará assinando novos valores gravados no atributo. Novos valores gravados no atributo serão incluídos no contexto de criptografia.

Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Usar uma classe de dados anotada

Se você definiu suas ações de atributo com um`TableSchema`, atualize a ação de atributo associada ao atributo de `@DynamoDBEncryptionSignOnly` para`@DynamoDBEncryptionSignAndIncludeInEncryptionContext`.

Usar um objeto de modelo

Se você definiu manualmente as ações de atributo, atualize a ação do atributo correspondente ao atributo de `SIGN_ONLY` ou `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` no modelo de objeto.

Alterar um atributo **`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`** existente para **`SIGN_ONLY`**

Para alterar um atributo `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` existente para `SIGN_ONLY`, você deve atualizar suas ações de atributo. Depois de implantar a atualização, o cliente poderá verificar os valores existentes gravados no atributo e continuará assinando novos valores gravados no atributo. Novos valores gravados no atributo não serão incluídos no contexto de criptografia.

Antes de alterar um `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo existente para`SIGN_ONLY`, considere cuidadosamente como suas atualizações podem afetar a funcionalidade do seu fornecedor de ID de chave de filial.

Usar uma classe de dados anotada

Se você definiu suas ações de atributo com um`TableSchema`, atualize a ação de atributo associada ao atributo de `@DynamoDBEncryptionSignAndIncludeInEncryptionContext` para`@DynamoDBEncryptionSignOnly`.

Usar um objeto de modelo

Se você definiu manualmente as ações de atributo, atualize a ação do atributo correspondente ao atributo de `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` ou `SIGN_ONLY` no modelo de objeto.

AWS SDK de criptografia de banco de dados para linguagens de programação disponíveis do DynamoDB

O SDK AWS de criptografia de banco de dados para DynamoDB está disponível para as seguintes linguagens de programação. As bibliotecas específicas de linguagem variam, mas as implementações resultantes são interoperáveis. É possível criptografar com uma implementação de linguagem e descriptografar com outra. A interoperabilidade pode estar sujeita às restrições de linguagem. Em caso afirmativo, essas restrições estarão descritas no tópico sobre a implementação de linguagem.

Tópicos

- Java
- .NET
- Rust

Java

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Este tópico explica como instalar a versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB. Para obter detalhes sobre a programação com o SDK AWS de criptografia de banco de dados para DynamoDB, consulte os exemplos de Java no repositório -dynamodb em aws-database-encryption-sdk. GitHub

> **Note**
>
> Os tópicos a seguir se concentram na versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB.
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O SDK do AWS Database Encryption continua oferecendo suporte às versões antigas do DynamoDB Encryption Client.

Tópicos

- Pré-requisitos
- Instalação
- Uso do biblioteca Java de criptografia do lado do cliente para o DynamoDB
- Exemplos de Java
- Configurar uma tabela existente do DynamoDB para usar o SDK de criptografia de banco de dados para AWS o DynamoDB
- Migrar para a versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB

Pré-requisitos

Antes de instalar a versão 3.x da bibioteca Java de criptografia do lado do cliente, verifique se você tem os pré-requisitos a seguir.

Um ambiente de desenvolvimento Java

Você precisará do Java 8 ou posterior. No site da Oracle, acesse Java SE Downloads e faça download e instale o Java SE Development Kit (JDK).

Se você usa o Oracle JDK, também precisara fazer download e instalar os arquivos de política de jurisdição de força ilimitada JCE (Java Cryptography Extension).

AWS SDK for Java 2.x

O SDK AWS de criptografia de banco de dados para DynamoDB requer o módulo DynamoDB Enhanced Client do. AWS SDK for Java 2.xÉ possível instalar todo o SDK ou apenas esse módulo.

Para obter informações sobre como atualizar sua versão do AWS SDK para Java, consulte Migração da versão 1.x para a 2.x do. AWS SDK para Java

O AWS SDK para Java está disponível por meio do Apache Maven. Você pode declarar uma dependência para todo AWS SDK para Java o módulo ou apenas para o `dynamodb-enhanced` módulo.

Instale o AWS SDK para Java usando o Apache Maven

- Para importar todo o AWS SDK para Java como uma dependência, declare-o no arquivo `pom.xml`.
- Para criar uma dependência somente para o módulo Amazon DynamoDB no AWS SDK para Java, siga as instruções para especificar módulos específicos. Defina o `groupId` como `software.amazon.awssdk` e `artifactID` como `dynamodb-enhanced`.

> **Note**
>
> Se você usar o AWS KMS chaveiro ou o AWS KMS chaveiro hierárquico, também precisará criar uma dependência para o módulo. AWS KMS Defina o `groupId` como `software.amazon.awssdk` e `artifactID` como `kms`.

Instalação

É possível instalar a versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB das formas a seguir.

Uso do Apache Maven

O Amazon DynamoDB Encryption Client para Java está disponível por meio do Apache Maven com a definição de dependência a seguir.

```
<dependency>
  <groupId>software.amazon.cryptography</groupId>
```

```
  <artifactId>aws-database-encryption-sdk-dynamodb</artifactId>
  <version>version-number</version>
</dependency>
```

Uso do Gradle Kotlin

É possível usar o Gradle para declarar uma dependência no Amazon DynamoDB Encryption Client para Java adicionando o que se segue à seção de dependências do projeto Gradle.

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

Manualmente

Para instalar a biblioteca de criptografia Java do lado do cliente para o DynamoDB, clone ou baixe o repositório -dynamodb. aws-database-encryption-sdk GitHub

Depois de instalar o SDK, comece examinando o código de exemplo neste guia e os exemplos de Java no repositório aws-database-encryption-sdk -dynamodb em. GitHub

Uso do biblioteca Java de criptografia do lado do cliente para o DynamoDB

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Este tópico explica algumas das funções e das classes auxiliares da versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB.

Para obter detalhes sobre a programação com a biblioteca de criptografia Java do lado do cliente para o DynamoDB, consulte os exemplos de Java, os exemplos de Java no repositório -dynamodb em. aws-database-encryption-sdk GitHub

Tópicos

- Criptografadores de itens
- Ações de atributos no SDK AWS de criptografia de banco de dados para DynamoDB
- Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB

- Atualização de itens com o SDK AWS de criptografia de banco de dados
- Descriptografar conjuntos assinados

Criptografadores de itens

Basicamente, o SDK AWS de criptografia de banco de dados para DynamoDB é um criptografador de itens. É possível usar a versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB para criptografar, assinar, verificar e descriptografar os itens da tabela do DynamoDB das maneiras a seguir.

O DynamoDB Enhanced Client

É possível configurar o DynamoDB Enhanced Client com `DynamoDbEncryptionInterceptor` o para criptografar e assinar automaticamente itens do lado do cliente com suas solicitações `PutItem` do DynamoDB. Com o DynamoDB Enhanced Client, é possível definir as ações de atributos usando uma classe de dados anotada. Recomendamos usar o DynamoDB Enhanced Client sempre que possível.

O DynamoDB Enhanced Client não oferece suporte à criptografia pesquisável.

> **Note**
>
> O SDK AWS de criptografia de banco de dados não oferece suporte a anotações em atributos aninhados.

A API de nível inferior do DynamoDB

É possível configurar a API de nível inferior do DynamoDB com `DynamoDbEncryptionInterceptor` para criptografar e assinar automaticamente itens no lado do cliente com suas solicitações `PutItem` do DynamoDB.

Você deve usar a API de nível inferior do DynamoDB para usar a criptografia pesquisável.

O **`DynamoDbItemEncryptor`** de nível inferior

O `DynamoDbItemEncryptor` de nível inferior criptografa, assina ou descriptografa e verifica diretamente os itens da tabela sem chamar o DynamoDB. Ele não faz solicitações `PutItem` ou `GetItem` para o DynamoDB. Por exemplo, é possível usar o `DynamoDbItemEncryptor`

de nível inferior para descriptografar e verificar diretamente um item do DynamoDB que você já recuperou.

O nível inferior do `DynamoDbItemEncryptor` não oferece suporte à criptografia pesquisável.

Ações de atributos no SDK AWS de criptografia de banco de dados para DynamoDB

As ações de atributo determinam quais valores de atributos são criptografados e assinados, quais são somente assinados, quais são assinados e incluídos no contexto de criptografia e quais são ignorados.

> **Note**
>
> Para usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica, você deve usar a versão 3.3 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de atualizar seu modelo de dados para incluí-la`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Se você usar a API do DynamoDB de nível inferior ou a `DynamoDbItemEncryptor` de nível inferior, deverá definir manualmente suas ações de atributos. Se você usar o DynamoDB Enhanced Client, poderá definir manualmente suas ações de atributo ou usar uma classe de dados anotada para gerar um `TableSchema`. Para simplificar o processo de configuração, recomendamos o uso de uma classe de dados anotada. Ao usar uma classe de dados anotada, você só precisa modelar seu objeto uma vez.

> **Note**
>
> Depois de definir suas ações de atributo, você deverá definir quais atributos serão excluídos das assinaturas. Para facilitar a adição de novos atributos não assinados no futuro, recomendamos escolher um prefixo distinto (como `":"`) para identificar os atributos não assinados. Inclua esse prefixo no nome do atributo para todos os atributos marcados como `DO_NOTHING` ao definir o esquema e as ações de atributos do DynamoDB.

Uso de uma classe de dados anotada

Use uma classe de dados anotada para especificar suas ações de atributos com o DynamoDB Enhanced Client e `DynamoDbEncryptionInterceptor`. O SDK de criptografia de banco de

dados da AWS usa as anotações de atributo padrão do DynamoDB que definem o tipo do atributo para determinar como proteger um atributo. Por padrão, todos os atributos são criptografados e assinados, exceto as chaves primárias, que são assinadas, mas não são criptografadas.

> **Note**
>
> Para usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica, você deve usar a versão 3.3 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de atualizar seu modelo de dados para incluí-la`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

Consulte SimpleClass.java no repositório aws-database-encryption-sdk -dynamodb em GitHub para obter mais orientações sobre as anotações do DynamoDB Enhanced Client.

Por padrão, os atributos da chave primária são assinados, mas não criptografados (`SIGN_ONLY`), e todos os outros atributos são criptografados e assinados (`ENCRYPT_AND_SIGN`). Se você definir qualquer atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Para especificar exceções, use as anotações de criptografia definidas na biblioteca de criptografia do lado do cliente para Java do DynamoDB. Por exemplo, se você quiser que um atributo específico seja somente assinado, use a anotação `@DynamoDbEncryptionSignOnly`. Se você quiser que um atributo específico seja assinado e incluído no contexto de criptografia, use `@DynamoDbEncryptionSignAndIncludeInEncryptionContext` o. Se desejar que um atributo específico não seja assinado nem criptografado (`DO_NOTHING`), use a anotação `@DynamoDbEncryptionDoNothing`.

> **Note**
>
> O SDK AWS de criptografia de banco de dados não oferece suporte a anotações em atributos aninhados.

O exemplo a seguir mostra as anotações usadas para definir `ENCRYPT_AND_SIGN` e `DO_NOTHING` atribuir ações. `SIGN_ONLY` Para ver um exemplo que mostra as anotações usadas para definir`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, consulte SimpleClass 4.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 a classe de dados anotada para criar o `TableSchema`, conforme mostrado no snippet a seguir.

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

Definir as ações de atributos manualmente

Para especificar manualmente ações de atributos, crie um objeto `Map` em que pares de nome-valor representam os nomes de atributos e as ações especificadas.

Especifique `ENCRYPT_AND_SIGN` para criptografar e assinar um atributo.
Especifique `SIGN_ONLY` para assinar, mas não criptografar um atributo. Especifique `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` para assinar um atributo e incluí-lo no contexto de criptografia. Não é possível criptografar um atributo sem também assiná-lo. Especifique `DO_NOTHING` para ignorar um atributo.

Os atributos de partição e classificação devem ser `SIGN_ONLY` ou`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você definir qualquer atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> Para usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica, você deve usar a versão 3.3 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de atualizar seu modelo de dados para incluí-la`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);
```

Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB

Ao usar o AWS Database Encryption SDK, você deve definir explicitamente uma configuração de criptografia para sua tabela do DynamoDB. Os valores necessários em sua configuração de criptografia dependem se você definiu suas ações de atributo manualmente ou com uma classe de dados anotada.

O snippet a seguir define uma configuração de criptografia de tabela do DynamoDB usando o DynamoDB Enhanced Client, `TableSchema`, e permite atributos não assinados definidos por um prefixo distinto.

```
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());
```

Nome da tabela lógica

Um nome de tabela lógica para sua tabela do DynamoDB.

O nome da tabela lógica é vinculado criptograficamente a todos os dados armazenados na tabela para simplificar as operações de restauração do DynamoDB. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Você deve sempre especificar o mesmo nome de tabela lógica. Para que a descriptografia seja bem-sucedida, o nome da tabela lógica deve corresponder ao nome especificado na criptografia. Se o nome da tabela do DynamoDB mudar após a restauração da tabela do DynamoDB a partir de um backup, o nome da tabela lógica garantirá que a operação de descriptografia ainda reconheça a tabela.

Atributos não assinados permitidos

Os atributos marcados `DO_NOTHING` em suas ações de atributos.

Os atributos não assinados permitidos informam ao cliente quais atributos são excluídos das assinaturas. O cliente presume que todos os outros atributos estão incluídos na assinatura. Em seguida, ao descriptografar um registro, o cliente determina quais atributos ele precisa verificar e quais ignorar dos atributos não assinados permitidos que você especificou. Não é possível remover um atributo dos atributos não assinados permitidos.

É possível definir explicitamente os atributos não assinados permitidos criando uma matriz que lista todos os atributos `DO_NOTHING`. Também é possível especificar um prefixo distinto ao nomear os atributos `DO_NOTHING` e usar o prefixo para informar ao cliente quais atributos não estão assinados. É altamente recomendável especificar um prefixo distinto, pois isso simplifica o processo de adicionar um novo atributo `DO_NOTHING` no futuro. Para obter mais informações, consulte Atualizar seu modelo de dados.

Se você não especificar um prefixo para todos os atributos `DO_NOTHING`, poderá configurar uma matriz `allowedUnsignedAttributes` que liste explicitamente todos os atributos que o cliente deve esperar que não estejam assinados ao encontrá-los na descriptografia. Você só deve definir explicitamente seus atributos não assinados permitidos se for absolutamente necessário.

Configuração de pesquisa (opcional)

O `SearchConfig` define a versão do beacon.

O `SearchConfig` deve ser especificado para usar criptografia pesquisável ou beacons assinados.

Conjunto de algoritmos (opcional)

O `algorithmSuiteId` define qual conjunto de algoritmos o SDK de criptografia de banco de dados da AWS usará.

A menos que você especifique explicitamente um conjunto alternativo de algoritmos, o SDK do AWS Database Encryption usa o conjunto de algoritmos padrão. O conjunto de algoritmos padrão usa o algoritmo AES-GCM com derivação de chaves, assinaturas digitais e comprometimento de chaves. Embora o conjunto de algoritmos padrão provavelmente seja adequado para a maioria dos aplicativos, é possível escolher um conjunto alternativo de algoritmos. Por exemplo, alguns modelos de confiança seriam satisfeitos com um pacote de algoritmos sem assinaturas digitais. Para obter informações sobre os conjuntos de algoritmos compatíveis com o SDK do AWS Database Encryption, consulteSuítes de algoritmos compatíveis no SDK AWS de criptografia de banco de dados.

Para selecionar o conjunto de algoritmos AES-GCM sem assinaturas digitais ECDSA, inclua o seguinte trecho em sua configuração de criptografia de tabela.

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

Atualização de itens com o SDK AWS de criptografia de banco de dados

O SDK AWS de criptografia de banco de dados não oferece suporte a ddb: UpdateItem para itens que foram criptografados ou assinados. Para atualizar um item criptografado ou assinado, você deve usar ddb: PutItem. Se algum item existir em uma tabela específica com a mesma chave primária de um item existente na consulta `PutItem`, o novo item substituirá completamente o item já existente. Também é possível usar o CLOBBER para limpar e substituir todos os atributos ao salvar após atualizar seus itens.

Descriptografar conjuntos assinados

Nas versões 3.0.0 e 3.1.0 do SDK de criptografia de AWS banco de dados, se você definir um atributo de tipo de conjunto como`SIGN_ONLY`, os valores do conjunto serão canonizados na ordem em que são fornecidos. O DynamoDB não preserva a ordem dos conjuntos. Como resultado, é possível que a validação da assinatura do item que contém o conjunto falhe. A validação da assinatura falha quando os valores do conjunto são retornados em uma ordem diferente da fornecida ao SDK do AWS Database Encryption, mesmo que os atributos do conjunto contenham os mesmos valores.

> **Note**
>
> As versões 3.1.1 e posteriores do SDK do AWS Database Encryption canonizam os valores de todos os atributos do tipo definido, de forma que os valores sejam lidos na mesma ordem em que foram gravados no DynamoDB.

Se houver falha na validação da assinatura, a operação de descriptografia falhará e retornará a seguinte mensagem de erro:

```
software.amazon.cryptography.dbencryptionsdk.structuredencryption.model. StructuredEncrypti
onException: Nenhuma etiqueta de destinatário correspondeu.
```

Se você receber a mensagem de erro acima e acreditar que o item que está tentando descriptografar inclui um conjunto que foi assinado usando a versão 3.0.0 ou 3.1.0, consulte o DecryptWithPermutediretório do repositório aws-database-encryption-sdk -dynamodb-java em para obter detalhes sobre GitHub como validar o conjunto com êxito.

Exemplos de Java

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Os exemplos a seguir mostram como usar a biblioteca de criptografia do lado do cliente para o DynamoDB para proteger os itens da tabela no aplicativo. Você pode encontrar mais exemplos (e contribuir com os seus) nos exemplos de Java no repositório aws-database-encryption-sdk - dynamodb em. GitHub

Os exemplos a seguir demonstram como configurar a biblioteca Java de criptografia do lado do cliente para o DynamoDB em uma nova tabela não preenchida do Amazon DynamoDB. Se você quiser configurar suas tabelas existentes do Amazon DynamoDB para criptografia do lado do cliente, consulte Adicionar versão 3.x a uma tabela existente.

Tópicos

- Uso do cliente aprimorado do DynamoDB

- Uso da API de nível inferior do DynamoDB
- Usando o nível inferior DynamoDbItemEncryptor

Uso do cliente aprimorado do DynamoDB

O exemplo a seguir mostra como usar o DynamoDB Enhanced Client e o `DynamoDbEncryptionInterceptor` com um token de autenticação do AWS KMS para criptografar itens da tabela do DynamoDB como parte de suas chamadas de API do DynamoDB.

Você pode usar qualquer chaveiro compatível com o DynamoDB Enhanced Client, mas recomendamos usar um dos AWS KMS chaveiros sempre que possível.

> **Note**
>
> O DynamoDB Enhanced Client não oferece suporte à criptografia pesquisável. Use o `DynamoDbEncryptionInterceptor` com a API de nível inferior do DynamoDB para usar criptografia pesquisável.

Veja a amostra de código completa: EnhancedPutGetExample.java

Etapa 1: criar o AWS KMS chaveiro

O exemplo a seguir é usado `CreateAwsKmsMrkMultiKeyring` para criar um AWS KMS chaveiro com uma chave KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

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

Etapa 2: criar um esquema de tabela a partir da classe de dados anotada

O exemplo a seguir usa a classe de dados anotada para criar o `TableSchema`.

Este exemplo pressupõe que as ações de classe e atributo de dados anotadas foram definidas usando o .java. SimpleClass Para obter mais orientações sobre como anotar suas ações de atributos, consulte Uso de uma classe de dados anotada.

> **Note**
>
> O SDK AWS de criptografia de banco de dados não oferece suporte a anotações em atributos aninhados.

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

Etapa 3: definir quais atributos são excluídos das assinaturas

O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto `":"` e usam o prefixo para definir os atributos não assinados permitidos. O cliente presume que qualquer nome de atributo com o prefixo `":"` está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Etapa 4: criar a configuração de criptografia

O exemplo a seguir define um mapa `tableConfigs` que representa a configuração de criptografia dessa tabela do DynamoDB.

Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Para obter mais informações, consulte Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB.

> **Note**
>
> Para usar criptografia pesquisável ou beacons assinados, você também deve incluir `SearchConfig` na configuração de criptografia.

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

Etapa 5: cria o **`DynamoDbEncryptionInterceptor`**

O exemplo a seguir cria o `DynamoDbEncryptionInterceptor` usando `tableConfigs` da Etapa 4.

```
final DynamoDbEncryptionInterceptor interceptor =
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

Etapa 6: criar um novo cliente AWS SDK do DynamoDB

O exemplo a seguir cria um novo cliente AWS SDK do DynamoDB usando **`interceptor`** o da Etapa 5.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                       .addExecutionInterceptor(interceptor)
                       .build())
        .build();
```

Etapa 7: criar o DynamoDB Enhanced Client e criar uma tabela

O exemplo a seguir cria o DynamoDB Enhanced Client usando o cliente AWS SDK DynamoDB criado na Etapa 6 e cria uma tabela usando a classe de dados anotada.

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
```

```
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

Etapa 8: criptografar e salvar um item da tabela

O exemplo a seguir coloca um item na tabela do DynamoDB usando o DynamoDB Enhanced Client. O item é criptografado e assinado no lado do cliente antes de ser enviado ao 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);
```

Uso da API de nível inferior do DynamoDB

O exemplo a seguir mostra como usar a API de nível inferior do DynamoDB com um token de autenticação do AWS KMS para criptografar e assinar automaticamente itens no lado do cliente com suas solicitações `PutItem` do DynamoDB.

Você pode usar qualquer chaveiro compatível, mas recomendamos usar um dos AWS KMS chaveiros sempre que possível.

Veja a amostra de código completa: BasicPutGetExample.java

Etapa 1: criar o AWS KMS chaveiro

O exemplo a seguir é usado `CreateAwsKmsMrkMultiKeyring` para criar um AWS KMS chaveiro com uma chave KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
```

```
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

Etapa 2: Configurar ações de atributos

O exemplo a seguir define um mapa `attributeActionsOnEncrypt` que representa exemplos de ações de atributos para um item da tabela.

> **Note**
>
> O exemplo a seguir não define nenhum atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`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);
```

Etapa 3: definir quais atributos são excluídos das assinaturas

O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto `":"` e usam o prefixo para definir os atributos não assinados permitidos. O cliente presume que qualquer nome de atributo com o prefixo `":"` está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Etapa 4: definir a configuração de criptografia de tabelas do DynamoDB

O exemplo a seguir define um mapa `tableConfigs` que representa a configuração de criptografia dessa tabela do DynamoDB.

Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Para obter mais informações, consulte Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB.

> **Note**
>
> Para usar criptografia pesquisável ou beacons assinados, você também deve incluir `SearchConfig` na configuração de criptografia.

```
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);
```

Etapa 5: criar o perfil do **`DynamoDbEncryptionInterceptor`**

O exemplo a seguir cria o `DynamoDbEncryptionInterceptor` usando `tableConfigs` da Etapa 4.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

Etapa 6: criar um novo cliente AWS SDK do DynamoDB

O exemplo a seguir cria um novo cliente AWS SDK do DynamoDB usando **`interceptor`** o da Etapa 5.

```
final DynamoDbClient ddb = DynamoDbClient.builder()
```

```
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

Etapa 7: criptografar e assinar um item da tabela do DynamoDB

O exemplo a seguir define um mapa `item` que representa um item da tabela de exemplo e coloca o item na tabela do DynamoDB. O item é criptografado e assinado no lado do cliente antes de ser enviado ao 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);
```

Usando o nível inferior DynamoDbItemEncryptor

O exemplo a seguir mostra como usar o nível inferior de `DynamoDbItemEncryptor` com um token de autenticação do AWS KMS para criptografar e assinar diretamente os itens da tabela. O `DynamoDbItemEncryptor` não coloca o item na tabela do DynamoDB.

Você pode usar qualquer chaveiro compatível com o DynamoDB Enhanced Client, mas recomendamos usar um dos AWS KMS chaveiros sempre que possível.

> **Note**
>
> O nível inferior do `DynamoDbItemEncryptor` não oferece suporte à criptografia pesquisável. Use o `DynamoDbEncryptionInterceptor` com a API de nível inferior do DynamoDB para usar criptografia pesquisável.

Veja a amostra de código completa: ItemEncryptDecryptExample.java

Etapa 1: criar o AWS KMS chaveiro

O exemplo a seguir é usado `CreateAwsKmsMrkMultiKeyring` para criar um AWS KMS chaveiro com uma chave KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

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

Etapa 2: Configurar ações de atributos

O exemplo a seguir define um mapa `attributeActionsOnEncrypt` que representa exemplos de ações de atributos para um item da tabela.

> **Note**
>
> O exemplo a seguir não define nenhum atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`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);
```

Etapa 3: definir quais atributos são excluídos das assinaturas

O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto `":"` e usam o prefixo para definir os atributos não assinados permitidos. O cliente presume que qualquer nome de atributo com o prefixo `":"` está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

```
final String unsignedAttrPrefix = ":";
```

Etapa 4: definir a configuração de **`DynamoDbItemEncryptor`**

O exemplo a seguir define a configuração para `DynamoDbItemEncryptor`.

Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Para obter mais informações, consulte Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB.

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

Etapa 5: criar o perfil do **`DynamoDbItemEncryptor`**

O exemplo a seguir cria um novo `DynamoDbItemEncryptor` usando `config` da Etapa 4.

