Guia do Desenvolvedor

AWS Encryption SDK

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Encryption SDK: Guia do Desenvolvedor

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

As marcas comerciais e imagens de marcas 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 AWS Encryption SDK? 1
Desenvolvido em repositórios de código aberto 2
Compatibilidade com bibliotecas e serviços de criptografia 3
Suporte e manutenção 4
Saiba mais 4
Enviar comentários 5
Conceitos 6
criptografia envelopada 7
Chave de dados 9
Chave de encapsulamento 9
Tokens de autenticação e provedores de chaves mestras 10
Contexto de criptografia 12
Mensagem criptografada 13
Pacote de algoritmos 14
Gerenciador de material de criptografia 14
Criptografia simétrica e assimétrica 15
Confirmação de chave 16
Política de compromisso 17
Assinaturas digitais 19
Saiba como o SDK funciona 20
Como o AWS Encryption SDK criptografa os dados 20
Como o AWS Encryption SDK decifra uma mensagem criptografada 21
Pacotes de algoritmos compatíveis 22
Recomendado: AES-GCM com derivação de chave, assinatura e confirmação de chave 22
Outros pacotes de algoritmos compatíveis 23
Interagindo com AWS KMS 25
Práticas recomendadas 27
Como configurar o SDK 32
Seleção de uma linguagem de programação 32
Seleção de chaves de encapsulamento 33
Usando várias regiões AWS KMS keys 34
Escolher um pacote de algoritmo 56
Limitar as chaves de dados criptografadas 68
Criação de um filtro de descoberta 74

Exigindo contextos de criptografia ... 77
Como definir uma política de compromisso ... 85
Trabalhar com streaming de dados ... 85
Armazenamento em cache de chaves de dados ... 86
Repositórios de chaves ... 87
Principais conceitos e terminologia da loja ... 87
Implementação de permissões de privilégio mínimo ... 88
Crie um armazenamento de chaves ... 89
Configurar as principais ações do armazenamento ... 90
Configure suas principais ações de armazenamento ... 91
Crie chaves de ramificação ... 96
Alternar a chave de ramificação ativa ... 100
Tokens de autenticação ... 103
Como os tokens de autenticação funcionam ... 103
Compatibilidade dos tokens de autenticação ... 105
Requisitos variados para tokens de autenticação de criptografia ... 106
Tokens de autenticação e provedores de chaves mestras compatíveis ... 106
AWS KMS chaveiros ... 108
Permissões necessárias para AWS KMS chaveiros ... 110
Identificação AWS KMS keys em um AWS KMS chaveiro ... 111
Criando um AWS KMS chaveiro ... 112
Usando um chaveiro AWS KMS Discovery ... 127
Usando um chaveiro de descoberta AWS KMS regional ... 134
AWS KMS Chaveiros hierárquicos ... 143
Como funciona ... 145
Pré-requisitos ... 147
Permissões obrigatórias ... 148
Escolha um cache ... 149
Criar um token de autenticação hierárquico ... 162
AWS KMS chaveiros ECDH ... 173
Permissões necessárias para AWS KMS chaveiros ECDH ... 175
Criando um AWS KMS chaveiro ECDH ... 175
Criando um AWS KMS chaveiro de descoberta ECDH ... 182
Tokens de autenticação AES Raw ... 188
Tokens de autenticação brutos do RSA ... 196
Chaveiros ECDH brutos ... 205

Criando um chaveiro ECDH bruto 207
Multi-keyrings 224
Linguagens de programação 235
C 235
Instalar 236
Uso do C SDK 237
Exemplos 242
.NET 250
Instalar e compilar 251
Depuração 252
Exemplos 252
Go 261
Pré-requisitos 262
Instalação 262
Exemplos 263
Java 266
Pré-requisitos 266
Instalação 268
Exemplos 269
JavaScript 282
Compatibilidade 283
Instalação 285
Módulos 286
Exemplos 289
Python 297
Pré-requisitos 298
Instalação 298
Exemplos 300
Rust 307
Pré-requisitos 308
Instalação 308
Exemplos 309
Interface de linha de comando 311
Instalar a CLI do 313
Como usar a CLI 316
Exemplos 331

Referência da sintaxe e de parâmetros 355
Versões 370
Armazenamento em cache de chaves de dados 373
Como usar o armazenamento em cache de chaves de dados 374
Usando o cache de chaves de dados: Step-by-step 375
Armazenamento em cache de chaves de dados de exemplo: criptografar uma string 383
Definir limites de segurança do cache 399
Detalhes do armazenamento em cache de chaves de dados 401
Como o armazenamento em cache de chaves de dados funciona 402
Criar um cache de material de criptografia 405
Criar um gerenciador de material de criptografia de armazenamento em cache 406
O que é uma entrada de chave de dados em cache? 407
Contexto de criptografia: como selecionar entradas do cache 408
Meu aplicativo está usando chaves de dados armazenadas em cache? 409
Exemplo de armazenamento em cache de chaves de dados 409
Resultados do cache local 411
Código de exemplo 411
CloudFormation modelo 423
Versões do AWS Encryption SDK 438
C 439
C#/.NET 440
Interface de linha de comando (CLI) 441
Java 443
Go 445
JavaScript 446
Python 447
Rust 449
Detalhes da versão 449
Versões anteriores à 1.7.x 450
Versão 1.7.x 450
Versão 2.0.x 453
Versão 2.2.x 455
Versão 2.3.x 456
Migrando seu AWS Encryption SDK 457
Como migrar e implantar 459
Etapa 1: atualize a aplicação para a versão 1.x mais recente 459

Etapa 2: atualize a aplicação para a versão mais recente 460
Atualizando provedores de chaves AWS KMS mestras 462
Migração para o modo estrito 463
Migrar para o modo de descoberta 466
Atualizando AWS KMS chaveiros 470
Como definir sua política de compromisso 472
Como definir sua política de compromisso 474
Solução de problemas de migração para as versões mais recentes 485
Objetos descontinuados ou removidos 486
Conflito de configuração: política de compromisso e pacote de algoritmos 486
Conflito de configuração: política de compromisso e texto cifrado 487
Falha na validação do confirmação de chave 488
Outras falhas de criptografia 488
Outras falhas de decriptografia 488
Considerações sobre reversão 489
Perguntas frequentes 490
Como o é AWS Encryption SDK diferente do AWS SDKs? 490
Como ele é AWS Encryption SDK diferente do cliente de criptografia Amazon S3? 491
Quais algoritmos criptográficos são suportados pelo AWS Encryption SDK e qual é o padrão? 491
Como o vetor de inicialização (IV) é gerado e onde é armazenado? 492
Como cada chave de dados é gerada, criptografada e descriptografada? 492
Como faço para controlar as chaves de dados que foram usadas para criptografar meus dados? 493
Como eles AWS Encryption SDK armazenam chaves de dados criptografadas com seus dados criptografados? 493
Quanta sobrecarga o formato da AWS Encryption SDK mensagem adiciona aos meus dados criptografados? 493
Posso usar meu próprio provedor de chaves mestras? 494
Posso criptografar dados com mais de uma chave de encapsulamento? 494
Com quais tipos de dados posso criptografar? AWS Encryption SDK 495
Como os fluxos AWS Encryption SDK criptografam e descriptografam input/output (E/S)? 495
Referência 496
Referência do formato de mensagens 496
Estrutura do cabeçalho 497
Estrutura do corpo 505

Estrutura do rodapé 511
Exemplos de formatos de mensagens 511
Dados emoldurados (formato de mensagem versão 1) 512
Dados emoldurados (formato de mensagem versão 2) 516
Non-framed dados (formato de mensagem versão 1) 518
Referência de AAD de corpo 522
Referência de algoritmos 523
Referência do vetor de inicialização 528
AWS KMS Detalhes técnicos do chaveiro hierárquico 529
Histórico do documento 531
Atualizações recentes 531
Atualizações anteriores 534
..... dxxxvi

O que é o AWS Encryption SDK?

AWS Encryption SDK É uma biblioteca de criptografia do lado do cliente projetada para facilitar que todos criptografem e descriptografem dados usando os padrões e as melhores práticas do setor. Ele permite que você se concentre na funcionalidade principal do aplicativo, em vez de sobre como melhor criptografar e descriptografar os dados. AWS Encryption SDK É fornecido gratuitamente sob a licença Apache 2.0.

As AWS Encryption SDK respostas a perguntas como as seguintes para você:

- Qual algoritmo de criptografia devo usar?
- Como, ou em qual modo, devo usar esse algoritmo?
- Como faço para gerar a chave de criptografia?
- Como faço para proteger a chave de criptografia e onde devo armazená-la?
- Como posso tornar meus dados criptografados portáteis?
- Como faço para garantir que o destinatário pretendido possa ler meus dados criptografados?
- Como posso garantir que meus dados criptografados não sejam modificados entre o momento em que são gravados e o momento em que são lidos?
- Como faço para usar as chaves de dados que AWS KMS retornam?

Com o AWS Encryption SDK, você define um provedor de chave mestra ou um chaveiro que determina quais chaves de agrupamento você usa para proteger seus dados. Em seguida, você criptografa e descriptografa seus dados usando métodos simples fornecidos pelo. AWS Encryption SDK O AWS Encryption SDK faz o resto.

Sem isso AWS Encryption SDK, você pode se esforçar mais na criação de uma solução de criptografia do que na funcionalidade principal do seu aplicativo. Ele AWS Encryption SDK responde a essas perguntas fornecendo as seguintes coisas.

Uma implementação padrão que segue as melhores práticas de criptografia

> Por padrão, o AWS Encryption SDK gera uma chave de dados exclusiva para cada objeto de dados que ele criptografa. Isso segue a melhor prática de criptografia de usar chaves de dados exclusivas para cada operação de criptografia.

Ele AWS Encryption SDK criptografa seus dados usando um algoritmo de chave simétrica, autenticado e seguro. Para obter mais informações, consulte the section called “Pacotes de algoritmos compatíveis”.

Uma estrutura para proteção de chaves de dados com chaves de encapsulamento

O AWS Encryption SDK protege as chaves de dados que criptografam seus dados, criptografando-as sob uma ou mais chaves de encapsulamento. Ao fornecer uma estrutura para criptografar chaves de dados com mais de uma chave de empacotamento, isso AWS Encryption SDK ajuda a tornar seus dados criptografados portáteis.

Por exemplo, criptografe dados com uma entrada AWS KMS key AWS KMS e uma chave do seu HSM local. É possível usar qualquer uma das duas chaves de encapsulamento para descriptografar os dados, caso alguma não esteja disponível ou o chamador não tenha permissão para usar as duas chaves.

Uma mensagem formatada que armazena chaves de dados criptografadas com os dados criptografados

Ele AWS Encryption SDK armazena os dados criptografados e a chave de dados criptografados juntos em uma mensagem criptografada que usa um formato de dados definido. Isso significa que você não precisa acompanhar ou proteger as chaves de dados que criptografam seus dados, pois elas AWS Encryption SDK fazem isso por você.

Algumas implementações de linguagem do AWS Encryption SDK exigem um AWS SDK, mas AWS Encryption SDK não exigem um Conta da AWS e não dependem de nenhum AWS serviço. Você Conta da AWS só precisa de um se optar por usar AWS KMS keyspara proteger seus dados.

Desenvolvido em repositórios de código aberto

O AWS Encryption SDK é desenvolvido em repositórios de código aberto em. GitHub É possível usar esses repositórios para visualizar o código, ler e enviar problemas e encontrar informações específicas para sua implementação de linguagem.

- AWS Encryption SDK for C — aws-encryption-sdk-c
- AWS Encryption SDK para o.NET — diretório.NET do `aws-encryption-sdk` repositório.
- AWS CLI de criptografia — aws-encryption-sdk-cli
- AWS Encryption SDK for Java — aws-encryption-sdk-java
- AWS Encryption SDK para JavaScript — aws-encryption-sdk-javascript

- AWS Encryption SDK for Python — aws-encryption-sdk-python
- AWS Encryption SDK para Rust — diretório Rust do repositório. `aws-encryption-sdk`
- AWS Encryption SDK para Go — diretório Go do `aws-encryption-sdk` repositório

Compatibilidade com bibliotecas e serviços de criptografia

O AWS Encryption SDK é suportado em várias linguagens de programação. As implementações de linguagem 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. Além disso, ao criptografar e descriptografar, é necessário usar tokens de autenticação compatíveis ou chaves mestras e provedores de chaves mestras. Para obter detalhes, consulte the section called “Compatibilidade dos tokens de autenticação”.

No entanto, eles AWS Encryption SDK não podem interoperar com outras bibliotecas. Como cada biblioteca retorna dados criptografados em um formato diferente, você não pode criptografar com uma biblioteca e descriptografar com outra.

DynamoDB Encryption Client e criptografia do lado do cliente do Amazon S3

: AWS Encryption SDK Não é possível descriptografar dados criptografados pelo DynamoDB Encryption Client ou pela criptografia do lado do cliente do Amazon S3. Essas bibliotecas não conseguem decifrar a mensagem criptografada que retornam. AWS Encryption SDK

AWS Key Management Service (AWS KMS)

: Eles AWS Encryption SDK podem usar chaves AWS KMS keysde dados para proteger seus dados, incluindo chaves KMS multirregionais. Por exemplo, você pode configurar o AWS Encryption SDK para criptografar seus dados em um ou mais AWS KMS keys em seu Conta da AWS. No entanto, você deve usar o AWS Encryption SDK para descriptografar esses dados.

  AWS Encryption SDK Não é possível descriptografar o texto cifrado que as operações Encrypt ou retornam. AWS KMSReEncrypt Da mesma forma, a operação AWS KMSDecrypt não pode descriptografar a mensagem criptografada que ela retorna. AWS Encryption SDK

  O AWS Encryption SDK suporta somente chaves KMS de criptografia simétrica. Não é possível usar uma chave assimétrica do KMS para criptografia ou assinatura no AWS Encryption SDK. O AWS Encryption SDK gera suas próprias chaves de assinatura ECDSA para pacotes de algoritmos que assinam mensagens.

Suporte e manutenção

O AWS Encryption SDK 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, recomendamos que você use a versão mais recente disponível do AWS Encryption SDK para sua linguagem de programação e atualize à medida que novas versões forem lançadas. Quando uma versão exige alterações significativas, como a atualização de AWS Encryption SDK versões anteriores à 1.7. x para as versões 2.0. x e posteriormente, fornecemos instruções detalhadas para ajudá-lo.

Cada implementação de linguagem de programação do AWS Encryption SDK é desenvolvida em um GitHub repositório de código aberto separado. É provável que o ciclo de vida e a fase do suporte de cada versão variem entre os repositórios. Por exemplo, uma determinada versão do AWS Encryption SDK pode estar na fase de disponibilidade geral (suporte total) em uma linguagem de programação, mas a end-of-support fase em uma linguagem de programação diferente. Recomendamos que você use uma versão totalmente compatível sempre que possível e evite versões que já não sejam compatíveis.

Para encontrar a fase do ciclo de vida das AWS Encryption SDK versões da sua linguagem de programação, consulte o `SUPPORT_POLICY.rst` arquivo em cada AWS Encryption SDK repositório.

- AWS Encryption SDK for C — Support_policy.rst
- AWS Encryption SDK para o.NET — Support_policy.rst
- AWS CLI de criptografia — Support_policy.rst
- AWS Encryption SDK for Java — Support_policy.rst
- AWS Encryption SDK para JavaScript — Support_policy.rst
- AWS Encryption SDK for Python — Support_policy.rst

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

Saiba mais

Para obter mais informações sobre a AWS Encryption SDK criptografia do lado do cliente, experimente essas fontes.

- Para obter ajuda com os termos e conceitos usados neste SDK, consulte Conceitos no AWS Encryption SDK.
- Para obter as diretrizes de práticas recomendadas, consulte Melhores práticas para o AWS Encryption SDK.
- Para obter informações sobre como o SDK funciona, consulte Saiba como o SDK funciona.
- Para obter exemplos que mostram como configurar opções no AWS Encryption SDK, consulteConfigurando o AWS Encryption SDK.
- Para obter informações técnicas, consulte a Referência.
- Para obter as especificações técnicas do AWS Encryption SDK, consulte a AWS Encryption SDK Especificação em GitHub.
- Para obter respostas às suas perguntas sobre o uso do AWS Encryption SDK, leia e publique no Fórum de discussão sobre ferramentas AWS criptográficas.

Para obter informações sobre implementações do AWS Encryption SDK em diferentes linguagens de programação.

- C: Veja AWS Encryption SDK for C a documentação em AWS Encryption SDK C e o aws-encryption-sdk-crepositório ativado GitHub.
- C#/.NET: Consulte AWS Encryption SDK para o.NET e ative o aws-encryption-sdk-netdiretório do `aws-encryption-sdk` repositório. GitHub
- Interface de linha de comando: consulteAWS Encryption SDK interface de linha de comando, leia os documentos da CLI de AWS criptografia e aws-encryption-sdk-clido repositório em. GitHub
- Java: vejaAWS Encryption SDK for Java, o AWS Encryption SDK Javadoc e o aws-encryption-sdk-javarepositório ativado. GitHub

  JavaScript: Veja the section called “JavaScript” e ative o aws-encryption-sdk-javascriptrepositório. GitHub
- Python: veja AWS Encryption SDK for Python a documentação do AWS Encryption SDK Python e o repositório em. aws-encryption-sdk-python GitHub

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 AWS Encryption SDK, notifique a AWS segurança. Não crie um GitHub problema público.
- Para fornecer feedback sobre o AWS Encryption SDK, registre um problema no GitHub repositório da linguagem de programação que você está usando.
- Para fornecer comentários sobre esta documentação, use os links Feedback nesta página. Você também pode registrar um problema ou contribuir para aws-encryption-sdk-docso repositório de código aberto desta documentação em. GitHub

Conceitos no AWS Encryption SDK

Esta seção apresenta os conceitos usados no AWS Encryption SDK e fornece um glossário e uma referência. Ele foi projetado para ajudar você a entender como AWS Encryption SDK funciona e os termos que usamos para descrevê-lo.

Precisa de ajuda?

- Saiba como ele AWS Encryption SDK 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 encapsulamento que protegem suas chaves de dados.
- Saiba mais sobre os tokens de autenticação e osprovedores de chaves mestras que determinam quais chaves de encapsulamento você usa.
- Saiba mais sobre o contexto de criptografia que adiciona integridade ao seu processo de criptografia. É opcional, mas é uma prática recomendada que incentivamos.
- Saiba mais sobre a mensagem criptografada que os métodos de criptografia retornam.
- Então você está pronto para usar o AWS Encryption SDK em sua linguagem de programação preferida.

Tópicos

- criptografia envelopada
- Chave de dados
- Chave de encapsulamento
- Tokens de autenticação e provedores de chaves mestras
- Contexto de criptografia

- Mensagem criptografada
- Pacote de algoritmos
- Gerenciador de material de criptografia
- Criptografia simétrica e assimétrica
- Confirmação de chave
- Política de compromisso
- Assinaturas digitais

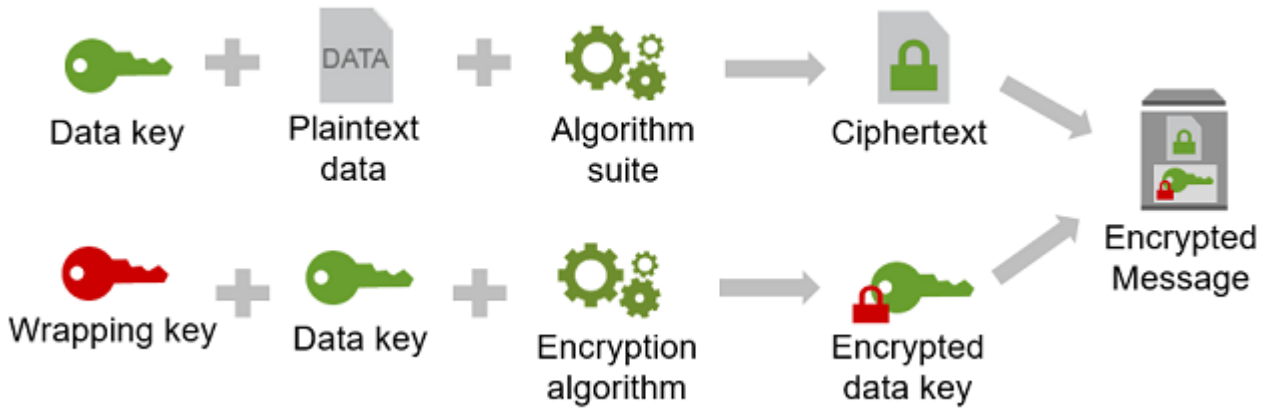
criptografia envelopada

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 AWS Encryption SDK criptografa cada mensagem com uma chave de dados exclusiva. Em seguida, ele criptografa cada chave de dados sob a chave de encapsulamento especificada. Ele armazena as chaves de dados criptografadas junto com os dados criptografados na mensagem criptografada que as operações de criptografia retornam.

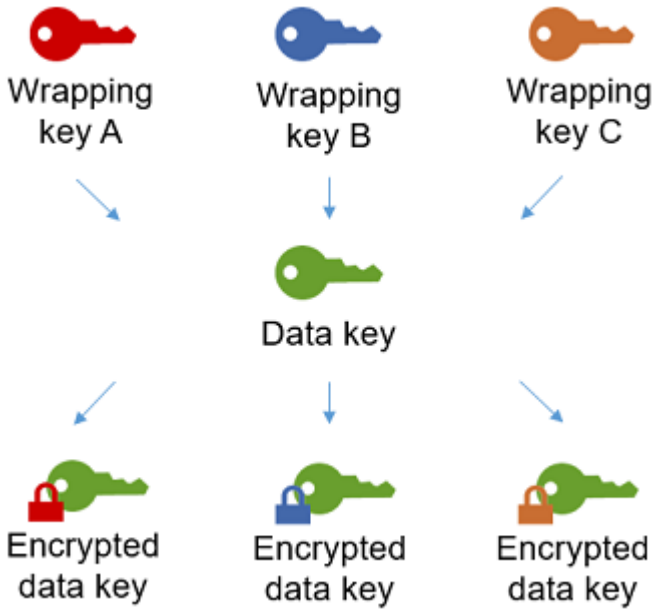
Para especificar sua chave de encapsulamento, use um token de autenticação ou umprovedor de chave mestra.



Criptografar os mesmos dados com várias chaves de encapsulamento

É possível criptografar a chave de dados sob várias chaves de encapsulamento. Talvez você queira fornecer chaves de encapsulamento distintas para usuários diferentes ou chaves de encapsulamento de tipos variados ou em locais diferentes. Cada uma das chaves de encapsulamento criptografa a mesma chave de dados. Ele AWS Encryption SDK armazena todas as chaves de dados criptografadas com os dados criptografados na mensagem criptografada.

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



Combinação de pontos fortes de vários algoritmos

Para criptografar seus dados, por padrão, o AWS Encryption SDK usa um sofisticado conjunto de algoritmos com criptografia simétrica AES-GCM, uma função de derivação de chave (HKDF) e assinatura. 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 e gerenciamento de chaves mais fácil. Para combinar as forças de cada um, você pode criptografar dados brutos com criptografia de chave simétrica e, em seguida, criptografar a chave de dados com criptografia de chave pública.

Chave de dados

A chave de ados é uma chave de criptografia que o AWS Encryption SDK usa para criptografar os dados. Cada chave de dados é uma matriz de bytes que cumpre os requisitos para chaves criptográficas. A menos que você esteja usando o cache de chaves de dados, ele AWS Encryption SDK usa uma chave de dados exclusiva para criptografar cada mensagem.

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

Para proteger suas chaves de dados, eles as AWS Encryption SDK criptografam sob uma ou mais chaves de criptografia de chave conhecidas como chaves de empacotamento ou chaves mestras. Depois de AWS Encryption SDK usar suas chaves de dados em texto simples para criptografar seus dados, ele os remove da memória o mais rápido possível. Depois, ele armazena as chaves de dados criptografadas junto com os dados criptografados na mensagem criptografada que as operações de criptografia retornam. Para obter detalhes, consulte the section called “Saiba como o SDK funciona”.

> **Tip**
>
> No AWS Encryption SDK, distinguimos as chaves de dados das chaves de criptografia de dados. Vários dos pacotes de algoritmos, incluindo o pacote padrão, usam uma função de derivação de chaves que impede que a chave de dados atinja seus limites de criptografia. 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 dados. 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 facilitam a identificação de chaves de empacotamento válidas durante a descriptografia. AWS Encryption SDK

Chave de encapsulamento

Uma chave de encapsulamento é uma chave de criptografia que o AWS Encryption SDK 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

encapsulamento serão usadas para proteger seus dados ao configurar um token de autenticação ou um provedor de chave mestra.

> **Note**
>
> A chave de encapsulamento refere-se às chaves em um token de autenticação ou provedor de chave mestra. A chave mestra geralmente está associada à classe `MasterKey` que você instancia ao usar um provedor de chave mestra.