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

Etapa 6: criptografar e assinar diretamente um item da tabela

O exemplo a seguir criptografa e assina diretamente um item usando o `DynamoDbItemEncryptor`. O `DynamoDbItemEncryptor` não coloca o item na tabela do DynamoDB.

```
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();
```

Configurar uma tabela existente do DynamoDB para usar o SDK de criptografia de banco de dados para AWS o DynamoDB

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Com a versão 3.x da biblioteca Java de criptografia do lado do cliente para DynamoDB, é possível configurar as tabelas existentes do Amazon DynamoDB para a criptografia do lado do cliente. Este tópico fornece orientação sobre as três etapas que você deve seguir para adicionar a versão 3.x para uma tabela existente e preenchida do DynamoDB.

Pré-requisitos

A versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB requer o DynamoDB Enhanced Client fornecido em AWS SDK for Java 2.x . Se você ainda usa o DynamoDBMapper, deve migrar AWS SDK for Java 2.x para o DynamoDB Enhanced Client.

Siga as instruções para migrar da versão 1.x para a 2.x do AWS SDK para Java.

Em seguida, siga as instruções para Começar a usar a API do DynamoDB Enhanced Client.

Antes de configurar sua tabela para usar a biblioteca Java de criptografia do lado do cliente para o DynamoDB, você precisa gerar `TableSchema` usando uma classe de dados anotada e criar um cliente aprimorado.

Etapa 1: preparar para ler e gravar itens criptografados

Conclua as etapas a seguir para preparar seu cliente SDK AWS de criptografia de banco de dados para ler e gravar itens criptografados. Depois de implantar as alterações a seguir, seu cliente continuará lendo e gravando itens de texto simples. Ele não criptografará nem assinará nenhum novo item gravado na tabela, mas poderá descriptografar itens criptografados assim que eles aparecerem. Essas mudanças preparam o cliente para começar a criptografar novos itens. As alterações a seguir devem ser implantadas em cada leitor antes de prosseguir para a próxima etapa.

1. Definir suas ações de atributos

   Atualize sua classe de dados anotada para incluir ações de atributos que definam quais valores de atributos serão criptografados e assinados, quais serão somente assinados e quais serão ignorados.

   Consulte o SimpleClassarquivo.java no repositório aws-database-encryption-sdk -dynamodb em GitHub para obter mais orientações sobre as anotações do DynamoDB Enhanced Client.

   Por padrão, os atributos da chave primária são assinados, mas não criptografados (`SIGN_ONLY`), e todos os outros atributos são criptografados e assinados (`ENCRYPT_AND_SIGN`). Para especificar exceções, use as anotações de criptografia definidas na biblioteca de criptografia do lado do cliente para Java do DynamoDB. Por exemplo, se você quiser que um atributo específico seja assinado, use apenas a anotação `@DynamoDbEncryptionSignOnly`. Se você quiser que um atributo específico seja assinado e incluído no contexto de criptografia, use a `@DynamoDbEncryptionSignAndIncludeInEncryptionContext` anotação. Se desejar que um atributo específico não seja assinado nem criptografado (`DO_NOTHING`), use a anotação `@DynamoDbEncryptionDoNothing`.

   > **Note**
   >
   > Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Para ver um exemplo que mostra as anotações usadas para definir`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, consulte SimpleClass 4.java.

Por obter exemplos de anotações, consulte Uso de uma classe de dados anotada.

2. Definir quais atributos são excluídos das assinaturas

   O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto `":"` e usam o prefixo para definir os atributos não assinados permitidos. O cliente presumirá que qualquer nome de atributo com o prefixo `":"` está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

   ```
   final String unsignedAttrPrefix = ":";
   ```

3. Criar um token de autenticação

   O exemplo a seguir cria um token de autenticação do AWS KMS. O AWS KMS chaveiro usa criptografia simétrica ou RSA assimétrica AWS KMS keys para gerar, criptografar e descriptografar chaves de dados.

   Este exemplo usa `CreateMrkMultiKeyring` para criar um token de autenticação do AWS KMS com uma chave do KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

   ```
   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. Definir a configuração de criptografia de tabelas do DynamoDB

   O exemplo a seguir define um mapa `tableConfigs` que representa a configuração de criptografia dessa tabela do DynamoDB.

   Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Para obter mais informações, consulte Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB.

Você deve especificar `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` como substituição de texto simples. Essa política continua lendo e gravando itens de texto simples, lendo itens criptografados e preparando o cliente para gravar itens criptografados.

```
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. Criar a **`DynamoDbEncryptionInterceptor`**

   O exemplo a seguir cria o `DynamoDbEncryptionInterceptor` usando `tableConfigs` da Etapa 3.

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

Etapa 2: gravar itens criptografados e assinados

Atualize a política de texto simples em sua configuração `DynamoDbEncryptionInterceptor` para permitir que o cliente grave itens criptografados e assinados. Depois de implantar a seguinte alteração, o cliente criptografará e assinará novos itens com base nas ações de atributos que você configurou na Etapa 1. O cliente poderá ler itens de texto simples e itens criptografados e assinados.

Antes de prosseguir para a Etapa 3, você deve criptografar e assinar todos os itens de texto sem formatação existentes em sua tabela. Não há uma única métrica ou consulta que você possa executar para criptografar rapidamente seus itens de texto sem formatação existentes. Use o processo que faz mais sentido para o seu sistema. Por exemplo, é possível usar um processo assíncrono que varre lentamente a tabela e reescreve os itens usando as ações de atributos e a

configuração de criptografia que você definiu. Para identificar os itens de texto simples em sua tabela, recomendamos escanear todos os itens que não contêm os `aws_dbe_foot` atributos `aws_dbe_head` e que o SDK de criptografia de AWS banco de dados adiciona aos itens quando eles são criptografados e assinados.

O exemplo a seguir atualiza a configuração de criptografia de tabela da Etapa 1. Você deve atualizar a substituição de texto simples com `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`. Essa política continua lendo itens de texto simples, mas também lê e grava itens criptografados. Crie um novo `DynamoDbEncryptionInterceptor` usando o atualizado`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);
```

Etapa 3: somente ler itens criptografados e assinados

Depois de criptografar e assinar todos os seus itens, atualize a substituição de texto simples na configuração `DynamoDbEncryptionInterceptor` para permitir que o cliente somente leia e grave itens criptografados e assinados. Depois de implantar a seguinte alteração, o cliente criptografará e assinará novos itens com base nas ações de atributos que você configurou na Etapa 1. O cliente só poderá ler itens criptografados e assinados.

O exemplo a seguir atualiza a configuração de criptografia de tabela da Etapa 2. É possível atualizar a substituição de texto sem formatação com `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT` ou remover a política de texto sem formatação da sua configuração. Por padrão, o cliente só lê e grava itens criptografados e assinados. Crie um novo `DynamoDbEncryptionInterceptor` usando o atualizado`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);
```

Migrar para a versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB

> Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

A Versão 3.x da biblioteca Java de criptografia do lado do cliente para DynamoDB é uma grande reescrita da base de código 2.x. Ela inclui muitas atualizações, como um novo formato de dados estruturados, suporte aprimorado para multilocação, alterações de esquema contínuas e suporte à criptografia pesquisável. Este tópico fornece orientação sobre como migrar seu código para a versão 3.x.

Migrar da versão 1.x para a versão 2.x

Migre para a versão 2.x antes de migrar para a versão 3.x.. A versão 2.x mudou o símbolo do provedor mais recente de `MostRecentProvider` para `CachingMostRecentProvider`. Se você usa atualmente a versão 1.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB com o símbolo `MostRecentProvider`, você deverá atualizar o nome do símbolo em seu código para `CachingMostRecentProvider`. Para obter mais informações, consulte Atualizações do provedor mais recente.

Migrar da versão 2.x para a versão 3.x

Os procedimentos a seguir descrevem como migrar seu código da versão 2.x para a versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB.

Etapa 1. Preparar para ler itens no novo formato

Conclua as etapas a seguir para preparar seu cliente SDK do AWS Database Encryption para ler itens no novo formato. Depois de implantar as alterações a seguir, seu cliente continuará se comportando da mesma maneira que na versão 2.x. Seu cliente continuará lendo e gravando itens no formato da versão 2.x, mas essas alterações preparam o cliente para ler itens no novo formato.

Atualize seu AWS SDK para Java para a versão 2.x

A versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB requer o DynamoDB Enhanced Client. O DynamoDB Enhanced Client substitui o DBMapperDynamo usado nas versões anteriores. Para usar o cliente aprimorado, você deve usar o AWS SDK for Java 2.x.

Siga as instruções para migrar da versão 1.x para a 2.x do AWS SDK para Java.

Para obter mais informações sobre quais AWS SDK for Java 2.x módulos são necessários, consultePré-requisitos.

Configurar seu cliente para ler itens criptografados por versões legadas

Os procedimentos a seguir fornecem uma visão geral das etapas demonstradas no exemplo de código abaixo.

1. Crie um token de autenticação.

   Os tokens de autenticação e os gerenciadores de materiais criptográficos substituem os fornecedores de materiais criptográficos usados nas versões anteriores da biblioteca Java de criptografia do lado do cliente para o DynamoDB.

   > **Important**
   >
   > As chaves de empacotamento que você especifica ao criar um token de autenticação devem ser as mesmas que você usou com seu provedor de materiais criptográficos na versão 2.x.

2. Crie um esquema de tabela sobre sua classe anotada.

   Essa etapa define as ações de atributos que serão usadas quando você começar a gravar itens no novo formato.

Para obter orientação sobre como usar o novo DynamoDB Enhanced Client, consulte Gerar um `TableSchema` no Guia do desenvolvedor do AWS SDK para Java .

O exemplo a seguir pressupõe que você atualizou sua classe anotada da versão 2.x usando as novas anotações de ações de atributos. Para obter mais orientações sobre como anotar suas ações de atributos, consulte Uso de uma classe de dados anotada.

> **Note**
>
> Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.
> Para ver um exemplo que mostra as anotações usadas para definir`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, consulte SimpleClass 4.java.

3. Defina quais atributos serão excluídos das assinaturas.
4. Configure um mapa explícito das ações de atributos configuradas em sua classe modelada na versão 2.x.

   Essa etapa define as ações de atributos usadas para gravar itens no formato antigo.
5. Configure o `DynamoDBEncryptor` que você usou na versão 2.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB.
6. Configure o comportamento legado.
7. Crie um `DynamoDbEncryptionInterceptor`.
8. Crie um novo cliente AWS SDK do DynamoDB.
9. Crie o `DynamoDBEnhancedClient` e crie uma tabela com sua classe modelada.

   Para obter mais informações sobre o DynamoDB Enhanced Client, consulte criar um cliente aprimorado.

```
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);
    }
}
```

Etapa 2. Gravar itens no novo formato

Depois de implantar as alterações da Etapa 1 em todos os leitores, conclua as etapas a seguir para configurar seu cliente SDK do AWS Database Encryption para gravar itens no novo formato. Depois de implantar as seguintes alterações, seu cliente continuará lendo itens no formato antigo e começará a gravar e ler itens no novo formato.

Os procedimentos a seguir fornecem uma visão geral das etapas demonstradas no exemplo de código abaixo.

1. Continue configurando seu token de autenticação, esquema de tabela, ações de atributos herdados e `allowedUnsignedAttributes` e `DynamoDBEncryptor` como você fez na Etapa 1.
2. Atualize seu comportamento legado para gravar somente novos itens usando o novo formato.
3. Criar uma `DynamoDbEncryptionInterceptor`
4. Crie um novo cliente AWS SDK do DynamoDB.
5. Crie o `DynamoDBEnhancedClient` e crie uma tabela com sua classe modelada.

   Para obter mais informações sobre o DynamoDB Enhanced Client, consulte criar um cliente aprimorado.

```
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);
    }
}
```

Depois de implantar as alterações da Etapa 2, você deve criptografar novamente todos os itens antigos em sua tabela com o novo formato antes de continuar na Etapa 3. Não há uma única métrica ou consulta que você possa executar para criptografar rapidamente seus itens existentes. Use o processo que faz mais sentido para o seu sistema. Por exemplo, é possível usar um processo assíncrono que varre lentamente a tabela e reescreve os itens usando as ações do novo atributo e a configuração de criptografia que você definiu.

Etapa 3. Ler e gravar somente itens no novo formato

Depois de criptografar novamente todos os itens da tabela com o novo formato, é possível remover o comportamento legado da configuração. Conclua as etapas a seguir para configurar o cliente para ler e gravar somente itens no novo formato.

Os procedimentos a seguir fornecem uma visão geral das etapas demonstradas no exemplo de código abaixo.

1. Continue configurando seu token de autenticação, esquema de tabela e `allowedUnsignedAttributes` como você fez na Etapa 1. Remova as ações do atributo legado e `DynamoDBEncryptor` da sua configuração.
2. Crie um `DynamoDbEncryptionInterceptor`.
3. Crie um novo cliente AWS SDK do DynamoDB.
4. Crie o `DynamoDBEnhancedClient` e crie uma tabela com sua classe modelada.

   Para obter mais informações sobre o DynamoDB Enhanced Client, consulte criar um cliente aprimorado.

```
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

Este tópico explica como instalar e usar a versão 3. x da biblioteca de criptografia do lado do cliente.NET para o DynamoDB. Para obter detalhes sobre a programação com o SDK AWS de

criptografia de banco de dados para DynamoDB, consulte os exemplos do.NET no repositório - dynamodb em aws-database-encryption-sdk. GitHub

A biblioteca de criptografia do lado do cliente.NET para o DynamoDB é para desenvolvedores que estão escrevendo aplicativos em C# e em outras linguagens de programação.NET. É compatível com Windows, macOS e Linux.

Todas as implementações da linguagem de programação do SDK de criptografia de AWS banco de dados para DynamoDB são interoperáveis. No entanto, o SDK para .NET não suporta valores vazios para tipos de dados de lista ou mapa. Isso significa que, se você usar a biblioteca de criptografia Java do lado do cliente para o DynamoDB para escrever um item que contém valores vazios para um tipo de dados de lista ou mapa, não poderá descriptografar e ler esse item usando a biblioteca de criptografia do lado do cliente.NET para o DynamoDB.

Tópicos

- Instalação da biblioteca de criptografia do lado do cliente.NET para o DynamoDB
- Depuração com o.NET
- Usando a biblioteca de criptografia do lado do cliente.NET para o DynamoDB
- Exemplos do.NET
- Configurar uma tabela existente do DynamoDB para usar o SDK de criptografia de banco de dados da AWS para DynamoDB

Instalação da biblioteca de criptografia do lado do cliente.NET para o DynamoDB

A biblioteca de criptografia do lado do cliente.NET para o DynamoDB está disponível como AWS.Cryptography. DbEncryptionSDK. DynamoDbpacote em NuGet. Para obter detalhes sobre como instalar e criar a biblioteca, consulte o arquivo.NET README.md no repositório -dynamodb. aws-database-encryption-sdk A biblioteca de criptografia do lado do cliente.NET para o DynamoDB exige as chaves SDK para .NET mesmo que você não esteja usando (). AWS Key Management Service AWS KMS O SDK para .NET é instalado com o NuGet pacote.

Versão 3. x da biblioteca de criptografia do lado do cliente.NET para DynamoDB é compatível com o.NET 6.0 e .NET Framework net48 e versões posteriores.

Depuração com o.NET

A biblioteca de criptografia do lado do cliente.NET para o DynamoDB não gera nenhum registro. As exceções na biblioteca de criptografia do lado do cliente.NET para o DynamoDB geram uma mensagem de exceção, mas não rastreiam a pilha.

Para ajudar na depuração, certifique-se de habilitar o login no SDK para .NET. Os registros e as mensagens de erro do SDK para .NET podem ajudá-lo a distinguir os erros decorrentes do e os da biblioteca de criptografia SDK para .NET do lado do cliente.NET para o DynamoDB. Para obter ajuda com o SDK para .NET registro, consulte AWSLoggingo Guia do AWS SDK para .NET desenvolvedor. (Para ver o tópico, expanda a seção Abrir para ver o conteúdo do .NET Framework.)

Usando a biblioteca de criptografia do lado do cliente.NET para o DynamoDB

Este tópico explica algumas das funções e classes auxiliares na versão 3. x da biblioteca de criptografia do lado do cliente.NET para o DynamoDB.

Para obter detalhes sobre a programação com a biblioteca de criptografia do lado do cliente.NET para o DynamoDB, consulte os exemplos do.NET no repositório -dynamodb em. aws-database-encryption-sdk GitHub

Tópicos

- Criptografadores de itens
- Ações de atributos no SDK de criptografia de banco de dados da AWS para DynamoDB
- Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB
- Atualização de itens com o SDK AWS de criptografia de banco de dados

Criptografadores de itens

Basicamente, o SDK AWS de criptografia de banco de dados para DynamoDB é um criptografador de itens. Você pode usar a versão 3. x da biblioteca de criptografia do lado do cliente.NET para que o DynamoDB criptografe, assine, verifique e descriptografe os itens da tabela do DynamoDB das seguintes maneiras.

O SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB

> Você pode usar sua configuração de criptografia de tabela para criar um cliente do DynamoDB que criptografe e assine automaticamente itens do lado do cliente com suas solicitações

do DynamoDB. `PutItem` Você pode usar esse cliente diretamente ou criar um modelo de documento ou modelo de persistência de objetos.

Você deve usar o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB para usar a criptografia pesquisável.

O **`DynamoDbItemEncryptor`** de nível inferior

O `DynamoDbItemEncryptor` de nível inferior criptografa, assina ou descriptografa e verifica diretamente os itens da tabela sem chamar o DynamoDB. Ele não faz solicitações `PutItem` ou `GetItem` para o DynamoDB. Por exemplo, é possível usar o `DynamoDbItemEncryptor` de nível inferior para descriptografar e verificar diretamente um item do DynamoDB que você já recuperou. Se você usar o nível inferior`DynamoDbItemEncryptor`, recomendamos usar o modelo de programação de baixo nível SDK para .NET fornecido para comunicação com o DynamoDB.

O nível inferior do `DynamoDbItemEncryptor` não oferece suporte à criptografia pesquisável.

Ações de atributos no SDK de criptografia de banco de dados da AWS para DynamoDB

As ações de atributo determinam quais valores de atributos são criptografados e assinados, quais são somente assinados, quais são assinados e incluídos no contexto de criptografia e quais são ignorados.

Para especificar ações de atributos com o cliente.NET, defina manualmente ações de atributos usando um modelo de objeto. Especifique suas ações de atributo criando um `Dictionary` objeto no qual os pares nome-valor representam os nomes dos atributos e as ações especificadas.

Especifique `ENCRYPT_AND_SIGN` para criptografar e assinar um atributo.
Especifique `SIGN_ONLY` para assinar, mas não criptografar um atributo. Especifique `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` para assinar um atributo e incluí-lo no contexto de criptografia. Não é possível criptografar um atributo sem também assiná-lo. Especifique `DO_NOTHING` para ignorar um atributo.

Os atributos de partição e classificação devem ser `SIGN_ONLY` ou`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você definir qualquer atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> Depois de definir suas ações de atributo, você deverá definir quais atributos serão excluídos das assinaturas. Para facilitar a adição de novos atributos não assinados no futuro, recomendamos escolher um prefixo distinto (como ":") para identificar os atributos não assinados. Inclua esse prefixo no nome do atributo para todos os atributos marcados como `DO_NOTHING` ao definir o esquema e as ações de atributos do DynamoDB.

O modelo de objeto a seguir demonstra como especificar`ENCRYPT_AND_SIGN`, `SIGN_ONLY`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, e `DO_NOTHING` atribuir ações com o cliente.NET. Este exemplo usa o prefixo ":" para identificar `DO_NOTHING` atributos.

> **Note**
>
> Para usar a ação `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` criptográfica, você deve usar a versão 3.3 ou posterior do SDK de criptografia de AWS banco de dados. Implante a nova versão para todos os leitores antes de atualizar seu modelo de dados para incluí-la`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
};
```

Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB

Ao usar o AWS Database Encryption SDK, você deve definir explicitamente uma configuração de criptografia para sua tabela do DynamoDB. Os valores necessários em sua configuração de criptografia dependem se você definiu suas ações de atributo manualmente ou com uma classe de dados anotada.

O trecho a seguir define uma configuração de criptografia de tabela do DynamoDB usando o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB e atributos não assinados permitidos definidos por um prefixo distinto.

```
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);
```

Nome da tabela lógica

Um nome de tabela lógica para sua tabela do DynamoDB.

O nome da tabela lógica é vinculado criptograficamente a todos os dados armazenados na tabela para simplificar as operações de restauração do DynamoDB. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Você deve sempre especificar o mesmo nome de tabela lógica. Para que a descriptografia seja bem-sucedida, o nome da tabela lógica deve corresponder ao nome especificado na criptografia. Se o nome da tabela do DynamoDB mudar após a restauração da tabela do DynamoDB a partir de um backup, o nome da tabela lógica garantirá que a operação de descriptografia ainda reconheça a tabela.

Atributos não assinados permitidos

Os atributos marcados `DO_NOTHING` em suas ações de atributos.

Os atributos não assinados permitidos informam ao cliente quais atributos são excluídos das assinaturas. O cliente presume que todos os outros atributos estão incluídos na assinatura. Em

seguida, ao descriptografar um registro, o cliente determina quais atributos ele precisa verificar e quais ignorar dos atributos não assinados permitidos que você especificou. Não é possível remover um atributo dos atributos não assinados permitidos.

É possível definir explicitamente os atributos não assinados permitidos criando uma matriz que lista todos os atributos `DO_NOTHING`. Também é possível especificar um prefixo distinto ao nomear os atributos `DO_NOTHING` e usar o prefixo para informar ao cliente quais atributos não estão assinados. É altamente recomendável especificar um prefixo distinto, pois isso simplifica o processo de adicionar um novo atributo `DO_NOTHING` no futuro. Para obter mais informações, consulte Atualizar seu modelo de dados.

Se você não especificar um prefixo para todos os atributos `DO_NOTHING`, poderá configurar uma matriz `allowedUnsignedAttributes` que liste explicitamente todos os atributos que o cliente deve esperar que não estejam assinados ao encontrá-los na descriptografia. Você só deve definir explicitamente seus atributos não assinados permitidos se for absolutamente necessário.

Configuração de pesquisa (opcional)

O `SearchConfig` define a versão do beacon.

O `SearchConfig` deve ser especificado para usar criptografia pesquisável ou beacons assinados.

Conjunto de algoritmos (opcional)

O `algorithmSuiteId` define qual conjunto de algoritmos o SDK de criptografia de banco de dados da AWS usará.

A menos que você especifique explicitamente um conjunto alternativo de algoritmos, o SDK do AWS Database Encryption usa o conjunto de algoritmos padrão. O conjunto de algoritmos padrão usa o algoritmo AES-GCM com derivação de chaves, assinaturas digitais e comprometimento de chaves. Embora o conjunto de algoritmos padrão provavelmente seja adequado para a maioria dos aplicativos, é possível escolher um conjunto alternativo de algoritmos. Por exemplo, alguns modelos de confiança seriam satisfeitos com um pacote de algoritmos sem assinaturas digitais. Para obter informações sobre os conjuntos de algoritmos compatíveis com o SDK do AWS Database Encryption, consulteSuítes de algoritmos compatíveis no SDK AWS de criptografia de banco de dados.

Para selecionar o conjunto de algoritmos AES-GCM sem assinaturas digitais ECDSA, inclua o seguinte trecho em sua configuração de criptografia de tabela.

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Atualização de itens com o SDK AWS de criptografia de banco de dados

O SDK AWS de criptografia de banco de dados não oferece suporte a ddb: UpdateItem para itens que incluem atributos criptografados ou assinados. Para atualizar um atributo criptografado ou assinado, você deve usar ddb: PutItem. Se algum item existir em uma tabela específica com a mesma chave primária de um item existente na consulta `PutItem`, o novo item substituirá completamente o item já existente. Também é possível usar o CLOBBER para limpar e substituir todos os atributos ao salvar após atualizar seus itens.

Exemplos do.NET

Os exemplos a seguir mostram como usar a biblioteca de criptografia do lado do cliente.NET para o DynamoDB para proteger os itens da tabela em seu aplicativo. Para encontrar mais exemplos (e contribuir com seus próprios), consulte os exemplos do.NET no repositório aws-database-encryption-sdk -dynamodb em. GitHub

Os exemplos a seguir demonstram como configurar a biblioteca de criptografia do lado do cliente.NET para o DynamoDB em uma nova tabela não preenchida do Amazon DynamoDB. Se você quiser configurar suas tabelas existentes do Amazon DynamoDB para criptografia do lado do cliente, consulte Adicionar versão 3.x a uma tabela existente.

Tópicos

- Usando o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB
- Usando o nível inferior DynamoDbItemEncryptor

Usando o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB

O exemplo a seguir mostra como usar o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB com um AWS KMS chaveiro para criptografar e assinar automaticamente itens do lado do cliente com suas solicitações do DynamoDB. `PutItem`

Você pode usar qualquer chaveiro compatível, mas recomendamos usar um dos AWS KMS chaveiros sempre que possível.

Veja o exemplo de código completo: BasicPutGetExample.cs

Etapa 1: criar o AWS KMS chaveiro

O exemplo a seguir é usado `CreateAwsKmsMrkMultiKeyring` para criar um AWS KMS chaveiro com uma chave KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

Etapa 2: Configurar ações de atributos

O exemplo a seguir define um `attributeActionsOnEncrypt` dicionário que representa exemplos de ações de atributos para um item da tabela.

> **Note**
>
> O exemplo a seguir não define nenhum atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`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
};
```

Etapa 3: definir quais atributos são excluídos das assinaturas

O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto `":"` e usam o prefixo para definir os atributos não assinados permitidos. O cliente presume que qualquer nome de atributo com o prefixo `":"` está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

```
const String unsignAttrPrefix = ":";
```

Etapa 4: definir a configuração de criptografia de tabelas do DynamoDB

O exemplo a seguir define um mapa `tableConfigs` que representa a configuração de criptografia dessa tabela do DynamoDB.

Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Para obter mais informações, consulte Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB.

> **Note**
>
> Para usar criptografia pesquisável ou beacons assinados, você também deve incluir `SearchConfig` na configuração de criptografia.

```
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);
```

Etapa 5: criar um novo cliente AWS SDK do DynamoDB

O exemplo a seguir cria um novo cliente AWS SDK do DynamoDB usando **`TableEncryptionConfigs`** o da Etapa 4.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

Etapa 6: criptografar e assinar um item de tabela do DynamoDB

O exemplo a seguir define um `item` dicionário que representa um item de tabela de amostra e coloca o item na tabela do DynamoDB. O item é criptografado e assinado no lado do cliente antes de ser enviado ao 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);
```

Usando o nível inferior **`DynamoDbItemEncryptor`**

O exemplo a seguir mostra como usar o nível inferior de `DynamoDbItemEncryptor` com um token de autenticação do AWS KMS para criptografar e assinar diretamente os itens da tabela. O `DynamoDbItemEncryptor` não coloca o item na tabela do DynamoDB.

Você pode usar qualquer chaveiro compatível com o DynamoDB Enhanced Client, mas recomendamos usar um dos AWS KMS chaveiros sempre que possível.

> **Note**
>
> O nível inferior do `DynamoDbItemEncryptor` não oferece suporte à criptografia pesquisável. Use o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB para usar criptografia pesquisável.

Veja o exemplo de código completo: ItemEncryptDecryptExample.cs

Etapa 1: criar o AWS KMS chaveiro

O exemplo a seguir é usado `CreateAwsKmsMrkMultiKeyring` para criar um AWS KMS chaveiro com uma chave KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

Etapa 2: Configurar ações de atributos

O exemplo a seguir define um `attributeActionsOnEncrypt` dicionário que representa exemplos de ações de atributos para um item da tabela.

> **Note**
>
> O exemplo a seguir não define nenhum atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`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
};
```

Etapa 3: definir quais atributos são excluídos das assinaturas

O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto `":"` e usam o prefixo para definir os atributos não assinados permitidos. O cliente presume que qualquer nome de atributo com o prefixo `":"` está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

```
String unsignAttrPrefix = ":";
```

Etapa 4: definir a configuração de **`DynamoDbItemEncryptor`**

O exemplo a seguir define a configuração para `DynamoDbItemEncryptor`.

Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Para obter mais informações, consulte Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB.

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
```

Etapa 5: criar o perfil do **`DynamoDbItemEncryptor`**

O exemplo a seguir cria um novo `DynamoDbItemEncryptor` usando `config` da Etapa 4.

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

Etapa 6: criptografar e assinar diretamente um item da tabela

O exemplo a seguir criptografa e assina diretamente um item usando o `DynamoDbItemEncryptor`. O `DynamoDbItemEncryptor` não coloca o item na tabela do DynamoDB.

```
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;
```

Configurar uma tabela existente do DynamoDB para usar o SDK de criptografia de banco de dados da AWS para DynamoDB

Com a versão 3. x da biblioteca de criptografia do lado do cliente.NET para o DynamoDB, você pode configurar suas tabelas existentes do Amazon DynamoDB para criptografia do lado do cliente. Este tópico fornece orientação sobre as três etapas que você deve seguir para adicionar a versão 3.x para uma tabela existente e preenchida do DynamoDB.

Etapa 1: preparar para ler e gravar itens criptografados

Conclua as etapas a seguir para preparar seu cliente SDK AWS de criptografia de banco de dados para ler e gravar itens criptografados. Depois de implantar as alterações a seguir, seu cliente continuará lendo e gravando itens de texto simples. Ele não criptografará nem assinará nenhum novo item gravado na tabela, mas poderá descriptografar itens criptografados assim que eles aparecerem. Essas mudanças preparam o cliente para começar a criptografar novos itens. As alterações a seguir devem ser implantadas em cada leitor antes de prosseguir para a próxima etapa.

1. Definir suas ações de atributos

   Crie um modelo de objeto para definir quais valores de atributos serão criptografados e assinados, quais serão somente assinados e quais serão ignorados.

   Por padrão, os atributos da chave primária são assinados, mas não criptografados (`SIGN_ONLY`), e todos os outros atributos são criptografados e assinados (`ENCRYPT_AND_SIGN`).

   Especifique `ENCRYPT_AND_SIGN` para criptografar e assinar um atributo.
   Especifique `SIGN_ONLY` para assinar, mas não criptografar um atributo. Especifique `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` para assinar e atribuir e incluí-los no contexto de criptografia. Não é possível criptografar um atributo sem também assiná-lo. Especifique `DO_NOTHING` para ignorar um atributo. Para obter mais informações, consulte Ações de atributos no SDK de criptografia de banco de dados da AWS para DynamoDB.

> **Note**
>
> Se você especificar algum `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` atributo, os atributos de partição e classificação também deverão ser`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. Definir quais atributos são excluídos das assinaturas

   O exemplo a seguir pressupõe que todos os atributos `DO_NOTHING` compartilham o prefixo distinto "`:`" e usam o prefixo para definir os atributos não assinados permitidos. O cliente presumirá que qualquer nome de atributo com o prefixo "`:`" está excluído das assinaturas. Para obter mais informações, consulte Allowed unsigned attributes.

   ```
   const String unsignAttrPrefix = ":";
   ```

3. Criar um token de autenticação

   O exemplo a seguir cria um token de autenticação do AWS KMS. O AWS KMS chaveiro usa criptografia simétrica ou RSA assimétrica AWS KMS keys para gerar, criptografar e descriptografar chaves de dados.

   Este exemplo usa `CreateMrkMultiKeyring` para criar um token de autenticação do AWS KMS com uma chave do KMS de criptografia simétrica. O método `CreateAwsKmsMrkMultiKeyring` garante que o token de autenticação manipule corretamente chaves de região única e de várias regiões.

   ```
   var matProv = new MaterialProviders(new MaterialProvidersConfig());
   var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
   ```

```
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. Definir a configuração de criptografia de tabelas do DynamoDB

   O exemplo a seguir define um mapa `tableConfigs` que representa a configuração de criptografia dessa tabela do DynamoDB.

   Este exemplo especifica o nome da tabela do DynamoDB como o nome lógico da tabela. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez.

   Você deve especificar `FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` como substituição de texto simples. Essa política continua lendo e gravando itens de texto simples, lendo itens criptografados e preparando o cliente para gravar itens criptografados.

   Para obter mais informações sobre os valores incluídos na configuração de criptografia da tabela, consulteConfiguração de criptografia no SDK de criptografia de banco de dados da AWS para 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. Crie um novo cliente AWS SDK do DynamoDB

   O exemplo a seguir cria um novo cliente AWS SDK do DynamoDB usando **`TableEncryptionConfigs`** o da Etapa 4.

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

Etapa 2: gravar itens criptografados e assinados

Atualize a política de texto simples em sua configuração de criptografia de tabela para permitir que o cliente grave itens criptografados e assinados. Depois de implantar a seguinte alteração, o cliente criptografará e assinará novos itens com base nas ações de atributos que você configurou na Etapa 1. O cliente poderá ler itens de texto simples e itens criptografados e assinados.

Antes de prosseguir para a Etapa 3, você deve criptografar e assinar todos os itens de texto sem formatação existentes em sua tabela. Não há uma única métrica ou consulta que você possa executar para criptografar rapidamente seus itens de texto sem formatação existentes. Use o processo que faz mais sentido para o seu sistema. Por exemplo, é possível usar um processo assíncrono que varre lentamente a tabela e reescreve os itens usando as ações de atributos e a configuração de criptografia que você definiu. Para identificar os itens de texto simples em sua tabela, recomendamos escanear todos os itens que não contêm os `aws_dbe_foot` atributos `aws_dbe_head` e que o SDK de criptografia de AWS banco de dados adiciona aos itens quando eles são criptografados e assinados.

O exemplo a seguir atualiza a configuração de criptografia de tabela da Etapa 1. Você deve atualizar a substituição de texto simples com `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT`. Essa política continua lendo itens de texto simples, mas também lê e grava itens criptografados. Crie um novo cliente AWS SDK do DynamoDB usando o atualizado. `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);
```

Etapa 3: somente ler itens criptografados e assinados

Depois de criptografar e assinar todos os seus itens, atualize a substituição de texto simples na configuração de criptografia de tabela para permitir que o cliente leia e grave somente itens

criptografados e assinados. Depois de implantar a seguinte alteração, o cliente criptografará e assinará novos itens com base nas ações de atributos que você configurou na Etapa 1. O cliente só poderá ler itens criptografados e assinados.

O exemplo a seguir atualiza a configuração de criptografia de tabela da Etapa 2. É possível atualizar a substituição de texto sem formatação com `FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT` ou remover a política de texto sem formatação da sua configuração. Por padrão, o cliente só lê e grava itens criptografados e assinados. Crie um novo cliente AWS SDK do DynamoDB usando o atualizado. `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

Este tópico explica como instalar e usar a versão 1. x da biblioteca de criptografia do lado do cliente Rust para DynamoDB. Para obter detalhes sobre a programação com o SDK AWS de criptografia de banco de dados para DynamoDB, consulte os exemplos de Rust no repositório -dynamodb em. aws-database-encryption-sdk GitHub

Todas as implementações da linguagem de programação do SDK de criptografia de AWS banco de dados para DynamoDB são interoperáveis.

Tópicos

- Pré-requisitos
- Instalação
- Usando a biblioteca de criptografia do lado do cliente Rust para o DynamoDB

Pré-requisitos

Antes de instalar a biblioteca de criptografia do lado do cliente Rust para o DynamoDB, verifique se você tem os seguintes pré-requisitos.

Instale Rust and Cargo

Instale a versão estável atual do Rust usando o rustup.

Para obter mais informações sobre como baixar e instalar o rustup, consulte os procedimentos de instalação no The Cargo Book.

Instalação

A biblioteca de criptografia do lado do cliente Rust para o DynamoDB está disponível como caixa em Crates.io. aws-db-esdk Para obter detalhes sobre como instalar e criar a biblioteca, consulte o arquivo README.md no repositório -dynamodb. aws-database-encryption-sdk GitHub

Manualmente

Para instalar a biblioteca de criptografia do lado do cliente Rust para o DynamoDB, clone ou baixe o repositório -dynamodb. aws-database-encryption-sdk GitHub

Para instalar a versão mais recente

Execute o seguinte comando Cargo no diretório do seu projeto:

```
cargo add aws-db-esdk
```

Ou adicione a seguinte linha ao seu Cargo.toml:

```
aws-db-esdk = "<version>"
```

Usando a biblioteca de criptografia do lado do cliente Rust para o DynamoDB

Este tópico explica algumas das funções e classes auxiliares na versão 1. x da biblioteca de criptografia do lado do cliente Rust para DynamoDB.

Para obter detalhes sobre a programação com a biblioteca de criptografia do lado do cliente Rust para o DynamoDB, consulte os exemplos do Rust no repositório -dynamodb em. aws-database-encryption-sdk GitHub

Tópicos

- Criptografadores de itens
- Ações de atributos no SDK de criptografia de banco de dados da AWS para DynamoDB
- Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB
- Atualização de itens com o SDK AWS de criptografia de banco de dados

Criptografadores de itens

Basicamente, o SDK AWS de criptografia de banco de dados para DynamoDB é um criptografador de itens. Você pode usar a versão 1. x da biblioteca de criptografia do lado do cliente Rust para que o DynamoDB criptografe, assine, verifique e descriptografe os itens da tabela do DynamoDB das seguintes maneiras.

O SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB

Você pode usar sua configuração de criptografia de tabela para criar um cliente do DynamoDB que criptografe e assine automaticamente itens do lado do cliente com suas solicitações do DynamoDB. `PutItem`

Você deve usar o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB para usar a criptografia pesquisável.

Para ver um exemplo de como usar o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB, consulte basic_get_put_example.rs no repositório -dynamodb em. aws-database-encryption-sdk GitHub

O **`DynamoDbItemEncryptor`** de nível inferior

O `DynamoDbItemEncryptor` de nível inferior criptografa, assina ou descriptografa e verifica diretamente os itens da tabela sem chamar o DynamoDB. Ele não faz solicitações `PutItem` ou `GetItem` para o DynamoDB. Por exemplo, é possível usar o `DynamoDbItemEncryptor` de nível inferior para descriptografar e verificar diretamente um item do DynamoDB que você já recuperou.

O nível inferior do `DynamoDbItemEncryptor` não oferece suporte à criptografia pesquisável.

Para ver um exemplo demonstrando como usar o nível inferior, consulte `DynamoDbItemEncryptor` item_encrypt_decrypt.rs no repositório -dynamodb em. aws-database-encryption-sdk GitHub

Ações de atributos no SDK de criptografia de banco de dados da AWS para DynamoDB

As ações de atributo determinam quais valores de atributos são criptografados e assinados, quais são somente assinados, quais são assinados e incluídos no contexto de criptografia e quais são ignorados.

Para especificar ações de atributos com o cliente Rust, defina manualmente as ações de atributos usando um modelo de objeto. Especifique suas ações de atributo criando um `HashMap` objeto no qual os pares nome-valor representam os nomes dos atributos e as ações especificadas.

Especifique `ENCRYPT_AND_SIGN` para criptografar e assinar um atributo.
Especifique `SIGN_ONLY` para assinar, mas não criptografar um atributo. Especifique `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT` para assinar um atributo e incluí-lo no contexto de criptografia. Não é possível criptografar um atributo sem também assiná-lo. Especifique `DO_NOTHING` para ignorar um atributo.

Os atributos de partição e classificação devem ser `SIGN_ONLY` ou`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`. Se você definir qualquer atributo como`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, os atributos de partição e classificação também deverão ser`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`.

> **Note**
>
> Depois de definir suas ações de atributo, você deverá definir quais atributos serão excluídos das assinaturas. Para facilitar a adição de novos atributos não assinados no futuro, recomendamos escolher um prefixo distinto (como "`:`") para identificar os atributos não assinados. Inclua esse prefixo no nome do atributo para todos os atributos marcados como `DO_NOTHING` ao definir o esquema e as ações de atributos do DynamoDB.

O modelo de objeto a seguir demonstra como especificar`ENCRYPT_AND_SIGN`, `SIGN_ONLY`, `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`, e `DO_NOTHING` atribuir ações com o cliente Rust. Este exemplo usa o prefixo "`:`" para identificar `DO_NOTHING` atributos.

```
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),
```

```
]);
```

Configuração de criptografia no SDK de criptografia de banco de dados da AWS para DynamoDB

Ao usar o AWS Database Encryption SDK, você deve definir explicitamente uma configuração de criptografia para sua tabela do DynamoDB. Os valores necessários em sua configuração de criptografia dependem se você definiu suas ações de atributo manualmente ou com uma classe de dados anotada.

O trecho a seguir define uma configuração de criptografia de tabela do DynamoDB usando o SDK de criptografia de AWS banco de dados de baixo nível para a API do DynamoDB e atributos não assinados permitidos definidos por um prefixo distinto.

```
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()?;
```

Nome da tabela lógica

Um nome de tabela lógica para sua tabela do DynamoDB.

O nome da tabela lógica é vinculado criptograficamente a todos os dados armazenados na tabela para simplificar as operações de restauração do DynamoDB. É altamente recomendável especificar o nome da tabela do DynamoDB como o nome lógico da tabela ao definir a configuração de criptografia pela primeira vez. Você deve sempre especificar o mesmo nome de tabela lógica. Para que a descriptografia seja bem-sucedida, o nome da tabela lógica deve corresponder ao nome especificado na criptografia. Se o nome da tabela do DynamoDB mudar

após a restauração da tabela do DynamoDB a partir de um backup, o nome da tabela lógica garantirá que a operação de descriptografia ainda reconheça a tabela.

Atributos não assinados permitidos

Os atributos marcados `DO_NOTHING` em suas ações de atributos.

Os atributos não assinados permitidos informam ao cliente quais atributos são excluídos das assinaturas. O cliente presume que todos os outros atributos estão incluídos na assinatura. Em seguida, ao descriptografar um registro, o cliente determina quais atributos ele precisa verificar e quais ignorar dos atributos não assinados permitidos que você especificou. Não é possível remover um atributo dos atributos não assinados permitidos.

É possível definir explicitamente os atributos não assinados permitidos criando uma matriz que lista todos os atributos `DO_NOTHING`. Também é possível especificar um prefixo distinto ao nomear os atributos `DO_NOTHING` e usar o prefixo para informar ao cliente quais atributos não estão assinados. É altamente recomendável especificar um prefixo distinto, pois isso simplifica o processo de adicionar um novo atributo `DO_NOTHING` no futuro. Para obter mais informações, consulte Atualizar seu modelo de dados.

Se você não especificar um prefixo para todos os atributos `DO_NOTHING`, poderá configurar uma matriz `allowedUnsignedAttributes` que liste explicitamente todos os atributos que o cliente deve esperar que não estejam assinados ao encontrá-los na descriptografia. Você só deve definir explicitamente seus atributos não assinados permitidos se for absolutamente necessário.

Configuração de pesquisa (opcional)

O `SearchConfig` define a versão do beacon.

O `SearchConfig` deve ser especificado para usar criptografia pesquisável ou beacons assinados.

Suíte de algoritmos (opcional)

O `algorithmSuiteId` define qual conjunto de algoritmos o SDK de criptografia de banco de dados da AWS usará.

A menos que você especifique explicitamente um conjunto alternativo de algoritmos, o SDK do AWS Database Encryption usa o conjunto de algoritmos padrão. O conjunto de algoritmos padrão usa o algoritmo AES-GCM com derivação de chaves, assinaturas digitais e comprometimento de chaves. Embora o conjunto de algoritmos padrão provavelmente seja adequado para a maioria dos aplicativos, é possível escolher um conjunto alternativo de algoritmos. Por exemplo, alguns

modelos de confiança seriam satisfeitos com um pacote de algoritmos sem assinaturas digitais. Para obter informações sobre os conjuntos de algoritmos compatíveis com o SDK do AWS Database Encryption, consulteSuítes de algoritmos compatíveis no SDK AWS de criptografia de banco de dados.

Para selecionar o conjunto de algoritmos AES-GCM sem assinaturas digitais ECDSA, inclua o seguinte trecho em sua configuração de criptografia de tabela.

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

Atualização de itens com o SDK AWS de criptografia de banco de dados

O SDK AWS de criptografia de banco de dados não oferece suporte a ddb: UpdateItem para itens que incluem atributos criptografados ou assinados. Para atualizar um atributo criptografado ou assinado, você deve usar ddb: PutItem. Se algum item existir em uma tabela específica com a mesma chave primária de um item existente na consulta `PutItem`, o novo item substituirá completamente o item já existente.

Cliente legado de criptografia do DynamoDB

Em 9 de junho de 2023, nossa biblioteca de criptografia do lado do cliente foi renomeada para AWS Database Encryption SDK. O SDK do AWS Database Encryption continua oferecendo suporte às versões antigas do DynamoDB Encryption Client. Para obter mais informações sobre as diferentes partes da biblioteca de criptografia do lado do cliente que foram alteradas com a renomeação, consulte Renomeação do Amazon DynamoDB Encryption Client.

Para migrar para a versão mais recente da biblioteca Java de criptografia do lado do cliente para o DynamoDB, consulte Migrar para a versão 3.x.

Tópicos

- AWS Suporte à versão SDK de criptografia de banco de dados para DynamoDB
- Como o DynamoDB Encryption Client funciona
- Conceitos do Amazon DynamoDB Encryption Client
- Provedor de materiais de criptografia
- Linguagens de programação disponíveis do Amazon DynamoDB Encryption Client

- Alterar seu modelo de dados
- Solução de problemas em seu aplicativo DynamoDB Encryption Client

AWS Suporte à versão SDK de criptografia de banco de dados para DynamoDB

Os tópicos a seguir do capítulo Legao fornecem informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e das versões 1.x—3x do DynamoDB Encryption Client para Python.

A tabela a seguir lista os idiomas e as versões que oferecem suporte à criptografia do lado do cliente no Amazon DynamoDB.

Linguagem de programação	Versão	Fase do ciclo de vida da versão principal do SDK
Java	Versões 1.x	End-of-Support fase, em vigor em julho de 2022
Java	Versões 2.x	Disponibilidade geral (GA)
Java	Versão 3.x	Disponibilidade geral (GA)
Python	Versões 1.x	End-of-Support fase, em vigor em julho de 2022
Python	Versões 2.x	End-of-Support fase, em vigor em julho de 2022
Python	Versões 3.x	Disponibilidade geral (GA)

Como o DynamoDB Encryption Client funciona

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—

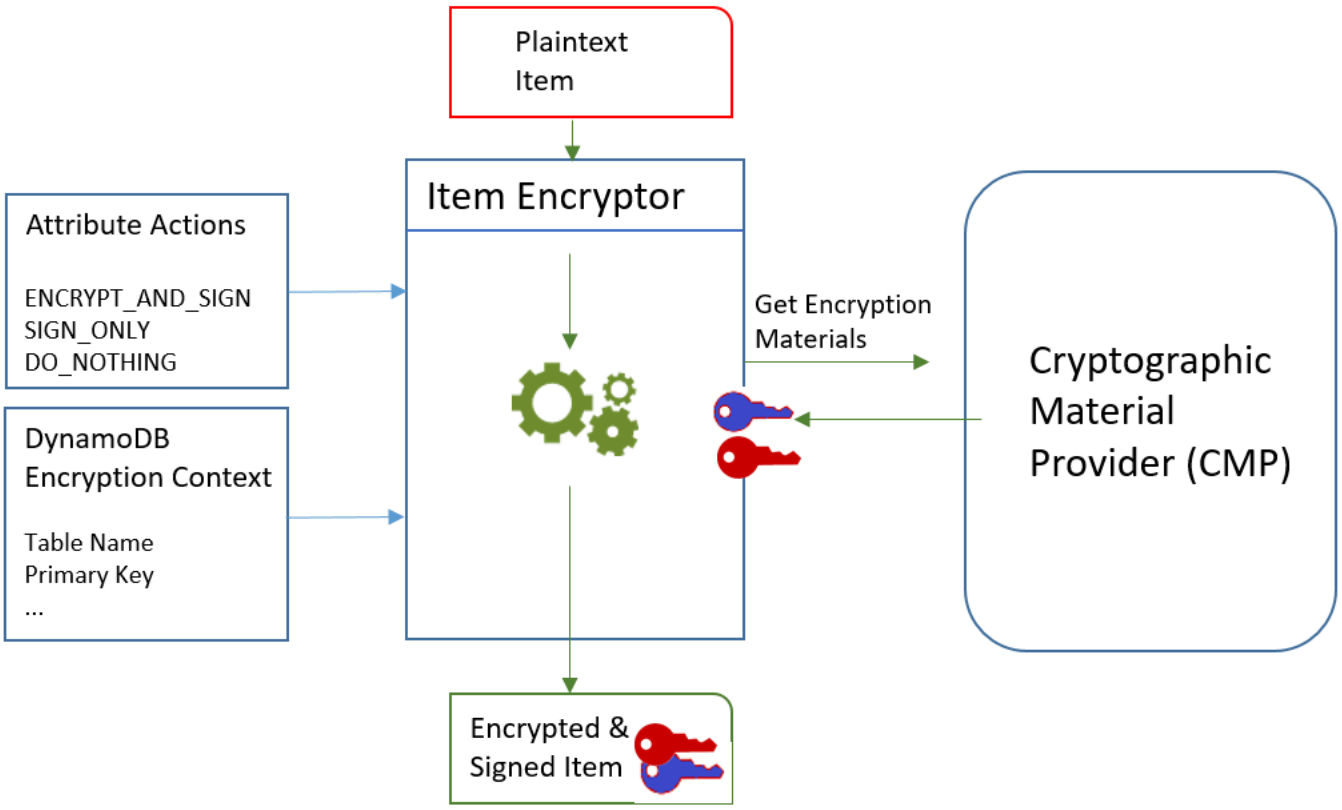
2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

O DynamoDB Encryption Client foi criado especificamente para proteger os dados que você armazena no DynamoDB. As bibliotecas incluem implementações seguras que você pode estender ou usar inalteradas. Além disso, a maioria dos elementos é representada por elementos abstratos para que você possa criar e usar componentes personalizados compatíveis.

Criptografar e assinar itens de tabela

No núcleo do DynamoDB Encryption Client está um item de criptografador que criptografa, autentica, verifica e descriptografa os itens da tabela. Ele obtém informações sobre os itens da tabela e instruções sobre quais itens devem ser criptografados e assinados. Ele obtém os materiais de criptografia e as instruções sobre como usá-los de um provedor de materiais de criptografia que você seleciona e configura.

O diagrama a seguir mostra uma visão de alto nível desse processo.



Para criptografar e assinar um item de tabela, o DynamoDB Encryption Client precisa:

- Informações sobre a tabela. Ele obtém informações sobre a tabela de um contexto de criptografia do DynamoDB que você fornece. Alguns auxiliares obtêm as informações necessárias do DynamoDB e criam o contexto de criptografia do DynamoDB para você.