O AWS Encryption SDK suporta várias chaves de agrupamento comumente usadas, como AWS Key Management Service (AWS KMS) simétricas AWS KMS keys(incluindo chaves KMS multirregionais), chaves brutas AES-GCM (Advanced Encryption Standard/Galois Counter Mode) e chaves RSA brutas. Você também pode estender ou implementar suas próprias chaves de encapsulamento.

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

- Use um serviço web 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. Ele AWS Encryption SDK se integra AWS KMS para ajudar você a proteger e usar suas chaves de embalagem. No entanto, AWS Encryption SDK não requer AWS nenhum AWS serviço.

Tokens de autenticação e provedores de chaves mestras

Para especificar as chaves de agrupamento que você usa para criptografia e decodificação, use um chaveiro ou um provedor de chave mestra. Você pode usar os chaveiros e os provedores de chaves mestras que eles AWS Encryption SDK fornecem ou criar suas próprias implementações. O AWS Encryption SDK fornece tokens de autenticação e provedores de chaves mestras compatíveis entre si, sujeitos a restrições de linguagem. Para obter detalhes, consulte Compatibilidade dos tokens de autenticação.

Um token de autenticação gera, criptografa e descriptografa chaves de dados. 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. Você também pode definir um token de autenticação sem chaves de encapsulamento ou um token de autenticação mais complexo com opções de configuração adicionais. Para obter ajuda para escolher e usar os chaveiros que AWS Encryption SDK definem, consulteTokens de autenticação.

Os chaveiros são compatíveis com as seguintes linguagens de programação:

- AWS Encryption SDK for C
- AWS Encryption SDK para JavaScript
- AWS Encryption SDK para o.NET
- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
- Versão 1. x do AWS Encryption SDK para Rust
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go

Um provedor de chave mestra é uma alternativa a um token de autenticação. O provedor de chave mestra retorna as chaves de encapsulamento (ou chaves mestras) que você especificar. Cada chave mestra é associada a um provedor de chaves mestras, mas um provedor de chaves mestras normalmente fornece várias chaves mestras. Os provedores de chaves mestras são compatíveis com Java, Python e a AWS CLI de criptografia.

Você deve especificar um token de autenticação (ou provedor de chave mestra) para criptografia. Você pode especificar o mesmo token de autenticação (ou provedor de chave mestra), ou um diferente, para descriptografia. Ao criptografar, ele AWS Encryption SDK usa todas as chaves de empacotamento que você especificar para criptografar a chave de dados. Ao descriptografar, o AWS Encryption SDK usa somente as chaves de encapsulamento que você especificar para descriptografar uma chave de dados criptografada. Especificar chaves de encapsulamento para decodificação é opcional, mas é uma prática recomendada. AWS Encryption SDK

Para obter detalhes sobre como especificar chaves de encapsulamento, consulte. Seleção de chaves de encapsulamento

Contexto de criptografia

Para melhorar a segurança de suas operações de criptografia, inclua um contexto de criptografia em todas as solicitações para criptografar dados. O uso de um contexto de criptografia é opcional, mas é uma melhor prática de criptografia que recomendamos.

Um contexto de criptografia é um conjunto de pares de chave-valor que contêm dados autenticados adicionais arbitrários e não secretos. O contexto de criptografia pode conter todos os dados que você escolher, mas geralmente consiste em dados que são úteis para registro em log e rastreamento, como dados sobre o tipo de arquivo, a finalidade ou a propriedade. Quando você criptografa dados, o contexto de criptografia é associado de maneira criptográfica aos dados criptografados de forma que o mesmo contexto de criptografia seja necessário para descriptografar os dados. O AWS Encryption SDK inclui o contexto de criptografia em texto simples no cabeçalho da mensagem criptografada retornada por ele.

O contexto de criptografia AWS Encryption SDK usado consiste no contexto de criptografia que você especifica e em um par de chaves públicas que o gerenciador de materiais criptográficos (CMM) adiciona. Especificamente, sempre que você usar um algoritmo de criptografia com assinatura, o 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. O `aws-crypto-public-key` nome no contexto de criptografia é reservado pelo AWS Encryption SDK e não pode ser usado como nome em nenhum outro par no contexto de criptografia. Para obter detalhes, consulte AAD na Referência do formato de mensagens.

O exemplo de contexto de criptografia a seguir consiste nos dois pares de contexto de criptografia especificados na solicitação e no par de chaves públicas adicionado pelo CMM.

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

Para descriptografar os dados, você passa a mensagem criptografada. Como o AWS Encryption SDK pode extrair o contexto de criptografia do cabeçalho da mensagem criptografada, você não precisa fornecer o contexto de criptografia separadamente. No entanto, o contexto de criptografia pode ajudar a confirmar se você está descriptografando a mensagem criptografada correta.

- Na interface de linha de comando do AWS Encryption SDK (CLI), se você fornecer um contexto de criptografia em um comando de descriptografia, a CLI verificará se os valores estão presentes no contexto da mensagem criptografada antes de retornar os dados em texto simples.

- Em outras implementações de linguagens de programação, a resposta de descriptografia inclui o contexto de criptografia e os dados em texto simples. A função de descriptografia em seu aplicativo sempre deve verificar se o contexto de criptografia na resposta de descriptografia inclui o contexto de criptografia na solicitação de criptografia (ou um subconjunto) antes de retornar os dados em texto simples.

> **Note**
>
> As versões a seguir oferecem suporte ao contexto de criptografia necessário CMM, que você pode usar para exigir um contexto de criptografia em todas as solicitações de criptografia.
>
> - Versão 3. x do AWS Encryption SDK for Java
> - Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
> - Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
> - Versão 1. x do AWS Encryption SDK para Rust
> - Versão 0.1. x ou posterior do AWS Encryption SDK for Go

Ao escolher um contexto de criptografia, lembre-se de que ele não é um segredo. O contexto de criptografia é exibido em texto não criptografado no cabeçalho da mensagem criptografada retornada pelo AWS Encryption SDK . Se você estiver usando AWS Key Management Service, o contexto de criptografia também poderá aparecer em texto simples em registros e registros de auditoria, como. AWS CloudTrail

Para obter exemplos de envio e verificação de um contexto de criptografia no seu código, consulte os exemplos da linguagem de programação de sua preferência.

Mensagem criptografada

Quando você criptografa dados com o AWS Encryption SDK, ele retorna uma mensagem criptografada.

Uma mensagem criptografada é uma estrutura de dados formatados portátil que inclui os dados criptografados junto com as cópias criptografadas das chaves de dados, o ID do algoritmo e, opcionalmente, um contexto de criptografia e uma assinatura digital. As operações de criptografia

no AWS Encryption SDK retornam uma mensagem criptografada, e as operações de descriptografia usam uma mensagem criptografada como entrada.

A combinação de dados criptografados e de suas chaves de dados criptografadas 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 mensagem criptografada, consulte Formato da mensagem criptografada.

Pacote de algoritmos

O AWS Encryption SDK usa um conjunto de algoritmos para criptografar e assinar os dados na mensagem criptografada que as operações de criptografia e descriptografia retornam. O AWS Encryption SDK é compatível com vários pacotes de algoritmos. Todos os conjuntos compatíveis usam o Advanced Encryption Standard (AES) como o algoritmo principal e o combinam com outros algoritmos e valores.

O AWS Encryption SDK estabelece um conjunto de algoritmos recomendado como padrão para todas as operações de criptografia. O padrão pode ser alterado conforme os padrões e as práticas recomendadas são aprimoradas. Você pode especificar um pacote de algoritmos alternativo em solicitações de criptografia de dados ou ao criar um gerenciador de materiais criptográficos (CMM), mas, a menos que um alternativo seja necessário para sua situação, é melhor usar o padrão. O padrão atual é AES-GCM com uma função de derivação de extract-and-expand chave (HKDF) baseada em HMAC, compromisso de chave, uma assinatura de algoritmo de assinatura digital de curva elíptica (ECDSA) e uma chave de criptografia de 256 bits.

Se sua aplicação exigir alta performance e os usuários que criptografam dados e aqueles que os descriptografam forem igualmente confiáveis, considere especificar um pacote de algoritmos sem uma assinatura digital. No entanto, recomendamos fortemente um pacote de algoritmos que inclua confirmação de chave e uma função de derivação de chave. Os pacotes de algoritmos que não têm esses atributos são compatíveis apenas para compatibilidade com versões anteriores.

Gerenciador de material de criptografia

O gerenciador de material de criptografia (CMM) monta o material criptográfico usado para criptografar e descriptografar dados. O material criptográfico inclui texto não criptografado e chaves de dados criptografadas e uma chave de assinatura de mensagem opcional. Você nunca interage diretamente com o CMM. Os métodos de criptografia e descriptografia o processam para você.

Você pode usar o CMM padrão ou o CMM de cache que ele AWS Encryption SDK fornece, ou escrever um CMM personalizado. E é possível especificar explicitamente um CMM, mas isso não é obrigatório. Quando você especifica um provedor de chaveiro ou chave mestra, AWS Encryption SDK ele cria um CMM padrão para você. O CMM padrão obtém o material de criptografia ou de descriptografia do token de autenticação ou do provedor de chave mestra que você especificar. Isso pode envolver uma chamada a um serviço criptográfico, como o AWS Key Management Service (AWS KMS).

Como o CMM atua como uma ligação entre o AWS Encryption SDK e um chaveiro (ou provedor de chave mestra), é um ponto ideal para personalização e extensão, como suporte para aplicação de políticas e armazenamento em cache. O AWS Encryption SDK fornece um CMM de cache para suportar o cache de chaves de dados.

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 AWS Encryption SDK 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 encapsulamento simétricas ou assimétricas. Ele retorna uma mensagem criptografada que inclui os dados criptografados e pelo menos uma cópia criptografada da chave de dados.

Criptografar dados (criptografia simétrica)

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

Criptografar chave de dados (criptografia simétrica ou assimétrica)

> O token de autenticação ou o provedor de chave mestra que você fornece para uma operação de criptografia e descriptografia determina como a chave de dados simétrica será criptografada e descriptografada. Você pode escolher um provedor de chaveiro ou chave mestra que use criptografia simétrica, como um AWS KMS chaveiro, ou um que use criptografia assimétrica, como um chaveiro RSA bruto ou. `JceMasterKey`

Confirmação de chave

O AWS Encryption SDK suporta o comprometimento da chave (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, a confirmação de chave garante que somente a chave de dados que criptografou sua mensagem seja usada para descriptografá-la. Criptografar e descriptografar com o confirmação de chave é uma prática recomendada do AWS Encryption SDK.

A maioria das cifras simétricas modernas (incluindo AES) criptografa um texto simples com uma única chave secreta, como a chave de dados exclusiva que o AWS Encryption SDK usa para criptografar cada mensagem de texto simples. Descriptografar esses dados com a mesma chave de dados retorna um texto sem formatação idêntico ao original. A decodificação com uma chave diferente geralmente falhará. No entanto, é possível decifrar um texto cifrado com duas chaves diferentes. Em casos raros, é possível encontrar uma chave que possa descriptografar alguns bytes do o texto cifrado em um texto simples diferente, mas ainda inteligível.

AWS Encryption SDK Sempre criptografa cada mensagem de texto simples em uma chave de dados exclusiva. Ele pode criptografar essa chave de dados em várias chaves de encapsulamento, mas as chaves de encapsulamento (ou chaves mestras) sempre criptografam a mesma chave de dados. No entanto, uma mensagem criptografada sofisticada e criada manualmente pode, na verdade, conter chaves de dados diferentes, cada uma criptografada por uma chave de encapsulamento diferente. Por exemplo, se um usuário descriptografar a mensagem criptografada, ela retornará 0x0 (falso), enquanto outro usuário descriptografando a mesma mensagem criptografada obterá 0x1 (verdadeiro).

Para evitar esse cenário, o AWS Encryption SDK suporta o comprometimento da chave ao criptografar e descriptografar. Quando AWS Encryption SDK criptografa uma mensagem com comprometimento de chave, ele vincula criptograficamente a chave de dados exclusiva que produziu o texto cifrado à cadeia de caracteres de confirmação da chave, um identificador de chave de dados não secreto. Em seguida, ele armazena o string de compromisso chave nos metadados da mensagem criptografada. Ao decifrar uma mensagem com comprometimento de chave, AWS Encryption SDK verifica se a chave de dados é a única chave para essa mensagem criptografada. Se a verificação da chave de dados falhar, a operação de descriptografia falhará.

O suporte para confirmação de chaves foi apresentado na versão 1.7. x, que pode descriptografar mensagens com confirmação de chave, mas não pode criptografar com confirmação de chave. Você pode usar essa versão para implantar totalmente a capacidade de descriptografar texto cifrado com confirmação de chave. A versão 2.0.x inclui suporte total para os compromissos de chave.

Por padrão, ela criptografa e descriptografa somente com confirmação de chave. Essa é uma configuração ideal para aplicativos que não precisam decifrar texto cifrado criptografado por versões anteriores do. AWS Encryption SDK

Embora criptografar e descriptografar com confirmação de chave seja uma prática recomendada, deixamos que você decida quando ela será usada e ajustamos o ritmo em que você a adota. A partir da versão 1.7. x, AWS Encryption SDK suporta uma política de compromisso que define o conjunto de algoritmos padrão e limita os conjuntos de algoritmos que podem ser usados. Essa política determina se seus dados são criptografados e descriptografados com confirmação de chave.

A confirmação de chave resulta em uma mensagem criptografada um pouco maior (+ 30 bytes) e que leva mais tempo para ser processada. Se sua aplicação for muito sensível ao tamanho ou à performance, você poderá optar por não aceitar a confirmação de chave. Mas faça isso somente se for necessário.

Para obter mais informações sobre a migração para as versões 1.7.x e 2.0.x, incluindo seus atributos de confirmação de chave, consulte Migrando seu AWS Encryption SDK. Para obter informações técnicas sobre confirmação de chave, consulte the section called “Referência de algoritmos” ethe section called “Referência do formato de mensagens”.

Política de compromisso

Uma política de compromisso é uma definição de configuração que determina se a aplicação criptografa e descriptografa com confirmação de chave. Criptografar e descriptografar com o confirmação de chave é uma prática recomendada do AWS Encryption SDK.

A política de compromisso tem três valores.

> **Note**
>
> Talvez seja necessário rolar horizontalmente ou verticalmente para ver a tabela inteira.

Valores da política de compromisso

Valor	Criptografa com confirmação de chave	Criptografa sem confirmação de chave	Descriptografa com confirmação de chave	Descriptografa sem confirmação de chave
ForbidEncryptAllowDecrypt				
RequireEncryptAllowDecrypt				
RequireEncryptRequireDecrypt				

A configuração da política de compromisso foi introduzida na AWS Encryption SDK versão 1.7. x. Ela é válida em todas as linguagens de programação suportadas.

- O `ForbidEncryptAllowDecrypt` descriptografa com ou sem confirmação de chave, mas não criptografa com confirmação de chave. Esse valor, introduzido na versão 1.7. x, foi projetado para preparar todos os hosts que executam seu aplicativo para decifrar com comprometimento de chave antes mesmo de encontrarem um texto cifrado criptografado com comprometimento de chave.
- O `RequireEncryptAllowDecrypt` sempre criptografa com confirmação de chave. Ele pode descriptografar textos cifrados criptografados com ou sem confirmação de chave. Este valor, introduzido na versão 2.0.x, permite que você comece a criptografar com confirmação de chave, mas ainda descriptografe textos cifrados herdados sem confirmação de chave.
- `RequireEncryptRequireDecrypt`: criptografa e descriptografa somente com confirmação de chave. Esse valor é o padrão para a versão 2.0x. Use-o valor quando tiver certeza de que todos os seus textos cifrados estão criptografados com a confirmação de chave.

A configuração da política de compromisso determina quais pacotes de algoritmos você pode usar. A partir da versão 1.7. x, o pacote de algoritmos de AWS Encryption SDK suporte para

comprometimento de chaves; com e sem assinatura. Se você especificar um pacote de algoritmos que entre em conflito com sua política de compromisso, o AWS Encryption SDK retornará um erro.

Para obter ajuda para definir sua política de compromisso, consulte Como definir sua política de compromisso.

Assinaturas digitais

Ele AWS Encryption SDK criptografa seus dados usando um algoritmo de criptografia autenticado, o AES-GCM, e o processo de decodificação 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 AWS Encryption SDK suporta a adição de uma assinatura do Algoritmo de Assinatura Digital de Curva Elíptica (ECDSA) ao final das mensagens criptografadas. Quando um conjunto de algoritmos de assinatura é usado, AWS Encryption SDK ele gera uma chave privada temporária e um par de chaves públicas para cada mensagem criptografada. O AWS Encryption SDK armazena a chave pública no contexto de criptografia da chave de dados 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 adicionais autenticados no cabeçalho da mensagem, impedindo que usuários que só podem descriptografar mensagens alterem a chave pública ou afetem a verificação da assinatura.

A verificação de assinatura adiciona um custo significativo de performance à descriptografia. Se os usuários que criptografam dados e os usuários que decifram os dados forem igualmente confiáveis, considere usar um pacote de algoritmos que não inclua assinatura.

> **Note**
>
> Se o chaveiro ou o acesso ao material criptográfico da embalagem não delimitarem entre criptografadores e decodificadores, as assinaturas digitais não fornecerão 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 AWS Encryption SDK funciona

Os fluxos de trabalho desta seção explicam como AWS Encryption SDK criptografa dados e descriptografa mensagens criptografadas. Esses fluxos de trabalho descrevem o processo básico usando os atributos padrão. Para obter detalhes sobre como definir e usar componentes personalizados, consulte o GitHub repositório de cada implementação de linguagem compatível.

O AWS Encryption SDK usa criptografia de envelope para proteger seus dados. Cada mensagem é criptografada em uma chave de dados exclusiva. Em seguida, a chave de dados é criptografada pelas chaves de encapsulamento que você especificar. Para descriptografar a mensagem criptografada, ele AWS Encryption SDK 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 mensagem de texto simples.

Precisa de ajuda com a terminologia que usamos no AWS Encryption SDK? Consulte the section called “Conceitos”.

Como o AWS Encryption SDK criptografa os dados

AWS Encryption SDK Ele fornece métodos que criptografam cadeias de caracteres, matrizes de bytes e fluxos de bytes. Para obter exemplos de código, consulte o tópico Exemplos em cada seção de Linguagens de programação.

1. Crie um token de autenticação (ou um provedor de chave mestra) que especifique as chaves de agrupamento que protegem seus dados.

2. Transmita o chaveiro e os dados do texto simples para um método de criptografia. Recomendamos transmitir um contexto de criptografia opcional, não secreto.
3. O método de criptografia solicita materiais de criptografia ao token de autenticação. O chaveiro retorna chaves de criptografia de dados exclusivas para a mensagem: uma chave de dados em texto simples e uma cópia dessa chave de dados criptografada por cada uma das chaves de encapsulamento especificadas.
4. O método de criptografia usa a chave de dados de texto não criptografado para criptografar os dados e, em seguida, descarta a chave de dados de texto não criptografado. Se você fornecer um contexto de criptografia (uma prática recomendada do AWS Encryption SDK), o método de criptografia também vinculará de forma criptográfica o contexto de criptografia aos dados criptografados.
5. O método de criptografia retorna uma mensagem criptografada que contém os dados criptografados, as chaves de dados criptografadas e outros metadados, incluindo o contexto de criptografia, se você o usou.

Como o AWS Encryption SDK decifra uma mensagem criptografada

AWS Encryption SDK Fornece métodos que decifram a mensagem criptografada e retornam texto sem formatação. Para obter exemplos de código, consulte o tópico Exemplos em cada seção de Linguagens de programação.

O token de autenticação (ou o provedor de chave mestra) que descriptografará a mensagem criptografada deve ser compatível com aquele usado para criptografar a mensagem. Uma das chaves de encapsulamento dele deve ser capaz descriptografar uma chave de dados criptografada na mensagem criptografada. Para obter informações sobre compatibilidade com tokens de autenticação e provedores de chaves mestra, consulte the section called “Compatibilidade dos tokens de autenticação”.

1. Crie um token de autenticação ou provedor de chave mestra com chaves de encapsulamento que possam descriptografar seus dados. É possível usar o mesmo token de autenticação fornecido para o método de criptografia ou um token diferente.
2. Transmita a mensagem criptografada e o token de autenticação para um método de descriptografia.
3. O método de descriptografia solicita que o token de autenticação ou o provedor de chave mestra descriptografe uma das chaves de dados criptografadas na mensagem criptografada. Ele passa informações da mensagem criptografada, incluindo as chaves de dados criptografadas.

4. O token de autenticação usa suas chaves de empacotamento para descriptografar uma das chaves de dados criptografadas. A resposta incluirá a chave de dados em texto simples, se for bem-sucedida. Caso nenhuma das chaves de encapsulamento especificadas pelo token de autenticação ou provedor da chave mestra possa descriptografar uma chave de dados criptografada, a chamada de descriptografia falhará.
5. O método de descriptografia usa a chave de dados de texto simples para descriptografar os dados, descarta a chave de dados de texto simples e retorna os dados de texto simples.

Suítes de algoritmos compatíveis no AWS Encryption SDK

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

O conjunto de algoritmos usa o AWS Encryption SDK algoritmo Advanced Encryption Standard (AES) em Galois/Counter Modo (GCM), conhecido como AES-GCM, para criptografar dados brutos. O AWS Encryption SDK suporta chaves de criptografia de 256 bits, 192 bits e 128 bits. O tamanho do vetor de inicialização (IV) é sempre 12 bytes. O tamanho da tag de autenticação é sempre 16 bytes.

Por padrão, o AWS Encryption SDK usa um conjunto de algoritmos com AES-GCM com uma função de derivação de extract-and-expand chave (HKDF) baseada em HMAC, assinatura e uma chave de criptografia de 256 bits. Se a política de compromisso exigir comprometimento de chave, ela AWS Encryption SDK seleciona um conjunto de algoritmos que também oferece suporte ao comprometimento de chave; caso contrário, seleciona um conjunto de algoritmos com derivação e assinatura de chaves, mas não com compromisso de chave.

Recomendado: AES-GCM com derivação de chave, assinatura e confirmação de chave

Ele AWS Encryption SDK recomenda um conjunto de algoritmos que deriva uma chave de criptografia AES-GCM fornecendo uma chave de criptografia de dados de 256 bits para a função de derivação de chave (HKDF) baseada em HMAC. extract-and-expand AWS Encryption SDK Isso adiciona uma assinatura do Algoritmo de Assinatura Digital de Curva Elíptica (ECDSA). Para oferecer suporte ao comprometimento da chave, esse pacote de algoritmos também deriva uma sequência de caracteres de comprometimento da chave, um identificador de chave de dados não secreto, que é armazenado nos metadados da mensagem criptografada. Essa sequência

de comprometimento da chave também é derivada por meio do HKDF usando um procedimento semelhante à derivação da chave de criptografia de dados.

AWS Encryption SDK Suíte de algoritmos

Algoritmo de criptografia	Tamanho da chave de criptografia de dados (em bits)	Algoritmo de derivação de chave	Algoritmo de assinatura	Confirmação de chave
AES-GCM	256	HKDF com SHA-384	ECDSA com P-384 e SHA-384	HKDF com SHA-512

A HKDF ajuda a evitar a reutilização acidental de uma chave de criptografia de dados e reduz o risco de sobreuso de chaves de dados.

Para assinatura, esse pacote de algoritmos usa ECDSA com um algoritmo de função hash criptográfica (SHA-384). O ECDSA é usado por padrão, mesmo quando ele não é especificado pela política da chave mestra subjacente. A assinatura da mensagem verifica se o remetente foi autorizado a criptografar mensagens e fornece o não repúdio. Isso é especialmente útil quando a política de autorização de uma chave mestra permite que um conjunto de usuários criptografe dados e outro conjunto de usuários descriptografe os dados.

Conuuntos de algoritmos com confirmação de chave garantem que cada texto cifrado seja descriptografado em apenas um texto simples. Eles fazem isso validando a chave de dados usada como entrada para o algoritmo de criptografia. Ao criptografar, esses conjuntos de algoritmos derivam um HMAC de comprometimento fundamental. Antes de descriptografar, eles validam que a chave de dados corresponde à sequência de conformação da chave. Caso contrário, a chamada de descriptografia falhará.

Outros pacotes de algoritmos compatíveis

O AWS Encryption SDK suporta os seguintes conjuntos de algoritmos alternativos para compatibilidade com versões anteriores. Em geral, não recomendamos esses pacotes de algoritmos. No entanto, reconhecemos que a assinatura pode prejudicar significativamente a performance, por isso oferecemos um pacote de confirmação de chaves com derivação de chaves para esses casos.

Para aplicações que precisam fazer concessões de performance mais significativas, continuamos oferecendo pacotes que não possuem assinatura, confirmação de chaves e derivação de chaves.