  > **Note**
  >
  > O contexto de criptografia do DynamoDB no DynamoDB Encryption Client não está relacionado ao contexto de criptografia em () e o. AWS Key Management Service AWS KMS AWS Encryption SDK

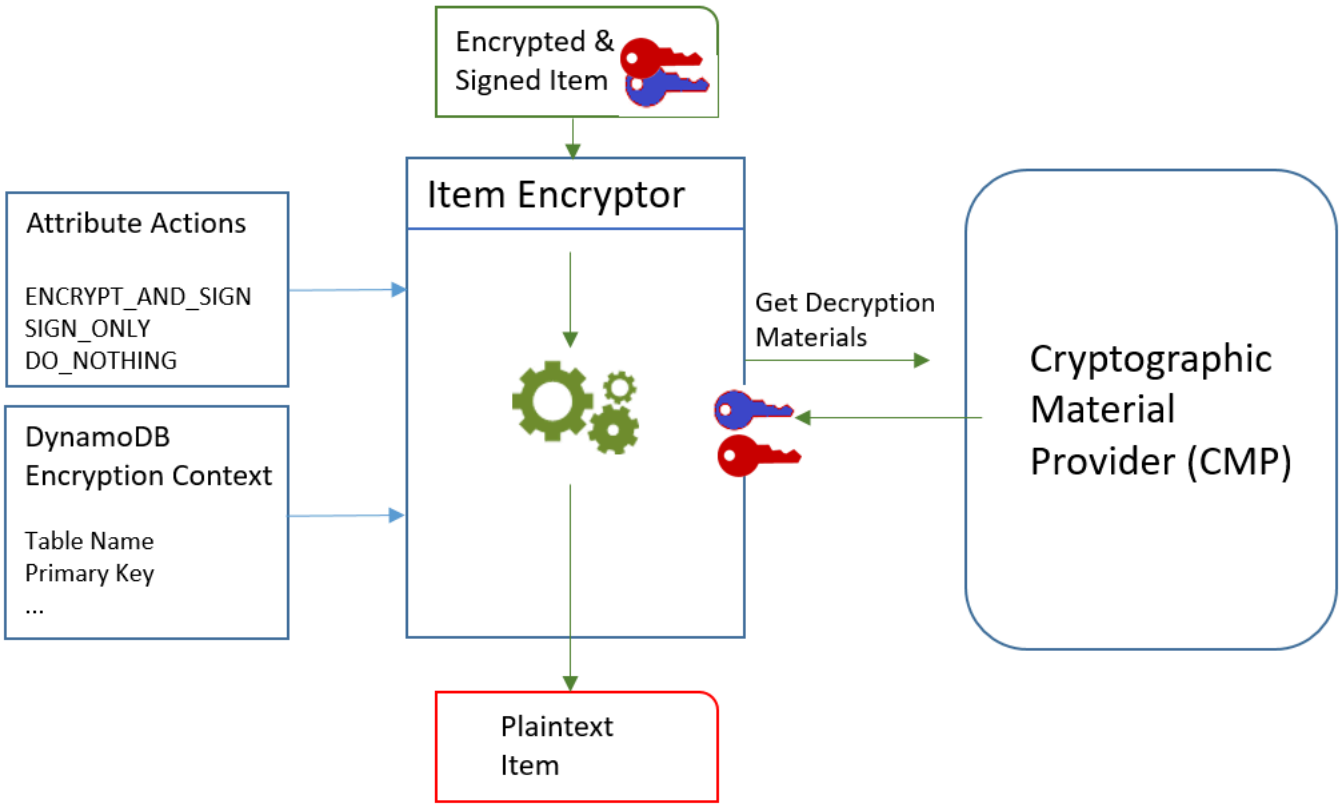
- Quais atributos devem ser criptografados e assinados. Ele obtém essas informações das ações de atributos que você fornece.
- Materiais de criptografia, incluindo chaves de criptografia e de assinatura. Ele os obtém de um provedor de materiais de criptografia (CMP) que você seleciona e configura.
- Instruções para criptografar e assinar o item. O CMP adiciona instruções para usar os materiais de criptografia, incluindo algoritmos de criptografia e assinatura, à descrição real do material.

O criptografador de itens usa todos esses elementos para criptografar e assinar o item. O criptografador de itens também adiciona dois atributos ao item: um atributo de descrição do material que contém as instruções de criptografia e assinatura (a descrição real do material) e um atributo que contém a assinatura. Você pode interagir com o criptografador do item diretamente ou usar os recursos auxiliares que interagem com o criptografador do item para que você implemente um comportamento padrão seguro.

O resultado é um item do DynamoDB que contém dados criptografados e assinados.

Verificar e descriptografar itens de tabela

Esses componentes também funcionam juntos para verificar e descriptografar o item, conforme mostrado no diagrama a seguir.



Para verificar e descriptografar um item, o DynamoDB Encryption Client precisa dos mesmos componentes, componentes com a mesma configuração ou criados especialmente para a descriptografia de itens da seguinte maneira:

- Informações sobre a tabela do contexto de criptografia do .
- Quais atributos verificar e descriptografar. Ele os obtém das ações de atributos.
- Materiais de descriptografia, incluindo chaves de verificação e de descriptografia, do provedor de materiais de criptografia (CMP) que você seleciona e configura.

  O item criptografado não inclui registros do CMP que foi usado para criptografá-lo. Você deve fornecer o mesmo CMP, um CMP com a mesma configuração ou que foi criado para descriptografar itens.
- Informações sobre como o item foi criptografado e assinado, incluindo os algoritmos de criptografia e assinatura. O cliente os obtém do atributo de descrição do material no item.

O criptografador de itens usa todos esses elementos para verificar e descriptografar o item. Ele também remove os atributos de assinatura e descrição do material. O resultado é um item do DynamoDB em formato de texto simples.

Conceitos do Amazon DynamoDB Encryption Client

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Este tópico explica a terminologia e os conceitos usados no Amazon DynamoDB Encryption Client.

Para saber como os componentes do DynamoDB Encryption Client interagem, consulte Como o DynamoDB Encryption Client funciona.

Tópicos

- Provedor de materiais de criptografia (CMP)
- Criptografadores de itens
- Ações de atributos
- Descrição do material
- Contexto de criptografia do DynamoDB
- Armazenamento de provedores

Provedor de materiais de criptografia (CMP)

Ao implementar o DynamoDB Encryption Client, uma das suas primeiras tarefas é selecionar um provedor de materiais de criptografia (CMP) (também conhecido como provedor de materiais criptográficos). Sua escolha determina muito do restante da implementação.

O provedor de materiais de criptografia (CMP) coleta, monta e retorna os materiais criptográficos que o criptografador de itens usa para criptografar e assinar os itens de sua tabela. O CMP determina

os algoritmos de criptografia a serem usados e como gerar e proteger a criptografia e as chaves de assinatura.

O CMP interage com o criptografador do item. O criptografador do item solicita materiais de criptografia ou de descriptografia do CMP, e o CMP os retorna ao criptografador do item. Então, o criptografador do item usa os materiais de criptografia para criptografar e assinar, ou verificar e descriptografar, o item.

Você especifica o CMP ao configurar o cliente. Você pode criar uma CMP personalizada compatível ou usar uma das várias CMPs da biblioteca. A maioria CMPs está disponível para várias linguagens de programação.

Criptografadores de itens

O criptografador do itens é um componente de nível inferior que executa operações de criptografia para o DynamoDB Encryption Client. Ele solicita materiais de criptografia de um provedor de materiais de criptografia (CMP) e usa os materiais que o CMP retorna para criptografar e assinar, ou verificar e descriptografar, o item da tabela.

É possível interagir com o criptografador do item diretamente ou usar os auxiliares fornecidos pela biblioteca. Por exemplo, o DynamoDB Encryption Client para Java inclui uma classe auxiliar `AttributeEncryptor` que é possível usar com o `DynamoDBMapper`, em vez de interagir diretamente com o criptografador de itens `DynamoDBEncryptor`. A biblioteca Python inclui as classes auxiliares `EncryptedTable`, `EncryptedClient` e `EncryptedResource` que interagem com o criptografador do item para você.

Ações de atributos

As Ações de atributos informam ao criptografador de itens quais ações executar em cada atributo de item.

Os valores das ações de atributo podem ser um destes:

- Criptografar e assinar – Criptografa o valor do atributo. Incluir o atributo (nome e valor) na assinatura do item.
- Apenas assinar – Inclui o atributo na assinatura do item.
- Não fazer nada – Não criptografa nem assina o atributo.

Para qualquer atributo que possa armazenar dados confidenciais, use Criptografar e assinar. Para atributos de chave primária (chave de partição e chave de classificação), use Apenas assinar. O atributo de descrição do material e o atributo de assinatura não são assinados nem criptografados. Não é necessário especificar ações para esses atributos.

Escolha suas ações de atributos com cuidado. Em caso de dúvida, use Criptografar e assinar. Depois de usar o DynamoDB Encryption Client para proteger seus itens de tabela, não será possível alterar a ação de um atributo sem arriscar um erro de validação de assinatura. Para obter detalhes, consulte Alterar seu modelo de dados.

> ⚠ Warning
>
> Não criptografe os atributos da chave primária. Eles devem permanecer em texto simples para que o DynamoDB possa encontrar o item sem executar uma varredura completa da tabela.

Se o contexto de criptografia do DynamoDB identificar os atributos de chave primária, o cliente gerará um erro se você tentar criptografá-los.

A técnica usada para especificar as ações de atributo é diferente para cada linguagem de programação. Ela também pode ser específica das classes auxiliares que você usa.

Para ver detalhes, consulte a documentação da sua linguagem de programação.

- Python
- Java

Descrição do material

A descrição do material para um item de tabela criptografado consiste em informações, como algoritmos de criptografia, sobre como o item de tabela é criptografado e assinado. O provedor de materiais de criptografia (CMP) registra a descrição do material à medida que monta os materiais para criptografia e assinatura. Depois, quando precisar montar materiais de criptografia para verificar e descriptografar o item, ele usará a descrição do material como guia.

No DynamoDB Encryption Client, a descrição do material refere-se a três elementos relacionados:

Descrição do material solicitado

Alguns fornecedores de materiais criptográficos (CMPs) permitem que você especifique opções avançadas, como um algoritmo de criptografia. Para indicar suas opções, adicione pares de nome-valor à propriedade de descrição do material do contexto de criptografia do DynamoDB na solicitação para criptografar um item da tabela. Esse elemento é conhecido como a descrição do material solicitado. Os valores válidos na descrição solicitada do material são definidos pelo CMP escolhido.

> **Note**
>
> Como a descrição do material pode substituir valores padrão seguros, recomendamos que você omita a descrição solicitada do material, a menos que tenha um bom motivo para usá-la.

Descrição real do material

A descrição do material que os fornecedores de materiais criptográficos (CMPs) retornam é conhecida como a descrição real do material. Ela descreve os valores reais que o CMP usou quando montou os materiais de criptografia. Ela consiste na descrição solicitada do material, se houver, com adições e alterações.

Atributo de descrição do material

O cliente salva a descrição real do material no atributo de descrição do material do item criptografado. O nome do atributo de descrição do material é `amzn-ddb-map-desc`, e seu valor é a descrição real do material. O cliente usa os valores do atributo de descrição do material para verificar e descriptografar o item.

Contexto de criptografia do DynamoDB

O contexto de criptografia do DynamoDB fornece informações sobre a tabela e o item ao provedor de materiais de criptografia (CMP). Em implementações avançadas, o contexto de criptografia do DynamoDB pode incluir uma descrição do material solicitado.

Quando você criptografa itens de tabela, o contexto de criptografia do DynamoDB é vinculado criptograficamente aos valores dos atributos criptografados. Ao descriptografar, se o contexto de criptografia do DynamoDB não for correspondência exata de maiúsculas e minúsculas do contexto de criptografia do DynamoDB usado para criptografar, a operação de descriptografia falhará. Se

você interagir com o criptografador de itens diretamente, forneça um contexto de criptografia do DynamoDB ao chamar um método de criptografia ou de descriptografia. A maioria das classes auxiliares cria o contexto de criptografia do DynamoDB para você.

> **Note**
>
> O contexto de criptografia do DynamoDB no DynamoDB Encryption Client não está relacionado ao contexto de criptografia em () e o. AWS Key Management Service AWS KMS AWS Encryption SDK

O contexto de criptografia do DynamoDB pode incluir os campos a seguir. Todos os campos e valores são opcionais.

- Nome da tabela
- Nome da chave de partição
- Nome da chave de classificação
- Pares de nome-valor do atributo
- Descrição do material solicitado

Armazenamento de provedores

Uma loja de fornecedores é um componente que retorna fornecedores de materiais criptográficos (CMPs). A loja do provedor pode criá-los CMPs ou obtê-los de outra fonte, como outra loja do provedor. O repositório do provedor salva as versões do CMPs que ele cria em um armazenamento persistente, no qual cada CMP armazenado é identificado pelo nome do material do solicitante e pelo número da versão.

O provedor mais recente no DynamoDB Encryption Client os obtém CMPs de uma loja de provedores, mas você pode usar a loja de provedores para CMPs fornecer qualquer componente. Cada provedor mais recente está associado a uma loja de provedores, mas uma loja de provedores pode fornecer CMPs a vários solicitantes em vários hosts.

A loja do provedor cria novas versões CMPs sob demanda e retorna versões novas e existentes. Ele também retorna o número da versão mais recente de um determinado nome de material. Assim, o solicitante sabe quando o armazenamento do provedor tem uma nova versão do CMP que ele pode solicitar.

O DynamoDB Encryption Client inclui MetaStoreum, que é um repositório de provedores que cria CMPs Wrapped com chaves armazenadas no DynamoDB e criptografadas usando um DynamoDB Encryption Client interno.

Saiba mais:

- Armazenamento de provedores: Java, Python
- MetaStore: Java, Python

Provedor de materiais de criptografia

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Uma das decisões mais importantes que você precisa tomar ao usar o DynamoDB Encryption Client é selecionar um provedor de materiais de criptografia (CMP). O CMP monta e retorna materiais de criptografia ao criptografador do item. Ele também determina como as chaves de criptografia e assinatura são geradas, se os novos materiais de chaves são gerados para cada item ou reutilizados e os algoritmos de criptografia e assinatura que são usados.

Você pode escolher um CMP das implementações fornecidas nas bibliotecas do DynamoDB Encryption Client ou criar um CMP compatível personalizado. Sua escolha de CMP também pode ter como base a linguagem de programação usada.

Este tópico descreve os mais comuns CMPs e oferece alguns conselhos para ajudá-lo a escolher o melhor para seu aplicativo.

Provedor direto de materiais do KMS

O provedor direto de materiais do KMS protege os itens da sua tabela sob uma AWS KMS key que nunca deixa o AWS Key Management Service (AWS KMS) sem criptografia. Seu aplicativo não precisa gerar ou gerenciar nenhum material de criptografia. Como ele usa o AWS KMS key

para gerar chaves exclusivas de criptografia e assinatura para cada item, esse provedor liga AWS KMS sempre que criptografa ou descriptografa um item.

Se você usa AWS KMS e uma AWS KMS chamada por transação é prática para seu aplicativo, esse provedor é uma boa escolha.

Para obter detalhes, consulte Provedor direto de materiais do KMS.

Provedor encapsulado de materiais (CMP encapsulado)

O provedor encapsulado de materiais (CMP encapsulado) permite gerar e gerenciar chaves encapsuladas e de assinatura fora do DynamoDB Encryption Client.

O CMP encapsulado gera uma chave exclusiva de criptografia para cada item. E, então, ele usa as chaves encapsuladas (ou desencapsuladas) e de assinatura que você forneceu. Desse modo, você pode determinar como as chaves encapsuladas e de assinatura serão geradas e se elas serão exclusivas para cada item ou reutilizadas. O Wrapped CMP é uma alternativa segura ao Direct KMS Provider para aplicativos que não usam AWS KMS e podem gerenciar com segurança materiais criptográficos.

Para obter detalhes, consulte Provedor encapsulado de materiais.

Provedor mais recente

O Provedor mais recente é um provedor de materiais de criptografia (CMP) que foi projetado para trabalhar com um armazenamento de provedores. Ele é CMPs obtido da loja do fornecedor e obtém os materiais criptográficos que retorna do CMPs. O provedor mais recente normalmente usa cada CMP para atender a várias solicitações de materiais de criptografia, mas você pode usar os recursos do armazenamento de provedor para gerenciar a frequência com a qual os materiais são reutilizados, determinar a frequência de rotação do CMP e até mesmo alterar o tipo de CMP usado sem alterar o provedor mais recente.

Você pode usar o provedor mais recente com qualquer armazenamento compatível de provedor. O DynamoDB Encryption Client inclui MetaStore um, que é um provedor de armazenamento que retorna Wrapped. CMPs

O provedor mais recente é uma boa opção para aplicativos que precisam minimizar as chamadas para sua origem de criptografia e para aplicativos que podem reutilizar alguns materiais de criptografia sem violar os requisitos de segurança. Por exemplo, ele permite que você proteja seus materiais criptográficos sob um AWS KMS keyin AWS Key Management Service(AWS KMS) sem chamar AWS KMS toda vez que criptografar ou descriptografar um item.

Para obter detalhes, consulte Provedor mais recente.

Provedor estático de materiais

O Static Materials Provider foi projetado para testes, proof-of-concept demonstrações e compatibilidade antiga. Ele não gera material exclusivo de criptografia para cada item. No entanto, ele retorna as mesmas chaves de criptografia e assinatura que você oferece, e essas chaves são usadas diretamente para criptografar, descriptografar e assinar os itens da sua tabela.

> **Note**
>
> O Provedor estático assimétrico na biblioteca Java não é um provedor estático. Ele apenas oferece construtores alternativos para o CMP encapsulado. Ele é seguro para fins de produção, mas você deve usar o CMP encapsulado diretamente sempre que possível.

Tópicos

- Provedor direto de materiais do KMS
- Provedor encapsulado de materiais
- Provedor mais recente
- Provedor estático de materiais

Provedor direto de materiais do KMS

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

O Provedor direto de materiais do KMS (Direct KMS Provider) protege os itens da sua tabela sob um AWS KMS key que nunca deixa AWS Key Management Service (AWS KMS) sem criptografia. O provedor de materiais de criptografia retorna uma chave de criptografia exclusiva e uma chave

de assinatura para cada item da tabela. Para fazer isso, ele chama AWS KMS toda vez que você criptografa ou descriptografa um item.

Se você estiver processando itens do DynamoDB em alta frequência e em grande escala, poderá exceder os limites, causando atrasos AWS KMS requests-per-secondno processamento. Caso seja necessário ultrapassar esses limites, visite o Centro do AWS Support e crie um caso. Também é possível considerar usar um provedor de materiais criptográficos com reutilização limitada de chaves, como o provedor mais recente.

Para usar o Direct KMS Provider, o chamador deve ter pelo menos uma AWS KMS key permissão para ligar para as operações GenerateDataKeye Decrypt no. Conta da AWS AWS KMS key O AWS KMS key deve ser uma chave de criptografia simétrica; o DynamoDB Encryption Client não oferece suporte à criptografia assimétrica. Se você estiver usando uma tabela global do DynamoDB, talvez queira especificar uma chave multirregional do AWS KMS. Para obter detalhes, consulte Como usar.

> **Note**
>
> Quando você usa o Direct KMS Provider, os nomes e valores de seus atributos de chave primária aparecem em texto simples no contexto de AWS KMS criptografia e nos AWS CloudTrail registros de operações relacionadas. AWS KMS No entanto, o DynamoDB Encryption Client nunca expõe o texto simples de nenhum valor de atributo criptografado.

O Direct KMS Provider é um dos vários provedores de materiais criptográficos (CMPs) suportados pelo DynamoDB Encryption Client. Para obter informações sobre o outro CMPs, consulteProvedor de materiais de criptografia.

Para ver um código de exemplo, consulte:

- Java: AwsKmsEncryptedItem
- Python:, aws-kms-encrypted-tableaws-kms-encrypted-item

Tópicos

- Como usar
- Como funciona

Como usar

Para criar um Direct KMS Provider, use o parâmetro ID da chave para especificar uma chave do KMS de criptografia simétrica em sua conta. O valor do parâmetro do ID da chave pode ser o ID, o ARN da chave ou um nome de alias ou um ARN de alias do AWS KMS key. Para obter detalhes sobre os identificadores de chave, consulte Identificadores de chave no Guia do desenvolvedor do AWS Key Management Service .

O Direct KMS Provider exige uma chave do KMS de criptografia simétrica. Não é possível usar uma chave do KMS assimétrica. É possível usar uma chave do KMS multirregional, chaves do KMS com material de chave importado ou uma chave do KMS em um armazenamento de chaves personalizado. Você deve ter as permissões kms: GenerateDataKey e kms:decrypt na chave KMS. Dessa forma, você deve usar uma chave gerenciada pelo cliente, não uma chave KMS AWS gerenciada ou de AWS propriedade.

O DynamoDB Encryption Client for Python determina a região para AWS KMS chamadas da região no valor do parâmetro de ID chave, se ele incluir um. Caso contrário, ele usa a Região no AWS KMS cliente, se você especificar uma, ou a Região que você configura no AWS SDK para Python (Boto3). Para obter informações sobre a seleção de regiões em Python, consulte Configuração na Referência da API AWS SDK for Python (Boto3).

O DynamoDB Encryption Client for Java determina a região para AWS KMS chamadas da região no cliente, se AWS KMS o cliente que você especificar incluir uma região. Caso contrário, ela usa a região que você configura em AWS SDK para Java. Para obter informações sobre a seleção de regiões no AWS SDK para Java, consulte a Região da AWS seleção no Guia do AWS SDK para Java desenvolvedor.

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

O exemplo a seguir usa o ARN de chave para especificar o AWS KMS key. Se seu identificador de chave não incluir um Região da AWS, o DynamoDB Encryption Client obtém a região da sessão de Botocore configurada, se houver, ou dos padrões do Boto.

```
# 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)
```

Se você estiver usando tabelas globais do Amazon DynamoDB, recomendamos que você criptografe seus dados com uma chave multirregional. AWS KMS As chaves multirregionais são AWS KMS keys diferentes e Regiões da AWS podem ser usadas de forma intercambiável porque têm o mesmo ID de chave e material de chave. Para obter mais detalhes, consulte Usar chaves de várias regiões, no Guia do desenvolvedor do AWS Key Management Service .

> **Note**
>
> Se você estiver usando a versão 2017.11.29 de tabelas globais, deverá definir ações de atributos para que os campos de replicação reservados não sejam criptografados ou assinados. Para obter detalhes, consulte Problemas com tabelas globais de versões mais antigas.

Para usar uma chave multirregional com o DynamoDB Encryption Client, crie uma chave multirregional e replique-a nas regiões em que seu aplicativo é executado. Em seguida, configure o Direct KMS Provider para usar a chave multirregional na região em que o DynamoDB Encryption Client faz chamadas para o AWS KMS.

O exemplo a seguir configura o DynamoDB Encryption Client para criptografar dados na região Leste dos EUA (Norte da Virgínia) (us-east-1) e descriptografá-los na região Oeste dos EUA (Oregon) (us-west-2).

Java

Neste exemplo, o DynamoDB Encryption Client obtém a região para fazer AWS KMS chamadas da região no cliente. AWS KMS O valor `keyArn` identifica uma chave de várias regiões na mesma região.

```
// 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

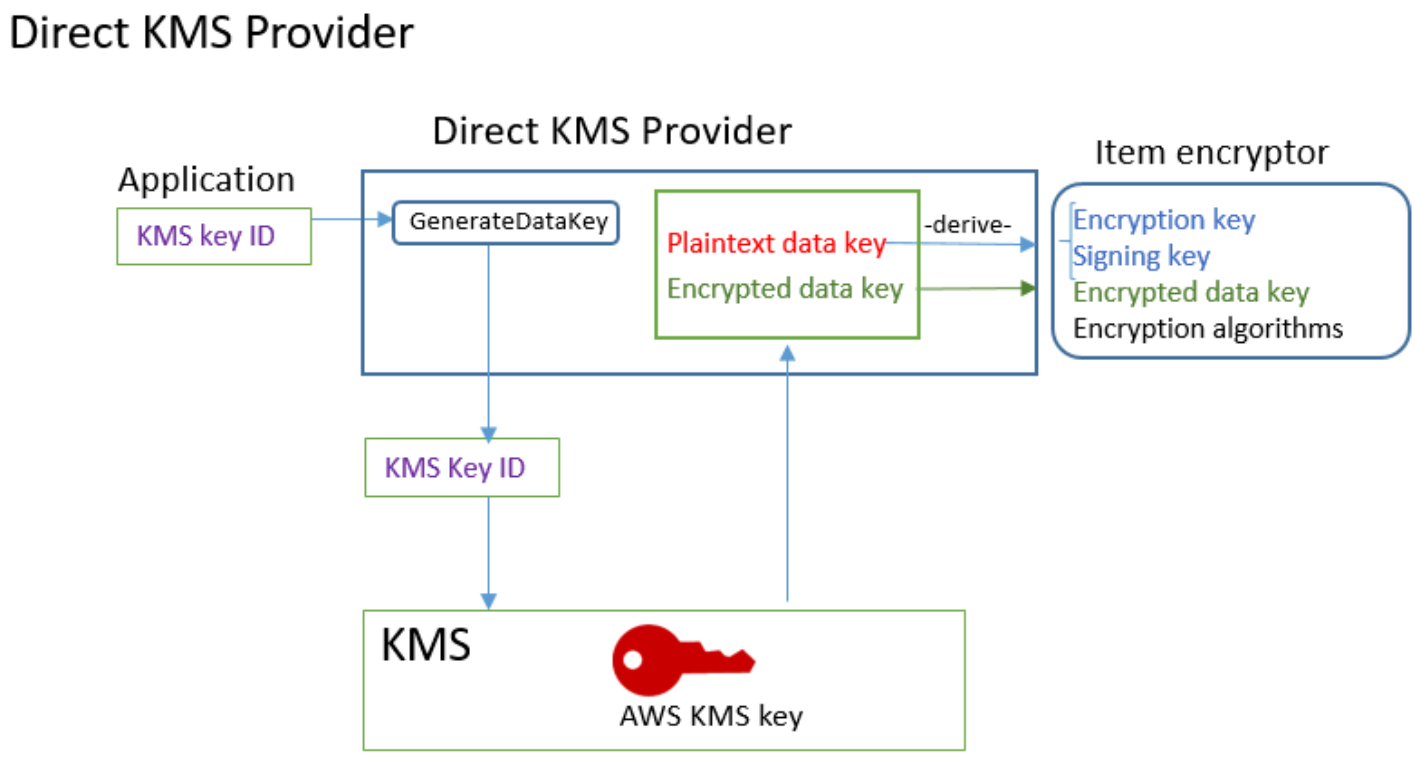
Neste exemplo, o DynamoDB Encryption Client obtém a região para fazer AWS KMS chamadas da região no ARN da chave.

```
# 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)
```

Como funciona

O Direct KMS Provider retorna as chaves de criptografia e assinatura que são protegidas por um AWS KMS key especificado, conforme exibido no diagrama a seguir.



- Para gerar materiais de criptografia, o Direct KMS Provider solicita AWS KMS a geração de uma chave de dados exclusiva para cada item usando uma AWS KMS key que você especifica. Ele deriva as chaves de criptografia e de assinatura do item da cópia de texto simples da chave de dados e retorna essas chaves junto com a chave de dados criptografada, que é armazenada no atributo de descrição do material do item.

  O criptografador do item usa as chaves de criptografia e assinatura e as remove da memória o mais rápido possível. Somente a cópia criptografada da chave de dados, da qual eles foram originados, é salva no item criptografado.

- Para gerar materiais de decodificação, o Direct KMS Provider solicita a decodificação da chave AWS KMS de dados criptografada. Então, ele obtém chaves de verificação e assinatura provenientes da chave de dados em texto simples e as retorna para o criptografador de item.

  O criptografador de item verifica o item e, se a verificação for bem-sucedida, ele descriptografa os valores criptografados. Então, ele remove as chaves da memória o mais rápido possível.

Obter materiais de criptografia

Esta seção descreve detalhadamente as entradas, as saídas e o processamento do Direct KMS Provider quando ele recebe uma solicitação de materiais de criptografia do criptografador de item.

Entrada (do aplicativo)

- O ID da chave de um AWS KMS key.

Entrada (do criptografador de itens)

- Contexto de criptografia do DynamoDB

Saída (para o criptografador de itens)

- Chave de criptografia (texto simples)
- Chave de assinatura
- Na descrição do material atual: esses valores são salvos no atributo da descrição do material que o cliente adiciona ao item.
  - amzn-ddb-env-key: chave de dados codificada em Base64 criptografada pelo AWS KMS key
  - amzn-ddb-env-alg: Algoritmo de criptografia, por padrão AES/256
  - amzn-ddb-sig-alg: algoritmo de assinatura, por padrão, Hmac /256 SHA256
  - amzn-ddb-wrap-alg: kms

Processamento

1. O Direct KMS Provider envia AWS KMS uma solicitação para usar o especificado AWS KMS key para gerar uma chave de dados exclusiva para o item. A operação retorna uma chave de texto simples e uma cópia criptografada de acordo com a AWS KMS key. Essa operação também é conhecida como o material de chave inicial.

   A solicitação inclui os seguintes valores em texto simples no contexto de criptografia do AWS KMS. Esses valores não confidenciais estão vinculados de maneira criptográfica ao objeto criptografado, assim, o mesmo contexto de criptografia será necessário na descriptografia. Você pode usar esses valores para identificar a chamada AWS KMS nos AWS CloudTrail registros.

   - amzn-ddb-env-alg — Algoritmo de criptografia, por padrão AES/256

- amzn-ddb-sig-alg — Algoritmo de assinatura, por padrão Hmac /256 SHA256
- (Opcional) aws-kms-table — *table name*
- (Opcional) *partition key name* — *partition key value* (os valores binários são codificados em Base64)
- (Opcional) *sort key name* — *sort key value* (os valores binários são codificados em Base64)

O Direct KMS Provider obtém os valores para o contexto de AWS KMS criptografia do contexto de criptografia do DynamoDB para o item. Se o contexto de criptografia do DynamoDB não incluir um valor, como o nome da tabela, esse par nome-valor será omitido do contexto de criptografia. AWS KMS

2. O Direct KMS Provider obtém uma chave de criptografia simétrica e uma chave de assinatura a partir da chave de dados. Por padrão, ele usa o Secure Hash Algorithm (SHA) 256 e a função de derivação de chave RFC5869 baseada em HMAC para derivar uma chave de criptografia simétrica AES de 256 bits e uma chave de assinatura HMAC-SHA-256 de 256 bits.
3. O Direct KMS Provider retorna a saída para o criptografador do item.
4. O criptografador do item usa a chave de criptografia para criptografar os atributos especificados e a chave de assinatura para assiná-los, usando os algoritmos especificados na real descrição do material. Ele remove as chaves de texto simples da memória o mais rápido possível.

Obter materiais de descriptografia

Esta seção descreve detalhadamente as entradas, as saídas e o processamento do Direct KMS Provider quando ele recebe uma solicitação de materiais de descriptografia do criptografador de itens.

Entrada (do aplicativo)

- O ID da chave de um AWS KMS key.

  O valor do ID da chave pode ser o ID, o ARN da chave ou um nome de alias ou um ARN de alias do AWS KMS key. Todos os valores que não forem incluídos no ID, como a região, deverão estar disponíveis no perfil nomeado da AWS. O ARN da chave fornece todos os valores necessários para o AWS KMS .

Entrada (do criptografador de itens)

- Uma cópia do contexto de criptografia do DynamoDB com o conteúdo do atributo de descrição do material.

Saída (para o criptografador de itens)

- Chave de criptografia (texto simples)
- Chave de assinatura

Processamento

1. O Direct KMS Provider obtém a chave de dados criptografada a partir do atributo de descrição do material no item criptografado.
2. Ele solicita AWS KMS o uso do especificado AWS KMS key para descriptografar a chave de dados criptografada. A operação retorna uma chave de texto simples.

   Essa solicitação deve usar o mesmo contexto de criptografia do AWS KMS que foi usado para gerar e criptografar a chave de dados.

   - aws-kms-table – *`table name`*
   - *`partition key name`*— *`partition key value`* (os valores binários são codificados em Base64)
   - (Opcional) *`sort key name`* — *`sort key value`* (os valores binários são codificados em Base64)
   - amzn-ddb-env-alg — Algoritmo de criptografia, por padrão AES/256
   - amzn-ddb-sig-alg — Algoritmo de assinatura, por padrão Hmac /256 SHA256
3. O Direct KMS Provider usa o Secure Hash Algorithm (SHA) 256 e a função de derivação de chave RFC5869 baseada em HMAC para derivar uma chave de criptografia simétrica AES de 256 bits e uma chave de assinatura HMAC-SHA-256 de 256 bits da chave de dados.
4. O Direct KMS Provider retorna a saída para o criptografador do item.
5. O criptografador do item usa a chave de assinatura para verificar o item. Se ele for bem-sucedido, usará a chave de criptografia simétrica para descriptografar os valores de atributos criptografados. Essas operações usam os algoritmos de criptografia e assinatura especificados na real descrição material. O criptografador do item remove as chaves de texto simples da memória o mais rápido possível.

Provedor encapsulado de materiais

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

O provedor empacotado de materiais (CMP empacotado) permite usar chaves encapsuladas e de assinatura a partir de qualquer origem com o DynamoDB Encryption Client. O Wrapped CMP não depende de nenhum AWS serviço. No entanto, você deve gerar e gerenciar suas chaves de empacotamento e assinatura fora do cliente, incluindo o fornecimento das chaves corretas para verificar e descriptografar o item.

O CMP encapsulado gera uma chave exclusiva de criptografia para cada item. Ele encapsula a chave de criptografia do item com a chave de empacotamento que você fornece, e salva a chave de criptografia de item encapsulada no atributo de descrição do material do item. Como fornece as chaves de empacotamento e assinatura, você determina como as chaves de empacotamento e assinatura serão geradas e se elas serão exclusivas de cada item ou reutilizadas.

O CMP encapsulado é uma implementação segura e uma boa opção para aplicativos que podem gerenciar materiais de criptografia.

O Wrapped CMP é um dos vários fornecedores de materiais criptográficos (CMPs) compatíveis com o DynamoDB Encryption Client. Para obter informações sobre o outro CMPs, consulteProvedor de materiais de criptografia.

Para ver um código de exemplo, consulte:

- Java: AsymmetricEncryptedItem
- Python:, wrapped-rsa-encrypted-tablewrapped-symmetric-encrypted-table

Tópicos

- Como usar

- Como funciona

Como usar

Para criar um CMP empacotado, especifique uma chave de empacotamento (necessária para a criptografia), uma chave de desempacotamento (necessária para a descriptografia) e uma chave de assinatura. É necessário fornecer chaves ao criptografar e descriptografar itens.

As chaves de empacotamento, desempacotamento e assinatura podem ser chaves simétricas ou pares de chaves assimétricos.

Java

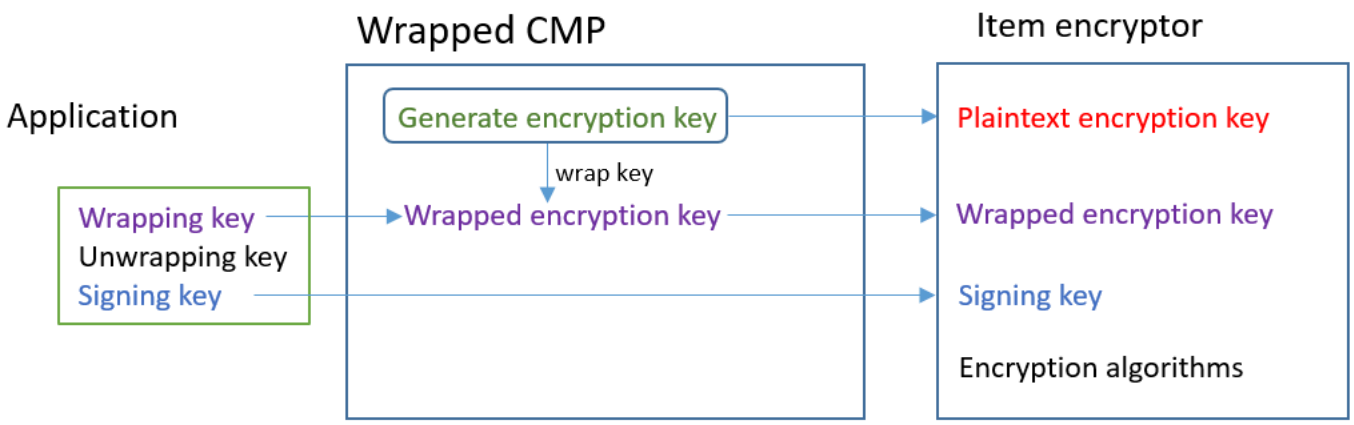
```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

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
)
```

Como funciona

O CMP encapsulado gera uma nova chave de criptografia para cada item. Ele usa as chaves de empacotamento, desempacotamento e assinatura que você fornece, conforme mostrado no diagrama a seguir.



Obter materiais de criptografia

Esta seção descreve em detalhes as entradas, as saídas e o processamento do provedor encapsulado de materiais (CMP encapsulado) quando ele recebe uma solicitação de materiais de criptografia.

Entrada (do aplicativo)

- Chave de empacotamento: uma chave simétrica do Advanced Encryption Standard (AES) ou uma chave pública RSA. Obrigatória se houver valores de atributo criptografados. Caso contrário, ela é opcional e ignorada.
- Chave de descriptografia: opcional e ignorada.
- Chave de assinatura

Entrada (do criptografador de itens)

- Contexto de criptografia do DynamoDB

Saída (para o criptografador de itens):

- Chave de criptografia do item de texto simples
- Chave de assinatura (inalterada)
- Descrição real do material: esses valores são salvos no atributo de descrição do material que o cliente adiciona ao item.
  - `amzn-ddb-env-key`: chave de criptografia de item encapsulado codificada em Base64

- `amzn-ddb-env-alg`: algoritmo de criptografia usado para criptografar o item. O padrão é AES-256-CBC.
- `amzn-ddb-wrap-alg`: o algoritmo de empacotamento que o CMP empacotado usou para encapsular a chave de criptografia de item. Se a chave de empacotamento for uma chave do AES, ela será encapsulada com o `AES-Keywrap` não preenchido, conforme definido na RFC 3394. Se a chave de empacotamento for uma chave RSA, a chave será criptografada usando RSA OAEP com preenchimento. MGF1

Processamento

Quando você criptografa um item, transmite uma chave de empacotamento e outra de assinatura. A chave de descriptografia é opcional e ignorada.

1. O CMP encapsulado gera uma chave exclusiva de criptografia simétrica para o item de tabela.
2. Ele usa a chave de empacotamento que você especifica para encapsular a chave de criptografia de item. Depois, ele a remove da memória o mais rápido possível.
3. Ele retorna a chave de criptografia do item de texto sem formatação, a chave de assinatura que você forneceu e uma descrição real do material que inclui a chave de criptografia do item empacotado e os algoritmos de criptografia e empacotamento.
4. O criptografador do item usa a chave de criptografia de texto simples para criptografar o item. Ele usa a chave de assinatura que você forneceu para assinar o item. Depois, ele remove as chaves de texto simples da memória o mais rápido possível. Ele copia os campos na descrição real do material, incluindo a chave de criptografia encapsulada (`amzn-ddb-env-key`), para o atributo de descrição do material do item.

Obter materiais de descriptografia

Esta seção descreve em detalhes as entradas, as saídas e o processamento do provedor encapsulado de materiais (CMP encapsulado) quando ele recebe uma solicitação de materiais de descriptografia.

Entrada (do aplicativo)

- Chave de criptografia: opcional e ignorada.
- Chave de descriptografia: a mesma chave simétrica Advanced Encryption Standard (AES) ou a chave privada RSA que corresponde à chave pública RSA usada para criptografia. Obrigatória se houver valores de atributo criptografados. Caso contrário, ela é opcional e ignorada.

- Chave de assinatura

Entrada (do criptografador de itens)

- Uma cópia do contexto de criptografia do DynamoDB com o conteúdo do atributo de descrição do material.

Saída (para o criptografador de itens)

- Chave de criptografia do item de texto simples
- Chave de assinatura (inalterada)

Processamento

Quando você descriptografa um item, transmite uma chave de desempacotamento e outra de assinatura. A chave de empacotamento é opcional e ignorada.

1. O CMP encapsulado obtém a chave de criptografia de item encapsulado do atributo de descrição do material do item.
2. Ele usa o algoritmo e a chave de desempacotamento para desencapsular a chave de criptografia de item.
3. Ele retorna a chave de criptografia de item de texto simples, a chave de assinatura e os algoritmos de criptografia e assinatura para o criptografador do item.
4. O criptografador do item usa a chave de assinatura para verificar o item. Quando consegue fazer isso, ele usa a chave de criptografia de item para descriptografar o item. Depois, ele remove as chaves de texto simples da memória o mais rápido possível.

Provedor mais recente

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption

Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

O Provedor mais recente é um provedor de materiais de criptografia (CMP) que foi projetado para trabalhar com um armazenamento de provedores. Ele é CMPs obtido da loja do fornecedor e obtém os materiais criptográficos que retorna do CMPs. Normalmente, ele usa cada CMP para atender a várias solicitações de materiais de criptografia. Mas você pode usar os recursos do armazenamento de provedores para gerenciar a frequência com a qual os materiais são reutilizados, determinar a frequência de rotação do CMP e até mesmo alterar o tipo de CMP usado sem alterar o provedor mais recente.

> **Note**
>
> O código associado ao símbolo `MostRecentProvider` do provedor mais recente pode armazenar materiais criptográficos na memória durante a vida útil do processo. Isso pode permitir que um chamador use chaves que não está mais autorizado a usar.
> O símbolo `MostRecentProvider` está obsoleto nas versões mais antigas compatíveis do DynamoDB Encryption Client e foi removido da versão 2.0.0. Ele é substituído pelo símbolo `CachingMostRecentProvider`. Para obter detalhes, consulte Atualizações do provedor mais recente.

O provedor mais recente é uma boa opção para aplicativos que precisam minimizar as chamadas para o armazenamento de provedores, sua origem de criptografia e aplicativos que podem reutilizar alguns materiais de criptografia sem violar os requisitos de segurança. Por exemplo, ele permite que você proteja seus materiais criptográficos sob um AWS KMS keyin AWS Key Management Service(AWS KMS) sem chamar AWS KMS toda vez que você criptografa ou descriptografa um item.

O repositório do provedor que você escolher determina o tipo do CMPs que o provedor mais recente usa e com que frequência ele obtém um novo CMP. Você pode usar qualquer armazenamento compatível de provedores com o provedor mais recente, incluindo os armazenamentos de provedores personalizados que você criar.

O DynamoDB Encryption Client inclui MetaStoreum que cria e retorna Wrapped Materials Providers (Wrapped). CMPs Ele MetaStore salva várias versões do Wrapped CMPs que ele gera em uma tabela interna do DynamoDB e as protege com criptografia do lado do cliente por uma instância interna do DynamoDB Encryption Client.

Você pode configurar o MetaStore para usar qualquer tipo de CMP interno para proteger os materiais na tabela, incluindo um provedor de KMS direto que gera materiais criptográficos protegidos por você AWS KMS key, um CMP empacotado que usa chaves de empacotamento e assinatura fornecidas por você ou um CMP personalizado compatível que você cria.

Para ver um código de exemplo, consulte:

- Java: MostRecentEncryptedItem
- Python: most_recent_provider_encrypted_table

Tópicos

- Como usar
- Como funciona
- Atualizações do provedor mais recente

Como usar

Para criar um provedor mais recente, você precisa criar e configurar um armazenamento de provedores e, em seguida, criar um provedor mais recente que usa o armazenamento de provedores.

Os exemplos a seguir mostram como criar um provedor mais recente que usa MetaStore e protege as versões em sua tabela interna do DynamoDB com materiais criptográficos de um provedor de KMS direto. Estes exemplos usam o símbolo `CachingMostRecentProvider`.

Cada provedor mais recente tem um nome que o identifica CMPs na MetaStore tabela, uma configuração time-to-live(TTL) e uma configuração de tamanho de cache que determina quantas entradas o cache pode conter. Esses exemplos definem o tamanho do cache para 1000 entradas e um TTL de 60 segundos.

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
)
```

Como funciona

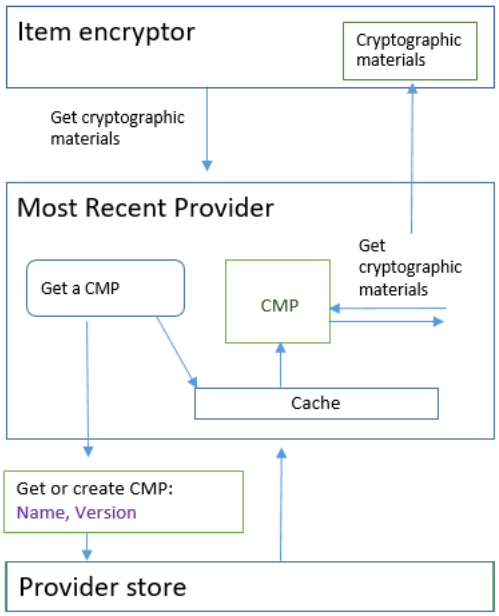
O fornecedor mais recente CMPs vem de uma loja de fornecedores. Em seguida, ele usa o CMP para gerar os materiais de criptografia que retorna ao criptografador de item.

Sobre o provedor mais recente

O Provedor mais recente obtém um provedor de materiais de criptografia (CMP) de um armazenamento de provedores. Em seguida, ele usa o CMP para gerar os materiais de criptografia que retorna. Cada provedor mais recente está associado a uma loja de provedores, mas uma loja de provedores pode fornecer CMPs a vários provedores em vários hosts.

O provedor mais recente pode trabalhar com qualquer CMP compatível de qualquer armazenamento de provedores. Ele solicita materiais de criptografia ou de descriptografia do CMP e retorna a saída ao criptografador do item. Não executa nenhuma operação de criptografia.

Para solicitar um CMP do armazenamento de provedores, o provedor mais recente fornece o nome de material e a versão de um CMP existente que deseja usar. Para materiais de criptografia, o provedor mais recente sempre solicita a versão mais recente. Para materiais de descriptografia, ele solicita a versão do CMP que foi usada para criar os materiais de criptografia, conforme exibido no diagrama a seguir.



O provedor mais recente salva as versões do CMPs que o provedor armazena retornou em um cache local de uso menos recente (LRU) na memória. O cache permite que o provedor mais recente obtenha o CMPs que precisa sem chamar a loja do provedor para cada item. Você pode limpar o cache sob demanda.

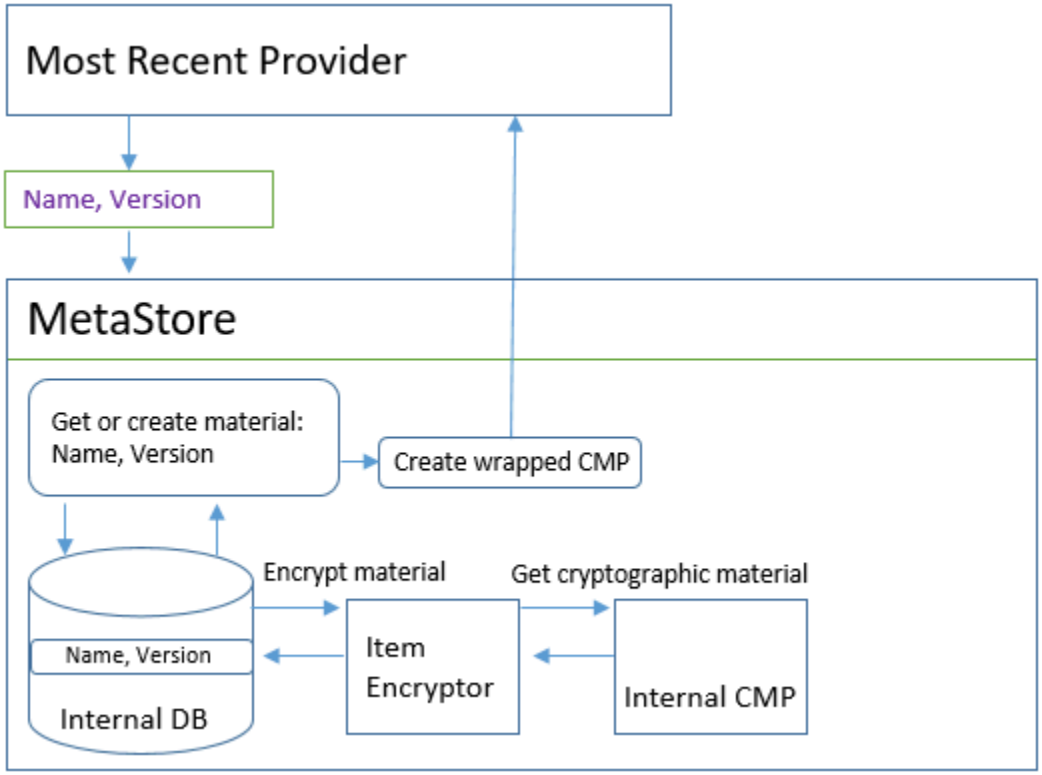
O provedor mais recente usa um time-to-livevalor configurável que você pode ajustar com base nas características do seu aplicativo.

Sobre o MetaStore

Você pode usar um provedor mais recente com qualquer armazenamento de provedores, incluindo um armazenamento de provedores personalizado compatível. O DynamoDB Encryption Client inclui MetaStore uma implementação segura que você pode configurar e personalizar.

MetaStoreA é um repositório de provedores que cria e retorna Wrapped CMPs configurados com a chave de empacotamento, a chave de desempacotamento e a chave de assinatura exigidas pelo Wrapped. CMPs MetaStore A é uma opção segura para um provedor mais recente porque o Wrapped CMPs sempre gera chaves de criptografia de item exclusivas para cada item. Somente a chave de empacotamento que protege a chave de criptografia do item e as chaves de assinatura é reutilizada.

O diagrama a seguir mostra os componentes do MetaStore e como ele interage com o provedor mais recente.



O MetaStore gera o Wrapped e CMPs, em seguida, o armazena (em formato criptografado) em uma tabela interna do DynamoDB. A chave de partição é o nome do material do provedor mais recente; a chave de classificação, seu número de versão. Os materiais na tabela são protegidos por um DynamoDB Encryption Client interno, incluindo um criptografador de item e um provedor de materiais de criptografia (CMP) interno.

Você pode usar qualquer tipo de CMP interno em seu MetaStore, incluindo um provedor de KMS direto, um CMP empacotado com materiais criptográficos fornecidos por você ou um CMP personalizado compatível. Se o CMP interno do seu MetaStore for um provedor de KMS direto, suas chaves reutilizáveis de empacotamento e assinatura serão protegidas por um in (). AWS KMS keyAWS Key Management ServiceAWS KMS As MetaStore chamadas AWS KMS sempre que ele adiciona uma nova versão do CMP à tabela interna ou obtém uma versão do CMP da tabela interna.