AES-GCM sem confirmação de chave

Os conjuntos de algoritmos sem confirmação de chave não validam a chave de dados antes da descriptografia. Como resultado, esses conjuntos de algoritmos podem descriptografar um único texto cifrado em diferentes mensagens de texto simples. No entanto, como os pacotes de algoritmos com confirmação de chave produzem uma mensagem criptografada um pouco maior (+30 bytes) e demoram mais para serem processados, eles podem não ser a melhor opção para cada aplicação.

O AWS Encryption SDK suporta um conjunto de algoritmos com derivação de chave, compromisso de chave, assinatura e outro com derivação de chave e compromisso de chave, mas não assinatura. Não recomendamos usar um pacote de algoritmos sem confirmação de chave. Se necessário, recomendamos um pacote de algoritmos com derivação e confirmação de chaves, mas sem assinatura. No entanto, se o perfil de performance da aplicação for compatível com o uso de um pacote de algoritmos, usar um pacote de algoritmos com confirmação de chaves, derivação de chaves e assinatura é uma prática recomendada.

AES-GCM sem assinatura

Os conjuntos de algoritmos sem assinatura carecem da assinatura ECDSA, que fornece autenticidade e não repúdio. Use esse conjunto apenas quando os usuários que criptografam dados e os que os descriptografam são igualmente confiáveis.

Ao usar um pacote de algoritmos sem assinatura, recomendamos escolher um com derivação e confirmação de chave.

AES-GCM sem derivação de chaves

Pacotes de algoritmos sem derivação usam a criptografia de dados como a chave de criptografia do AES-GCM, em vez de usar uma função de derivação de chaves para derivar uma chave exclusiva. Nós desencorajamos o uso dessa suíte para gerar texto cifrado, mas ela é AWS Encryption SDK compatível por motivos de compatibilidade.

Para obter mais informações sobre como esses pacotes são representados e usados na biblioteca, consulte the section called “Referência de algoritmos”.

Usar o AWS Encryption SDK por AWS KMS

Para usar o AWS Encryption SDK, você precisa configurar chaveiros ou provedores de chaves mestras com chaves agrupadas. Se você não tiver uma infraestrutura de chaves, recomendamos usar o AWS Key Management Service (AWS KMS). Muitos dos exemplos de código no AWS Encryption SDK exigem um AWS KMS key.

Para interagir AWS KMS, o AWS Encryption SDK requer o AWS SDK para sua linguagem de programação preferida. A biblioteca AWS Encryption SDK cliente funciona com os AWS SDKs para oferecer suporte às chaves mestras armazenadas em AWS KMS.

Para se preparar para usar o AWS Encryption SDK por AWS KMS

1. Crie um Conta da AWS. Para saber como, consulte Como faço para criar e ativar 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 ID da chave ou do nome de recurso da Amazon (ARN) do. AWS KMS key Para ajudar a descorbri o ID ou o ARN de uma AWS KMS key, consulte Descobrir 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 pode usar o 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 ou usuário AWS (raiz) do IAM.

   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 Java, AWS SDK para JavaScript, AWS SDK para Python (Boto) ou AWS SDK para C++ (para C) e 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.

   Esse procedimento permite que AWS os SDKs assinem solicitações AWS para você. As amostras de código no AWS Encryption SDK que interagem com AWS KMS pressupõem que você tenha concluído essa etapa.
5. Baixe e instale AWS Encryption SDK o. Para saber como, consulte as instruções de instalação da linguagem de programação que você deseja usar.

Melhores práticas para o AWS Encryption SDK

O foi AWS Encryption SDK projetado para facilitar a proteção de seus dados usando os padrões e as melhores práticas do setor. Embora muitas práticas recomendadas tenham sido selecionadas para você nos valores padrão, algumas delas são opcionais, mas recomendadas sempre que seja prático.

Use a versão mais recente

Ao começar a usar o AWS Encryption SDK, use a versão mais recente oferecida em sua linguagem de programação preferida. Se você estiver usando o AWS Encryption SDK, atualize para cada versão mais recente o mais rápido possível. Isso assegura que você esteja usando a configuração recomendada e aproveitando as novas propriedades de segurança para proteger seus dados. Para obter detalhes sobre as versões compatíveis, incluindo orientações para migração e implantação, consulte Suporte e manutenção e Versões do AWS Encryption SDK.

Se uma nova versão descontinuar elementos em seu código, substitua-os assim que possível. Os avisos de descontinuação e os comentários de código geralmente recomendam uma boa alternativa.

Para tornar as atualizações significativas mais fáceis e menos propensas a erros, ocasionalmente fornecemos uma versão temporária ou transitória. Use essas versões e a documentação que as acompanha para garantir que você possa atualizar a aplicação sem interromper seu fluxo de trabalho de produção.

Use os valores padrão

As melhores práticas de AWS Encryption SDK design em seus valores padrão. Sempre que possível, use-os. Para casos em que aplicar o padrão seja pouco prático, fornecemos alternativas, como pacotes de algoritmos sem assinatura. Também oferecemos oportunidades de personalização para usuários avançados, como tokens de autenticação, fornecedores de chaves mestras e gerenciadores de material criptográfico (CMMs) personalizados. Use essas alternativas avançadas com cuidado e faça com que um engenheiro de segurança verifique suas escolhas sempre que possível.

Usar um contexto de criptografia

Para melhorar a segurança de suas operações de criptografia, inclua um contexto de criptografia com um valor significativo em todas as solicitações para criptografar dados. O uso de um contexto de criptografia é opcional, mas é uma melhor prática de criptografia que recomendamos.

Ele oferece dados autenticados adicionais (AAD) para criptografia autenticada no AWS Encryption SDK. Embora não seja secreto, o contexto de criptografia pode ajudar você a proteger a integridade e a autenticidade de seus dados criptografados.

No AWS Encryption SDK, você especifica um contexto de criptografia somente ao criptografar. Ao descriptografar, o AWS Encryption SDK usa o contexto de criptografia no cabeçalho da mensagem criptografada que retorna. AWS Encryption SDK Antes da aplicação retornar os dados de texto simples, verifique se o contexto de criptografia usado para criptografar a mensagem está incluído no contexto de criptografia usado ao descriptografar a mensagem. Para obter detalhes, consulte os exemplos na sua linguagem de programação.

Quando você usa a interface de linha de comando, ele AWS Encryption SDK verifica o contexto de criptografia para você.

Contexto de criptografia necessário com a Biblioteca de Provedores de Materiais

Começando com as versões do AWS Encryption SDK que integram a Biblioteca de Provedores de Material AWS Criptográfico (MPL) (por exemplo, versão 3). x e versões posteriores do AWS Encryption SDK for Java e da versão 4. x e posterior do AWS Encryption SDK for Python), você pode configurar uma CMM de contexto de criptografia necessária que marca as chaves de contexto de criptografia selecionadas como necessárias. As chaves de contexto de criptografia necessárias e seus valores ainda estão vinculados ao texto cifrado como dados autenticados adicionais (AAD), mas as chaves não são armazenadas no cabeçalho da mensagem criptografada. O chamador deve fornecer as mesmas chaves e valores de contexto de criptografia necessários ao descriptografar; caso contrário, a descriptografia falhará. Isso reduz o tamanho do texto cifrado para o contexto de criptografia que o chamador já tem, preservando as propriedades de autenticação e integridade do contexto de criptografia.

Cross-version e compatibilidade entre idiomas

As mensagens que incluem todo o contexto de criptografia no cabeçalho (produzidas por AWS Encryption SDK versões que não configuram uma CMM de contexto de criptografia necessária) podem ser descriptografadas por qualquer AWS Encryption SDK versão compatível, incluindo versões anteriores à integração MPL. As mensagens produzidas com um CMM de contexto de criptografia necessário só podem ser descriptografadas por AWS Encryption SDK versões que integram a MPL, e o chamador deve fornecer as chaves e valores de contexto de criptografia necessários no momento da descriptografia. Para obter mais informações sobre quais versões de idioma integram o MPL, consulteAWS Encryption SDK linguagens de programação. Para obter detalhes sobre compatibilidade entre versões, consulteVersões do AWS Encryption SDK.

Proteja suas chaves de encapsulamento

O AWS Encryption SDK gera uma chave de dados exclusiva para criptografar cada mensagem de texto sem formatação. Em seguida, ele criptografa a chave de dados com as chaves de encapsulamento fornecidas por você. Se suas chaves de encapsulamento forem perdidas ou excluídas, seus dados criptografados serão irrecuperáveis. Se suas chaves não estiverem protegidas, seus dados podem ficar vulneráveis.

Use chaves de encapsulamento protegidas por uma infraestrutura de chave segura, como o AWS Key Management Service (AWS KMS). Ao usar chaves AES ou RSA brutas, utilize uma fonte de randomização e armazenamento durável que atenda aos seus requisitos de segurança. Gerar e armazenar chaves de empacotamento em um módulo de segurança de hardware (HSM) ou em um serviço que forneça HSMs, como AWS CloudHSM, é uma prática recomendada.

Use os mecanismos de autorização da sua infraestrutura de chaves para limitar o acesso às chaves de encapsulamento somente aos usuários que exigem elas. Implemente princípios de práticas recomendadas, como privilégio mínimo. Ao usar AWS KMS keys, use as principais políticas e políticas do IAM que implementam os princípios das melhores práticas.

Especifique suas chaves de encapsulamento

A especificação das chaves de encapsulamento de forma explícita ao descriptografar e criptografar é sempre um prática recomendada. Quando você faz isso, o AWS Encryption SDK usa somente as chaves que você especifica. Essa prática garante que você use somente as chaves de criptografia que pretende utilizar. Para AWS KMS agrupar chaves, ele também melhora o desempenho ao impedir que você use chaves inadvertidamente em uma região diferente Conta da AWS ou tente descriptografar com chaves que você não tem permissão para usar.

Ao criptografar, os chaveiros e os provedores de chaves mestras fornecidos exigem que você especifique as AWS Encryption SDK chaves de encapsulamento. Eles utilizam somente as chaves de encapsulamento que você especificar, nenhuma outra. Você também precisa especificar chaves de encapsulamento ao criptografar e descriptografar com chaveiros AES brutos, chaveiros RSA brutos e. JCEMasterKeys

No entanto, ao descriptografar com AWS KMS chaveiros e provedores de chaves mestras, você não precisa especificar chaves de encapsulamento. Eles AWS Encryption SDK podem obter o identificador da chave a partir dos metadados da chave de dados criptografada. No entanto, recomendamos que você especifique as chaves de encapsulamento, já que esta é uma prática recomendada.

Para apoiar essa prática recomendada ao trabalhar com chaves de AWS KMS encapsulamento, recomendamos o seguinte:

- Use AWS KMS chaveiros que especificam as chaves de empacotamento. Ao criptografar e descriptografar, esses tokens de autenticação usam somente as chaves de encapsulamento especificadas por você.
- Ao usar chaves AWS KMS mestras e provedores de chaves mestras, use os construtores de modo estrito introduzidos na versão 1.7. x do AWS Encryption SDK. Eles criam provedores que criptografam e descriptografam somente com as chaves de encapsulamento que você especificar. Os construtores de provedores de chave mestra que sempre descriptografam com qualquer chave de encapsulamento foram descontinuados na versão 1.7.x e excluídos na versão 2.0.x.

Quando a especificação de chaves de AWS KMS encapsulamento para decodificação é impraticável, você pode usar provedores de descoberta. Os chaveiros AWS KMS Discovery AWS Encryption SDK em C e JavaScript suporte. Os provedores de chaves mestras com um modo de descoberta estão disponíveis para Java e Python nas versões 1.7.x e posteriores. Esses provedores de descoberta, que são usados somente para descriptografar com chaves de AWS KMS encapsulamento, orientam explicitamente o AWS Encryption SDK uso de qualquer chave de encapsulamento que criptografe uma chave de dados.

Se você precisar usar um provedor de descoberta, use os atributos de filtro de descoberta para limitar as chaves de encapsulamento que eles usam. Por exemplo, o token de autenticação de descoberta regional do AWS KMS usa somente as chaves de encapsulamento em uma determinada Região da AWS. Você também pode configurar AWS KMS chaveiros e provedores de chaves AWS KMS mestras para usar somente as chaves de empacotamento em particular. Contas da AWS Além disso, como sempre, use políticas de chaves e políticas do IAM para controlar o acesso às suas chaves de AWS KMS empacotamento.

Use assinaturas digitais

É uma prática recomendada usar um pacote de algoritmos com assinatura. Assinaturas digitais confirmam que o remetente da mensagem foi autorizado a enviá-la e protegem a integridade da mensagem. Todas as versões do AWS Encryption SDK uso de conjuntos de algoritmos com assinatura por padrão.

Se seus requisitos de segurança não incluírem assinaturas digitais, você pode selecionar um pacote de algoritmos sem assinaturas digitais. No entanto, recomendamos o uso de assinaturas digitais, especialmente quando um grupo de usuários criptografa dados e um grupo diferente de usuários descriptografa esses dados.

Use o confirmação de chave

É uma prática recomendada usar o atributo de segurança de confirmação de chave. Ao verificar a identidade da chave de dados exclusiva que criptografou seus dados, a confirmação de chave impede que você descriptografe qualquer texto cifrado que possa resultar em mais de uma mensagem de texto simples.

O AWS Encryption SDK fornece suporte completo para criptografia e descriptografia com compromisso de chave a partir da versão 2.0. x. Por padrão, todas as suas mensagens são criptografadas e decriptografadas com confirmação de chave. Versão 1.7. x do AWS Encryption SDK pode decifrar textos cifrados com o comprometimento da chave. Ela foi projetada para ajudar os usuários de versões anteriores a implantar a versão 2.0.x com sucesso.

O suporte para confirmação de chaves inclui novos pacotes de algoritmos e um novo formato de mensagem que produz um texto cifrado apenas 30 bytes maior do que um texto cifrado sem confirmação de chave. O design minimiza seu impacto na performance para que a maioria dos usuários possa aproveitar os benefícios da confirmação de chave. Se seu aplicativo for muito sensível ao tamanho e ao desempenho, você pode decidir usar a configuração da política de comprometimento para desativar o AWS Encryption SDK comprometimento de chaves ou permitir que ele decifre mensagens sem compromisso, mas faça isso somente se necessário.

Limite o número de chaves de dados criptografadas

É uma prática recomendada limitar o número de chaves de dados criptografadas nas mensagens que você descriptografa, especialmente mensagens de fontes não confiáveis. Descriptografar uma mensagem com várias chaves de dados criptografadas que você não pode descriptografar pode gerar atrasos prolongados, aumentar despesas, limitar a capacidade da aplicação e de outros que compartilham sua conta e potencialmente esgotar sua infraestrutura de chaves. Uma mensagem criptografada pode conter até 65.535 (2^16 - 1) chaves de dados criptografadas. Para obter detalhes, consulte Limitar as chaves de dados criptografadas.

Para obter mais informações sobre os recursos AWS Encryption SDK de segurança subjacentes a essas melhores práticas, consulte Criptografia aprimorada do lado do cliente: compromisso explícito KeyIds e fundamental no Blog de segurança. AWS

Configurando o AWS Encryption SDK

O foi AWS Encryption SDK projetado para ser fácil de usar. Embora o AWS Encryption SDK 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.

Ao configurar sua implementação, analise as AWS Encryption SDK melhores práticas e implemente o máximo possível.

Tópicos

- Seleção de uma linguagem de programação
- Seleção de chaves de encapsulamento
- Usando várias regiões AWS KMS keys
- Escolher um pacote de algoritmo
- Limitar as chaves de dados criptografadas
- Criação de um filtro de descoberta
- Configurando o contexto de criptografia necessário (CMM)
- Como definir uma política de compromisso
- Trabalhar com streaming de dados
- Armazenamento em cache de chaves de dados

Seleção de uma linguagem de programação

O AWS Encryption SDK 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. No entanto, pode selecionar uma linguagem de programação para uma implementação específica. Por exemplo, se você preferir trabalhar com chaveiros, você pode escolher o AWS Encryption SDK for C ou o. AWS Encryption SDK para JavaScript

Seleção de chaves de encapsulamento

Isso AWS Encryption SDK gera uma chave de dados simétrica exclusiva para criptografar cada mensagem. A menos que você esteja usando o armazenamento em cache de chaves de dados, você não precisa configurar, gerenciar ou usar as chaves de dados. Ele AWS Encryption SDK faz isso por você.

No entanto, você deve selecionar uma ou mais chaves de encapsulamento para criptografar cada chave de dados. O AWS Encryption SDK é compatível com chaves simétricas AES e chaves assimétricas RSA em tamanhos diferentes. Ele também é compatível com a criptografia simétrica AWS KMS keys do AWS Key Management Service(AWS KMS). Você é responsável pela segurança e durabilidade de suas chaves de empacotamento, por isso recomendamos que você 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 agrupamento para criptografia e descriptografia, você usa um chaveiro (C, Java, JavaScript .NET e Python) ou um provedor de chave mestra (Java, Python, CLI de criptografia). AWS É possível especificar uma chave de encapsulamento ou várias chaves de encapsulamento do mesmo tipo ou de tipos diferentes. Se você usar várias chaves de encapsulamento para empacotar uma chave de dados, cada chave de encapsulamento criptografará uma cópia da mesma chave de dados. As chaves de dados criptografadas (uma por chave de empacotamento) são armazenadas com os dados criptografados na mensagem criptografada que elas AWS Encryption SDK retornam. Para descriptografar os dados, primeiro use uma de suas chaves de empacotamento para AWS Encryption SDK descriptografar uma chave de dados criptografada.

Para especificar um AWS KMS key em um chaveiro ou provedor de chave mestra, use um identificador de AWS KMS chave compatível. 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 AWS Encryption SDK for Java,, AWS Encryption SDK para JavaScript AWS Encryption SDK for Python, ou com a CLI de AWS criptografia, você pode usar qualquer identificador de chave válido (ID da chave, ARN da chave, nome do alias ou ARN do alias) para uma chave KMS. Ao criptografar com o AWS Encryption SDK for C, você só pode usar um ID de chave ou ARN de chave.

  Se você especificar um nome de alias ou ARN de alias para uma chave KMS ao criptografar, o AWS Encryption SDK 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.

- Ao descriptografar no modo estrito (onde você especifica chaves de encapsulamento específicas), você deve usar um ARN de chave para identificar as AWS KMS keys. Esse requisito aplica-se a todas as implementações de linguagem do AWS Encryption SDK.

  Quando você criptografa com um AWS KMS chaveiro, ele AWS Encryption SDK armazena o ARN da chave AWS KMS key nos metadados da chave de dados criptografada. Ao descriptografar no modo estrito, AWS Encryption SDK verifica se o mesmo ARN da chave aparece no chaveiro (ou no provedor da chave mestra) antes de tentar usar a chave de empacotamento para descriptografar a chave de dados criptografada. Se você usar um identificador de chave diferente, eles não AWS Encryption SDK reconhecerão nem usarão o AWS KMS key, mesmo que os identificadores se refiram à mesma chave.

Para especificar uma chave AES bruta ou um par de chaves RSA brutas como chave de agrupamento em um token de autenticação, você deve especificar um namespace e um nome. Em um provedor de chave mestra, o `Provider ID` é o equivalente do namespace e o `Key ID` é o equivalente do nome. Ao descriptografar, você deve usar exatamente o mesmo namespace e nome para cada chave de encapsulamento bruta que você usou ao criptografar. Se você usar um namespace ou nome diferente, eles não AWS Encryption SDK reconhecerão nem usarão a chave de encapsulamento, mesmo que o material da chave seja o mesmo.

Usando várias regiões AWS KMS keys

Você pode usar chaves multirregionais AWS Key Management Service (AWS KMS) como chaves de encapsulamento no. AWS Encryption SDK Se você criptografar com uma chave multirregional em uma Região da AWS, poderá descriptografar usando uma chave multirregional relacionada em outra. Região da AWS O suporte para chaves multirregionais foi introduzido na versão 2.3. x do AWS Encryption SDK e versão 3.0. x da CLI AWS de criptografia.

AWS KMS As chaves 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, AWS Encryption SDK inclui chaveiros com AWS KMS reconhecimento de várias regiões e fornecedores de chaves mestras. O novo multi-Region-aware símbolo em cada linguagem de programação oferece suporte às chaves de região única e multirregião.

- Para chaves de região única, o multi-Region-aware símbolo se comporta exatamente como o AWS KMS chaveiro de região única e o provedor da chave mestra. Ele tenta descriptografar o texto cifrado somente com a chave de região única que criptografou os dados.
- 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 de réplica multirregional relacionada na região especificada.

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

Você também pode usar uma chave multirregional com os fornecedores padrão de AWS KMS chaveiros de região única e chave mestra. No entanto, deve usar a mesma chave multirregional na mesma região para criptografar e descriptografar. Os tokens de autenticação de região única e os provedores de chaves mestras tentam descriptografar o texto cifrado somente com as chaves que criptografaram os dados.

Os exemplos a seguir mostram como criptografar e descriptografar dados usando chaves multirregionais e os novos fornecedores de multi-Region-aware chaveiros e chaves mestras. Esses exemplos criptografam dados na `us-east-1` região e descriptografam os dados na região usando chaves de réplica `us-west-2` multirregionais relacionadas em cada região. Antes de executar esses exemplos, substitua o exemplo de ARN de chave multirregional por um valor válido da sua Conta da AWS.

C

Para criptografar com uma chave multirregional, use o método `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` para instanciar o token de autenticação. Especifique uma chave multirregional.

Esse exemplo simples não inclui um contexto de criptografia. Para obter um exemplo que usa um contexto de criptografia em C, consulte Criptografar e descriptografar strings.

Para ver um exemplo completo, consulte kms_multi_region_keys.cpp no AWS Encryption SDK for C repositório em GitHub.

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

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

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

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

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

aws_cryptosdk_keyring_release(mrk_keyring);

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

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

C# / .NET

Para criptografar com uma chave multirregional na região Leste dos EUA (Norte da Virgínia) (us-east-1), instancie um `CreateAwsKmsMrkKeyringInput` objeto com um identificador de chave para a chave multirregional e um cliente para a região especificada. AWS KMS Em seguida, use o método `CreateAwsKmsMrkKeyring()` para criar o token de autenticação.

O método `CreateAwsKmsMrkKeyring()` cria um token de autenticação com exatamente uma chave multirregional. Para criptografar com várias chaves de encapsulamento, incluindo uma chave multirregional, use o método `CreateAwsKmsMrkMultiKeyring()`.

Para obter um exemplo completo, consulte AwsKmsMrkKeyringExample.cs no AWS Encryption SDK repositório.NET em. GitHub

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

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();


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

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

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

// Encrypt your plaintext data.
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = mrkEncryptKeyring,
    EncryptionContext = encryptionContext
```

```
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

AWS Encryption CLI

Este exemplo criptografa o arquivo `hello.txt` com uma chave multirregional na região us-east-1. Como o exemplo especifica um ARN de chave que tem um elemento de região, esse exemplo não usa oatributo region do parâmetro `--wrapping-keys`.

Quando o ID da chave de encapsulamento não especifica uma região, você pode usar o atributo region de `--wrapping-keys` para especificar a região, como `--wrapping-keys key=$keyID region=us-east-1`.

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

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

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

Java

Para criptografar com uma chave multirregional, instancie uma chave `AwsKmsMrkAwareMasterKeyProvider` e especifique uma chave multirregional.

Para ver um exemplo completo, consulte BasicMultiRegionKeyEncryptionExample.javano AWS Encryption SDK for Java repositório em. GitHub

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

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

// Multi-Region keys have a distinctive key ID that begins with 'mrk'
// Specify a multi-Region key in us-east-1
```

```
final String mrkUSEast1 = "arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab";

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

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

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

JavaScript Browser

Para criptografar com uma chave mulirregional, use o método `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` para criar o token de autenticação e especificar uma chave multirregional.

Para ver um exemplo completo, consulte kms_multi_region_simple.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

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


/* Instantiate an AWS Encryption SDK client */
const { encrypt } = buildClient(
```

```
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

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

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


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

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


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

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

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

JavaScript Node.js

Para criptografar com uma chave mulirregional, use o método `buildAwsKmsMrkAwareStrictMultiKeyringNode()` para criar o token de autenticação e especificar uma chave multirregional.

Para ver um exemplo completo, consulte kms_multi_region_simple.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

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

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

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

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

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

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

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

Python

Para criptografar com uma chave AWS KMS multirregional, use o `MRKAwareStrictAwsKmsMasterKeyProvider()` método e especifique uma chave multirregional.

Para ver um exemplo completo, consulte mrk_aware_kms_provider.py no AWS Encryption SDK for Python repositório em GitHub.

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

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

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

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

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

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

Em seguida, mova seu texto cifrado para a região `us-west-2`. Não é necessário recriptografar o texto cifrado.

Para descriptografar o texto cifrado no modo estrito na `us-west-2` região, instancie o símbolo multi-Region-aware com o ARN da chave multirregional relacionada na região. `us-west-2` Se

você especificar o ARN da chave de uma chave multirregional relacionada em uma região diferente (incluindo `us-east-1` onde ela foi criptografada), 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.