Definindo um time-to-live valor

Você pode definir um valor time-to-live (TTL) para cada provedor mais recente que você criar. Em geral, use o valor TTL mais baixo que seja prático para a sua aplicação.

O uso do valor TTL é alterado no símbolo `CachingMostRecentProvider` do provedor mais recente.

> **Note**
>
> O símbolo `MostRecentProvider` do Provedor mais recente está obsoleto nas versões mais antigas compatíveis do DynamoDB Encryption Client e foi removido da versão 2.0.0. Ele é substituído pelo símbolo `CachingMostRecentProvider`. Recomendamos que você atualize seu código o mais rápido possível. Para obter detalhes, consulte Atualizações do provedor mais recente.

`CachingMostRecentProvider`

O `CachingMostRecentProvider` usa o valor TTL de duas maneiras diferentes.

- O TTL determina com que frequência o provedor mais recente verifica o armazenamento do provedor em busca de uma nova versão do CMP. Se uma nova versão estiver disponível, o provedor mais recente substituirá o CMP e atualizará os materiais criptográficos. Caso contrário, ele continuará usando o CMP atual e os materiais criptográficos.
- O TTL determina por quanto tempo CMPs o cache pode ser usado. Antes de usar uma CMP em cache para criptografia, o provedor mais recente avalia seu tempo no cache. Se o tempo de cache do CMP exceder o TTL, o CMP será removido do cache e o provedor mais recente obterá um novo CMP da versão mais recente do repositório do provedor.

`MostRecentProvider`

No `MostRecentProvider`, o TTL determina com que frequência o provedor mais recente verifica o armazenamento do provedor em busca de uma nova versão do CMP. Se uma nova versão estiver disponível, o provedor mais recente substituirá o CMP e atualizará os materiais criptográficos. Caso contrário, ele continuará usando o CMP atual e os materiais criptográficos.

O TTL não determina com que frequência uma nova versão do CMP é criada. Crie novas versões do CMP alternando os materiais criptográficos.

Um valor ideal de TTL varia de acordo com o aplicativo e suas metas de latência e disponibilidade. Um TTL mais baixo melhora seu perfil de segurança ao reduzir o tempo em que os materiais criptográficos são armazenados na memória. Além disso, um TTL mais baixo atualiza as informações críticas com mais frequência. Por exemplo, se seu CMP interno for um Direct KMS Provider, ele verificará com mais frequência se o chamador ainda está autorizado a usar um AWS KMS key.

No entanto, se o TTL for muito breve, as chamadas frequentes para o armazenamento do provedor podem aumentar seus custos e fazer com que o armazenamento do provedor reduza as solicitações do seu aplicativo e de outros aplicativos que compartilham sua conta de serviço. Também é possível se beneficiar da coordenação do TTL com a taxa na qual você alterna os materiais criptográficos.

Durante o teste, varie o tamanho do TTL e do cache em diferentes cargas de trabalho até encontrar uma configuração que funcione para seu aplicativo e seus padrões de segurança e desempenho.

Alternar os materiais de criptografia

Quando um provedor mais recente precisa de materiais de criptografia, ele sempre usa a versão mais recente de seu CMP que conhece. A frequência com que ele verifica uma versão mais recente é determinada pelo valor time-to-live(TTL) que você define ao configurar o provedor mais recente.

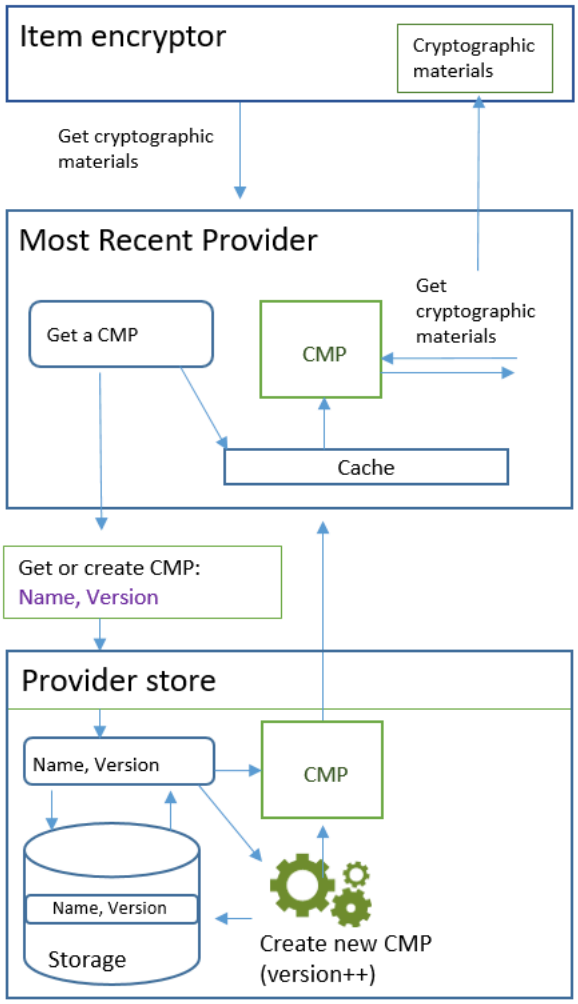
Quando o TTL expira, o provedor mais recente verifica o armazenamento do provedor em busca de uma nova versão do CMP. Se houver um disponível, o provedor mais recente o obterá e substituirá o CMP em seu cache. Ele usa esse CMP e seus materiais criptográficos até descobrir que o armazenamento do provedor tem uma versão mais recente.

Para solicitar que o armazenamento de provedores crie uma nova versão de um CMP para um provedor mais recente, chame a operação Criar Novo Provedor do armazenamento de provedores com o nome do material do provedor mais recente. O armazenamento de provedores cria um novo CMP e salva uma cópia criptografada em seu armazenamento interno com um número da versão mais recente. (Ele também retorna um CMP, mas você pode descartá-lo.) Como resultado, na próxima vez que o provedor mais recente consultar o repositório do provedor para obter o número máximo de versão CMPs, ele obterá o novo número de versão maior e o usará em solicitações subsequentes à loja para ver se uma nova versão da CMP foi criada.

Você pode programar as chamadas da operação Criar Novo Provedor com base no tempo, no número de itens ou de atributos processados ou em qualquer outra métrica que seja aceitável para seu aplicativo.

Obter materiais de criptografia

O provedor mais recente usa o seguinte processo, mostrado neste diagrama, para obter os materiais de criptografia retornados ao criptografador de item. A saída depende do tipo de CMP que o armazenamento de provedores retorna. O provedor mais recente pode usar qualquer loja de provedores compatível, incluindo a MetaStore que está incluída no DynamoDB Encryption Client.



Ao criar um provedor mais recente usando o `CachingMostRecentProvider`símbolo, você especifica um repositório de provedores, um nome para o provedor mais recente e um valor time-to-live(TTL). Também é possível especificar opcionalmente um tamanho de cache, que determina o número máximo de materiais criptográficos que podem existir no cache.

Quando o criptografador de item solicita ao provedor mais recente os materiais de criptografia, esse provedor começa pesquisando em seu cache a versão mais recente do CMP.

- Se ele encontrar a versão mais recente do CMP no cache e o CMP não tiver excedido o valor TTL, o provedor mais recente usará o CMP para gerar materiais de criptografia. Em seguida, ele retorna os materiais de criptografia ao criptografador de item. Essa operação não requer uma chamada para o armazenamento de provedores.
- Se a versão mais recente do CMP não estiver no cache, ou se estiver no cache mas tiver excedido o valor TTL, o provedor mais recente solicitará um CMP do armazenamento de provedores. A

solicitação inclui o nome do material do provedor mais recente e o número da versão mais recente que ele conhece.

1. O armazenamento de provedores retorna um CMP de seu armazenamento persistente. Se o repositório do provedor for um MetaStore, ele obterá uma CMP encriptada encriptada de sua tabela interna do DynamoDB usando o nome do material Most Recent Provider como chave de partição e o número da versão como chave de classificação. O MetaStore usa seu criptografador de itens interno e CMP interno para descriptografar o Wrapped CMP. Em seguida, ele retorna o CMP com texto simples ao provedor mais recente. Se o CMP interno for um Direct KMS Provider, esta etapa incluirá uma chamada ao AWS Key Management Service (AWS KMS).
2. O CMP adiciona o campo `amzn-ddb-meta-id` à descrição real do material. O valor é o nome do material e a versão do CMP em sua tabela interna. O armazenamento de provedores retorna o CMP ao provedor mais recente.
3. O provedor mais recente armazena o CMP na memória.
4. O provedor mais recente usa o CMP para gerar materiais de criptografia. Em seguida, ele retorna os materiais de criptografia ao criptografador de item.

Obter materiais de descriptografia

Quando o criptografador do item solicita ao provedor mais recente os materiais de descriptografia, esse provedor usa o seguinte processo para obtê-los e retorná-los.

1. O provedor mais recente solicita ao armazenamento de provedores o número da versão dos materiais de criptografia que foram usados para criptografar o item. Ele passa a descrição real do material a partir do atributo de descrição do material do item.
2. O armazenamento de provedores obtém o número da versão do CMP criptografado a partir do campo `amzn-ddb-meta-id` na descrição real do material e o retorna ao provedor mais recente.
3. O provedor mais recente pesquisa seu cache em busca da versão do CMP que foi usada para criptografar e assinar o item.

- Se descobrir que a versão correspondente do CMP está em seu cache e o CMP não excedeu o valor time-to-live (TTL), o provedor mais recente usa o CMP para gerar materiais de descriptografia. Em seguida, ele retorna os materiais de descriptografia ao criptografador de item. Essa operação não requer uma chamada para o armazenamento de provedores ou qualquer outro CMP.

- Se a versão do CMP correspondente não estiver no cache, ou se o AWS KMS key estiver no cache mas tiver excedido o valor TTL, o provedor mais recente solicitará um CMP do armazenamento de provedores. Ele envia o nome do material e o número da versão do CMP criptografado na solicitação.
  1. O armazenamento de provedores pesquisa seu armazenamento persistente em busca do CMP usando o nome do provedor mais recente como a chave de partição e o número da versão como a chave de classificação.
     - Se o nome e o número da versão não estiverem no armazenamento persistente, o armazenamento de provedores gera uma exceção. Se o armazenamento de provedores foi usado para gerar o CMP, o CMP deve ser armazenado no armazenamento persistente, a menos que tenha sido intencionalmente excluído.
     - Se o CMP com o nome e o número de versão correspondentes estiverem no armazenamento persistente do armazenamento de provedores, este retornará o CMP especificado ao provedor mais recente.

       Se o repositório do provedor for um MetaStore, ele obterá o CMP criptografado de sua tabela do DynamoDB. Em seguida, ele usa materiais de criptografia do CMP interno para descriptografar o CMP criptografado antes de retornar o CMP ao provedor mais recente. Se o CMP interno for um Direct KMS Provider, esta etapa incluirá uma chamada ao AWS Key Management Service (AWS KMS).
  2. O provedor mais recente armazena o CMP na memória.
  3. O provedor mais recente usa o CMP para gerar materiais de descriptografia. Em seguida, ele retorna os materiais de descriptografia ao criptografador de item.

Atualizações do provedor mais recente

O símbolo do provedor mais recente é alterado de `MostRecentProvider` para `CachingMostRecentProvider`.

> **Note**
>
> O símbolo `MostRecentProvider`, que representa o provedor mais recente, foi descontinuado na versão 1.15 do DynamoDB Encryption Client for Java e na versão 1.3 do DynamoDB Encryption Client for Python e removido das versões 2.0.0 do DynamoDB Encryption Client nas duas implementações de linguagem. Use a `CachingMostRecentProvider` em vez disso.

O `CachingMostRecentProvider` implementa as seguintes mudanças:

- O remove `CachingMostRecentProvider` periodicamente materiais criptográficos da memória quando seu tempo na memória excede o valor configurado time-to-live (TTL).

  O `MostRecentProvider` pode armazenar materiais criptográficos na memória durante toda a vida útil do processo. Como resultado, o provedor mais recente pode não estar ciente das alterações na autorização. Ele pode usar chaves de criptografia depois que as permissões do chamador para usá-las forem revogadas.

  Se você não conseguir atualizar para essa nova versão, poderá obter um efeito semelhante chamando periodicamente o método `clear()` no cache. Esse método limpa manualmente o conteúdo do cache e exige que o provedor mais recente solicite um novo CMP e novos materiais criptográficos.
- O `CachingMostRecentProvider` também inclui uma configuração de tamanho de cache que oferece mais controle sobre o cache.

Para atualizar para o `CachingMostRecentProvider`, você precisa alterar o nome do símbolo em seu código. Em todos os outros aspectos, o `CachingMostRecentProvider` é totalmente compatível com versões anteriores do `MostRecentProvider`. Você não precisa criptografar novamente nenhum item da tabela.

No entanto, o `CachingMostRecentProvider` gera mais chamadas para a infraestrutura principal subjacente. Ele chama a loja do provedor pelo menos uma vez em cada intervalo time-to-live (TTL). Aplicativos com vários ativos CMPs (devido à rotação frequente) ou aplicativos com grandes frotas provavelmente serão sensíveis a essa mudança.

Antes de lançar seu código atualizado, teste-o minuciosamente para garantir que as chamadas mais frequentes não prejudiquem seu aplicativo nem causem limitação por serviços dos quais seu provedor depende, como AWS Key Management Service () ou AWS KMS Amazon DynamoDB. Para mitigar quaisquer problemas de desempenho, ajuste o tamanho do cache e o time-to-live do `CachingMostRecentProvider` com base nas características de desempenho observadas. Para obter orientações, consulte Definindo um time-to-live valor.

Provedor estático de materiais

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

O Static Materials Provider (Static CMP) é um provedor de materiais criptográficos (CMP) muito simples, destinado a testes, proof-of-concept demonstrações e compatibilidade antiga.

Para usar o CMP estático para criptografar um item de tabela, forneça uma chave de criptografia simétrica do Advanced Encryption Standard (AES) e uma chave de assinatura ou um par de chaves. Você deve fornecer as mesmas chaves para descriptografar o item criptografado. O CMP estático não realiza operações de criptografia. Em vez disso, ele transmite inalteradas as chaves de criptografia que você fornece ao criptografador do item. O criptografador do item criptografa os itens diretamente na chave de criptografia. Depois, ele usa a chave de assinatura diretamente para assiná-los.

Como o CMP estático não gera nenhum material exclusivo de criptografia, todos os itens da tabela que você processa são criptografados com a mesma chave de criptografia e assinados pela mesma chave de assinatura. Ao usar a mesma chave para criptografar os valores de atributos em diversos itens ou a mesma chave ou par de chaves para assinar todos os itens, você corre o risco de ultrapassar os limites de criptografia das chaves.

> **Note**
>
> O Provedor estático assimétrico na biblioteca Java não é um provedor estático. Ele apenas oferece construtores alternativos para o CMP encapsulado. Ele é seguro para fins de produção, mas você deve usar o CMP encapsulado diretamente sempre que possível.

O Static CMP é um dos vários fornecedores de materiais criptográficos (CMPs) compatíveis com o DynamoDB Encryption Client. Para obter informações sobre o outro CMPs, consulteProvedor de materiais de criptografia.

Para ver um código de exemplo, consulte:

- Java: SymmetricEncryptedItem

Tópicos

- Como usar
- Como funciona

Como usar

Para criar um provedor estático, forneça uma chave de criptografia ou um par de chaves e uma chave de assinatura ou um par de chaves. É necessário fornecer material de chave para criptografar e descriptografar os itens de tabela.

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);
```

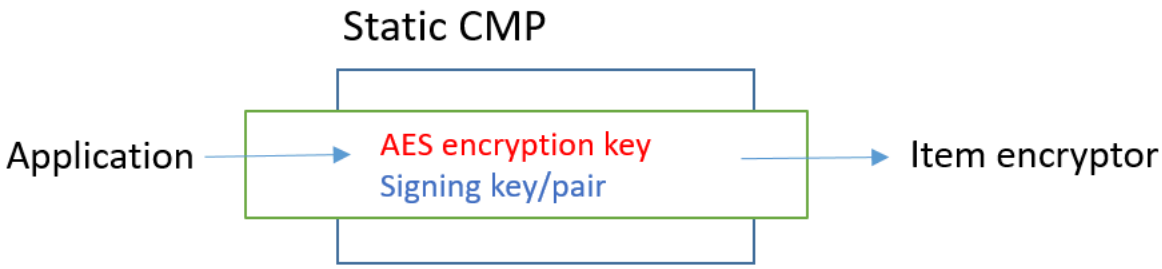
Python

```
# 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
)
```

Como funciona

O provedor estático transmite as chaves de criptografia e assinatura que você fornece ao criptografador do item, onde elas são usadas diretamente para criptografar e assinar os itens da tabela. As mesmas chaves são usadas para todos os itens, a menos que você forneça chaves diferentes para cada um deles.



Obter materiais de criptografia

Esta seção descreve em detalhes as entradas, as saídas e o processamento do provedor estático de materiais (CMP estático) quando ele recebe uma solicitação de materiais de criptografia.

Entrada (do aplicativo)

- Chave de criptografia - deve ser uma chave simétrica, como uma chave do Advanced Encryption Standard (AES).
- Chave de assinatura - Pode ser uma chave simétrica ou um par de chaves assimétrico.

Entrada (do criptografador de itens)

- Contexto de criptografia do DynamoDB

Saída (para o criptografador de itens)

- A chave de criptografia transmitida como entrada.
- A chave de assinatura transmitida como entrada.

- Descrição real do material: a descrição solicitada do material, se houver, inalterada.

Obter materiais de descriptografia

Esta seção descreve em detalhes as entradas, as saídas e o processamento do provedor estático de materiais (CMP estático) quando ele recebe uma solicitação de materiais de descriptografia.

Embora ela inclua métodos separados para obter materiais de criptografia e de descriptografia, o comportamento é o mesmo.

Entrada (do aplicativo)

- Chave de criptografia - deve ser uma chave simétrica, como uma chave do Advanced Encryption Standard (AES).
- Chave de assinatura - Pode ser uma chave simétrica ou um par de chaves assimétrico.

Entrada (do criptografador de itens)

- Contexto de criptografia do DynamoDB (não usado)

Saída (para o criptografador de itens)

- A chave de criptografia transmitida como entrada.
- A chave de assinatura transmitida como entrada.

Linguagens de programação disponíveis do Amazon DynamoDB Encryption Client

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

O Amazon DynamoDB Encryption Client está disponível para as linguagens de programação a seguir. As bibliotecas específicas de linguagem variam, mas as implementações resultantes são interoperáveis. Por exemplo, é possível criptografar (e assinar) um item com o cliente de Java e descriptografá-lo com o cliente Python.

Para obter mais informações, consulte o tópico correspondente.

Tópicos

- Amazon DynamoDB Encryption Client para Java
- DynamoDB Encryption Client para Python

Amazon DynamoDB Encryption Client para Java

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Este tópico explica como instalar e usar o Amazon DynamoDB Encryption Client para Java. Para obter detalhes sobre a programação com o DynamoDB Encryption Client, consulte os exemplos de Java, os exemplos no repositório GitHub em e o Javadoc para aws-dynamodb-encryption-java o DynamoDB Encryption Client.

> **Note**
>
> Versões 1. x. x do DynamoDB Encryption Client for Java estão end-of-support em fase a partir de julho de 2022. Atualize para uma versão mais recente o mais rápido possível.

Tópicos

- Pré-requisitos
- Instalação

- Uso do DynamoDB Encryption Client para Java
- Código de exemplo para o DynamoDB Encryption Client para Java

Pré-requisitos

Antes de instalar o Amazon DynamoDB Encryption Client para Java, verifique se você tem os pré-requisitos a seguir.

Um ambiente de desenvolvimento Java

Você precisará do Java 8 ou posterior. No site da Oracle, acesse Java SE Downloads e faça download e instale o Java SE Development Kit (JDK).

Se você usa o Oracle JDK, também precisara fazer download e instalar os arquivos de política de jurisdição de força ilimitada JCE (Java Cryptography Extension).

AWS SDK para Java

O DynamoDB Encryption Client exige o módulo DynamoDB do mesmo que seu aplicativo não interaja com AWS SDK para Java o DynamoDB. É possível instalar todo o SDK ou apenas esse módulo. Se você usa o Maven, adicione `aws-java-sdk-dynamodb` ao arquivo `pom.xml`.

Para obter mais informações sobre como instalar e configurar o AWS SDK para Java, consulte AWS SDK para Java.

Instalação

É possível instalar o Amazon DynamoDB Encryption Client para Java usando as opções a seguir.

Manualmente

Para instalar o Amazon DynamoDB Encryption Client para Java, clone ou baixe o repositório. aws-dynamodb-encryption-java GitHub

Uso do Apache Maven

O Amazon DynamoDB Encryption Client para Java está disponível por meio do Apache Maven com a definição de dependência a seguir.

```
<dependency>
```

```
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

Depois de instalar o SDK, comece examinando o código de exemplo neste guia e o Javadoc do DynamoDB Encryption Client ativado. GitHub

Uso do DynamoDB Encryption Client para Java

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Este tópico explica alguns dos recursos do DynamoDB Encryption Client para Java que talvez não sejam encontrados em outras implementações de linguagem de programação.

Para obter detalhes sobre a programação com o DynamoDB Encryption Client, consulte os exemplos em Java, os exemplos em GitHub on e o Javadoc para `aws-dynamodb-encryption-java repository` o DynamoDB Encryption Client.

Tópicos

- Criptografadores de itens: AttributeEncryptor e Dynamo DBEncryptor
- Configurar o comportamento de salvamento
- Ações de atributos em Java
- Substituir nomes de tabelas

Criptografadores de itens: AttributeEncryptor e Dynamo DBEncryptor

O DynamoDB Encryption Client em Java tem dois criptografadores de itens: o Dynamo de nível inferior e o. DBEncryptor AttributeEncryptor

`AttributeEncryptor`É uma classe auxiliar que ajuda você a usar o Dynamo AWS SDK para Java com o DBMapper no DynamoDB `DynamoDB Encryptor` Encryption Client. Ao usar o `AttributeEncryptor` com o `DynamoDBMapper`, ele criptografa e assina seus itens de forma transparente quando você os salva. Ele também verifica e descriptografa seus itens de forma transparente quando você os carrega.

Configurar o comportamento de salvamento

É possível usar o `AttributeEncryptor` e o `DynamoDBMapper` para adicionar ou substituir itens de tabela com atributos assinados somente ou criptografados e assinados. Para essas tarefas, recomendamos que você o configure para usar o comportamento de salvamento PUT, conforme mostrado no exemplo a seguir. Caso contrário, talvez você não possa descriptografar os dados.

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

Se você usar o comportamento padrão de salvamento, que atualiza somente os atributos que são modelados no item da tabela, os atributos não serão incluídos na assinatura e não serão alterado nas gravações da tabela. Como resultado, em leituras posteriores de todos os atributos, a assinatura não será validada porque não inclui atributos não modelados.

Também é possível usar o comportamento de salvamento `CLOBBER`. Esse comportamento é idêntico ao comportamento de salvamento PUT, exceto pelo fato de que ele desabilita o bloqueio otimista e substitui o item na tabela.

Para evitar erros de assinatura, o DynamoDB Encryption Client lança uma exceção de runtime se um `AttributeEncryptor` for usado com um `DynamoDBMapper` que não esteja configurado com um comportamento de salvamento de `CLOBBER` ou PUT.

Para ver esse código usado em um exemplo, consulte Usando o Dynamo DBMapper o exemplo AwsKmsEncryptedObjectde.java no `aws-dynamodb-encryption-java` repositório em. GitHub

Ações de atributos em Java

As Ações de atributos determinam quais valores de atributo são criptografados e assinados, quais são apenas assinados e quais são ignorados. O método usado para especificar ações de atributos depende de você usar o `DynamoDBMapper` e ou o `AttributeEncryptor` Dynamo de nível inferior. DBEncryptor

> **Important**
>
> Depois de usar as ações do atributo para criptografar os itens da tabela, adicionar ou remover atributos do modelo de dados poderá gerar um erro de validação de assinatura que impede a descriptografia dos dados. Para obter uma explicação detalhada, consulte Alterar seu modelo de dados.

Ações de atributos para o Dynamo DBMapper

Ao usar o `DynamoDBMapper` e o `AttributeEncryptor`, use anotações para especificar as ações de atributos. O DynamoDB Encryption Client usa as anotações de atributo padrão do DynamoDB que definem o tipo do atributo para determinar como proteger um atributo. Por padrão, todos os atributos são criptografados e assinados, exceto as chaves primárias, que são assinadas, mas não são criptografadas.

> **Note**
>
> Não criptografe o valor dos atributos com a anotação @Dynamo DBVersion Attribute, embora você possa (e deva) assiná-los. Caso contrário, as condições que usam o valor terão efeitos indesejados.

```
// 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")
```

Para especificar exceções, use as anotações de criptografia definidas no DynamoDB Encryption Client para Java. Se você especificá-las no nível da classe, elas se tornam o valor padrão para a classe.

```
// Sign only
@DoNotEncrypt
```

```
// Do nothing; not encrypted or signed
@DoNotTouch
```

Por exemplo, essas anotações assinam, mas não criptografam o atributo `PublicationYear`, e não criptografam nem assinam o valor de atributo `ISBN`.

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

Ações de atributos para o Dynamo DBEncryptor

Para especificar ações de atributos ao usar o Dynamo DBEncryptor diretamente, crie um `HashMap` objeto no qual os pares nome-valor representem os nomes dos atributos e as ações especificadas.

Os valores válidos para as ações de atributo estão definidos no tipo enumerado de `EncryptionFlags`. Você pode usar `ENCRYPT` e `SIGN` juntos, usar `SIGN` isoladamente ou omitir os dois. No entanto, se você usar `ENCRYPT` sozinho, o DynamoDB Encryption Client gerará um erro. Você não pode criptografar um atributo que você não assine.

```
ENCRYPT
SIGN
```

> **Warning**
>
> Não criptografe os atributos da chave primária. Eles devem permanecer em texto simples para que o DynamoDB possa encontrar o item sem executar uma varredura completa da tabela.

Se você especificar uma chave primária no contexto de criptografia e especificar `ENCRYPT` na ação de um atributo de chave primária, o DynamoDB Encryption Client gerará uma exceção.

Por exemplo, o código Java a seguir cria um `actions` HashMap que criptografa e assina todos os atributos no `record` item. As exceções são os atributos de chave de partição e de chave de

classificação que são assinados, mas não criptografados, e o atributo `test` que não é assinado nem criptografado.

```
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;
  }
}
```

Ao chamar o método encryptRecord do `DynamoDBEncryptor`, especifique o mapa como o valor do parâmetro `attributeFlags`. Por exemplo, esta chamada para `encryptRecord` usa o mapa `actions`.

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Substituir nomes de tabelas

No DynamoDB Encryption Client, o nome da tabela do DynamoDB é um elemento do contexto de criptografia do DynamoDB que é passado para os métodos de criptografia e de descriptografia. Quando você criptografa ou assina itens de tabela, o contexto de criptografia do DynamoDB, inclusive o nome da tabela, é vinculado criptograficamente ao texto cifrado. Se o contexto de criptografia do DynamoDB passado para o método de descriptografia não corresponder ao contexto

de criptografia do DynamoDB passado para o método de criptografia, a operação de descriptografia falhará.

Ocasionalmente, o nome de uma tabela muda, como quando você faz backup de uma tabela ou executa uma point-in-time recuperação. Ao descriptografar ou verificar a assinatura desses itens, passe o mesmo contexto de criptografia do DynamoDB usado para criptografar e assinar os itens, inclusive o nome da tabela original. O nome da tabela atual não é necessário.

Quando você usa o `DynamoDBEncryptor`, você monta o contexto de criptografia do manualmente. No entanto, se você estiver usando o `DynamoDBMapper`, o `AttributeEncryptor` criará o contexto de criptografia do DynamoDB para você, incluindo o nome da tabela atual. Para informar ao `AttributeEncryptor` para criar um contexto de criptografia com um nome de tabela diferente, use o `EncryptionContextOverrideOperator`.

Por exemplo, o código a seguir cria instâncias do provedor de materiais de criptografia (CMP) e do `DynamoDBEncryptor`. Depois, ele chama o método `setEncryptionContextOverrideOperator` do `DynamoDBEncryptor`. Ele usa o operador `overrideEncryptionContextTableName`, que substitui um nome de tabela. Quando ele é configurado dessa maneira, o `AttributeEncryptor` cria um contexto de criptografia do DynamoDB que inclui `newTableName` no lugar de `oldTableName`. Para ver um exemplo completo, consulte EncryptionContextOverridesWithDynamoDBMapper.java.

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

Quando você chama o método de carregamento do `DynamoDBMapper`, que descriptografa e verifica o item, você especifica o nome da tabela original.

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

Também é possível usar o operador `overrideEncryptionContextTableNameUsingMap`, que substitui vários nomes de tabela.

Normalmente, os operadores de substituição de nome de tabela são usados ao descriptografar dados e verificar assinaturas. No entanto, é possível usá-los para definir o nome da tabela no contexto de criptografia do DynamoDB como um valor diferente ao criptografar e assinar.

Não use os operadores de substituição de nome de tabela se estiver usando o `DynamoDBEncryptor`. Em vez disso, crie um contexto de criptografia com o nome da tabela original e envie-o para o método de descriptografia.

Código de exemplo para o DynamoDB Encryption Client para Java

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Os exemplos a seguir mostram como usar o DynamoDB Encryption Client para Java para proteger a tabela do DynamoDB no aplicativo. Você pode encontrar mais exemplos (e contribuir com os seus) no diretório de exemplos do aws-dynamodb-encryption-javarepositório em GitHub.

Tópicos

- Usando o Dynamo DBEncryptor
- Usando o Dynamo DBMapper

Usando o Dynamo DBEncryptor

Este exemplo mostra como usar o Dynamo de nível inferior DBEncryptor com o Direct KMS Provider. O Direct KMS Provider gera e protege seus materiais criptográficos sob um AWS KMS keyin AWS Key Management Service (AWS KMS) especificado por você.

Você pode usar qualquer provedor de materiais criptográficos (CMP) compatível com o. e você pode usar o Direct KMS Provider com e. `DynamoDBEncryptor` `DynamoDBMapper` AttributeEncryptor

Veja a amostra de código completa: AwsKmsEncryptedItem.java

Etapa 1: crie um Direct KMS Provider

Crie uma instância do AWS KMS cliente com a região especificada. Em seguida, use a instância do cliente para criar uma instância de Direct KMS Provider com o AWS KMS key de sua preferência.

Este exemplo usa o Amazon Resource Name (ARN) para identificar o AWS KMS key, mas você pode usar qualquer identificador de chave válido.

```
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);
```

Etapa 2: crie um item

Este exemplo define um `record` HashMap que representa um item de tabela de amostra.

```
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"));
```

Etapa 3: criar um Dynamo DBEncryptor

Crie uma instância do `DynamoDBEncryptor` com o Direct KMS Provider.

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

Etapa 4: crie um contexto de criptografia do DynamoDB

O Contexto de criptografia do DynamoDB contém informações sobre a estrutura da tabela e de como ela é criptografada e assinada. Se você usar o `DynamoDBMapper`, o `AttributeEncryptor` cria o contexto de criptografia para você.

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

Etapa 5: crie o objeto de ações de atributo

As Ações de atributos determinam os atributos do item que são criptografados e assinados, que são somente assinados e que não são criptografados nem assinados.

Em Java, para especificar ações de atributos, você cria pares HashMap de nome e `EncryptionFlags` valor do atributo.

Por exemplo, o código Java a seguir cria um `actions` HashMap que criptografa e assina todos os atributos no `record` item, exceto os atributos da chave de partição e da chave de classificação, que são assinados, mas não criptografados, e o `test` atributo, que não está assinado ou criptografado.

```
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;
  }
}
```

Etapa 6: criptografe e assine o item

Para criptografar e assinar o item da tabela, chame o método `encryptRecord` na instância do `DynamoDBEncryptor`. Especifique o item da tabela (`record`), as ações de atributo (`actions`) e o contexto de criptografia (`encryptionContext`).

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

Etapa 7: coloque o item na tabela do DynamoDB

Finalmente, coloque o item criptografado e assinado na tabela do DynamoDB.

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

Usando o Dynamo DBMapper

O exemplo a seguir mostra como usar a classe auxiliar do mapeador do DynamoDB com o Direct KMS Provider. O Direct KMS Provider gera e protege seus materiais criptográficos sob um AWS KMS key no AWS Key Management Service (AWS KMS) especificado por você.

Você pode usar qualquer provedor de materiais de criptografia (CMP) compatível com o `DynamoDBMapper`, e usar o Direct KMS Provider com o `DynamoDBEncryptor` de baixo nível.

Veja a amostra de código completa: AwsKmsEncryptedObject.java

Etapa 1: crie um Direct KMS Provider

Crie uma instância do AWS KMS cliente com a região especificada. Em seguida, use a instância do cliente para criar uma instância de Direct KMS Provider com o AWS KMS key de sua preferência.

Este exemplo usa o Amazon Resource Name (ARN) para identificar o AWS KMS key, mas você pode usar qualquer identificador de chave válido.

```
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);
```

Etapa 2: criar o DynamoDB Encryptor e o Dynamo DBMapper

Use o Direct KMS Provider que você criou na etapa anterior para criar uma instância do DynamoDB Encryptor. Você precisa instanciar o DynamoDB Encryptor de nível inferior para usar o DynamoDB Mapper.

Em seguida, crie uma instância de seu banco de dados do DynamoDB e uma configuração de mapeador e use-as para criar uma instância do Mapeador do DynamoDB.

> **Important**
>
> Ao usar o `DynamoDBMapper` para adicionar ou editar itens assinados (ou criptografados e assinados), configure-o para usar um comportamento de salvamento, como PUT, que inclua todos os atributos, conforme mostrado no exemplo a seguir. Caso contrário, talvez você não possa descriptografar os dados.

```
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));
```

Etapa 3: Definir a tabela do DynamoDB

Em seguida, defina sua tabela do DynamoDB. Use anotações para especificar as ações de atributos. Este exemplo cria uma tabela do DynamoDB, `ExampleTable`, e uma classe `DataPoJo` que representa itens da tabela.

Nessa tabela de exemplo, os atributos de chave primária serão assinados, mas não criptografados. Isso se aplica ao `partition_attribute`, que é anotado com a `@DynamoDBHashKey`, e ao `sort_attribute`, que é anotado com a `@DynamoDBRangeKey`.

Os atributos que são anotadas com o `@DynamoDBAttribute`, como o `some numbers`, serão criptografados e assinados. As exceções são os atributos que usam as anotações de criptografia `@DoNotEncrypt` (apenas assinar) ou `@DoNotTouch` (não criptografar nem assinar) definidos pelo DynamoDB Encryption Client. Por exemplo, como o atributo `leave me` tem uma anotação `@DoNotTouch`, ele não será criptografado nem assinado.

```
@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 +
 "]";
  }
}
```

Etapa 4: Criptografar e salvar um item da tabela

Agora, quando você cria um item da tabela e usa o Mapeador do DynamoDB para salvá-lo, o item é automaticamente criptografado e assinado antes de ser adicionado à tabela.

Este exemplo define um item da tabela chamado `record`. Antes de serem salvos na tabela, seus atributos são criptografados e assinados com base nas anotações na classe `DataPoJo`. Nesse caso, todos os atributos, com exceção de `PartitionAttribute`, `SortAttribute` e `LeaveMe` são criptografados e assinados. O `PartitionAttribute` e `SortAttributes` são só assinados. O atributo `LeaveMe` não é criptografado nem assinado.

Para criptografar e assinar o item `record` e, em seguida, adicioná-lo à `ExampleTable`, chame o método `save` da classe `DynamoDBMapper`. Como o DynamoDB Mapper é configurado para usar o comportamento de salvamento de PUT, o item substitui qualquer item com as mesmas chaves primárias, em vez de atualizá-los. Isso garante que as assinaturas correspondam e que você possa descriptografar o item ao obtê-lo da tabela.

```
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 para Python

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Este tópico explica como instalar e usar o DynamoDB Encryption Client para Python. Você pode encontrar o código no aws-dynamodb-encryption-pythonrepositório em GitHub, incluindo um código de amostra completo e testado para ajudar você a começar.

> **Note**
>
> Versões 1. x. x e 2. x. x do DynamoDB Encryption Client para Python estão end-of-support em fase a partir de julho de 2022. Atualize para uma versão mais recente o mais rápido possível.

Tópicos

- Pré-requisitos
- Instalação
- Uso do DynamoDB Encryption Client para Python
- Código de exemplo para o DynamoDB Encryption Client para Python

Pré-requisitos

Antes de instalar o Amazon DynamoDB Encryption Client para Python, verifique se você tem os pré-requisitos a seguir.

Uma versão compatível do Python

O Python 3.8 ou posterior é exigido pelo Amazon DynamoDB Encryption Client para Python nas versões 3.3.0 e posteriores. Para fazer download do Python, consulte Downloads do Python.

As versões anteriores do Amazon DynamoDB Encryption Client for Python oferecem suporte ao Python 2.7 e ao Python 3.4 e versões posteriores, mas recomendamos que você use a versão mais recente do DynamoDB Encryption Client.

A ferramenta de instalação do pip para Python

O Python 3.6 e versões posteriores incluem pip, embora você possa querer atualizá-lo. Para obter mais informações sobre a atualização ou a instalação do pip, consulte Installation na documentação do pip.

Instalação

Use o pip para instalar o Amazon DynamoDB Encryption Client para Python, conforme mostrado nos exemplos a seguir.

Para instalar a versão mais recente

```
pip install dynamodb-encryption-sdk
```

Para obter mais detalhes sobre o uso do pip para instalar e atualizar pacotes, consulte Installing Packages.

O DynamoDB Encryption Client requer a biblioteca de criptografia em todas as plataformas. Todas as versões do pip instalam e criam a biblioteca de criptografia no Windows e no OS X. pip 8.1 e posterior instala e cria a criptografia no Linux. Se estiver usando uma versão anterior do pip, e seu ambiente Linux não tiver as ferramentas necessárias para criar a biblioteca de criptografia, será necessário instalá-las. Para obter mais informações, consulte Criação de criptografia no Linux.

Você pode obter a versão de desenvolvimento mais recente do DynamoDB Encryption Client no repositório em. aws-dynamodb-encryption-python GitHub

Depois de instalar o DynamoDB Encryption Client, veja o código de exemplo do Python neste guia.

Uso do DynamoDB Encryption Client para Python

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Este tópico explica alguns dos recursos do DynamoDB Encryption Client para Python que talvez não sejam encontrados em outras implementações de linguagem de programação. Esses atributos são projetados para facilitar o uso do DynamoDB Encryption Client da forma mais confiável possível. A menos que você tenha um caso de uso incomum, recomendamos que você os use.

Para obter detalhes sobre a programação com o DynamoDB Encryption Client, consulte os exemplos do Python neste guia, os exemplos no repositório GitHub e a documentação do Python para aws-dynamodb-encryption-python o DynamoDB Encryption Client.

Tópicos

- Classes auxiliares do cliente
- TableInfo classe
- Ações de atributos em Python

Classes auxiliares do cliente

O DynamoDB Encryption Client para Python inclui várias classes auxiliares do cliente que espelham as classes do Boto 3 para o DynamoDB. Essas classes auxiliares são projetadas para facilitar a adição da criptografia e da assinatura ao seu aplicativo DynamoDB existente e evitar os problemas mais comuns:

- Evite que você criptografe a chave primária em seu item, seja adicionando uma ação de substituição da chave primária ao AttributeActionsobjeto ou lançando uma exceção se seu `AttributeActions` objeto solicitar explicitamente ao cliente que criptografe a chave primária. Se a ação padrão no objeto `AttributeActions` for `DO_NOTHING`, as classes auxiliares do cliente usarão a ação para a chave primária. Caso contrário, eles usarão `SIGN_ONLY`.
- Crie um TableInfo objeto e preencha o contexto de criptografia do DynamoDB com base em uma chamada para o DynamoDB. Isso ajuda a garantir que o contexto de criptografia do DynamoDB seja preciso e o cliente possa identificar a chave primária.
- Métodos de suporte, como `put_item` e `get_item`, que criptografam e descriptografam de modo transparente os itens da tabela quando você grava ou lê em uma tabela do DynamoDB. Somente o método `update_item` não é compatível.

É possível usar a classe auxiliar do cliente em vez de interagir diretamente com o criptografador de itens de nível inferior. Use essas classes a menos que você precise definir opções avançadas no criptografador do item.

As classes auxiliares do cliente incluem:

- EncryptedTablepara aplicativos que usam o recurso Tabela no DynamoDB para processar uma tabela por vez.

- EncryptedResourcepara aplicativos que usam a classe Service Resource no DynamoDB para processamento em lote.
- EncryptedClientpara aplicativos que usam o cliente de nível inferior no DynamoDB.

Para usar as classes auxiliares do cliente, o chamador deve ter permissão para chamar a operação do DynamoDB na tabela de destino. DescribeTable

TableInfo classe

A TableInfoclasse é uma classe auxiliar que representa uma tabela do DynamoDB, completa com campos para sua chave primária e índices secundários. Com ela, você pode obter informações precisas e em tempo real sobre a tabela.

Se você utilizar uma classe auxiliar do cliente, ela criará e usará um objeto `TableInfo` para você. Caso contrário, você pode criar um explicitamente. Para obter um exemplo, consulte Usar o criptografador de item.

Quando você chama o `refresh_indexed_attributes` método em um `TableInfo` objeto, ele preenche os valores da propriedade do objeto chamando a operação do DynamoDB. DescribeTable Consultar a tabela é muito mais confiável que consultar os nomes de índice de hard-coding. A classe `TableInfo` também inclui uma propriedade `encryption_context_values` que fornece os valores necessários para o contexto de criptografia do DynamoDB.

Para usar o `refresh_indexed_attributes` método, o chamador deve ter permissão para chamar a operação do DescribeTableDynamoDB na tabela de destino.

Ações de atributos em Python

As Ações de atributos informam ao criptografador de itens quais ações executar em cada atributo de item. Para especificar ações de atributo em Python, crie um objeto `AttributeActions` com uma ação padrão e todas as exceções dos atributos específicos. Os valores válidos estão definidos no tipo enumerado `CryptoAction`.

> **Important**
>
> Depois de usar as ações do atributo para criptografar os itens da tabela, adicionar ou remover atributos do modelo de dados poderá gerar um erro de validação de assinatura que impede a descriptografia dos dados. Para obter uma explicação detalhada, consulte Alterar seu modelo de dados.

```
DO_NOTHING = 0
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

Por exemplo, o objeto `AttributeActions` estabelece `ENCRYPT_AND_SIGN` como o padrão para todos os atributos e define as exceções para os atributos `ISBN` e `PublicationYear`.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

Se você usar uma classe auxiliar do cliente, não será necessário especificar uma ação de atributo para os atributos de chave primária. As classes auxiliares do cliente evitam que você criptografe sua chave primária.

Se você não utiliza uma classe auxiliar do cliente e a ação padrão é `ENCRYPT_AND_SIGN`, é necessário especificar uma ação para a chave primária. A ação recomendada para chaves primárias é `SIGN_ONLY`. Para facilitar esse procedimento, use o método `set_index_keys`, que usa SIGN_ONLY para chaves primárias ou DO_NOTHING, quando essa é a ação padrão.

> **Warning**
>
> Não criptografe os atributos da chave primária. Eles devem permanecer em texto simples para que o DynamoDB possa encontrar o item sem executar uma varredura completa da tabela.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Código de exemplo para o DynamoDB Encryption Client para Python

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Os exemplos a seguir mostram como usar o DynamoDB Encryption Client para Python para proteger os dados do DynamoDB no aplicativo. Você pode encontrar mais exemplos (e contribuir com os seus) no diretório de exemplos do aws-dynamodb-encryption-pythonrepositório em GitHub.

Tópicos

- Use a classe auxiliar EncryptedTable do cliente
- Usar o criptografador de item

Use a classe auxiliar EncryptedTable do cliente

O exemplo a seguir mostra como usar o Direct KMS Provider com a `EncryptedTable` classe auxiliar do cliente. Este exemplo usa o mesmo provedor de materiais de criptografia que o Usar o criptografador de item exemplo a seguir. No entanto, ele usa a classe `EncryptedTable` em vez de interagir diretamente com o criptografador de itens de nível inferior.

Comparando esses casos, você poderá visualizar o trabalho que a classe auxiliar do cliente faz para você. Isso inclui a criação do Contexto de criptografia do DynamoDB e a verificação de que os atributos de chave primária são sempre assinados, mas nunca criptografados. Para criar o contexto de criptografia e descobrir a chave primária, as classes auxiliares do cliente chamam a operação do DynamoDB. DescribeTable Para executar esse código, você deve ter permissão para chamar essa operação.

Consulte o exemplo de código completo: aws_kms_encrypted_table.py

Etapa 1: crie a tabela

Comece criando uma instância de uma tabela padrão do DynamoDB com o nome da tabela.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Etapa 2: crie um provedor de materiais de criptografia

Crie uma instância do provedor de materiais de criptografia (CMP) que você selecionou.

Este exemplo usa o Direct KMS Provider, mas você pode usar qualquer CMP compatível. Para criar um Direct KMS Provider, especifique um AWS KMS key. Este exemplo usa o Amazon Resource Name (ARN) do AWS KMS key, mas você pode usar qualquer identificador de chave válido.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Etapa 3: crie o objeto de ações de atributo

As Ações de atributos informam ao criptografador de itens quais ações executar em cada atributo de item. O objeto `AttributeActions` neste exemplo criptografa e assina todos os itens exceto o atributo `test`, que é ignorado.

Não especifique ações de atributo para os atributos de chave primária ao usar uma classe auxiliar do cliente. A classe `EncryptedTable` assina, mas nunca criptografa os atributos de chave primária.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

Etapa 4: crie a tabela criptografada

Crie a tabela criptografada usando a tabela padrão, o Direct KMS Provider e as ações de atributo. Essa etapa conclui a configuração.

```
encrypted_table = EncryptedTable(
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
```

```
)
```

Etapa 5: coloque o item de texto simples na tabela

Ao chamar o método `put_item` no `encrypted_table`, os itens da tabela são criptografados, assinados e adicionados à tabela do DynamoDB de maneira transparente.

Primeiro, defina o item da tabela.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Em seguida, coloque-o na tabela.

```
encrypted_table.put_item(Item=plaintext_item)
```

Para obter o item da tabela do `get_item` na forma criptografada, chame o método no objeto `table`. Para obter o item descriptografado, chame o método `get_item` no objeto `encrypted_table`.

Usar o criptografador de item

Este exemplo mostra como interagir diretamente com o criptografador de itens no DynamoDB Encryption Client ao criptografar itens de tabela, em vez de usar as classes auxiliares do cliente que interagem com o criptografador de itens para você.

Ao usar essa técnica, crie o contexto de criptografia e o objeto de configuração (`CryptoConfig`) do DynamoDB manualmente. Além disso, criptografe os itens em uma chamada e coloque-os na tabela do DynamoDB em uma chamada separada. Isso permite personalizar suas chamadas do `put_item` e usar o DynamoDB Encryption Client para criptografar e assinar dados estruturados que nunca são enviados ao DynamoDB.

Este exemplo usa o Direct KMS Provider, mas você pode usar qualquer CMP compatível.

Consulte o exemplo de código completo: aws_kms_encrypted_item.py

Etapa 1: crie a tabela

Comece criando uma instância de um recurso de tabela padrão do DynamoDB com o nome da tabela.

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

Etapa 2: crie um provedor de materiais de criptografia

Crie uma instância do provedor de materiais de criptografia (CMP) que você selecionou.

Este exemplo usa o Direct KMS Provider, mas você pode usar qualquer CMP compatível. Para criar um Direct KMS Provider, especifique um AWS KMS key. Este exemplo usa o Amazon Resource Name (ARN) do AWS KMS key, mas você pode usar qualquer identificador de chave válido.

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

Etapa 3: usar a TableInfo classe auxiliar

Para obter informações sobre a tabela do DynamoDB, crie uma instância da TableInfoclasse auxiliar. Ao trabalhar diretamente com o criptografador de item, você precisa criar uma instância `TableInfo` e chamar seus métodos. As classes auxiliares do cliente fazem isso para você.

O `refresh_indexed_attributes` método de `TableInfo` usa a operação do DescribeTableDynamoDB para obter informações precisas e em tempo real sobre a tabela. Isso inclui sua chave primária e seus índices secundários locais e globais. O chamador precisa ter permissão para chamar `DescribeTable`.

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

Etapa 4: crie o contexto de criptografia do DynamoDB

O Contexto de criptografia do DynamoDB contém informações sobre a estrutura da tabela e de como ela é criptografada e assinada. Este exemplo cria um contexto de criptografia do

DynamoDB explicitamente, pois interage com o criptografador de item. As classes auxiliares do cliente criam o contexto de criptografia do DynamoDB para você.

Para obter a chave de partição e a chave de classificação, você pode usar as propriedades da classe TableInfoauxiliar.

```
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)
)
```

Etapa 5: crie o objeto de ações de atributo

As Ações de atributos informam ao criptografador de itens quais ações executar em cada atributo de item. O objeto `AttributeActions` neste exemplo criptografa e assina todos os itens, exceto para os atributos de chave primária, que são assinados, mas não criptografados, e o atributo `test`, que é ignorado.

Ao interagir diretamente com o criptografador de item, e a ação padrão ser `ENCRYPT_AND_SIGN`, você deve especificar uma ação alternativa para a chave primária. Você pode usar o método `set_index_keys`, que utiliza `SIGN_ONLY` para a chave primária ou usa `DO_NOTHING` se é o padrão.

Para especificar a chave primária, este exemplo usa as chaves de índice no TableInfoobjeto, que são preenchidas por uma chamada para o DynamoDB. Esta técnica é mais segura do que nomes de chave primária de hard-code.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
actions.set_index_keys(*table_info.protected_index_keys())
```

Etapa 6: crie a configuração para o item

Para configurar o DynamoDB Encryption Client, use os objetos que você acabou de criar em CryptoConfiguma configuração para o item da tabela. As classes auxiliares do cliente criam o CryptoConfig para você.

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

Etapa 7: criptografe o item

Essa etapa criptografa e assina o item, mas não o coloca em uma tabela do DynamoDB.

Quando você usa uma classe auxiliar do cliente, seus itens são criptografados e assinados de maneira transparente e, em seguida, adicionados à tabela do DynamoDB quando você chama o método `put_item` da classe auxiliar. Ao usar o criptografador de item diretamente, as ações de criptografia e colocação são independentes.

Primeiro, crie um item de texto simples.

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

Em seguida, criptografe e assine o item. O método `encrypt_python_item` requer o objeto de configuração `CryptoConfig`.

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

Etapa 8: coloque o item na tabela

Essa etapa coloca o item criptografado e assinado na tabela do DynamoDB.