Antes de executar esses exemplos, substitua o exemplo de chave multirregional ARN por um valor válido de seu. Conta da AWS

C

Para descriptografar no modo estrito com uma chave multirregional, use o método `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()` para instanciar o token de autenticação. Especifique a chave multirregional relacionada na região local (us-west-2).

Para ver um exemplo completo, consulte kms_multi_region_keys.cpp no AWS Encryption SDK for C repositório em GitHub.

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

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

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

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

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

aws_cryptosdk_session_set_commitment_policy(session,
    COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT);

aws_cryptosdk_keyring_release(mrk_keyring);

/* Decrypt the ciphertext
 *   aws_cryptosdk_session_process_full is designed for non-streaming data
```

```
 */
aws_cryptosdk_session_process_full(
    session, plaintext, plaintext_buf_sz, &plaintext_len, ciphertext,
 ciphertext_len));

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

C# / .NET

Para descriptografar no modo estrito com uma única chave multirregional, use os mesmos construtores e métodos usados para montar a entrada e criar o token de autenticação para criptografia. Instancie um `CreateAwsKmsMrkKeyringInput` objeto com o ARN da chave de uma chave multirregional relacionada e um AWS KMS cliente para a região Oeste dos EUA (Oregon) (us-west-2). Em seguida, use o método `CreateAwsKmsMrkKeyring()` para criar um token de autenticação multirregional com uma chave do KMS multirregional.

Para obter um exemplo completo, consulte AwsKmsMrkKeyringExample.cs no AWS Encryption SDK repositório.NET em. GitHub

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

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

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

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

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

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

AWS Encryption CLI

Para descriptografar com a chave multirregional relacionada na região us-west-2, use o atributo key do parâmetro `--wrapping-keys` para especificar o ARN da chave.

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

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

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

Java

Para descriptografar no modo estrito, instancie uma `AwsKmsMrkAwareMasterKeyProvider` e especifique a chave multirregional na região local (us-west-2).

Para ver um exemplo completo, consulte BasicMultiRegionKeyEncryptionExample.java no AWS Encryption SDK for Java repositório em. GitHub

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

// Instantiate the client
final AwsCrypto crypto = AwsCrypto.builder()
```

```
    .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
    .build();

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

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

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

JavaScript Browser

Para descriptografar em modo estrito, use o método `buildAwsKmsMrkAwareStrictMultiKeyringBrowser()` para criar o token de autenticação e especifique a chave multirregional na região local (us-west-2).

Para ver um exemplo completo, consulte kms_multi_region_simple.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

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


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

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

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


/* Specify a multi-Region key in us-west-2 */
const multiRegionUsWestKey =
    'arn:aws:kms:us-west-2:111122223333:key/mrk-1234abcd12ab34cd56ef1234567890ab'

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


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

JavaScript Node.js

Para descriptografar em modo estrito, use o método `buildAwsKmsMrkAwareStrictMultiKeyringNode()` para criar o token de autenticação e especifique a chave multirregional na região local (us-west-2).

Para ver um exemplo completo, consulte kms_multi_region_simple.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

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

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

```
)

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

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

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

Python

Para descriptografar no modo estrito, use o método `MRKAwareStrictAwsKmsMasterKeyProvider()` para criar o provedor de chave mestra. Especifique a chave multirregional relacionada na região local (us-west-2).

Para ver um exemplo completo, consulte mrk_aware_kms_provider.py no AWS Encryption SDK for Python repositório em GitHub.

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

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

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

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

# Decrypt your ciphertext
```

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

Você também pode descriptografar no modo de descoberta com chaves 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, eles não AWS Encryption SDK tentarão fazer uma chamada entre regiões para a chave multirregional usada para criptografia.

> **Note**
>
> Se você usar um multi-Region-aware símbolo no modo de descoberta para criptografar dados, a operação de criptografia falhará.

O exemplo a seguir mostra como descriptografar com o multi-Region-aware símbolo no modo de descoberta. Como você não especifica um AWS KMS key, eles AWS Encryption SDK devem obter a região de uma fonte diferente. Quando possível, especifique explicitamente a região local. Caso contrário, AWS Encryption SDK obtém a região local da região configurada no AWS SDK para sua linguagem de programação.

Antes de executar esses exemplos, substitua o exemplo de ID da conta e ARN da chave multirregional por valores válidos do seu. Conta da AWS

C

Para descriptografar no modo de descoberta com uma chave multirregional, use o método `Aws::Cryptosdk::KmsMrkAwareSymmetricKeyring::Builder()`para criar o token de autenticação e o método `Aws::Cryptosdk::KmsKeyring::DiscoveryFilter::Builder()` para criar o filtro de descoberta. Para especificar a região local, defina uma `ClientConfiguration` e especifique-a no AWS KMS cliente.

Para ver um exemplo completo, consulte kms_multi_region_keys.cpp no AWS Encryption SDK for C repositório em GitHub.

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

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

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

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

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

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

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

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

aws_cryptosdk_keyring_release(mrk_keyring);
commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
/* Decrypt the ciphertext
 *   aws_cryptosdk_session_process_full is designed for non-streaming data
 */
aws_cryptosdk_session_process_full(
```

```
    session, plaintext, plaintext_buf_sz, &plaintext_len, ciphertext,
 ciphertext_len));

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

C# / .NET

Para criar um chaveiro de multi-Region-aware descoberta no AWS Encryption SDK para.NET, instancie um `CreateAwsKmsMrkDiscoveryKeyringInput` objeto que leve um AWS KMS cliente para um determinado Região da AWS cliente e um filtro de descoberta opcional que limita as chaves KMS a uma partição e conta específicas AWS . Em seguida, chame o método `CreateAwsKmsMrkDiscoveryKeyring()` do objeto . Para obter um exemplo completo, consulte AwsKmsMrkDiscoveryKeyringExample.cs no AWS Encryption SDK repositório.NET em. GitHub

Para criar um chaveiro de multi-Region-aware descoberta para mais de um Região da AWS, use o `CreateAwsKmsMrkDiscoveryMultiKeyring()` método para criar um chaveiro múltiplo ou use `CreateAwsKmsMrkDiscoveryKeyring()` para criar vários chaveiros de multi-Region-aware descoberta e, em seguida, use o `CreateMultiKeyring()` método para combiná-los em um chaveiro múltiplo.

Para ver um exemplo, consulte AwsKmsMrkDiscoveryMultiKeyringExample.cs.

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

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

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

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

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


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

AWS Encryption CLI

Para descriptografar no modo de descoberta, use o atributo discovery do parâmetro. `--wrapping-keys` Os atributos discovery-account e discovery-partition criam um filtro de descoberta que é opcional, mas recomendado.

Para especificar a região, esse comando inclui o atributo region do parâmetro `--wrapping-keys`.

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

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

Java

Para especificar a região local, use o parâmetro `builder().withDiscoveryMrkRegion`. Caso contrário, o AWS Encryption SDK obtém a região local da região configurada no AWS SDK para Java.

Para ver um exemplo completo, consulte DiscoveryMultiRegionDecryptionExample.java no AWS Encryption SDK for Java repositório em. GitHub

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

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

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

AwsKmsMrkAwareMasterKeyProvider mrkDiscoveryProvider =
 AwsKmsMrkAwareMasterKeyProvider
    .builder()
    .withDiscoveryMrkRegion(Region.US_WEST_2)
    .buildDiscovery(discoveryFilter);

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

JavaScript Browser

Para descriptografar no modo de descoberta com uma chave multirregional simétrica, use o método `AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser()`.

Para ver um exemplo completo, consulte kms_multi_region_discovery.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

import {
  AwsKmsMrkAwareSymmetricDiscoveryKeyringBrowser,
  buildClient,
  CommitmentPolicy,
  KMS,
```

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


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

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

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


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


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


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

JavaScript Node.js

Para descriptografar no modo de descoberta com uma chave multirregional simétrica, use o método `AwsKmsMrkAwareSymmetricDiscoveryKeyringNode()`.

Para ver um exemplo completo, consulte kms_multi_region_discovery.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

import {
  AwsKmsMrkAwareSymmetricDiscoveryKeyringNode,
  buildClient,
```

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

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

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

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

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

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

Python

Para descriptografar no modo de descoberta com uma chave multirregional, use o método `MRKAwareDiscoveryAwsKmsMasterKeyProvider()`.

Para ver um exemplo completo, consulte mrk_aware_kms_provider.py no AWS Encryption SDK for Python repositório em GitHub.

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

# Instantiate the client
client = aws_encryption_sdk.EncryptionSDKClient()

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

# Use the multi-Region method to create the master key provider
```

```
# in discovery mode
mrk_discovery_key_provider =
 MRKAwareDiscoveryAwsKmsMasterKeyProvider(**decrypt_kwargs)

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

Escolher um pacote de algoritmo

O AWS Encryption SDK suporta vários algoritmos de criptografia simétrica e assimétrica para criptografar suas chaves de dados sob as chaves de encapsulamento que você especificar. No entanto, quando ele usa essas chaves de dados para criptografar seus dados, o AWS Encryption SDK padrão é um conjunto de algoritmos recomendado que usa o algoritmo AES-GCM com derivação de chaves, assinaturas digitais e comprometimento de chaves. Embora o pacote de algoritmos padrão seja adequado para a maioria das aplicações, você pode 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 pacotes de algoritmos compatíveis com o AWS Encryption SDK , consulte Suítes de algoritmos compatíveis no AWS Encryption SDK.

Os exemplos a seguir mostram como selecionar um pacote de algoritmos alternativo ao criptografar. Esses exemplos selecionam um pacote de algoritmos recomendado AES-GCM com derivação e confirmação de chaves, mas sem assinaturas digitais. Ao criptografar com um pacote de algoritmos que não inclui assinaturas digitais, use o modo de descriptografia somente sem assinatura ao descriptografar. Esse modo, que falha se encontrar um texto cifrado assinado, é mais útil ao transmitir a decodificação.

C

Para especificar um conjunto alternativo de algoritmos no AWS Encryption SDK for C, você deve criar um CMM explicitamente. Em seguida, use o `aws_cryptosdk_default_cmm_set_alg_id` com o CMM e o pacote de algoritmos selecionado.

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

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

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

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

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

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

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

Ao descriptografar dados que foram criptografados sem assinaturas digitais, use. `AWS_CRYPTOSDK_DECRYPT_UNSIGNED` Isso faz com que a descriptografia falhe se encontrar um texto cifrado assinado.

```
/* Decrypt unsigned streaming data */
```

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

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

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

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

if (!session) {
    return AWS_OP_ERR;
}

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

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

C# / .NET

Para especificar um conjunto alternativo de algoritmos no AWS Encryption SDK para.NET, especifique a `AlgorithmSuiteId` propriedade de um EncryptInputobjeto. O AWS Encryption

SDK for.NET inclui constantes que você pode usar para identificar seu conjunto de algoritmos preferido.

O AWS Encryption SDK for.NET não tem um método para detectar texto cifrado assinado durante a decodificação de streaming porque essa biblioteca não oferece suporte a dados de streaming.

```
// Specify an algorithm suite without signing

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

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

// Encrypt your plaintext data
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    AlgorithmSuiteId = AlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY
};
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

AWS Encryption CLI

Ao criptografar o arquivo `hello.txt`, este exemplo usa o parâmetro `--algorithm` para especificar um pacote de algoritmos sem assinaturas digitais.

```
# Specify an algorithm suite without signing

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

$ aws-encryption-cli --encrypt \
                     --input hello.txt \
```

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

Ao descriptografar, este exemplo usa o parâmetro `--decrypt-unsigned`. Esse parâmetro é recomendado para garantir que você esteja descriptografando texto cifrado não assinado, especialmente com a CLI, que está sempre transmitindo entrada e saída.

```
# Decrypt unsigned streaming data

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

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

Java

Para especificar um pacote ade algoritmos alternativo, use o método `AwsCrypto.builder().withEncryptionAlgorithm()`. Este exemplo usa um pacote de algoritmos alternativo sem assinaturas digitais.

```
// Specify an algorithm suite without signing

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

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

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

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

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

Ao transmitir dados para descriptografia, use o método `createUnsignedMessageDecryptingStream()` para garantir que todo texto cifrado que você está descriptografando não esteja assinado.

```
// Decrypt unsigned streaming data

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

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

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

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

```
decryptingStream.close();
```

JavaScript Browser

Para especificar um pacote de algoritmos alternativo, use o parâmetro `suiteId` com um valor enum `AlgorithmSuiteIdentifier`.

```
// Specify an algorithm suite without signing

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

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

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

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

Ao descriptografar, use o método padrão `decrypt`. O AWS Encryption SDK para JavaScript no navegador não tem um modo `decrypt-unsigned`porque o navegador não é compatível com streaming.

```
// Decrypt unsigned streaming data

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

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

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

JavaScript Node.js

Para especificar um pacote de algoritmos alternativo, use o parâmetro `suiteId` com um valor enum `AlgorithmSuiteIdentifier`.

```
// Specify an algorithm suite without signing

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

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

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

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

Ao descriptografar dados que foram criptografados sem assinaturas digitais, use o Stream. decryptUnsignedMessage Esse método falhará se encontrar texto cifrado assinado.

```
// Decrypt unsigned streaming data

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

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

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

Python

Para especificar um algoritmo de criptografia alternativo, use o parâmetro `algorithm` com um valor enum `Algorithm`.

```
# Specify an algorithm suite without signing

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

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

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

Ao descriptografar mensagens que foram criptografadas sem assinaturas digitais, use o modo de `decrypt-unsigned` streaming, especialmente ao descriptografar durante o streaming.

```
# Decrypt unsigned streaming data

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

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

# Decrypt with decrypt-unsigned
```

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

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

Rust

Para especificar um conjunto alternativo de algoritmos no AWS Encryption SDK for Rust, especifique a `algorithm_suite_id` propriedade em sua solicitação de criptografia.

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

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

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

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

// Create Raw AES keyring
```

```
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

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

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

Go

```
import (
    "context"

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

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

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

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

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

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

Limitar as chaves de dados criptografadas

Você pode limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Esse atributo de práticas recomendadas pode ajudar você a detectar um token de autenticação mal configurado ao criptografar ou um texto cifrado malicioso ao descriptografar. Isso também evita chamadas desnecessárias, caras e potencialmente exaustivas para sua infraestrutura principal. Limitar as chaves de dados criptografadas tem mais valor quando você está descriptografando mensagens de uma fonte não confiável.

Embora a maioria das mensagens criptografadas tenha uma chave de dados criptografada para cada chave de encapsulamento usada na criptografia, uma mensagem criptografada pode conter até 65.535 chaves de dados criptografadas. Um agente mal-intencionado pode criar uma mensagem criptografada com milhares de chaves de dados criptografadas, nenhuma delas capaz de ser descriptografada. Como resultado, eles AWS Encryption SDK tentariam descriptografar cada chave de dados criptografada até esgotar as chaves de dados criptografadas na mensagem.

Para limitar as chaves de dados criptografadas, use o parâmetro `MaxEncryptedDataKeys`. Esse parâmetro está disponível para todas as linguagens de programação compatíveis a partir das versões 1.9.x e 2.2.x do AWS Encryption SDK. Ele é opcional e válido ao criptografar e descriptografar. Os exemplos a seguir descriptografam dados que foram criptografados sob três chaves de encapsulamento diferentes. O valor de `MaxEncryptedDataKeys` foi definido como 3.

C

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

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

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

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

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

C# / .NET

Para limitar as chaves de dados criptografadas no AWS Encryption SDK para.NET, instancie um cliente AWS Encryption SDK para o.NET e defina seu `MaxEncryptedDataKeys` parâmetro opcional com o valor desejado. Em seguida, chame o método`Decrypt()` na instância do AWS Encryption SDK configurada.

```
// Decrypt with limited data keys

// Instantiate the material providers
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{
    MaxEncryptedDataKeys = 3
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

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

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

AWS Encryption CLI

```
# Decrypt with limited encrypted data keys

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

Java

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

// Create an AWS KMS master key provider
final KmsMasterKeyProvider keyProvider = KmsMasterKeyProvider.builder()
      .buildStrict(keyArn1, keyArn2, keyArn3);

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

JavaScript Browser

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

declare const credentials: {
```

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

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

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

JavaScript Node.js

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

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

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

Python

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

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

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

Rust

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

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

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

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

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

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

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

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

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

let multi_keyring = mpl
    .create_multi_keyring()
```

```
    .generator(generator_keyring)
    .child_keyrings(raw_aes_keyrings)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

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

//  Generate `maxEncryptedDataKeys` raw AES keyrings to use with your keyring
rawAESKeyrings := make([]mpltypes.IKeyring, 0, maxEncryptedDataKeys)
var i int64 = 0
for i < maxEncryptedDataKeys {
    key, err := generate256KeyBytesAES()
```

```
 if err != nil {
  panic(err)
 }
    aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
  KeyName:      keyName,
  KeyNamespace: keyNamespace,
  WrappingKey:  key,
  WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
 }
 aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
    if err != nil {
  panic(err)
 }
    rawAESKeyrings = append(rawAESKeyrings, aesKeyring)
    i++
}

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

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, 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 precisar descriptografar no modo de descoberta, recomendamos que você sempre use um filtro de descoberta, que limita as chaves KMS que podem ser usadas às de uma partição especificada. 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`

Os exemplos nesta seção mostram como criar um filtro de descoberta. Antes de usar o código, substitua os valores de exemplo por valores válidos para a partição Conta da AWS e.

C

Para obter um exemplo completo, consulte: kms_discovery.cpp no AWS Encryption SDK for C.

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

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

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

C# / .NET

Para obter um exemplo completo, consulte DiscoveryFilterExample.cs no AWS Encryption SDK for.NET.

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

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

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

AWS Encryption CLI

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

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

Java

Para obter um exemplo completo, consulte DiscoveryDecryptionExample.java no AWS Encryption SDK for Java.

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

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

JavaScript (Node and Browser)

Para obter exemplos completos, consulte kms_filtered_discovery.ts (Node.js) e kms_multi_region_discovery.ts (Navegador) no. AWS Encryption SDK para JavaScript.

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

Python

Para obter um exemplo completo, consulte discovery_kms_provider.py no AWS Encryption SDK for Python.

```
# Create the discovery filter and specify the region
```

```
decrypt_kwargs = dict(
        discovery_filter=DiscoveryFilter(account_ids="111122223333",
 partition="aws"),
        discovery_region="us-west-2",
    )
```

Rust

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

Go

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

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

Configurando o contexto de criptografia necessário (CMM)

Você pode usar o contexto de criptografia necessário CMM para exigir contextos de criptografia em suas operações criptográficas. Um contexto de criptografia é um conjunto de pares de chave/valor não secretos. O contexto de criptografia é associado de maneira criptográfica aos dados criptografados de forma que o mesmo contexto de criptografia é necessário para descriptografar o campo. Ao usar o CMM de contexto de criptografia necessário, é possível especificar uma ou mais chaves de contexto de criptografia necessárias (chaves obrigatórias) que devem ser incluídas em todas as chamadas de criptografia e descriptografia.

> **Note**
>
> O contexto de criptografia necessário (CMM) só é suportado pelas seguintes versões:

- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go

Se você criptografar dados usando o contexto de criptografia necessário CMM, só poderá descriptografá-los com uma dessas versões suportadas.

Ao criptografar, AWS Encryption SDK verifica se todas as chaves de contexto de criptografia necessárias estão incluídas no contexto de criptografia que você especificou. Os AWS Encryption SDK sinais dos contextos de criptografia que você especificou. Somente os pares de chave/valor que não são chaves obrigatórias são serializados e armazenados em texto simples no cabeçalho da mensagem criptografada retornada pela operação de criptografia.

Ao descriptografar, você deve fornecer um contexto de criptografia que contenha todos os pares de chave/valor que representam as chaves necessárias. O AWS Encryption SDK usa esse contexto de criptografia e os pares de valores-chave armazenados no cabeçalho da mensagem criptografada para reconstruir o contexto de criptografia original que você especificou na operação de criptografia. Se o AWS Encryption SDK não puder reconstruir o contexto de criptografia original, a operação de descriptografia falhará. Se você fornecer um par de chave/valor que contenha a chave necessária com um valor incorreto, a mensagem criptografada não poderá ser descriptografada. Você deve fornecer o mesmo par de chave/valor especificado na criptografia.

Important

Considere cuidadosamente quais valores você escolhe para as chaves necessárias no contexto de criptografia. Você deverá fornecer as mesmas chaves e os valores correspondentes novamente na descriptografia. Se você não conseguir reproduzir as chaves necessárias, a mensagem criptografada não poderá ser descriptografada.

Os exemplos a seguir inicializam um AWS KMS chaveiro com o contexto de criptografia necessário CMM.

C# / .NET

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

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

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

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

var createCMMInput = new CreateRequiredEncryptionContextCMMInput
{
    UnderlyingCMM = mpl.CreateDefaultCryptographicMaterialsManager(new
 CreateDefaultCryptographicMaterialsManagerInput{Keyring = kmsKeyring}),
    // If you pass in a keyring but no underlying cmm, it will result in a failure
 because only cmm is supported.
    RequiredEncryptionContextKeys = new List<string>(encryptionContext.Keys)
};

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

Java

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

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

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

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

// Create the required encryption context CMM
ICryptographicMaterialsManager cmm =
    materialProviders.CreateDefaultCryptographicMaterialsManager(
        CreateDefaultCryptographicMaterialsManagerInput.builder()
            .keyring(kmsKeyring)
            .build()
    );
ICryptographicMaterialsManager requiredCMM =
    materialProviders.CreateRequiredEncryptionContextCMM(
        CreateRequiredEncryptionContextCMMInput.builder()
            .requiredEncryptionContextKeys(requiredEncryptionContextKeys)
            .underlyingCMM(cmm)
            .build()
        );
```

Python

Para usar o CMM AWS Encryption SDK for Python com o contexto de criptografia necessário, você também deve usar a biblioteca de fornecedores de materiais (MPL).

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

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

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

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

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

# Create the required encryption context CMM
underlying_cmm: ICryptographicMaterialsManager = \
    mpl.create_default_cryptographic_materials_manager(
        CreateDefaultCryptographicMaterialsManagerInput(
            keyring=kms_keyring
        )
    )

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

Rust

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

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

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

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

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

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

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

// Create the required encryption context CMM
let underlying_cmm = mpl
    .create_default_cryptographic_materials_manager()
```

```
    .keyring(kms_keyring)
    .send()
    .await?;

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

Go

```
import (
    "context"

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

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

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

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

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

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

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

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

```
if err != nil {
    panic(err)
}
```

Como definir uma política de compromisso

Uma política de compromisso é uma definição de configuração que determina se sua aplicação criptografa e descriptografa com confirmação de chave. Criptografar e descriptografar com o confirmação de chave é uma prática recomendada do AWS Encryption SDK.

Definir e ajustar sua política de compromisso é uma etapa fundamental na migração das versões 1.7.x e anteriores do AWS Encryption SDK àz versões 2.0x posteriores. Essa progressão é explicada em detalhes no tópico de migração.

O valor padrão da política de compromisso nas versões mais recentes do AWS Encryption SDK (a partir da versão 2.0.x), `RequireEncryptRequireDecrypt`, é ideal para a maioria das situações. No entanto, se você precisar descriptografar um texto cifrado que foi criptografado sem confirmação de chave, talvez seja necessário alterar sua política de compromisso para `RequireEncryptAllowDecrypt`. Para obter exemplos de como definir uma política de compromisso em cada linguagem de programação, consulte Como definir sua política de compromisso.

Trabalhar com streaming de dados

Ao transmitir dados para decodificação, lembre-se de que eles AWS Encryption SDK retornam texto simples descriptografado após a conclusão das verificações de integridade, mas antes da verificação da assinatura digital. Para garantir que você não retorne ou use texto simples até que a assinatura seja verificada, recomendamos que você armazene o texto simples transmitido até que todo o processo de descriptografia seja concluído.

Esse problema surge somente quando você está transmitindo texto cifrado para decodificação e somente quando você está usando um pacote de algoritmos, como opacote de algoritmos padrão, que inclui assinaturas digitais.

Para facilitar o armazenamento em buffer, algumas implementações de AWS Encryption SDK linguagem, como AWS Encryption SDK para JavaScript no Node.js, incluem um recurso de buffer como parte do método decrypt. A CLI de criptografia da AWS , que sempre transmite entrada e

saída, introduziu um parâmetro `--buffer` nas versões 1.9.x e 2.2.x.. Em outras implementações de linguagem, você pode usar os atributos de buffer existentes. (O AWS Encryption SDK for.NET não oferece suporte a streaming.)

Se você estiver usando um pacote de algoritmos sem assinaturas digitais, certifique-se de usar o atributo `decrypt-unsigned` em cada implementação de linguagem. Esse atributo descriptografa o texto cifrado, mas falhará se encontrar um texto cifrado assinado. Para obter detalhes, consulte Escolher um pacote de algoritmo.

Armazenamento em cache de chaves de dados

Em geral, a reutilização de chaves de dados é desencorajada, mas AWS Encryption SDK oferece uma opção de armazenamento em cache de chaves de dados que fornece reutilização limitada de chaves de dados. O armazenamento em cache de chaves de dados pode melhorar a performance de algumas aplicações e reduzir as chamadas para sua infraestrutura de chaves. Antes de usar o armazenamento em cache de chaves de dados em produção, ajuste os limites de segurança e teste, para garantir que os benefícios superem as desvantagens da reutilização de chaves de dados.

Lojas principais no AWS Encryption SDK

No AWS Encryption SDK, 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.

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 criptográficasGetActiveBranchKey, comoGetBranchKeyVersion, 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 “Crie 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 AWS Encryption SDK suporta as seguintes configurações de ações de armazenamento principais.

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 suak`msConfiguration`, 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);
```

Python

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

Rust

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

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

Go

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

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

Configuração de descoberta

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);
```

Python

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

Rust

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

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

Go

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

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

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

Crie 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 chaveiro hierárquico gera uma chave de dados exclusiva para cada solicitação de criptografia e criptografa cada chave de dados com uma chave de empacotamento 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.

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 devem 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.

Crie uma chave de ramificação

A operação a seguir cria uma nova chave de ramificação ativa usando a chave KMS que você especificou na configuração de ações do armazenamento de chaves e adiciona a chave de ramificação ativa à tabela do DynamoDB que serve como seu 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 chave-valor não secretos que fornecem 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
 });
```

Python

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

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

Rust

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

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

Go

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

```
branchKey, err := keyStore.CreateKey(context.Background(),
 keystoretypes.CreateKeyInput{
    BranchKeyIdentifier: &customBranchKeyId,
    EncryptionContext:   additional_encryption_context,
})
if err != nil {
    return "", err
}
```

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"
  }
```

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 de chave de ramificação ativa 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 agrupamento, mas ainda podem ser consultadas e fornecer chaves de agrupamento para descriptografar as chaves de dados que criptografaram enquanto estavam ativas.

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.

Gire uma chave de ramificação ativa

Use a `VersionKey` operação para girar 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});
```

Python

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

Rust

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

Go

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

```
}
```

Tokens de autenticação

As implementações de linguagem de programação suportadas usam 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 mensagem, bem como as chaves de encapsulamento 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. Você pode usar os tokens de autenticação fornecidos pelo SDK ou elaborar seus próprios tokens de autenticação personalizados compatíveis.

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. Para obter detalhes, consulte o tópico Interface do token de autenticação na Especificação do AWS Encryption SDK .

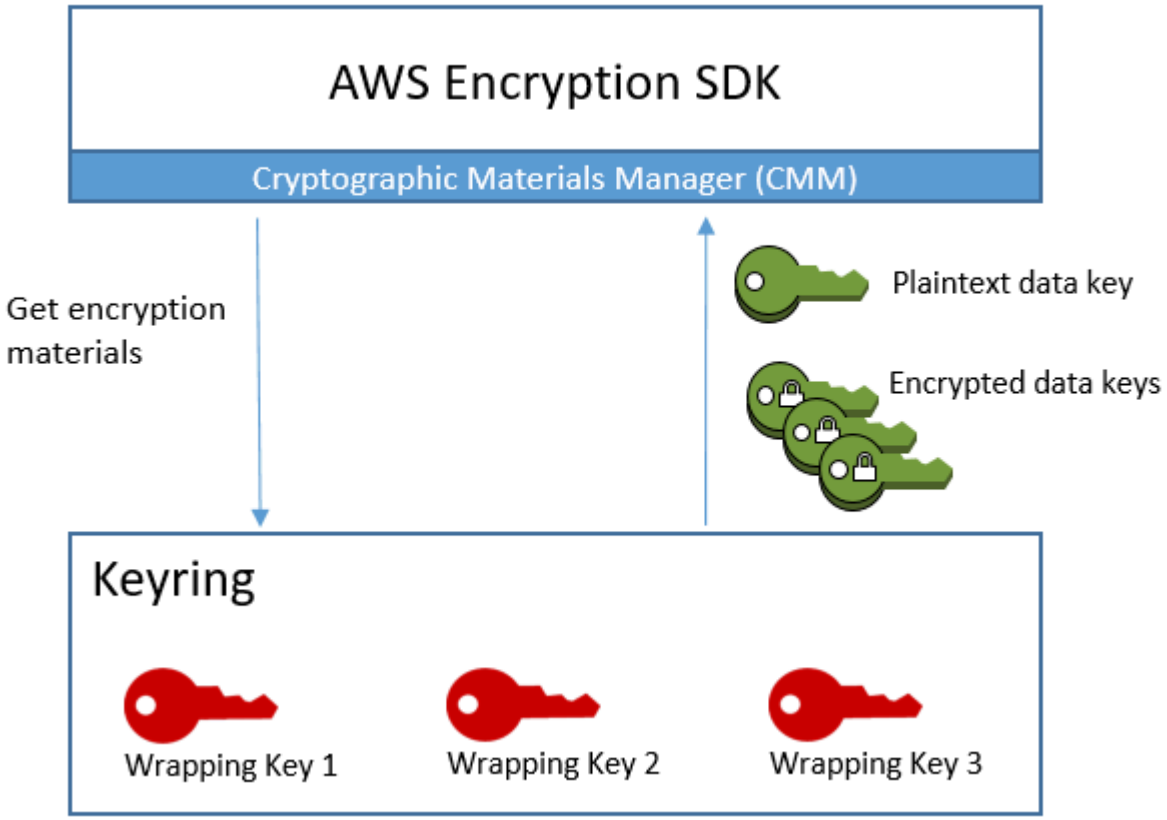
Os chaveiros desempenham o papel de chaves mestras e provedores de chaves mestras usados em outras implementações de linguagens de programação. Se você usar diferentes implementações de linguagem do AWS Encryption SDK para criptografar e descriptografar seus dados, certifique-se de usar tokens de autenticação e provedores de chaves mestras compatíveis. Para obter detalhes, consulte Compatibilidade dos tokens de autenticação.

Este tópico explica como usar o recurso de chaveiro do AWS Encryption SDK e como escolher um chaveiro.

Como os tokens de autenticação funcionam

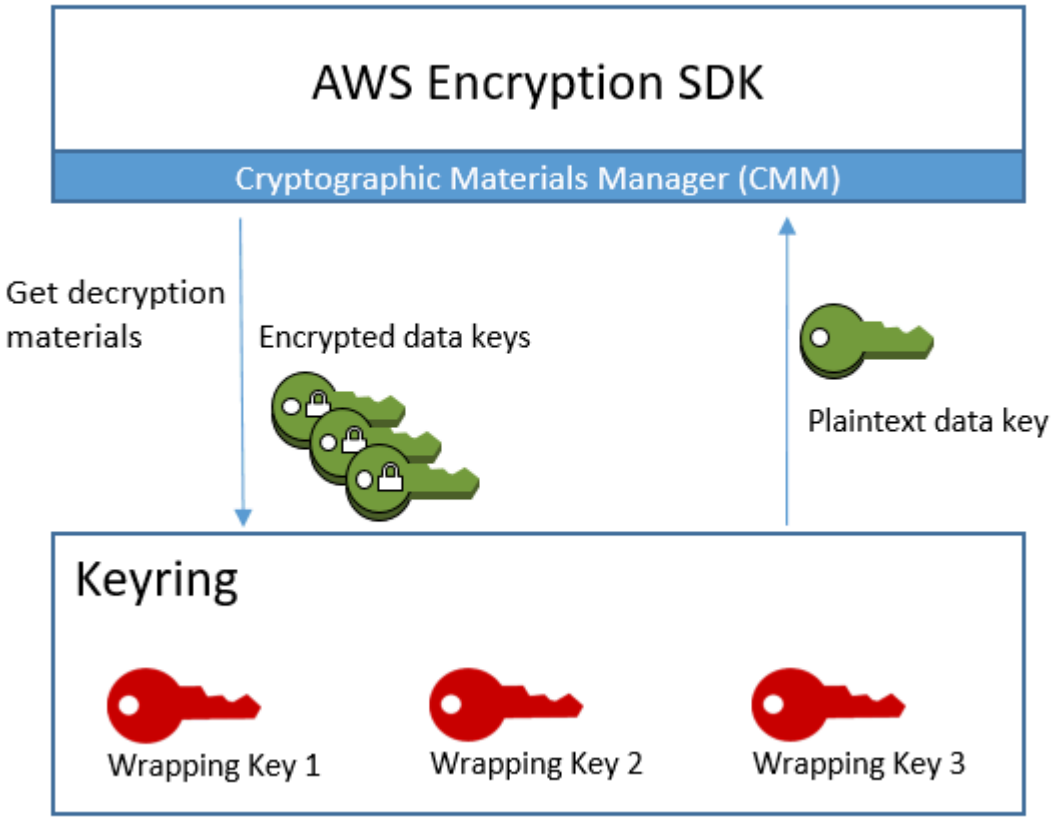
Quando você criptografa dados, AWS Encryption SDK ele solicita materiais de criptografia ao chaveiro. O token de autenticação retorna uma chave dados de texto simples e uma cópia da chave de dados que é criptografada por cada uma das chaves de encapsulamento no token de

autenticação. O AWS Encryption SDK usa a chave de texto simples para criptografar os dados e, em seguida, destrói a chave de dados de texto sem formatação. Em seguida, AWS Encryption SDK retorna uma mensagem criptografada que inclui as chaves de dados criptografadas e os dados criptografados.



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

Ele AWS Encryption SDK passa as chaves de dados criptografadas da mensagem criptografada 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 AWS Encryption SDK usa a chave de dados de 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. É possível descriptografar os dados usando um token de autenticação com qualquer uma das chaves de encapsulamento no multitoken de autenticação.

Compatibilidade dos tokens de autenticação

Embora as diferentes implementações de linguagem do AWS Encryption SDK tenham algumas diferenças arquitetônicas, elas são totalmente compatíveis, sujeitas às restrições de linguagem. Você pode criptografar seus dados usando uma implementação de linguagem e descriptografá-los em qualquer outra implementação de linguagem. No entanto, é necessário usar as mesmas chaves de encapsulamento, ou correspondentes, para criptografar e descriptografar suas chaves de dados. Para obter informações sobre restrições de linguagem, consulte o tópico sobre cada implementação de linguagem, como the section called “Compatibilidade” no AWS Encryption SDK para JavaScript tópico.

Os chaveiros são compatíveis com as seguintes linguagens de programação:

- AWS Encryption SDK for C
- AWS Encryption SDK para JavaScript
- AWS Encryption SDK para o.NET
- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
- AWS Encryption SDK para Rust
- AWS Encryption SDK para Go

Requisitos variados para tokens de autenticação de criptografia

Em implementações de AWS Encryption SDK linguagem diferentes da AWS Encryption SDK for C, todas as chaves agrupadas em um chaveiro de criptografia (ou vários chaveiros) ou provedor de chave mestra devem ser capazes de criptografar a chave de dados. Se alguma chave de encapsulamento 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.

A exceção é a AWS Encryption SDK for C, em que a operação de criptografia ignora um chaveiro de descoberta padrão, mas falha se você especificar um chaveiro de descoberta de várias regiões, sozinho ou em um chaveiro com várias chaves.

Tokens de autenticação e provedores de chaves mestras compatíveis

A tabela a seguir mostra quais chaves mestras e fornecedores de chaves mestras são compatíveis com os chaveiros fornecidos pela empresa AWS Encryption SDK . Qualquer pequena incompatibilidade devido às restrições de linguagem é explicada no tópico sobre a implementação de linguagem.

Token de autenticação:	Provedor de chaves mestras:
AWS KMS chaveiro	KMSMasterChave (Java) KMSMasterKeyProvider (Java)

Token de autenticação:	Provedor de chaves mestras:
	KMSMasterChave (Python) KMSMasterKeyProvider (Python) **Note** O AWS Encryption SDK for Python e AWS Encryption SDK for Java não inclui uma chave mestra ou um provedor de chave mestra que seja equivalente ao chaveiro AWS KMS regional Discovery.
AWS KMS Chaveiro hierárquico	Compatível com as seguintes linguagens e versões de programação: • Versão 3. x do AWS Encryption SDK for Java • Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET • Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL). • Versão 1. x do AWS Encryption SDK para Rust • Versão 0.1. x ou posterior do AWS Encryption SDK for Go
AWS KMS Chaveiro ECDH	Compatível com as seguintes linguagens e versões de programação: • Versão 3. x do AWS Encryption SDK for Java • Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET • Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL). • Versão 1. x do AWS Encryption SDK para Rust • Versão 0.1. x ou posterior do AWS Encryption SDK for Go

Token de autenticação:	Provedor de chaves mestras:
Token de autenticação bruto do AES	Quando são usados com chaves de criptografia simétrica: JceMasterKey(Java) RawMasterKey(Python)
Token de autenticação bruto do RSA	Quando são usados com chaves de criptografia assimétrica: JceMasterKey(Java) RawMasterKey(Python) **Note** O token de autenticação RSA bruto não oferece suporte a chaves do KMS assimétricas. Se você quiser usar chaves RSA KMS assimétricas, versão 4. O x e versões posteriores do AWS Encryption SDK for.NET oferecem suporte a AWS KMS chaveiros que usam criptografia simétrica (`SYMMETRIC_DEFAULT`) ou RSA assimétrica. AWS KMS keys
Chaveiro ECDH bruto	Compatível com as seguintes linguagens e versões de programação: • Versão 3. x do AWS Encryption SDK for Java • Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET • Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL). • Versão 1. x do AWS Encryption SDK para Rust • Versão 0.1. x ou posterior do AWS Encryption SDK for Go

AWS KMS chaveiros

Um AWS KMS chaveiro é usado 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. É recomendável usar um token de autenticação do AWS KMS ou um token de autenticação com propriedades de segurança semelhantes sempre que possível.

Todas as implementações de linguagem de programação que oferecem suporte a chaveiros oferecem suporte a AWS KMS chaveiros que usam chaves KMS de criptografia simétrica. As implementações de linguagem de programação a seguir também oferecem suporte a AWS KMS chaveiros que usam chaves RSA KMS assimétricas:

- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
- Versão 1. x do AWS Encryption SDK para Rust
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go

Se você tentar incluir uma chave do KMS assimétrica em um token de autenticação de criptografia que esteja em outra implementação de linguagem, a chamada de criptografia falhará. Se você incluí-la em um token de autenticação de descriptografia, ele será ignorado.

Você pode usar uma chave AWS KMS multirregional em um AWS KMS chaveiro ou provedor de chave mestra a partir da versão 2.3. x do AWS Encryption SDK e versão 3.0. x da CLI AWS de criptografia. Para obter detalhes e exemplos de uso do multi-Region-aware símbolo, consulteUsando várias regiões AWS KMS keys. Para obter mais informações sobre chaves multirregionais, consulte Usar chaves multirregionais no Guia do Desenvolvedor do AWS Key Management Service .

> **Note**
>
> Todas as menções aos chaveiros KMS AWS Encryption SDK se referem aos chaveiros. AWS KMS

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 mensagens. Quando um AWS KMS chaveiro tem apenas uma chave KMS, essa chave é usada para gerar e criptografar a chave de dados. Ao descriptografar, a chave geradora é opcional e a distinção entre chaves geradoras e chaves adicionais é ignorada.

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 AWS KMS chaveiros
- Identificação AWS KMS keys em um AWS KMS chaveiro
- Criando um AWS KMS chaveiro
- Usando um chaveiro AWS KMS Discovery
- Usando um chaveiro de descoberta AWS KMS regional

Permissões necessárias para AWS KMS chaveiros

O AWS Encryption SDK não requer 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 Acesso e permissões à chave KMS 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.

- Em um chaveiro de criptografia para o AWS Encryption SDK for C, você pode usar um ARN de chave ou um alias ARN para identificar chaves KMS. Em todas as outras implementações de linguagem, você pode usar um ID de chave, ARN de chave, nome de alias ou ARN de alias para criptografar dados.
- Em um token de autenticação de descriptografia, você deve usar um ARN de chave para identificar AWS KMS keys. Esse requisito aplica-se a todas as implementações de linguagem do AWS Encryption SDK. 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. Esse requisito aplica-se a todas as implementações de linguagem do AWS Encryption SDK.

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.

Criando um AWS KMS chaveiro

Você pode configurar cada AWS KMS chaveiro com um único AWS KMS key ou vários AWS KMS keys no mesmo ou em um diferente Contas da AWS e. Regiões da AWS AWS KMS keys Deve ser uma chave KMS de criptografia simétrica (SYMMETRIC_DEFAULT) ou uma chave KMS RSA assimétrica. Também é possível usar uma chave KMS multirregional criptografia simétrica. É possível usar um ou mais tokens de autenticação do AWS KMS em um multitoken de autenticação.

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.)

Em implementações de AWS Encryption SDK linguagem diferentes da AWS Encryption SDK for C, todas as chaves agrupadas em um chaveiro de criptografia ou em vários chaveiros devem ser capazes de criptografar a chave de dados. Se alguma chave de encapsulamento 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. A exceção é a AWS Encryption SDK for C, em que a operação de criptografia ignora um chaveiro de descoberta padrão, mas falha se você especificar um chaveiro de descoberta de várias regiões, sozinho ou em um chaveiro com várias chaves.

Os exemplos a seguir criam um AWS KMS chaveiro com uma chave geradora e uma chave adicional. Tanto a chave geradora quanto a chave adicional são chaves KMS de criptografia simétrica. Esses exemplos usam ARNs a chave para identificar as chaves KMS. Essa é uma prática recomendada para AWS KMS chaveiros usados para criptografia e um requisito para AWS KMS chaveiros usados para decodificação. Para obter detalhes, consulte Identificação AWS KMS keys em um AWS KMS chaveiro.

C

Para identificar um AWS KMS key em um chaveiro de criptografia no AWS Encryption SDK for C, especifique o ARN da chave ou o ARN do alias. Em um token de autenticação de descriptografia,

é necessário usar um ARN de chave. Para obter detalhes, consulte Identificação AWS KMS keys em um AWS KMS chaveiro.

Para obter um exemplo completo, consulte string.cpp.

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

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

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

C# / .NET

Para criar um chaveiro com uma ou mais chaves KMS no AWS Encryption SDK para.NET, use o `CreateAwsKmsMultiKeyring()` método. Este exemplo usa duas chaves do AWS KMS . Para especificar uma chave do KMS, use o parâmetro `Generator`. O parâmetro `KmsKeyIds`, que especifica chaves KMS adicionais, é opcional.

A entrada para este chaveiro não requer um AWS KMS cliente. Em vez disso, AWS Encryption SDK ele usa o AWS KMS cliente padrão para cada região representada por uma chave KMS no chaveiro. Por exemplo, se a chave KMS identificada pelo valor do `Generator` parâmetro estiver na região Oeste dos EUA (Oregon) (`us-west-2`), ela AWS Encryption SDK criará um AWS KMS cliente padrão para a `us-west-2` região. Se você precisar personalizar o cliente do AWS KMS , use o método `CreateAwsKmsKeyring()`.

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK para.NET, você pode usar qualquer identificador de chave válido: um ID de chave, ARN de chave, nome de alias ou ARN de alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

O exemplo a seguir usa a versão 4. x do AWS Encryption SDK para o.NET e o `CreateAwsKmsKeyring()` método para personalizar o AWS KMS cliente.

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

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

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

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

JavaScript Browser

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK para JavaScript, você pode usar qualquer identificador de chave válido: um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

Para ver um exemplo completo, consulte kms_simple.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

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

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

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

JavaScript Node.js

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK para JavaScript, você pode usar qualquer identificador de chave válido: um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

Para ver um exemplo completo, consulte kms_simple.ts no repositório em. AWS Encryption SDK para JavaScript GitHub

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

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

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

const additionalKey = 'alias/exampleAlias'

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

Java

Para criar um chaveiro com uma ou mais AWS KMS chaves, use o `CreateAwsKmsMultiKeyring()` método. Este exemplo usa duas chaves KMS. Para especificar uma chave do KMS, use o parâmetro `generator`. O parâmetro `kmsKeyIds`, que especifica chaves KMS adicionais, é opcional.

A entrada para este chaveiro não requer um AWS KMS cliente. Em vez disso, AWS Encryption SDK ele usa o AWS KMS cliente padrão para cada região representada por uma chave KMS no chaveiro. Por exemplo, se a chave KMS identificada pelo valor do `Generator` parâmetro estiver na região Oeste dos EUA (Oregon) (`us-west-2`), ela AWS Encryption SDK criará um AWS KMS cliente padrão para a `us-west-2` região. Se você precisar personalizar o cliente do AWS KMS , use o método `CreateAwsKmsKeyring()`.

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK for Java, você pode usar qualquer identificador de chave válido: um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

Para ver um exemplo completo, consulte BasicEncryptionKeyringExample.java no AWS Encryption SDK for Java repositório em. GitHub

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

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

Python

Para criar um chaveiro com uma ou mais AWS KMS chaves, use o `create_aws_kms_multi_keyring()` método. Este exemplo usa duas chaves KMS. Para especificar uma chave do KMS, use o parâmetro `generator`. O parâmetro `kms_key_ids`, que especifica chaves KMS adicionais, é opcional.

A entrada para este chaveiro não requer um AWS KMS cliente. Em vez disso, AWS Encryption SDK ele usa o AWS KMS cliente padrão para cada região representada por uma chave KMS no chaveiro. Por exemplo, se a chave KMS identificada pelo valor do `generator` parâmetro estiver na região Oeste dos EUA (Oregon) (`us-west-2`), ela AWS Encryption SDK criará um AWS KMS cliente padrão para a `us-west-2` região. Se você precisar personalizar o cliente do AWS KMS , use o método `create_aws_kms_keyring()`.

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK for Python, você pode usar qualquer identificador de chave válido: um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

O exemplo a seguir instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. `REQUIRE_ENCRYPT_REQUIRE_DECRYPT` Para ver um exemplo completo, consulte aws_kms_multi_keyring_example.py no AWS Encryption SDK for Python repositório em GitHub.

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

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

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

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

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

Rust

Para criar um chaveiro com uma ou mais AWS KMS chaves, use o `create_aws_kms_multi_keyring()` método. Este exemplo usa duas chaves KMS. Para especificar uma chave do KMS, use o parâmetro `generator`. O parâmetro `kms_key_ids`, que especifica chaves KMS adicionais, é opcional.

A entrada para este chaveiro não requer um AWS KMS cliente. Em vez disso, AWS Encryption SDK ele usa o AWS KMS cliente padrão para cada região representada por uma chave KMS no chaveiro. Por exemplo, se a chave KMS identificada pelo valor do `generator` parâmetro estiver na região Oeste dos EUA (Oregon) (`us-west-2`), ela AWS Encryption SDK criará um AWS KMS cliente padrão para a `us-west-2` região. Se você precisar personalizar o cliente do AWS KMS , use o método `create_aws_kms_keyring()`.

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK for Rust, você pode usar qualquer identificador de chave válido: um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

O exemplo a seguir instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. `REQUIRE_ENCRYPT_REQUIRE_DECRYPT` Para obter um exemplo completo, consulte aws_kms_keyring_example.rs no diretório Rust do repositório em. aws-encryption-sdk GitHub

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

// Create an AWS KMS client
```

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);

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

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

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

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

Go

Para criar um chaveiro com uma ou mais AWS KMS chaves, use o `create_aws_kms_multi_keyring()` método. Este exemplo usa duas chaves KMS. Para especificar uma chave do KMS, use o parâmetro `generator`. O parâmetro `kms_key_ids`, que especifica chaves KMS adicionais, é opcional.

A entrada para este chaveiro não requer um AWS KMS cliente. Em vez disso, AWS Encryption SDK ele usa o AWS KMS cliente padrão para cada região representada por uma chave KMS no chaveiro. Por exemplo, se a chave KMS identificada pelo valor do `generator` parâmetro estiver na região Oeste dos EUA (Oregon) (`us-west-2`), ela AWS Encryption SDK criará um AWS KMS

cliente padrão para a `us-west-2` região. Se você precisar personalizar o cliente do AWS KMS , use o método `create_aws_kms_keyring()`.

Ao especificar um AWS KMS key para um chaveiro de criptografia no AWS Encryption SDK for Go, você pode usar qualquer identificador de chave válido: um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

O exemplo a seguir instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. `REQUIRE_ENCRYPT_REQUIRE_DECRYPT`

```
import (
    "context"

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

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

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

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

```
}

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

AWS Encryption SDK Também suporta AWS KMS chaveiros que usam chaves RSA KMS assimétricas. Os AWS KMS chaveiros RSA assimétricos só podem conter um par de chaves.

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 chamada do AWS KMS é feita ao criptografar com este token de autenticação. Para descriptografar com um AWS KMS chaveiro RSA assimétrico, você precisa da permissão KMS:Decrypt.

> **Note**
>
> Para criar um AWS KMS chaveiro que use chaves RSA KMS assimétricas, você deve usar uma das seguintes implementações de linguagem de programação:
>
> - Versão 3. x do AWS Encryption SDK for Java
> - Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
> - Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
> - Versão 1. x do AWS Encryption SDK para Rust
> - Versão 0.1. x ou posterior do AWS Encryption SDK for Go

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`

C# / .NET

Para criar um AWS KMS chaveiro RSA assimétrico, você deve fornecer o ARN da chave pública e da chave privada da sua chave RSA KMS assimétrica. A chave pública deve ser codificada em PEM. O exemplo a seguir cria um AWS KMS chaveiro com um par de chaves RSA assimétrico.

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

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

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

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

Java

Para criar um AWS KMS chaveiro RSA assimétrico, você deve fornecer o ARN da chave pública e da chave privada da sua chave RSA KMS assimétrica. A chave pública deve ser codificada em PEM. O exemplo a seguir cria um AWS KMS chaveiro com um par de chaves RSA assimétrico.