```
table.put_item(Item=encrypted_item)
```

Para visualizar o item criptografado, chame o método `get_item` no objeto original `table`, em vez do objeto `encrypted_table`. Ele obtém o item de tabela do DynamoDB sem verificá-la e descriptografá-la.

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

A imagem a seguir mostra parte de um item de tabela criptografado e assinado.

Os valores de atributos criptografados são dados binários. Os nomes e valores de atributos de chave primária (`partition_attribute` e `sort_attribute`) e o atributo `test` permanecem em texto simples. A saída também mostra o atributo que contém a assinatura (`*amzn-ddb-map-sig*`) e o atributo de descrição de materiais (`*amzn-ddb-map-desc*`).

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

Alterar seu modelo de dados

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Sempre que criptografa ou descriptografa um item, você precisa fornecer dações de atributos que informam ao DynamoDB Encryption Client quais atributos criptografar e assinar, quais assinar (mas não criptografar) e quais ignorar. As ações de atributos não são salvas no item criptografado e o DynamoDB Encryption Client não atualiza as ações de atributos automaticamente.

> **Important**
>
> O DynamoDB Encryption Client não oferece suporte à criptografia de dados de tabela do DynamoDB existentes e não criptografados.

Sempre que alterar seu modelo de dados, ou seja, ao adicionar ou remover atributos de seus itens de tabela, você corre o risco de um erro. Se as ações de atributo especificadas por você não justificam todos os atributos no item, ele não poderá ser criptografado nem assinado como você deseja. O mais importante é que se as ações dos atributos fornecidas por você ao descriptografar um item forem diferentes das ações de atributos fornecidas ao criptografar o item, poderá ocorrer uma falha na verificação da assinatura.

Por exemplo, se as ações de atributo usadas para criptografar o item o instruem a assinar o atributo `test`, a assinatura no item incluirá o atributo `test`. Mas se as ações de atributo usadas para descriptografar o item não justificam o atributo `test`, ocorrerá uma falha na verificação porque o cliente tentará verificar uma assinatura que não inclui o atributo `test`.

Esse é um problema específico quando vários aplicativos leem e gravam os mesmos itens do DynamoDB porque o DynamoDB Encryption Client precisa calcular a mesma assinatura para itens em todos os aplicativos. Também é um problema para qualquer aplicativo distribuído porque as alterações nas ações de atributos devem ser propagadas para todos os hosts. Mesmo que suas tabelas do DynamoDB sejam acessadas por um único host em um processo, o estabelecimento de um processo de melhores práticas ajudará a evitar erros se o projeto se tornar mais complexo.

Para evitar erros de validação de assinatura que impedem a leitura de itens de tabela, use as orientações a seguir.

- Adicionar um atributo — Se o novo atributo alterar as ações de atributo, implante totalmente a alteração da ação de atributo antes de incluir o novo atributo em um item.
- Remover um atributo — Se você parar de usar um atributo nos itens, não altere as suas ações de atributo.

- Alterar a ação — Depois de usar uma configuração de ações de atributo para criptografar os itens da tabela, não será possível alterar com segurança a ação padrão ou a ação de um atributo existente sem recriptografar cada item da tabela.

Erros de validação de assinatura podem ser extremamente difíceis de resolver, portanto, a melhor abordagem é evitá-los.

Tópicos

- Adicionar um atributo
- Remover um atributo

Adicionar um atributo

Ao adicionar um novo atributo a itens de tabela, talvez seja necessário alterar as ações de seus atributos. Para evitar erros de validação de assinatura, é recomendável implementar essa alteração em um processo de dois estágios. Verifique se o primeiro estágio está completo antes de iniciar o segundo estágio.

1. Altere as ações de atributos em todos os aplicativos que leem ou gravam na tabela. Implante essas alterações e confirme se a atualização foi propagada para todos os hosts de destino.
2. Grave valores para o novo atributo em seus itens de tabela.

Essa abordagem em dois estágios garante que todos os aplicativos e hosts tenham as mesmas ações de atributos e calculará a mesma assinatura, antes de qualquer encontro com o novo atributo. Isso é importante mesmo quando a ação do atributo for Não fazer nada (não criptografar ou assinar), porque o padrão para alguns criptografadores é criptografar e assinar.

Os exemplos a seguir mostram o código para o primeiro estágio desse processo. Eles adicionam um novo atributo de item, `link`, que armazena um link para outro item da tabela. Como esse link deve permanecer em texto simples, o exemplo atribui a ele a ação somente assinar. Depois de implantar totalmente essa alteração e verificar se todos os aplicativos e hosts têm as novas ações de atributos, é possível começar a usar o atributo `link` em seus itens de tabela.

Java DynamoDB Mapper

> Por padrão, ao usar o `DynamoDB Mapper` e o `AttributeEncryptor`, todos os atributos são criptografados e assinados, exceto as chaves primárias que são assinadas, mas não

são criptografadas. Para especificar uma ação de somente assinatura, use a anotação `@DoNotEncrypt`.

Este exemplo usa a anotação `@DoNotEncrypt` para o novo atributo `link`.

```
@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

No DynamoDB Encryptor de nível inferior, você deve definir ações para cada atributo. Este exemplo usa uma instrução switch em que o padrão é `encryptAndSign` e exceções são especificadas para a chave de partição, a chave de classificação e o novo atributo `link`. Neste exemplo, se o código do atributo link não for totalmente implantado antes de ser usado, o atributo link será criptografado e assinado por alguns aplicativos, mas somente assinado por outros.

```
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

No DynamoDB Encryption Client para Python, é possível especificar uma ação padrão para todos os atributos e especificar exceções.

Se você usa uma classe auxiliar do cliente do Python, não será necessário especificar uma ação de atributo para os atributos de chave primária. As classes auxiliares do cliente evitam que você criptografe sua chave primária. No entanto, se você não estiver usando uma classe auxiliar do cliente, defina a ação SIGN_ONLY em sua chave de partição e em sua chave de classificação. Se você criptografar acidentalmente sua chave de partição ou de classificação, não será possível recuperar seus dados sem uma verificação completa da tabela.

Este exemplo especifica uma exceção para o novo atributo `link`, que obtém a ação `SIGN_ONLY`.

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

Remover um atributo

Se você não precisar mais de um atributo em itens que foram criptografados com o DynamoDB Encryption Client, poderá parar de usar o atributo. No entanto, não exclua nem altere a ação desse atributo. Se o fizer e depois encontrar um item com esse atributo, a assinatura calculada para o item não corresponderá à assinatura original, e a validação da assinatura falhará.

Embora você possa ser tentado a remover todos os traços do atributo do seu código, adicione um comentário informando que o item não é mais usado em vez de excluí-lo. Mesmo que você faça uma verificação de tabela completa para excluir todas as instâncias do atributo, um item criptografado com esse atributo pode ser armazenado em cache ou em processo em algum lugar da configuração.

Solução de problemas em seu aplicativo DynamoDB Encryption Client

> **Note**
>
> Nossa biblioteca de criptografia do lado do cliente foi renomeada como SDK de criptografia de banco de dados da AWS. O tópico a seguir fornece informações sobre as versões 1.x—2.x do DynamoDB Encryption Client para Java e versões 1.x—3x do DynamoDB Encryption Client para Python. Para obter mais informações, consulte SDK de criptografia de banco de dados da AWS para obter suporte à versão do DynamoDB.

Esta seção descreve os problemas que você pode encontrar ao usar o DynamoDB Encryption Client e oferece sugestões para resolvê-los.

Para fornecer feedback sobre o DynamoDB Encryption Client, registre um problema no aws-dynamodb-encryption-javarepositório or. aws-dynamodb-encryption-python GitHub

Para fornecer feedback sobre esta documentação, use o link de feedback em qualquer página.

Tópicos

- Acesso negado
- Falhas na verificação da assinatura
- Problemas com tabelas globais de versões mais antigas
- Baixo desempenho do fornecedor mais recente

Acesso negado

Problema: o acesso a um recurso necessário é negado ao aplicativo.

Sugestão: saiba mais sobre as permissões necessárias e adicione-as ao contexto de segurança em que o aplicativo é executado.

Detalhes

Para executar um aplicativo que usa uma biblioteca do DynamoDB Encryption Client, o chamador deve ter permissão para usar os componentes. Caso contrário, eles terão o acesso negado aos elementos necessários.

- O DynamoDB Encryption Client não exige uma conta Amazon Web Services (AWS) nem depende de nenhum serviço da AWS . No entanto, se seu aplicativo usa AWS, você precisa de um usuário Conta da AWS e que tenham permissão para usar a conta.
- O DynamoDB Encryption Client não exige o Amazon DynamoDB. No entanto, se o aplicativo que usa o cliente criar tabelas do DynamoDB, colocar os itens em uma tabela ou obtiver itens de uma tabela, o chamador deverá ter permissão para usar as operações do DynamoDB necessárias em sua Conta da AWS. Para obter detalhes, consulte os tópicos de controle de acesso no Amazon DynamoDB Developer Guide.
- Se seu aplicativo usa uma classe auxiliar de cliente no DynamoDB Encryption Client for Python, o chamador deverá ter permissão para chamar a operação do DynamoDB. DescribeTable
- O DynamoDB Encryption Client não AWS Key Management Service exige ().AWS KMS No entanto, se seu aplicativo usa um provedor direto de materiais KMS ou usa um provedor mais recente com uma loja de provedores que usa AWS KMS, o chamador deve ter permissão para usar as operações AWS KMS GenerateDataKeye Decrypt.

Falhas na verificação da assinatura

Problema: um item não pode ser descriptografado porque ocorre uma falha na verificação de assinatura. O item também pode não ser criptografado e assinado como você deseja.

Sugestão: verifique se as ações de atributos fornecidas justificam todos os atributos no item. Ao descriptografar um item, forneça as ações de atributos que correspondam às ações usadas para criptografar o item.

Detalhes

As ações de atributos fornecidas informam ao DynamoDB Encryption Client quais atributos criptografar e assinar, quais atributos assinar (mas não criptografar) e quais ignorar.

Se as ações de atributo especificadas por você não justificam todos os atributos no item, ele não poderá ser criptografado nem assinado como você deseja. Se as ações de atributo fornecidas por você ao descriptografar um item são diferentes das ações de atributo fornecidas por você ao criptografar o item, pode ocorrer uma falha na verificação de assinatura. Este é um problema específico para aplicativos distribuídos em que novas ações de atributo talvez não tenham sigo propagadas para todos os hosts.

Erros de validação de assinatura são difíceis de resolver. Para ajudar a evitá-los, tome precauções extras ao alterar seu modelo de dados. Para obter detalhes, consulte Alterar seu modelo de dados.

Problemas com tabelas globais de versões mais antigas

Problema: os itens em uma versão mais antiga da tabela global do Amazon DynamoDB não podem ser descriptografados porque a verificação da assinatura falha.

Sugestão: defina ações de atributo para que os campos de replicação reservados não sejam criptografados ou assinados.

Detalhes

É possível usar o DynamoDB Encryption Client com as tabelas globais do DynamoDB. Recomendamos que você use tabelas globais com uma chave KMS multirregional e replique a chave KMS em todos os Regiões da AWS lugares onde a tabela global é replicada.

A partir da versão 2019.11.21 de tabelas globais, é possível usar tabelas globais com o DynamoDB Encryption Client sem nenhuma configuração especial. No entanto, se você usar tabelas

globais versão 2017.11.29, deverá garantir que os campos de replicação reservados não sejam criptografados ou assinados.

Se você estiver usando a versão de tabelas globais 2017.11.29, deverá definir as ações de atributo para os atributos a seguir `DO_NOTHING` em Java ou `@DoNotTouch` em Python.

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

Se você estiver usando qualquer outra versão das tabelas globais, nenhuma ação será necessária.

Baixo desempenho do fornecedor mais recente

Problema: seu aplicativo responde menos, especialmente após a atualização para uma versão mais recente do DynamoDB Encryption Client.

Sugestão: ajuste o time-to-live valor e o tamanho do cache.

Detalhes

O provedor mais recente foi projetado para melhorar o desempenho dos aplicativos que usam o DynamoDB Encryption Client, permitindo a reutilização limitada de materiais criptográficos. Ao configurar o provedor mais recente para seu aplicativo, você precisa equilibrar o desempenho aprimorado com as preocupações de segurança decorrentes do armazenamento em cache e da reutilização.

Nas versões mais recentes do DynamoDB Encryption Client, time-to-live o valor (TTL) determina por quanto tempo os provedores de material criptográfico em cache () podem ser usados. CMPs O TTL também determina com que frequência o provedor mais recente verifica em busca de uma nova versão do CMP.

Se o TTL for muito longo, seu aplicativo poderá violar suas regras de negócios ou padrões de segurança. Se o TTL for muito breve, as chamadas frequentes para o armazenamento do provedor podem fazer com que o armazenamento do provedor reduza as solicitações do seu aplicativo e de outros aplicativos que compartilham sua conta de serviço. Para resolver esse problema, ajuste o TTL e o tamanho do cache para um valor que atenda às suas metas de latência e disponibilidade e esteja em conformidade com seus padrões de segurança. Para obter mais detalhes, consulte Definindo um time-to-live valor.

Renomeação do Amazon DynamoDB Encryption Client

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Em 9 de junho de 2023, nossa biblioteca de criptografia do lado do cliente foi renomeada para AWS Database Encryption SDK. O SDK AWS de criptografia de banco de dados é compatível com o Amazon DynamoDB. Ele pode descriptografar e ler itens criptografados pelo antigo DynamoDB Encryption Client. Para obter mais informações sobre as versões antigas do DynamoDB Encryption Client, consulte AWS Suporte à versão SDK de criptografia de banco de dados para DynamoDB.

O SDK AWS de criptografia de banco de dados fornece a versão 3. x da biblioteca de criptografia Java do lado do cliente para o DynamoDB, que é uma grande reescrita do DynamoDB Encryption Client for Java. Ela inclui muitas atualizações, como um novo formato de dados estruturados, suporte aprimorado para multilocação, alterações de esquema contínuas e suporte à criptografia pesquisável.

Para saber mais sobre os novos recursos introduzidos com o SDK AWS de criptografia de banco de dados, consulte os tópicos a seguir.

Criptografia pesquisável

É possível criar bancos de dados capazes de pesquisar registros criptografados sem descriptografar o banco de dados inteiro. Dependendo do modelo de ameaça e dos requisitos de consulta, você pode usar criptografia pesquisável para realizar pesquisas de correspondência exata ou consultas complexas mais personalizadas em seu registro criptografado.

Tokens de autenticação

O SDK AWS de criptografia de banco de dados usa chaveiros para realizar a criptografia de envelopes. Os tokens de autenticação geram, criptografam e descriptografam as chaves de dados que protegem seus registros. O SDK AWS de criptografia de banco de dados suporta AWS KMS chaveiros que usam criptografia simétrica ou RSA assimétrica AWS KMS keyspara proteger suas chaves de dados e AWS KMS chaveiros hierárquicos que permitem proteger seus materiais criptográficos sob uma chave KMS de criptografia simétrica sem ligar toda vez que você criptografa ou descriptografa um registro. AWS KMS Também é possível especificar seu próprio material de chave com tokens de autenticação AES brutos e tokens de autenticação RSA brutos.

Mudanças de esquema simplificadas

Ao configurar o SDK AWS de criptografia de banco de dados, você fornece ações criptográficas que informam ao cliente quais campos criptografar e assinar, quais campos assinar (mas não criptografar) e quais ignorar. Depois de usar o SDK AWS de criptografia de banco de dados para proteger seus registros, você ainda pode fazer alterações em seu modelo de dados. É possível atualizar ações criptográficas, como adicionar ou remover campos criptografados, em uma única implantação.

Configurar tabelas do DynamoDB existentes para criptografia do lado do cliente

As versões antigas do DynamoDB Encryption Client foram projetadas para serem implementadas em tabelas novas e não preenchidas. Com o SDK AWS de criptografia de banco de dados para DynamoDB, você pode migrar suas tabelas existentes do Amazon DynamoDB para a versão 3. x da biblioteca de criptografia Java do lado do cliente para o DynamoDB.

Referência

Nossa biblioteca de criptografia do lado do cliente foi renomeada para SDK de criptografia de AWS banco de dados. Este guia do desenvolvedor ainda fornece informações sobre o DynamoDB Encryption Client.

Os tópicos a seguir fornecem detalhes técnicos do SDK AWS de criptografia de banco de dados.

Formato de descrição do material

A descrição do material serve como cabeçalho para um registro criptografado. Quando você criptografa e assina campos com o SDK AWS de criptografia de banco de dados, o criptografador registra a descrição do material à medida que reúne os materiais criptográficos e armazena a descrição do material em um novo campo (`aws_dbe_head`) que o criptografador adiciona ao seu registro. A descrição do material é uma estrutura de dados formatada portátil que contém a chave de dados criptografada e informações sobre como o registro foi criptografado e assinado. A tabela a seguir descreve os valores que formam a descrição do material. Os bytes são anexados na ordem mostrada.

Valor	Tamanho em bytes
Version	1
Signatures Enabled	1
Record ID	32
Encrypt Legend	Variável
Encryption Context Length	2
???	Variável
Encrypted Data Key Count	1
Encrypted Data Keys	Variável

Valor	Tamanho em bytes
Record Commitment	1

Versão

A versão desse formato de campo `aws_dbe_head`.

Assinaturas habilitadas

Codifica se as assinaturas digitais ECDSA estão habilitadas para esse registro.

Valor de bytes	Significado
`0x01`	Assinaturas digitais ECDSA ativadas (padrão)
`0x00`	Assinaturas digitais ECDSA desativadas

ID do registro

Um valor de 256 bits gerado aleatoriamente que identifica a mensagem. O ID do registro:

- Identifica de forma exclusiva o registro criptografado.
- Vincula a descrição do material ao registro criptografado.

Criptografar legenda

Uma descrição serializada de quais campos autenticados foram criptografados. O Encrypt Legend é usada para determinar quais campos o método de descriptografia deve tentar descriptografar.

Valor de bytes	Significado
`0x65`	`ENCRYPT_AND_SIGN`
`0x73`	`SIGN_ONLY`

O Encrypt Legend é serializado da seguinte forma:

1. Lexicograficamente pela sequência de bytes que representa seu caminho canônico.
2. Para cada campo, em ordem, anexe um dos valores de byte especificados acima para indicar se esse campo deve ser criptografado.

Tamanho do contexto de criptografia

O tamanho do conteúdo criptografado. É um valor de 2 bytes interpretado como um inteiro não assinado de 16 bits. O comprimento máximo é de 65.535 bytes.

Contexto de criptografia

Um contexto de criptografia é um conjunto de pares de chave-valor que contêm dados autenticados adicionais arbitrários e não secretos.

Quando as assinaturas digitais ECDSA estão habilitadas, o contexto de criptografia contém o par de valores-chave. `{"aws-crypto-footer-ecdsa-key": Qtxt}` `Qtxt`representa o ponto da curva elíptica `Q` comprimido de acordo com a SEC 1 versão 2.0 e, em seguida, codificado em base64.

Contagem de chaves de dados criptografados

O número de chaves de dados criptografadas. É um valor de 1 byte interpretado como um número inteiro não assinado de 8 bits que especifica o número de chaves de dados criptografadas. O número máximo de chaves de dados criptografadas em cada registro é 255.

Chaves de dados criptografadas

Uma sequência de chaves de dados criptografadas. O tamanho da sequência é determinado pelo número de chaves de dados criptografadas e pelo tamanho de cada uma delas. A sequência contém pelo menos uma chave de dados criptografada.

A tabela a seguir descreve os campos que formam cada chave de dados criptografada. Os bytes são anexados na ordem mostrada.

Estrutura da chave de dados criptografada

Campo	Tamanho em bytes
Key Provider ID Length	2
Key Provider ID	Variável. Igual ao valor especificado nos 2 bytes anteriores (tamanho do ID do provedor de chave).

Campo	Tamanho em bytes
Key Provider Information Length	2
Key Provider Information	Variável. Igual ao valor especificado nos 2 bytes anteriores (tamanho das informações do provedor de chave).
Encrypted Data Key Length	2
Encrypted Data Key	Variável. Igual ao valor especificado nos 2 bytes anteriores (tamanho da chave de dados criptografada).

Tamanho do ID do provedor de chaves

O tamanho do identificador do provedor de chave. É um valor de 2 bytes interpretado como um número inteiro não assinado de 16 bits que especifica o número de bytes que contém o ID do provedor de chave.

ID do provedor de chave

O identificador do provedor de chave. É usado para indicar o provedor da chave de dados criptografada e deve ser extensível.

Tamanho das informações do principal provedor

O tamanho das informações do provedor de chave. É um valor de 2 bytes interpretado como um número inteiro não assinado de 16 bits que especifica o número de bytes que contém as informações do provedor de chave.

Informações sobre os principais fornecedores

As informações do provedor de chave. São determinadas pelo provedor de chaves.

Quando você está usando um AWS KMS chaveiro, esse valor contém o Amazon Resource Name (ARN) do. AWS KMS key

Comprimento da chave de dados criptografados

O tamanho da chave de dados criptografada. É um valor de 2 bytes interpretado como um número inteiro não assinado de 16 bits que especifica o número de bytes que contém a chave de dados criptografada.

Chave de dados criptografada

A chave de dados criptografada. É a chave de criptografia dos dados criptografada pelo provedor de chaves.

Compromisso de registro

Um hash distinto de código de autenticação de mensagens baseado em hash (HMAC) de 256 bits calculado sobre todos os bytes anteriores da descrição do material usando a chave de confirmação.

AWS KMS Detalhes técnicos do chaveiro hierárquico

O token de autenticação hierárquico do AWS KMS usa uma chave de dados exclusiva para criptografar cada campo e criptografa cada chave de dados com uma chave de empacotamento exclusiva derivada de uma chave de ramificação ativa. Ele usa uma derivação de chave no modo contador com uma função pseudoaleatória com HMAC SHA-256 para derivar a chave de empacotamento de 32 bytes com as seguintes entradas.

- Um sal aleatório de 16 bytes
- A chave de ramificação ativa
- O valor codificado em UTF-8 para o identificador do provedor de chaves "" aws-kms-hierarchy

O token de autenticação hierárquico usa a chave de empacotamento derivada para criptografar uma cópia da chave de dados em texto simples usando o AES-GCM-256 com uma tag de autenticação de 16 bytes e as seguintes entradas.

- A chave de empacotamento derivada é usada como a chave de cifra AES-GCM
- A chave de dados é usada como mensagem AES-GCM
- Um vetor de inicialização aleatória (IV) de 12 bytes é usado como o AES-GCM IV
- Dados autenticados adicionais (AAD) contendo os seguintes valores serializados.

Valor	Tamanho em bytes	Interpretada como
"aws-kms-hierarchy"	17	Codificada em UTF-8

Valor	Tamanho em bytes	Interpretada como
O identificador de chave de ramificação	Variável	Codificada em UTF-8
A versão da chave de ramificação	16	Codificada em UTF-8
Contexto de criptografia	Variável	Pares de valores-chave com codificação UTF-8

Histórico de documentos do AWS Database Encryption SDK Developer Guide

A tabela a seguir descreve alterações significativas feitas nesta documentação. Além dessas alterações principais, também atualizamos a documentação com frequência para melhorar as descrições e os exemplos e abordar os comentários que você nos envia. Para ser notificado sobre alterações significativas, inscreva-se no feed RSS.

Alteração	Descrição	Data
Novo recurso	Foi adicionada documentação para o AWS KMS chaveiro ECDH e o chaveiro ECDH bruto.	17 de junho de 2024
Versão de disponibilidade geral (GA)	Apresentando o suporte para a biblioteca de criptografia do lado do cliente.NET para o DynamoDB.	17 de janeiro de 2024
Versão de disponibilidade geral (GA)	Documentação atualizada para GA versão 3.x da biblioteca Java de criptografia do lado do cliente para o DynamoDB. **Warning** Não há mais suporte para as chaves de ramificação criadas durante a versão prévia para desenvolvedores.	24 de julho de 2023

Atualização da marca do DynamoDB Encryption Client	A biblioteca de criptografia do lado do cliente foi renomeada para AWS Database Encryptio n SDK.	9 de junho de 2023
Versão de visualização	Documentação adicionada e atualizada para a versão 3.x da biblioteca Java de criptografia do lado do cliente para DynamoDB, que inclui um novo formato de dados estruturados, suporte aprimorado à multilocação, alterações de esquema perfeitas e suporte à criptogra fia pesquisável.	9 de junho de 2023
Alteração na documentação	Substitua o AWS Key Management Service termo chave mestra do cliente (CMK) por uma AWS KMS keychave KMS.	30 de agosto de 2021
Novo recurso	Foi adicionado suporte para chaves multirregionais AWS Key Management Service (AWS KMS). As chaves multirregionais são AWS KMS chaves diferentes Regiões da AWS que podem ser usadas de forma intercambiável porque têm o mesmo ID de chave e material de chave.	8 de junho de 2021
Novo exemplo	Foi adicionado um exemplo de uso do Dynamo DBMapper em Java.	6 de setembro de 2018

Suporte ao Python	Adicionado suporte para Python, além do Java.	2 de maio de 2018
Lançamento inicial	Versão inicial desta documentação.	2 de maio de 2018

As traduções são geradas por tradução automática. Em caso de conflito entre o conteúdo da tradução e da versão original em inglês, a versão em inglês prevalecerá.