```
// Instantiate the AWS Encryption SDK and material providers
final AwsCrypto crypto = AwsCrypto.builder()
        // Specify algorithmSuite without asymmetric signing here
```

```
        //
        // ALG_AES_128_GCM_IV12_TAG16_NO_KDF("0x0014"),
        // ALG_AES_192_GCM_IV12_TAG16_NO_KDF("0x0046"),
        // ALG_AES_256_GCM_IV12_TAG16_NO_KDF("0x0078"),
        // ALG_AES_128_GCM_IV12_TAG16_HKDF_SHA256("0x0114"),
        // ALG_AES_192_GCM_IV12_TAG16_HKDF_SHA256("0x0146"),
        // ALG_AES_256_GCM_IV12_TAG16_HKDF_SHA256("0x0178")

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

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

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

Python

Para criar um AWS KMS chaveiro RSA assimétrico, você deve fornecer o ARN da chave pública e da chave privada da sua chave RSA KMS assimétrica. A chave pública deve ser codificada em PEM. O exemplo a seguir cria um AWS KMS chaveiro com um par de chaves RSA assimétrico.

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

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

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

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

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

Rust

Para criar um AWS KMS chaveiro RSA assimétrico, você deve fornecer o ARN da chave pública e da chave privada da sua chave RSA KMS assimétrica. A chave pública deve ser codificada em PEM. O exemplo a seguir cria um AWS KMS chaveiro com um par de chaves RSA assimétrico.

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

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

// Optional: Create an encryption context
let encryption_context = HashMap::from([
```

```
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);

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

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

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

Go

```
import (
    "context"

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

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

```
    panic(err)
}

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

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

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

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

Usando um chaveiro AWS KMS Discovery

Ao descriptografar, é uma prática recomendada especificar as chaves de encapsulamento que podem ser usadas. AWS Encryption SDK 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.

AWS Encryption SDK Fornece um chaveiro de AWS KMS descoberta padrão e um chaveiro de descoberta para chaves AWS KMS multirregionais. Para obter informações sobre como usar chaves de várias regiões com o AWS Encryption SDK, consulte Usando várias regiões AWS KMS keys.

Como não especifica nenhuma chave de encapsulamento, 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. A exceção é a AWS Encryption SDK for C, em que a operação de criptografia ignora um chaveiro de descoberta padrão, mas falha se você especificar um chaveiro de descoberta de várias regiões, sozinho ou em um chaveiro com várias chaves.

Ao descriptografar, um chaveiro de descoberta permite que você solicite AWS Encryption SDK 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 decodificação múltipla, 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. Um token de autenticação de descoberta do AWS KMS não tem efeito na criptografia quando usado sozinho ou em um multitoken de autenticação.

AWS Encryption SDK Ele fornece um chaveiro AWS KMS Discovery 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 chave usada para criptografar uma chave de dados na mensagem criptografada, apenas para AWS KMS key 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 AWS Encryption SDK tenta descriptografar todas as chaves de dados criptografadas, incluindo aquelas criptografadas AWS KMS keys em 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. Os filtros de descoberta são compatíveis com as versões 1.7.x e posteriores do AWS Encryption SDK. Para obter ajuda para encontrar o ID e a partição da sua conta, consulte Seus Conta da AWS identificadores e formato ARN no. Referência geral da AWS

O código a seguir instancia um chaveiro de AWS KMS descoberta com um filtro de descoberta que limita as chaves KMS que AWS Encryption SDK podem ser usadas às da `aws` partição e da 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`.

C

Para obter um exemplo completo, consulte: kms_discovery.cpp.

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

struct aws_cryptosdk_keyring *kms_discovery_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()
```

```
        .BuildDiscovery(discovery_filter));
```

C# / .NET

O exemplo a seguir usa a versão 4.x do AWS Encryption SDK para .NET.

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

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

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

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

JavaScript Browser

Em JavaScript, você deve especificar explicitamente a propriedade de descoberta.

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

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

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

const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

Em JavaScript, você deve especificar explicitamente a propriedade de descoberta.

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

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

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

const discovery = true

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

Java

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

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

Python

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

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

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

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

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

```
)

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

Rust

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

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


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

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

// Create the AWS KMS discovery keyring
let discovery_keyring = mpl
    .create_aws_kms_discovery_keyring()
    .kms_client(kms_client.clone())
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

Go

```
import (
    "context"
```

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

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

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

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

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

// Create discovery filter
discoveryFilter := mpltypes.DiscoveryFilter{
```

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

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 AWS Encryption SDK o decodifique usando somente as chaves KMS em particular. Regiões da AWS

Ao descriptografar com um chaveiro de descoberta AWS KMS regional, ele AWS Encryption SDK descriptografa qualquer chave de dados criptografada que tenha sido criptografada sob 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 token de autenticação de descoberta, o token de autenticação de descoberta regional não afeta a criptografia. Ele funciona somente ao descriptografar mensagens criptografadas. Se você usar um token de autenticação de descoberta regional em um multitoken de autenticação 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 nas AWS Encryption SDK for C tentativas de descriptografar somente com chaves KMS na região especificada. Ao usar um chaveiro de descoberta no AWS Encryption SDK para JavaScript e AWS Encryption SDK para.NET, você configura a região no AWS KMS cliente. Essas AWS Encryption SDK implementações não filtram as chaves KMS por região, mas AWS KMS falharão em uma solicitação de descriptografia de chaves KMS fora da região especificada.

Se você usa um chaveiro de descoberta, recomendamos que você use um filtro de descoberta para limitar as chaves KMS usadas na descriptografia àquelas em partições e partições especificadas. Contas da AWS Os filtros de descoberta são compatíveis com as versões 1.7.x e posteriores do AWS Encryption SDK.

Por exemplo, o código a seguir cria um chaveiro de descoberta AWS KMS regional com um filtro de descoberta. Esse chaveiro limita as duas AWS Encryption SDK chaves KMS na conta 111122223333 na região Oeste dos EUA (Oregon) (us-west-2).

C

Para exibir esse token de autenticação e o método `create_kms_client` em um exemplo funcional, consulte kms_discovery.cpp.

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

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

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

C# / .NET

O AWS Encryption SDK for.NET não tem um chaveiro de descoberta regional dedicado. Porém, você pode usar várias técnicas para limitar as chaves KMS usadas ao descriptografar para uma região específica.

A maneira mais eficiente de limitar as regiões em um chaveiro de descoberta é usar um chaveiro de multi-Region-aware descoberta, mesmo que você tenha criptografado os dados usando somente chaves de região única. Quando encontra chaves de região única, o multi-Region-aware chaveiro não usa nenhum recurso multirregional.

O token de autenticação retornado pelo método `CreateAwsKmsMrkDiscoveryKeyring()` filtra as chaves do por região antes de chamar o AWS KMS. Ele envia uma solicitação de descriptografia AWS KMS somente quando a chave de dados criptografada foi criptografada por uma chave KMS na região especificada pelo `Region` parâmetro no objeto. `CreateAwsKmsMrkDiscoveryKeyringInput`

Os exemplos a seguir usam a versão 4.x do AWS Encryption SDK para .NET.

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

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

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

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

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

Você também pode limitar as chaves KMS a uma determinada Região da AWS especificando uma região na sua instância do AWS KMS cliente () AmazonKeyManagementServiceClient. No entanto, essa configuração é menos eficiente e potencialmente mais cara do que usar um chaveiro de multi-Region-aware descoberta. Em vez de filtrar as chaves KMS por região antes da chamada AWS KMS, o AWS Encryption SDK for.NET chama AWS KMS cada chave de dados

criptografada (até decifrar uma) e se baseia em limitar as chaves KMS que usa AWS KMS à região especificada.

O exemplo a seguir usa a versão 4.x do AWS Encryption SDK para .NET.

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

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

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

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

JavaScript Browser

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

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

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

```
const clientProvider = getClient(KMS, { credentials })

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

JavaScript Node.js

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

Para ver esse chaveiro e a `limitRegions` função em um exemplo prático, consulte kms_regional_discovery.ts.

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

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

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

Java

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

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

Python

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

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

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

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

# Create the AWS KMS regional discovery keyring
regional_discovery_keyring_input: CreateAwsKmsMrkDiscoveryKeyringInput = \
    CreateAwsKmsMrkDiscoveryKeyringInput(
        kms_client=kms_client,
        region=mrk_replica_decrypt_region,
        discovery_filter=DiscoveryFilter(
            account_ids=[111122223333],
```

```
            partition="aws"
        )
)

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

Rust

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

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

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

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


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

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

Go

```
import (
    "context"

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

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

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

// Optional: Create an encryption context
```

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

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

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

Ele AWS Encryption SDK para JavaScript também exporta uma `excludeRegions` função para o Node.js e o navegador. Essa função cria um chaveiro de descoberta AWS KMS regional que é omitido AWS KMS keys em regiões específicas. O exemplo a seguir cria um chaveiro de descoberta AWS KMS regional que pode ser usado AWS KMS keys na conta 111122223333 em todos, Região da AWS exceto no Leste dos EUA (Norte da Virgínia) (us-east-1).

O AWS Encryption SDK for C não tem um método análogo, mas você pode implementá-lo criando um personalizado ClientSupplier.

Este exemplo mostra o código para Node.js.

```
const discovery = true
const clientProvider = excludeRegions(['us-east-1'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
```

```
    discovery,
    discoveryFilter: { accountIDs: [111122223333], partition: 'aws' }
})
```

AWS KMS Chaveiros hierárquicos

Com o chaveiro AWS KMS hierárquico, você pode proteger seus materiais criptográficos com uma chave KMS de criptografia simétrica sem precisar fazer chamadas AWS KMS sempre que criptografar ou descriptografar dados. 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 chaves 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 dados exclusiva para criptografar cada mensagem e criptografa cada chave de criptografia de dados para cada solicitação de criptografia e criptografa cada chave de criptografia de dados com uma chave de encapsulamento 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 que a `branch-key-id` é especificada em uma 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 chaves de ramificação 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 descriptografia. Os materiais de criptografia são montados 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 por vez. Quando o cache local é configurado para usar umbranch key ID supplier, ele também pode armazenar materiais de chaves de ramificação de várias chaves de ramificação ativas ao mesmo tempo.

> **Note**
>
> Todas as menções ao chaveiro hierárquico AWS Encryption SDK se referem ao chaveiro hierárquico. AWS KMS

Compatibilidade da linguagem de programação

O chaveiro hierárquico é suportado pelas seguintes linguagens e versões de programação:

- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x e posterior do AWS Encryption SDK para .NET
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional de MPL.
- Versão 1. x do AWS Encryption SDK para Rust
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go
- Versão 4.1. x e posterior do AWS Encryption SDK para JavaScript formulário JavaScript Node.js.

- O chaveiro AWS KMS hierárquico não é suportado no for Browser. AWS Encryption SDK para JavaScript JavaScript Para saber o status e as limitações atuais, consulte o repositório https://github.com/aws/aws-encryption-sdk-javascript/ aws-encryption-sdk-javascript em. GitHub

Tópicos

- Como funciona
- Pré-requisitos
- Permissões obrigatórias
- Escolha um cache
- Criar um token de autenticação hierárquico

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 token de autenticação gera uma chave de dados em texto simples e, em seguida, verifica se há materiais de ramificação válidos no cache local para gerar a chave de empacotamento. Se houver materiais de chave de filial válidos, o chaveiro prosseguirá 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 chama AWS KMS para descriptografar a chave de ramificação ativa e retorna a chave de ramificação ativa em texto sem formatação. 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 sem formatação 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 encapsulamento derivada para criptografar uma cópia da chave de dados em texto simples.

O método de criptografia usa os materiais de criptografia para criptografar os dados. Para obter mais informações, consulte Como AWS Encryption SDK criptografar dados.

Descriptografar e verificar

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

1. O método de descriptografia identifica a chave de dados criptografada da mensagem criptografada e a transmite 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 corresponda à versão da chave de ramificação identificada na Etapa 2.

   a. O armazenamento de chaves chama AWS KMS para descriptografar a chave de ramificação e retorna a chave de ramificação ativa em texto sem formatação. 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 sem formatação 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 encapsulamento 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 a mensagem criptografada. Para obter mais informações, consulte Como o AWS Encryption SDK descriptografa uma mensagem criptografada.

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 ações de armazenamento de chaves.

> **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 Ações de armazenamento de chaves.

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

Permissões obrigatórias

AWS Encryption SDK 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 na (s) criptografia AWS KMS key(ões) simétrica (s) em seu armazenamento de chaves.

- Para criptografar e descriptografar dados com o chaveiro hierárquico, você precisa de kms:Decrypt. https://docs.aws.amazon.com/kms/latest/APIReference/API_Decrypt.html
- Para criar e girar chaves de ramificação, você precisa de kms: GenerateDataKeyWithoutPlaintext e kms:. ReEncrypt

Permissões obrigatórias do Amazon DynamoDB na tabela de armazenamento de chaves

Os diretores que interagem com seu armazenamento de chaves também precisam de permissões na tabela do DynamoDB. O conjunto de permissões depende da função.

Usuário do armazenamento de chaves

Um usuário do armazenamento de chaves é o principal que usa o chaveiro hierárquico para criptografar e descriptografar dados. Um usuário de armazenamento de chaves precisa de dynamodb: GetItem na tabela de armazenamento de chaves.

Administrador do armazenamento de chaves

Um administrador de armazenamento de chaves é o diretor que cria e alterna as chaves das filiais. Um administrador de armazenamento de chaves precisa das seguintes permissões na tabela de armazenamento de chaves:

- Para leituras: dynamodb: GetItem e dynamodb:. ConditionCheckItem
- Para gravações transacionais: dynamodb: e dynamodb:ConditionCheckItem . PutItem O armazenamento de chaves hierárquico do chaveiro executa gravações TransactWriteItems; você pode definir o escopo das permissões de gravação para essa operação com uma condição. `dynamodb:EnclosingOperation`

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 AWS KMS armazenando localmente em cache os materiais da 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”

> **Important**
>
> Todos os tipos de cache compatíveis são projetados para suportar ambientes multisegmentados.
> No entanto, quando usado com o AWS Encryption SDK for Python, o chaveiro hierárquico não oferece suporte a ambientes de vários segmentos. Para obter mais informações, consulte o README.rst arquivo Python no repositório aws-cryptographic-material-providers-library em. GitHub

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 threads sejam chamados, AWS KMS notificando um thread 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 suportam o 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, não é necessário 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}
};
```

Python

```
default_cache = CacheTypeDefault(
    value=DefaultCache(
        entry_capacity=100
    )
)
```

Rust

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

Go

```
cache := mpltypes.CacheTypeMemberDefault{
  Value: mpltypes.DefaultCache{
   EntryCapacity: 100,
  },
 }
```

MultiThreaded cache

O MultiThreaded cache é seguro para uso em ambientes de vários segmentos, 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
```

```
    }
};
```

Python

```
multithreaded_cache = CacheTypeMultiThreaded(
    value=MultiThreadedCache(
        entry_capacity=100,
        entry_pruning_tail_size=1
    )
)
```

Rust

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

Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberMultiThreaded{
  Value: mpltypes.MultiThreadedCache{
   EntryCapacity:        100,
   EntryPruningTailSize: &entryPruningTailSize,
  },
 }
```

StormTracking 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 threads sejam chamados, AWS KMS notificando um thread de que a entrada de materiais da chave de ramificação expirará com antecedência. Isso garante que somente um thread envie uma solicitação AWS KMS para atualizar o cache. Para obter mais informações, consulte Storm Tracking Cryptographic Materials Cache no repositório GitHub aws-encryption-sdk-specification.

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
- Suspensão: define o número de milissegundos em que um thread deve dormir se o `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
    }
};
```

Python

```
storm_tracking_cache = CacheTypeStormTracking(
    value=StormTrackingCache(
        entry_capacity=100,
        entry_pruning_tail_size=1,
        fan_out=20,
        grace_interval=1,
        grace_period=10,
        in_flight_ttl=10,
        sleep_milli=20
    )
)
```

Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
```

```
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

Go

```
var entryPruningTailSize int32 = 1
 cache := mpltypes.CacheTypeMemberStormTracking{
  Value: mpltypes.StormTrackingCache{
   EntryCapacity:        100,
   EntryPruningTailSize: &entryPruningTailSize,
   GraceInterval:        1,
   GracePeriod:          10,
   FanOut:               20,
   InFlightTTL:          10,
   SleepMilli:           20,
  },
 }
```

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 ao permitir 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 em 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 umthe 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 gravando 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 IDs de partição diferentes, cada chaveiro acessará somente as entradas de cache de sua própria partição designada dentro do 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 nos dados armazenados na outra partição.

Se você pretende reutilizar ou compartilhar as entradas de cache em uma partição, você deve definir sua própria ID de partição. Quando você passa o 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 sempre 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 Biblioteca de Provedores de Materiais (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);
```

Python

```
# Instantiate the MPL
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

# Create a CacheType object for the default cache
cache: CacheType = CacheTypeDefault(
    value=DefaultCache(
        entry_capacity=100,
    )
)

# Create a CMC using the default cache
cryptographic_materials_cache_input = CreateCryptographicMaterialsCacheInput(
    cache=cache,
)

shared_cryptographic_materials_cache =
 mat_prov.create_cryptographic_materials_cache(
    cryptographic_materials_cache_input
```

```
)
```

Rust

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

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

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

Go

```
import (
    "context"

 mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
)

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

// Create a CacheType object for the default cache
cache := mpltypes.CacheTypeMemberDefault{
    Value: mpltypes.DefaultCache{
        EntryCapacity: 100,
```

```
    },
}

// Create a CMC using the default cache
cmcCacheInput := mpltypes.CreateCryptographicMaterialsCacheInput{
    Cache: &cache,
}
sharedCryptographicMaterialsCache, err :=
 matProv.CreateCryptographicMaterialsCache(context.Background(), cmcCacheInput)
if err != nil {
    panic(err)
}
```

2. 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 };
   ```

   Python

   ```
   # Create a CacheType object for the shared_cryptographic_materials_cache
   shared_cache: CacheType = CacheTypeShared(
       value=shared_cryptographic_materials_cache
   )
   ```

   Rust

   ```
   // Create a CacheType object for the shared_cryptographic_materials_cache
   ```

```
let shared_cache: CacheType =
 CacheType::Shared(shared_cryptographic_materials_cache);
```

Go

```
// Create a CacheType object for the shared_cryptographic_materials_cache
shared_cache :=
 mpltypes.CacheTypeMemberShared{sharedCryptographicMaterialsCache}
```

3. 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 e um branch key ID supplier limite de cache de 600 segundos. Para obter mais informações sobre os valores definidos na seguinte configuração hierárquica de chaveiros, 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);
```

Python

```
# Create the Hierarchical keyring
keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=shared_cache,
    partition_id=partition_id
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
```

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

Go

```
// Create the Hierarchical keyring
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:    keyStore1,
    BranchKeyId: &branchKeyId,
    TtlSeconds:  600,
    Cache:       &shared_cache,
    PartitionId: &partitionId,
}
keyring, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

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 seu 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 servida 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 ramificação usa os campos armazenados no contexto de criptografia para determinar qual chave de ramificação é necessária para descriptografar um registro.

  É altamente recomendável usar um fornecedor de ID de chave de ramificação para bancos de dados de vários locatários em que cada inquilino tenha sua própria chave de ramificação. Você pode usar o fornecedor de ID de chave de filial para criar um nome amigável para seus IDs de chave de filial a fim de facilitar o reconhecimento do ID de chave de filial correto 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) Um 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 gravando no cache. Se você pretende reutilizar ou compartilhar as entradas de cache em uma partição, você deve definir sua própria 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 estático, othe section called “Cache padrão”, e um limite de cache TTL 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,
   BranchKeyId = branch-key-id,
   TtlSeconds = 600
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Python

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id=branch_key_id,
    ttl_seconds=600
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

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

let hierarchical_keyring = mpl
        .create_aws_kms_hierarchical_keyring()
        .key_store(key_store.clone())
        .branch_key_id(branch_key_id)
        .ttl_seconds(600)
        .send()
```

```
        .await?;
```

Go

```
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:    keyStore,
    BranchKeyId: &branchKeyID,
    TtlSeconds:  600,
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

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 ramificação.

1. Crie um fornecedor de ID de chave de filial

   O exemplo a seguir define um fornecedor de ID de chave de ramificação que usa o contexto de criptografia no momento da criptografia ou descriptografia para selecionar a ID da chave de ramificação para cada inquilino. Para uma implementação funcional em cada idioma, consulte:

   - Java: ExampleBranchKeyIdSupplier.java
   - C# /.NET: ExampleBranchKeySupplier.cs
   - Python: branch_key_id_supplier_example.py https://github.com/aws/aws-encryption-sdk-python/tree/master/examples/src/branch_key_id_supplier_example.py
   - Ferrugem: example_branch_key_id_supplier.rs
   - Vá: branchkeysupplier.go

Java

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
public class ExampleBranchKeyIdSupplier implements IBranchKeyIdSupplier {
    private static String branchKeyIdForTenantA;
    private static String branchKeyIdForTenantB;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this.branchKeyIdForTenantA = tenant1Id;
        this.branchKeyIdForTenantB = tenant2Id;
    }

    @Override
    public GetBranchKeyIdOutput GetBranchKeyId(GetBranchKeyIdInput input) {
        Map<String, String> encryptionContext = input.encryptionContext();
        if (!encryptionContext.containsKey("tenant")) {
            throw new IllegalArgumentException(
                "EncryptionContext invalid, does not contain expected tenant key
 value pair.");
        }

        String tenantKeyId = encryptionContext.get("tenant");
        String branchKeyId;
        if (tenantKeyId.equals("TenantA")) {
            branchKeyId = branchKeyIdForTenantA;
        } else if (tenantKeyId.equals("TenantB")) {
            branchKeyId = branchKeyIdForTenantB;
        } else {
            throw new IllegalArgumentException("Item does not contain valid
 tenant ID");
        }

        return GetBranchKeyIdOutput.builder().branchKeyId(branchKeyId).build();
    }
}

// Create the branch key ID supplier
final IBranchKeyIdSupplier branchKeyIdSupplier = new ExampleBranchKeyIdSupplier(
    branch-key-ID-tenantA, branch-key-ID-tenantB);
```

C# / .NET

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
public class ExampleBranchKeySupplier : BranchKeyIdSupplierBase {
    private string branchKeyTenantA;
    private string branchKeyTenantB;

    public ExampleBranchKeySupplier(string branchKeyTenantA, string
 branchKeyTenantB) {
        this.branchKeyTenantA = branchKeyTenantA;
        this.branchKeyTenantB = branchKeyTenantB;
    }

    // The encryption context is used to determine the Branch Key ID.
    protected override GetBranchKeyIdOutput _GetBranchKeyId(GetBranchKeyIdInput
 input) {
        Dictionary<string, string> encryptionContext = input.EncryptionContext;
        if (!encryptionContext.ContainsKey("tenant")) {
            throw new Exception("EncryptionContext invalid, does not contain
 expected tenant key value pair.");
        }

        string tenant = encryptionContext["tenant"];
        if (tenant.Equals("TenantA")) {
            return new GetBranchKeyIdOutput { BranchKeyId = branchKeyTenantA };
        }
        if (tenant.Equals("TenantB")) {
            return new GetBranchKeyIdOutput { BranchKeyId = branchKeyTenantB };
        }
        throw new Exception("Item does not have a valid tenantID.");
    }
}

// Create the branch key ID supplier
var branchKeyIdSupplier = new ExampleBranchKeySupplier(
    branch-key-ID-tenantA, branch-key-ID-tenantB);
```

Python

```
# Define a branch key ID supplier that uses the encryption context to
# select a branch key ID for each tenant.
```

```
class ExampleBranchKeyIdSupplier(IBranchKeyIdSupplier):
    branch_key_id_for_tenant_A: str
    branch_key_id_for_tenant_B: str

    def __init__(self, tenant_1_id, tenant_2_id):
        self.branch_key_id_for_tenant_A = tenant_1_id
        self.branch_key_id_for_tenant_B = tenant_2_id

    def get_branch_key_id(
        self, param: GetBranchKeyIdInput
    ) -> GetBranchKeyIdOutput:
        encryption_context = param.encryption_context
        if "tenant" not in encryption_context:
            raise ValueError("EncryptionContext invalid, does not contain
 expected tenant key value pair.")

        tenant_key_id = encryption_context.get("tenant")
        if tenant_key_id == "TenantA":
            branch_key_id = self.branch_key_id_for_tenant_A
        elif tenant_key_id == "TenantB":
            branch_key_id = self.branch_key_id_for_tenant_B
        else:
            raise ValueError(f"Item does not contain valid tenant ID:
 {tenant_key_id=}")

        return GetBranchKeyIdOutput(branch_key_id=branch_key_id)

# Create the branch key ID supplier
branch_key_id_supplier: IBranchKeyIdSupplier = ExampleBranchKeyIdSupplier(
        tenant_1_id=branch_key_id_a,
        tenant_2_id=branch_key_id_b,
)
```

Rust

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant_a: String,
    branch_key_id_for_tenant_b: String,
}

impl ExampleBranchKeyIdSupplier {
```

```
    pub fn new(tenant_a_id: &str, tenant_b_id: &str) -> Self {
        Self {
            branch_key_id_for_tenant_a: tenant_a_id.to_string(),
            branch_key_id_for_tenant_b: tenant_b_id.to_string(),
        }
    }
}

// The encryption context is used to determine the Branch Key ID.
impl BranchKeyIdSupplier for ExampleBranchKeyIdSupplier {
    fn get_branch_key_id(&self, input: GetBranchKeyIdInput) ->
 Result<GetBranchKeyIdOutput, Error> {
        let encryption_context: HashMap<String, String> =
 input.encryption_context.unwrap();
        if !encryption_context.contains_key("tenant") {
            return Err(Error::AwsCryptographicMaterialProvidersException {
                message: "EncryptionContext invalid, does not contain expected
 tenant key value pair.".to_string(),
            });
        }

        let tenant_key_id: &str = encryption_context["tenant"].as_str();
        if tenant_key_id == "TenantA" {
            Ok(GetBranchKeyIdOutput::builder()
                .branch_key_id(self.branch_key_id_for_tenant_a.clone())
                .build()
                .unwrap())
        } else if tenant_key_id == "TenantB" {
            Ok(GetBranchKeyIdOutput::builder()
                .branch_key_id(self.branch_key_id_for_tenant_b.clone())
                .build()
                .unwrap())
        } else {
            Err(Error::AwsCryptographicMaterialProvidersException {
                message: "Item does not contain valid tenant ID.".to_string(),
            })
        }
    }
}

// Create the branch key ID supplier
let branch_key_id_supplier = ExampleBranchKeyIdSupplier::new(
    &branch_key_id_a,
    &branch_key_id_b,
```

```
);
```

Go

```
// Define a branch key ID supplier that uses the encryption context to
// select a branch key ID for each tenant.
type branchKeySupplier struct {
    branchKeyA string
    branchKeyB string
}

// The encryption context is used to determine the Branch Key ID.
func (b *branchKeySupplier) GetBranchKeyId(input mpltypes.GetBranchKeyIdInput)
 (*mpltypes.GetBranchKeyIdOutput, error) {
    ec := input.EncryptionContext
    if value, exists := ec["tenant"]; !exists || value == "" {
        return nil, fmt.Errorf("EncryptionContext invalid, does not contain
 expected tenant key value pair.")
    }

    branchKeyIdentifier := ec["tenant"]
    if branchKeyIdentifier == "TenantA" {
        return &mpltypes.GetBranchKeyIdOutput{BranchKeyId: b.branchKeyA}, nil
    } else if branchKeyIdentifier == "TenantB" {
        return &mpltypes.GetBranchKeyIdOutput{BranchKeyId: b.branchKeyB}, nil
    } else {
        return &mpltypes.GetBranchKeyIdOutput{}, fmt.Errorf("unknown branch key
 identifier")
    }
}

// Create the branch key ID supplier
keySupplier := branchKeySupplier{branchKeyA: branchKeyA, branchKeyB: branchKeyB}
```

2. Criar um token de autenticação hierárquico

Os exemplos a seguir inicializam um chaveiro hierárquico com o fornecedor de ID da chave de ramificação 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);
```

Python

```
mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

keyring_input: CreateAwsKmsHierarchicalKeyringInput =
 CreateAwsKmsHierarchicalKeyringInput(
    key_store=keystore,
    branch_key_id_supplier=branch_key_id_supplier,
    ttl_seconds=600,
    cache=CacheTypeDefault(
        value=DefaultCache(
```

```
            entry_capacity=100
        )
    ),
)

hierarchical_keyring: IKeyring = mat_prov.create_aws_kms_hierarchical_keyring(
    input=keyring_input
)
```

Rust

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

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

Go

```
hkeyringInput := mpltypes.CreateAwsKmsHierarchicalKeyringInput{
    KeyStore:            keyStore,
    BranchKeyIdSupplier: &keySupplier,
    TtlSeconds:          600,
}
hKeyRing, err := matProv.CreateAwsKmsHierarchicalKeyring(context.Background(),
 hkeyringInput)
if err != nil {
    panic(err)
}
```

AWS KMS chaveiros ECDH

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 AWS Encryption SDK usa (`KDF_CTR_HMAC_SHA384`) para derivar a chave de empacotamento 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, AWS Encryption SDK usam 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 encapsulamento compartilhada armazenadas no texto cifrado do cabeçalho da mensagem, a operação falhará. Por exemplo, se você criptografar dados 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 empacotamento compartilhada e poderá descriptografar os dados. Se a chave pública de Bob não for de um par de chaves KMS, Bob poderá criar um chaveiro ECDH bruto para descriptografar os dados.

O chaveiro AWS KMS ECDH criptografa os dados 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.

Compatibilidade com linguagens de programação

O chaveiro AWS KMS ECDH foi introduzido na versão 1.5.0 da Biblioteca de Provedores de Material Criptográfico (MPL) e é suportado pelas seguintes linguagens e versões de programação:

- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional do MPL.
- Versão 1. x do AWS Encryption SDK para Rust
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go

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

AWS Encryption SDK Não requer 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 dados 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 dados 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();
```

Python

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.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey,
    KmsPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

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

# Create the AWS KMS ECDH static keyring
sender_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 KmsEcdhStaticConfigurationsKmsPrivateKeyToStaticPublicKey(
```

```
        KmsPrivateKeyToStaticPublicKeyInput(
            sender_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
            sender_public_key = bob_public_key,
            recipient_public_key = alice_public_key,

        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(sender_keyring_input)
```

Rust

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.

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

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

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

// Retrieve public keys
// Must be DER-encoded X.509 keys
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
```

```
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();

let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

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

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"
```

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

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

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

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

// Retrieve public keys
// Must be DER-encoded X.509 keys
publicKeySender, err := utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameSender)
if err != nil {
    panic(err)
}
publicKeyRecipient, err :=
 utils.LoadPublicKeyFromPEM(kmsEccPublicKeyFileNameRecipient)
```

```
if err != nil {
    panic(err)
}

// Create KmsPrivateKeyToStaticPublicKeyInput
kmsEcdhStaticConfigurationInput := mpltypes.KmsPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey:  publicKeyRecipient,
    SenderKmsIdentifier: arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    SenderPublicKey:     publicKeySender,
}
kmsEcdhStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPrivateKeyToStaticPublicKey{
    Value: kmsEcdhStaticConfigurationInput,
}

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

// Create AWS KMS ECDH keyring
awsKmsEcdhKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhKeyring, err := matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

Criando um AWS KMS chaveiro de descoberta ECDH

Ao descriptografar, é uma prática recomendada especificar as chaves que eles podem usar. AWS Encryption SDK 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 mensagem em que a chave pública do par de chaves

KMS especificado corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

> **Important**
>
> Ao descriptografar mensagens 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 mensagem em que a chave pública do par de chaves KMS especificado corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

```
// 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 mensagem em que a chave pública do par de chaves KMS especificado corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

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

Python

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 mensagem em que a chave pública do par de chaves KMS especificado corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateAwsKmsEcdhKeyringInput,
    KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery,
    KmsPublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Create the AWS KMS ECDH discovery keyring
create_keyring_input = CreateAwsKmsEcdhKeyringInput(
    kms_client = boto3.client('kms', region_name="us-west-2"),
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = KmsEcdhStaticConfigurationsKmsPublicKeyDiscovery(
        KmsPublicKeyDiscoveryInput(
            recipient_kms_identifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321",
        )
    )
)

keyring = mat_prov.create_aws_kms_ecdh_keyring(create_keyring_input)
```

Rust

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

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

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

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

let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

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

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

Go

```
import (
    "context"

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

```
 esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

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

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

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

// Create KmsPublicKeyDiscoveryInput
kmsEcdhDiscoveryStaticConfigurationInput := mpltypes.KmsPublicKeyDiscoveryInput{
    RecipientKmsIdentifier: eccRecipientKeyArn,
}
kmsEcdhDiscoveryStaticConfiguration :=
 &mpltypes.KmsEcdhStaticConfigurationsMemberKmsPublicKeyDiscovery{
    Value: kmsEcdhDiscoveryStaticConfigurationInput,
}

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

```
// Create AWS KMS ECDH discovery keyring
awsKmsEcdhDiscoveryKeyringInput := mpltypes.CreateAwsKmsEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: kmsEcdhDiscoveryStaticConfiguration,
    KmsClient:          kmsClient,
}
awsKmsEcdhDiscoveryKeyring, err :=
 matProv.CreateAwsKmsEcdhKeyring(context.Background(),
 awsKmsEcdhDiscoveryKeyringInput)
if err != nil {
    panic(err)
}
```

Tokens de autenticação AES Raw

O AWS Encryption SDK 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 encapsulamento 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.

O chaveiro AES bruto é equivalente e interopera com a JceMasterKeyclasse no AWS Encryption SDK for Java e com a RawMasterKeyclasse no AWS Encryption SDK for Python quando são usados com chaves de criptografia AES. Você pode criptografar dados com uma implementação e descriptografá-los com qualquer outra implementação usando a mesma chave de encapsulamento. Para obter detalhes, consulte Compatibilidade dos tokens 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 no cabeçalho da mensagem criptografada que a operação

de criptografia retorna. 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 da chave e o nome da chave são equivalentes aos campos ID do provedor (ou provedor) e ID da chave no `JceMasterKey` e no `RawMasterKey`.
> O AWS Encryption SDK for C e AWS Encryption SDK para.NET reserva o valor do namespace `aws-kms` chave para as chaves KMS. Não use esse valor de namespace em um token de autenticação AES bruto ou um token de autenticação RSA bruto com essas bibliotecas.

Se você cria tokens de autenticação diferentes para criptografar e descriptografar determinada mensagem, o namespace e os valores do nome são cruciais. 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.

C

Para instanciar um chaveiro AES bruto no AWS Encryption SDK for C, use. `aws_cryptosdk_raw_aes_keyring_new()` Para obter um exemplo completo, consulte raw_aes_keyring.c.

```
struct aws_allocator *alloc = aws_default_allocator();

AWS_STATIC_STRING_FROM_LITERAL(wrapping_key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(wrapping_key_name, "AES_256_012");

struct aws_cryptosdk_keyring *raw_aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
```

```
        alloc, wrapping_key_namespace, wrapping_key_name, aes_wrapping_key,
 wrapping_key_len);
```

C# / .NET

Para criar um chaveiro AES bruto AWS Encryption SDK para o.NET, use o `materialProviders.CreateRawAesKeyring()` método. Para ver um exemplo completo, consulte Raw AESKeyring Example.cs.

O exemplo a seguir usa a versão 4. x e versões posteriores do AWS Encryption SDK para o.NET.

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

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

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

// Create the keyring that determines how your data keys are protected.
var createKeyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = aesWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var keyring = materialProviders.CreateRawAesKeyring(createKeyringInput);
```

JavaScript Browser

O AWS Encryption SDK para JavaScript no navegador obtém suas primitivas criptográficas da WebCryptoAPI. Antes de construir o chaveiro, você deve usá-lo `RawAesKeyringWebCrypto.importCryptoKey()` para importar o material bruto da chave para o WebCrypto backend. Isso garante que o chaveiro esteja completo, mesmo que todas as chamadas sejam WebCrypto assíncronas.

Em seguida, para instanciar um token de autenticação AES bruto, use o método `RawAesKeyringWebCrypto()`. Você deve especificar o algoritmo de encapsulamento AES ("pacote de encapsulamento") com base no tamanho do seu material de chave. Para obter um exemplo completo, consulte aes_simple.ts (Browser). JavaScript

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called "Limitar as chaves de dados criptografadas".

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

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

const keyNamespace = 'HSM_01'
const keyName = 'AES_256_012'

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

/* Import the plaintext AES key into the WebCrypto backend. */
const aesWrappingKey = await RawAesKeyringWebCrypto.importCryptoKey(
  rawAesKey,
  wrappingSuite
)

const rawAesKeyring = new RawAesKeyringWebCrypto({
  keyName,
  keyNamespace,
  wrappingSuite,
  aesWrappingKey
})
```

JavaScript Node.js

Para instanciar um chaveiro AES bruto no AWS Encryption SDK para JavaScript for Node.js, crie uma instância da classe. `RawAesKeyringNode` Você deve especificar o algoritmo de encapsulamento AES (“pacote de encapsulamento”) com base no tamanho do seu material de chave. Para obter um exemplo completo, consulte aes_simple.ts (Node.js). JavaScript

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

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

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

const keyName = 'AES_256_012'
const keyNamespace = 'HSM_01'

const wrappingSuite =
  RawAesWrappingSuiteIdentifier.AES256_GCM_IV12_TAG16_NO_PADDING

const rawAesKeyring = new RawAesKeyringNode({
  keyName,
  keyNamespace,
  aesWrappingKey,
  wrappingSuite,
})
```

Java

Para instanciar um chaveiro AES bruto no AWS Encryption SDK for Java, use. `matProv.CreateRawAesKeyring()`

```
final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
```

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

Python

O exemplo a seguir instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. REQUIRE_ENCRYPT_REQUIRE_DECRYPT Para ver um exemplo completo, consulte raw_aes_keyring_example.py no AWS Encryption SDK for Python repositório em GitHub.

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

# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "AES_256_012"

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

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

# Create Raw AES keyring
keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
```

```
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)

raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=keyring_input
)
```

Rust

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

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

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

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

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

Go

```
import (
    mpl "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygenerated"
 mpltypes "aws/aws-cryptographic-material-providers-library/releases/go/mpl/
awscryptographymaterialproviderssmithygeneratedtypes"
    client "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygenerated"
    esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
)
//Instantiate the AWS Encryption SDK client.
encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
// Define the key namespace and key name
var keyNamespace = "A managed aes keys"
var keyName = "My 256-bit AES wrapping key"

// Optional: Create an encryption context
encryptionContext := map[string]string{
    "encryption":               "context",
    "is not":                   "secret",
    "but adds":                 "useful metadata",
    "that can help you":        "be confident that",
    "the data you are handling": "is what you think it is",
}
// Instantiate the material providers library
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
// Create Raw AES keyring
aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  aesWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
if err != nil {
```

```
    panic(err)
}
```

Tokens de autenticação brutos do RSA

O token de autenticação bruto do RSA realiza a criptografia e a descriptografia assimétricas das chaves de dados na memória local com chaves de encapsulamento 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. Você pode 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. Algumas implementações de linguagem do não AWS Encryption SDK construirão um chaveiro RSA bruto com chaves de pares diferentes. Outras pessoas confiam em você para verificar se suas chaves são do mesmo par de chaves.

O chaveiro RSA bruto é equivalente e interopera com o JceMasterKeyin the AWS Encryption SDK for Java e o RawMasterKeyin the AWS Encryption SDK for Python quando são usados com chaves de criptografia assimétrica RSA. Você pode criptografar dados com uma implementação e descriptografá-los com qualquer outra implementação usando a mesma chave de encapsulamento. Para obter detalhes, consulte Compatibilidade dos tokens de autenticação.

> **Note**
>
> O token de autenticação bruto do RSA não oferece suporte a chaves assimétricas do KMS. Se você quiser usar chaves RSA KMS assimétricas, as seguintes linguagens de programação oferecem suporte a AWS KMS chaveiros que usam RSA assimétrico: AWS KMS keys
>
> - Versão 3. x do AWS Encryption SDK for Java

- Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional da Biblioteca de Provedores de Material Criptográfico (MPL).
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go

Se você criptografar dados com um chaveiro RSA bruto que inclua a chave pública de uma chave RSA KMS, nem o AWS Encryption SDK nem poderá descriptografá-lo. AWS KMS Você não pode exportar a chave privada de uma chave KMS AWS KMS assimétrica para um chaveiro RSA bruto. A operação de AWS KMS descriptografia não pode descriptografar a mensagem criptografada retornada. AWS Encryption SDK

Ao criar um chaveiro RSA bruto no AWS Encryption SDK for C, certifique-se de fornecer o conteúdo do arquivo PEM que inclui cada chave como uma string C terminada em nulo, não como um caminho ou nome de arquivo. Ao criar um token de autenticação bruto do RSA no JavaScript, lembre-se da potencial incompatibilidade com outras implementações de linguagem.

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 no cabeçalho da mensagem criptografada que a operação de criptografia retorna. 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 da chave e o nome da chave são equivalentes aos campos ID do provedor (ou provedor) e ID da chave no `JceMasterKey` e no `RawMasterKey`.
> O AWS Encryption SDK for C reserva o valor do namespace da `aws-kms` chave para as chaves KMS. Não o use em um token de autenticação bruto do AES ou em um token de autenticação bruto do RSA com o AWS Encryption SDK for C.

Se você cria tokens de autenticação diferentes para criptografar e descriptografar determinada mensagem, o namespace e os valores do nome são cruciais. Se o namespace e o nome da chave

no token de autenticação de descriptografia 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 descriptografia 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

Os exemplos a seguir mostram 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 fornecido por você.

C

Para criar um chaveiro RSA bruto no AWS Encryption SDK for C, use. `aws_cryptosdk_raw_rsa_keyring_new`

Ao criar um chaveiro RSA bruto no AWS Encryption SDK for C, certifique-se de fornecer o conteúdo do arquivo PEM que inclui cada chave como uma string C terminada em nulo, não como um caminho ou nome de arquivo. Para obter um exemplo completo, consulte raw_rsa_keyring.c.

```
struct aws_allocator *alloc = aws_default_allocator();

AWS_STATIC_STRING_FROM_LITERAL(key_namespace, "HSM_01");
AWS_STATIC_STRING_FROM_LITERAL(key_name, "RSA_2048_06");

struct aws_cryptosdk_keyring *rawRsaKeyring = aws_cryptosdk_raw_rsa_keyring_new(
    alloc,
    key_namespace,
    key_name,
    private_key_from_pem,
    public_key_from_pem,
    AWS_CRYPTOSDK_RSA_OAEP_SHA256_MGF1);
```

C# / .NET

Para instanciar um chaveiro RSA bruto no para.NET, AWS Encryption SDK use o método. `materialProviders.CreateRawRsaKeyring()` Para ver um exemplo completo, consulte Raw RSAKeyring Example.cs.

O exemplo a seguir usa a versão 4. x e versões posteriores do AWS Encryption SDK para o.NET.

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

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

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

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

// Create the keyring
var rawRsaKeyring = materialProviders.CreateRawRsaKeyring(createRawRsaKeyringInput);
```

JavaScript Browser

O AWS Encryption SDK para JavaScript no navegador obtém suas primitivas criptográficas da WebCryptobiblioteca. Antes de construir o chaveiro, você deve usá-lo `importPublicKey()` and/or `importPrivateKey()` para importar o material bruto da chave para o WebCrypto backend. Isso garante que o chaveiro esteja completo, mesmo que todas as chamadas sejam WebCrypto assíncronas. O objeto usado pelos métodos de importação inclui o algoritmo de encapsulamento e seu modo de preenchimento.

Depois de importar o material da chave, use o método `RawRsaKeyringWebCrypto()` para instanciar o token de autenticação. Ao criar um chaveiro RSA bruto JavaScript, esteja ciente da possível incompatibilidade com outras implementações de linguagem.

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

Para ver um exemplo completo, consulte rsa_simple.ts (Browser). JavaScript

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

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

```
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

const privateKey = await RawRsaKeyringWebCrypto.importPrivateKey(
  privateRsaJwKKey
)

const publicKey = await RawRsaKeyringWebCrypto.importPublicKey(
  publicRsaJwKKey
)

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringWebCrypto({
  keyName,
  keyNamespace,
  publicKey,
  privateKey,
})
```

JavaScript Node.js

Para instanciar um chaveiro RSA bruto no AWS Encryption SDK para JavaScript Node.js, crie uma nova instância da classe. `RawRsaKeyringNode` O parâmetro `wrapKey` contém a chave pública. O parâmetro `unwrapKey` contém a chave privada. O construtor `RawRsaKeyringNode` calcula um modo de preenchimento padrão, embora você possa especificar um modo de preenchimento preferencial.

Ao criar um chaveiro RSA bruto JavaScript, esteja ciente da possível incompatibilidade com outras implementações de linguagem.

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

Para ver um exemplo completo, consulte rsa_simple.ts (Node.js). JavaScript

```
import {
  RawRsaKeyringNode,
  buildClient,
```

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

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

const keyNamespace = 'HSM_01'
const keyName = 'RSA_2048_06'

const keyring = new RawRsaKeyringNode({ keyName, keyNamespace, rsaPublicKey,
 rsaPrivateKey})
```

Java

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

Python

O exemplo a seguir instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. REQUIRE_ENCRYPT_REQUIRE_DECRYPT Para ver um exemplo completo, consulte raw_rsa_keyring_example.py no AWS Encryption SDK for Python repositório em GitHub.

```
# Define the key namespace and key name
key_name_space = "HSM_01"
key_name = "RSA_2048_06"

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

# Create Raw RSA keyring
```

```
keyring_input: CreateRawRsaKeyringInput = CreateRawRsaKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    padding_scheme=PaddingScheme.OAEP_SHA256_MGF1,
    public_key=RSAPublicKey,
    private_key=RSAPrivateKey
)

raw_rsa_keyring: IKeyring = mat_prov.create_raw_rsa_keyring(
    input=keyring_input
)
```

Rust

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

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

// Define the key namespace and key name
let key_namespace: &str = "HSM_01";
let key_name: &str = "RSA_2048_06";

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

// Create Raw RSA keyring
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name(key_name)
    .key_namespace(key_namespace)
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(aws_smithy_types::Blob::new(RSAPublicKey))
```

```
    .private_key(aws_smithy_types::Blob::new(RSAPrivateKey))
    .send()
    .await?;
```

Go

```
// Instantiate the material providers library
matProv, err :=
 awscryptographymaterialproviderssmithygenerated.NewClient(awscryptographymaterialproviderss

// Create Raw RSA keyring
rsaKeyRingInput :=
 awscryptographymaterialproviderssmithygeneratedtypes.CreateRawRsaKeyringInput{
 KeyName:       "rsa",
 KeyNamespace:  "rsa-keyring",
 PaddingScheme:
 awscryptographymaterialproviderssmithygeneratedtypes.PaddingSchemePkcs1,
 PublicKey:     pem.EncodeToMemory(publicKeyBlock),
 PrivateKey:    pem.EncodeToMemory(privateKeyBlock),
}

rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
```

Go

```
import (
    "context"

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

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

```
}

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

// Define the key namespace and key name
var keyNamespace = "HSM_01"
var keyName = "RSA_2048_06"

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

// Create Raw RSA keyring
rsaKeyRingInput := mpltypes.CreateRawRsaKeyringInput{
    KeyName:       keyName,
    KeyNamespace:  keyNamespace,
    PaddingScheme: mpltypes.PaddingSchemeOaepSha512Mgf1,
    PublicKey:     (RSAPublicKey),
    PrivateKey:    (RSAPrivateKey),
}
rsaKeyring, err := matProv.CreateRawRsaKeyring(context.Background(),
 rsaKeyRingInput)
if err != nil {
    panic(err)
}
```

Chaveiros ECDH brutos

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 AWS Encryption SDK usa (`KDF_CTR_HMAC_SHA384`) para derivar a chave de empacotamento 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 chave. Para garantir que ambas as partes usem o material de chave correto, AWS Encryption SDK usam 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 encapsulamento compartilhada armazenadas no texto cifrado do cabeçalho da mensagem, a operação falhará. Por exemplo, se você criptografar dados 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 empacotamento compartilhada e poderá descriptografar os dados. Se a chave pública de Bob for de um AWS KMS key par, Bob poderá criar um chaveiro AWS KMS ECDH para descriptografar os dados.

O chaveiro ECDH bruto criptografa os dados 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 AWS Encryption SDK suporta as seguintes especificações de curva elíptica:

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

Compatibilidade com linguagens de programação

O chaveiro ECDH bruto foi introduzido na versão 1.5.0 da Biblioteca de Provedores de Material Criptográfico (MPL) e é suportado pelas seguintes linguagens e versões de programação:

- Versão 3. x do AWS Encryption SDK for Java
- Versão 4. x e versões posteriores do AWS Encryption SDK para o.NET
- Versão 4. x do AWS Encryption SDK for Python, quando usado com a dependência opcional do MPL.
- Versão 1. x do AWS Encryption SDK para Rust
- Versão 0.1. x ou posterior do AWS Encryption SDK for Go

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 chave pode criptografar e descriptografar dados.

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);
}
```

Python

O exemplo de Python a seguir usa o esquema de contrato de `RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey` 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.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey,
    RawPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Must be a PEM-encoded private key
bob_private_key = get_private_key_bytes()
# Must be a DER-encoded X.509 public key
alice_public_key = get_public_key_bytes()

# Create the raw ECDH static keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsRawPrivateKeyToStaticPublicKey(
        RawPrivateKeyToStaticPublicKeyInput(
            sender_static_private_key = bob_private_key,
            recipient_public_key = alice_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

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.

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

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

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

let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

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

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

Go

```
import (
    "context"
```

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

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

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

// Create keyring input
rawEcdhStaticConfigurationInput := mpltypes.RawPrivateKeyToStaticPublicKeyInput{
    SenderStaticPrivateKey: privateKeySender,
    RecipientPublicKey:     publicKeyRecipient,
}
rawECDHStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberRawPrivateKeyToStaticPublicKey{
    Value: rawEcdhStaticConfigurationInput,
}
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: rawECDHStaticConfiguration,
}

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

```
// Create raw ECDH static keyring
rawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

EphemeralPrivateKeyToStaticPublicKey

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 mensagens. Para descriptografar mensagens criptografadas 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);
}
```

Python

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

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey,
    EphemeralPrivateKeyToStaticPublicKeyInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

```
# Your get_public_key_bytes must return a DER-encoded X.509 public key
recipient_public_key = get_public_key_bytes()

# Create the raw ECDH ephemeral private key keyring
ephemeral_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme =
 RawEcdhStaticConfigurationsEphemeralPrivateKeyToStaticPublicKey(
        EphemeralPrivateKeyToStaticPublicKeyInput(
            recipient_public_key = recipient_public_key,
        )
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(ephemeral_input)
```

Rust

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.

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

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

// Load public key from UTF-8 encoded PEM files into a DER encoded public key.
let public_key_file_content =
 std::fs::read_to_string(Path::new(EXAMPLE_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content = parse(public_key_file_content)?;
let public_key_recipient_utf8_bytes = parsed_public_key_file_content.contents();

// Create EphemeralPrivateKeyToStaticPublicKeyInput
```

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

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

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

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

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

// Load public key from UTF-8 encoded PEM files into a DER encoded public key
publicKeyRecipient, err := LoadPublicKeyFromPEM(eccPublicKeyFileNameRecipient)
if err != nil {
    panic(err)
}

// Create EphemeralPrivateKeyToStaticPublicKeyInput
ephemeralRawEcdhStaticConfigurationInput :=
 mpltypes.EphemeralPrivateKeyToStaticPublicKeyInput{
    RecipientPublicKey: publicKeyRecipient,
}
ephemeralRawECDHStaticConfiguration :=
    mpltypes.RawEcdhStaticConfigurationsMemberEphemeralPrivateKeyToStaticPublicKey{
        Value: ephemeralRawEcdhStaticConfigurationInput,
    }

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

// Create raw ECDH ephemeral private key keyring
rawEcdhKeyRingInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: &ephemeralRawECDHStaticConfiguration,
}
ecdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 rawEcdhKeyRingInput)
if err != nil {
    panic(err)
}
```

PublicKeyDiscovery

Ao descriptografar, é uma prática recomendada especificar as chaves de encapsulamento que podem ser usadas. AWS Encryption SDK 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 mensagem em que a chave pública da chave especificada corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem. Esse esquema de contrato de chave só pode descriptografar mensagens.

> **Important**
>
> Ao descriptografar mensagens 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 mensagem em que a chave pública da chave privada especificada corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

```
// 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 mensagem em que a chave pública da chave privada especificada corresponda à chave pública do destinatário armazenada no texto cifrado da mensagem.

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

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
              PublicKeyDiscoveryInput.builder()
```

```
                // Must be a PEM-encoded private key

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

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

Python

O exemplo a seguir cria um chaveiro ECDH bruto com o esquema de contrato de `RawEcdhStaticConfigurationsPublicKeyDiscovery` 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.

```
import boto3
from aws_cryptographic_materialproviders.mpl.models import (
    CreateRawEcdhKeyringInput,
    RawEcdhStaticConfigurationsPublicKeyDiscovery,
    PublicKeyDiscoveryInput,
)
from aws_cryptography_primitives.smithygenerated.aws_cryptography_primitives.models
 import ECDHCurveSpec

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

# Your get_private_key_bytes must return a PEM-encoded private key
recipient_private_key = get_private_key_bytes()

# Create the raw ECDH discovery keyring
raw_keyring_input = CreateRawEcdhKeyringInput(
    curve_spec = ECDHCurveSpec.ECC_NIST_P256,
    key_agreement_scheme = RawEcdhStaticConfigurationsPublicKeyDiscovery(
        PublicKeyDiscoveryInput(
            recipient_static_private_key = recipient_private_key,
        )
```

```
    )
)

keyring = mat_prov.create_raw_ecdh_keyring(raw_keyring_input)
```

Rust

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.

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

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


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

// Load keys from UTF-8 encoded PEM files.
let mut file = File::open(Path::new(EXAMPLE_ECC_PRIVATE_KEY_FILENAME_RECIPIENT))?;
let mut private_key_recipient_utf8_bytes = Vec::new();
file.read_to_end(&mut private_key_recipient_utf8_bytes)?;

// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =
```

```
 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

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

// Load keys from UTF-8 encoded PEM files.
privateKeyRecipient, err := os.ReadFile(eccPrivateKeyFileNameRecipient)
```

```
if err != nil {
    panic(err)
}

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

// Create PublicKeyDiscoveryInput
discoveryRawEcdhStaticConfigurationInput := mpltypes.PublicKeyDiscoveryInput{
    RecipientStaticPrivateKey: privateKeyRecipient,
}

discoveryRawEcdhStaticConfiguration :=
 &mpltypes.RawEcdhStaticConfigurationsMemberPublicKeyDiscovery{
    Value: discoveryRawEcdhStaticConfigurationInput,
}

// Create raw ECDH discovery private key keyring
discoveryRawEcdhKeyringInput := mpltypes.CreateRawEcdhKeyringInput{
    CurveSpec:          ecdhCurveSpec,
    KeyAgreementScheme: discoveryRawEcdhStaticConfiguration,
}

discoveryRawEcdhKeyring, err := matProv.CreateRawEcdhKeyring(context.Background(),
 discoveryRawEcdhKeyringInput)
if err != nil {
    panic(err)
}
```

Multi-keyrings

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 token de autenticação gerador for um token de autenticação do KMS, a chave geradora no token de autenticação do AWS KMS gerará e criptografará 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.

Se você criar um chaveiro múltiplo sem gerador de chaves, poderá usá-lo sozinho para descriptografar dados, mas não para criptografar. Ou, para usar um chaveiro múltiplo sem gerador em operações de criptografia, você pode especificá-lo como um chaveiro secundário em outro chaveiro múltiplo. Um chaveiro múltiplo sem chaveiro gerador não pode ser designado como chaveiro gerador em outro chaveiro múltiplo.

Ao descriptografar, ele AWS Encryption SDK 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.

A partir da versão 1.7. x, quando uma chave de dados criptografada é criptografada em um chaveiro AWS Key Management Service (AWS KMS) (ou provedor de chave mestra), AWS Encryption SDK sempre passa o ARN da chave para AWS KMS key`KeyId` o parâmetro da AWS KMS operação Decrypt. Essa é uma prática AWS KMS recomendada que garante que você decodifique a chave de dados criptografada com a chave de empacotamento que você pretende usar.

Para ver um exemplo prático de um multitoken de autenticação, consulte:

- C: multi_keyring.cpp
- C# /.NET: MultiKeyringExample.cs
- JavaScript Node.js: multi_keyring.ts
- JavaScript Navegador: multi_keyring.ts
- Java: MultiKeyringExample.java
- Python: multi_keyring_example.py

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.

C

```
/* Define an AWS KMS keyring. For details, see string.cpp */
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(example_key);

// Define a Raw AES keyring. For details, see raw_aes_keyring.c */
struct aws_cryptosdk_keyring *aes_keyring = aws_cryptosdk_raw_aes_keyring_new(
        alloc, wrapping_key_namespace, wrapping_key_name, wrapping_key,
 AWS_CRYPTOSDK_AES256);
```

C# / .NET

```
// Define an AWS KMS keyring. For details, see AwsKmsKeyringExample.cs.
var kmsKeyring = materialProviders.CreateAwsKmsKeyring(createKmsKeyringInput);

// Define a Raw AES keyring. For details, see RawAESKeyringExample.cs.
var aesKeyring = materialProviders.CreateRawAesKeyring(createAesKeyringInput);
```

JavaScript Browser

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

```
import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  RawAesKeyringWebCrypto,
  RawAesWrappingSuiteIdentifier,
  MultiKeyringWebCrypto,
  buildClient,
  CommitmentPolicy,
  synchronousRandomValues,
} from '@aws-crypto/client-browser'
```

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

const clientProvider = getClient(KMS, { credentials })

// Define an AWS KMS keyring. For details, see kms_simple.ts.
const kmsKeyring = new KmsKeyringBrowser({ generatorKeyId: exampleKey })

// Define a Raw AES keyring. For details, see aes_simple.ts.
const aesKeyring = new RawAesKeyringWebCrypto({ keyName, keyNamespace,
 wrappingSuite, masterKey })
```

JavaScript Node.js

O exemplo a seguir usa a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

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

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

// Define an AWS KMS keyring. For details, see kms_simple.ts.
const kmsKeyring = new KmsKeyringNode({ generatorKeyId: exampleKey })

// Define a Raw AES keyring. For details, see raw_aes_keyring_node.ts.
const aesKeyring = new RawAesKeyringNode({ keyName, keyNamespace, wrappingSuite,
 unencryptedMasterKey })
```

Java

```
// Define the raw AES keyring.
```

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

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

Python

O exemplo a seguir instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. `REQUIRE_ENCRYPT_REQUIRE_DECRYPT`

```
# Create the AWS KMS keyring
kms_client = boto3.client('kms', region_name="us-west-2")

mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
    config=MaterialProvidersConfig()
)

kms_keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
    generator=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab,
    kms_client=kms_client
)

kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
    input=kms_keyring_input
)

# Create Raw AES keyring
```

```
key_name_space = "HSM_01"
key_name = "AES_256_012"

raw_aes_keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
    key_namespace=key_name_space,
    key_name=key_name,
    wrapping_key=AESWrappingKey,
    wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
)

raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
    input=raw_aes_keyring_input
)
```

Rust

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

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

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

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

// Create a Raw AES keyring
let key_namespace: &str = "my-key-namespace";
let key_name: &str = "my-aes-key-name";

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
```

```
    .key_name(key_name)
    .key_namespace(key_namespace)
    .wrapping_key(aws_smithy_types::Blob::new(AESWrappingKey))
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

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

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

```
}

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

// Create a Raw AES keyring
var keyNamespace = "my-key-namespace"
var keyName = "my-aes-key-name"

aesKeyRingInput := mpltypes.CreateRawAesKeyringInput{
    KeyName:      keyName,
    KeyNamespace: keyNamespace,
    WrappingKey:  AESWrappingKey,
    WrappingAlg:  mpltypes.AesWrappingAlgAlgAes256GcmIv12Tag16,
}
aesKeyring, err := matProv.CreateRawAesKeyring(context.Background(),
 aesKeyRingInput)
```

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.

C

No construtor de multitoken de autenticação no C, você especifica apenas seu token de autenticação gerador.

```
struct aws_cryptosdk_keyring *multi_keyring = aws_cryptosdk_multi_keyring_new(alloc,
 kms_keyring);
```

Para adicionar um token de autenticação filho ao multitoken de autenticação, use o método `aws_cryptosdk_multi_keyring_add_child`. Você precisa chamar o método uma vez para cada token de autenticação filho que adicionar.

```
// Add the Raw AES keyring (C only)
aws_cryptosdk_multi_keyring_add_child(multi_keyring, aes_keyring);
```

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 = kmsKeyring,
    ChildKeyrings = new List<IKeyring>() {aesKeyring}
};

var multiKeyring = materialProviders.CreateMultiKeyring(createMultiKeyringInput);
```

JavaScript Browser

JavaScript vários chaveiros são imutáveis. O construtor JavaScript de vários chaveiros permite que você especifique o chaveiro do gerador e vários chaveiros infantis.

```
const clientProvider = getClient(KMS, { credentials })

const multiKeyring = new MultiKeyringWebCrypto(generator: kmsKeyring, children:
 [aesKeyring]);
```

JavaScript Node.js

JavaScript vários chaveiros são imutáveis. O construtor JavaScript de vários chaveiros permite que você especifique o chaveiro do gerador e vários chaveiros infantis.

```
const multiKeyring = new MultiKeyringNode(generator: kmsKeyring, children:
 [aesKeyring]);
```

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);
```

Python

```
multi_keyring_input: CreateMultiKeyringInput = CreateMultiKeyringInput(
    generator=kms_keyring,
    child_keyrings=[raw_aes_keyring]
)

multi_keyring: IKeyring = mat_prov.create_multi_keyring(
    input=multi_keyring_input
)
```

Rust

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

Go

```
createMultiKeyringInput := mpltypes.CreateMultiKeyringInput{
  Generator:     awsKmsKeyring,
  ChildKeyrings: []mpltypes.IKeyring{rawAESKeyring},
 }
 multiKeyring, err := matProv.CreateMultiKeyring(context.Background(),
 createMultiKeyringInput)
 if err != nil {
  panic(err)
```

```
}
```

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

AWS Encryption SDK linguagens de programação

O AWS Encryption SDK está disponível para as seguintes linguagens de programação. As implementações de linguagem 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. Além disso, ao criptografar e descriptografar, é necessário usar tokens de autenticação compatíveis ou chaves mestras e provedores de chaves mestras. Para obter mais detalhes, consulte the section called “Compatibilidade dos tokens de autenticação”.

Tópicos

- AWS Encryption SDK for C
- AWS Encryption SDK para o.NET
- AWS Encryption SDK para Go
- AWS Encryption SDK for Java
- AWS Encryption SDK para JavaScript
- AWS Encryption SDK for Python
- AWS Encryption SDK para Rust
- AWS Encryption SDK interface de linha de comando

AWS Encryption SDK for C

O AWS Encryption SDK for C fornece uma biblioteca de criptografia do lado do cliente para desenvolvedores que estão escrevendo aplicativos em C. Ela também serve como base para implementações de linguagens de programação de nível superior. AWS Encryption SDK

Como todas as implementações do AWS Encryption SDK, o AWS Encryption SDK for C oferece recursos avançados de proteção de dados. Os recursos incluem criptografia de envelope, AAD (additional authenticated data - dados autenticados adicionais) e pacotes de algoritmos de chave simétrica segura e autenticada, como o AES-GCM de 256 bits com derivação de chave e assinatura.

Todas as implementações específicas de linguagem do AWS Encryption SDK são totalmente interoperáveis. Por exemplo, você pode criptografar dados com o AWS Encryption SDK for C e descriptografá-los com qualquer implementação de linguagem compatível, incluindo a CLI de criptografia.AWS

AWS Encryption SDK for C Isso requer que AWS SDK para C++ o interaja com AWS Key Management Service (AWS KMS). Você precisa usá-lo somente se estiver usando o AWS KMS token de autenticação opcional. No entanto, AWS Encryption SDK não requer AWS KMS nenhum outro AWS serviço.

Saiba mais

- Para obter detalhes sobre a programação com o AWS Encryption SDK for C, consulte os exemplos em C, os exemplos no aws-encryption-sdk-c repositório em GitHub e a documentação da AWS Encryption SDK for C API.
- Para uma discussão sobre como usar o para criptografar dados AWS Encryption SDK for C para que você possa descriptografá-los em vários Regiões da AWS, consulte Como descriptografar textos cifrados em várias regiões com o em C no Blog de Segurança. AWS Encryption SDK AWS

Tópicos

- Instalando o AWS Encryption SDK for C
- Usando o AWS Encryption SDK for C
- AWS Encryption SDK for C exemplos

Instalando o AWS Encryption SDK for C

Instale a versão mais recente do AWS Encryption SDK for C.

> **Note**
>
> Todas as versões AWS Encryption SDK for C anteriores à 2.0.0 estão em end-of-supportfase.
> Você pode atualizar com segurança a partir da versão 2.0.x e posteriores até a versão mais recente do AWS Encryption SDK for C sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.0.x não são compatíveis com versões anteriores. Para atualizar a partir de versões anteriores à 1.7.x até a versão 2.0.x e posteriores, primeiro será necessário atualizar para a versão 1.x mais recente do AWS Encryption SDK for C. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

Você pode encontrar instruções detalhadas para instalar e criar o AWS Encryption SDK for C no arquivo README do aws-encryption-sdk-crepositório. Ele inclui instruções para criar nas plataformas Amazon Linux, Ubuntu, macOS e Windows.

Antes de começar, decida se deseja usar tokens de autenticação do AWS KMS no AWS Encryption SDK. Se você usa um AWS KMS chaveiro, você precisa instalar o. AWS SDK para C++ O AWS SDK é necessário para interagir com AWS Key Management Service(AWS KMS). Quando você AWS Encryption SDK usa AWS KMS chaveiros, eles usam AWS KMS para gerar e proteger as chaves de criptografia que protegem seus dados.

Você não precisa instalar o AWS SDK para C++ se estiver usando outro tipo de chaveiro, como um chaveiro AES bruto, um chaveiro RSA bruto ou um chaveiro múltiplo que não inclua um chaveiro. AWS KMS No entanto, ao usar um tipo de token de autenticação bruto, você precisa gerar e proteger suas próprias chaves de encapsulamento brutas.

Se você estiver com problemas com a instalação, registre um problema no repositório do `aws-encryption-sdk-c` ou use os links de feedback desta página.

Usando o AWS Encryption SDK for C

Este tópico explica alguns dos recursos do AWS Encryption SDK for C que não são suportados em outras implementações de linguagem de programação.

Esses exemplos mostram como usar a versão 2.0.x e versões posteriores do AWS Encryption SDK for C. Para exemplos que usam versões anteriores, encontre sua versão na lista de lançamentos do aws-encryption-sdk-c repositório em. GitHub

Para obter detalhes sobre a programação com o AWS Encryption SDK for C, consulte os exemplos em C, os exemplos no aws-encryption-sdk-c repositório em GitHub e a documentação da AWS Encryption SDK for C API.

Consulte também: Tokens de autenticação

Tópicos

- Padrões para criptografar e descriptografar dados
- Contagem de referências

Padrões para criptografar e descriptografar dados

Ao usar o AWS Encryption SDK for C, você segue um padrão semelhante a este: cria um chaveiro, cria um CMM que usa o chaveiro, cria uma sessão que usa o CMM (e o chaveiro) e, em seguida, processa a sessão.

1. Carregar sequências de erro.

   Chame o método `aws_cryptosdk_load_error_strings()` no código C++ ou C++. Ele carrega informações de erro que são muito úteis para depuração.

   Você só precisa chamá-lo uma vez, como no método `main`.

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

2. Crie um token de autenticação.

   Configure o token de autenticação com as chaves de empacotamento que você deseja usar para criptografar suas chaves de dados. Este exemplo usa um AWS KMS chaveiro com um AWS KMS key, mas você pode usar qualquer tipo de chaveiro em seu lugar.

   Para identificar um AWS KMS key em um chaveiro de criptografia no AWS Encryption SDK for C, especifique o ARN da chave ou o ARN do alias. Em um token de autenticação de descriptografia, é necessário usar um ARN de chave. Para obter detalhes, consulte Identificação AWS KMS keys em um AWS KMS chaveiro.

   ```
   const char * KEY_ARN = "arn:aws:kms:us-
   west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
   struct aws_cryptosdk_keyring *kms_keyring =
          Aws::Cryptosdk::KmsKeyring::Builder().Build(KEY_ARN);
   ```

3. Crie uma sessão.

   No AWS Encryption SDK for C, você usa uma sessão para criptografar uma única mensagem de texto simples ou descriptografar uma única mensagem de texto cifrado, independentemente do tamanho. A sessão mantém o estado da mensagem durante todo o processamento.

   Configure a sessão com um alocador, um token de autenticação e um modo: `AWS_CRYPTOSDK_ENCRYPT` ou `AWS_CRYPTOSDK_DECRYPT`. Se você precisar alterar o modo da sessão, use o método `aws_cryptosdk_session_reset`.

Quando você cria uma sessão com um chaveiro, ele cria AWS Encryption SDK for C automaticamente um gerenciador de materiais criptográficos (CMM) padrão para você. Você não precisa criar, manter ou destruir esse objeto.

Por exemplo, a sessão a seguir usa o alocador e o token de autenticação definido na etapa 1. Ao criptografar dados, o modo é o `AWS_CRYPTOSDK_ENCRYPT`.

```
struct aws_cryptosdk_session * session =
 aws_cryptosdk_session_new_from_keyring_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);
```

4. Criptografe ou descriptografe os dados.

Para processar os dados na sessão, use o método `aws_cryptosdk_session_process`. Se o buffer de entrada for grande o suficiente para conter todo o texto simples, e o buffer de saída for grande o suficiente para conter todo o texto cifrado, você pode chamar `aws_cryptosdk_session_process_full`. No entanto, se precisar lidar com dados de streaming, você poderá chamar `aws_cryptosdk_session_process` em um loop. Para obter um exemplo, consulte o file_streaming.cpp . O `aws_cryptosdk_session_process_full` é introduzido nas AWS Encryption SDK versões 1.9. x e 2.2. x.

Quando a sessão é configurada para criptografar dados, os campos em texto simples descrevem a entrada e os campos de texto cifrado descrevem a saída. O campo `plaintext` contém a mensagem que você deseja criptografar, e o campo `ciphertext` obtém a mensagem criptografada retornada pelo método de criptografia.

```
/* Encrypting data */
aws_cryptosdk_session_process_full(session,
                                   ciphertext,
                                   ciphertext_buffer_size,
                                   &ciphertext_length,
                                   plaintext,
                                   plaintext_length)
```

Quando a sessão é configurada para descriptografar dados, os campos em texto cifrado descrevem a entrada e os campos em texto cifrado descrevem a saída. O campo `ciphertext` contém a mensagem criptografada retornada pelo método de criptografia, e o campo `plaintext` obtém a mensagem em texto simples retornada pelo método de descriptografia.

Para descriptografar os dados, chame o método `aws_cryptosdk_session_process_full`.

```
/* Decrypting data */
aws_cryptosdk_session_process_full(session,
                                   plaintext,
                                   plaintext_buffer_size,
                                   &plaintext_length,
                                   ciphertext,
                                   ciphertext_length)
```

Contagem de referências

Para evitar vazamentos de memória, libere as referências a todos os objetos que você criou ao concluir. Caso contrário, ocorrerão vazamentos de memória. O SDK fornece métodos para facilitar essa tarefa.

Sempre que você criar um objeto pai com um dos seguintes objetos filho, o objeto pai obtém e mantém uma referência ao objeto filho, da seguinte forma:

- Um token de autenticação, como criar uma sessão com um token de autenticação
- Um gerenciador de materiais criptográficos (CMM) padrão, como criar uma sessão ou um CMM personalizado com um CMM padrão
- Um cache de chaves de dados, como criar um CMM de armazenamento em cache com um token de autenticação e um cache

A menos que precise de uma referência independente ao objeto filho, você pode liberar a referência ao objeto filho assim que criar o objeto pai. A referência restante ao objeto filho é liberada quando o objeto pai é destruído. Esse padrão garante que você mantenha a referência a cada objeto somente pelo tempo necessário e não ocorra vazamento de memória causado por referências não liberadas.

Você só é responsável por liberar referências aos objetos filho que cria explicitamente. Você não é responsável por gerenciar referências a objetos criados pelo SDK para você. Se o SDK criar um objeto, como o CMM padrão que o método `aws_cryptosdk_caching_cmm_new_from_keyring` adiciona a uma sessão, o SDK gerenciará a criação e a destruição do objeto e suas referências.

No exemplo a seguir, ao criar uma sessão com um token de autenticação, a sessão obtém uma referência ao token de autenticação e mantém essa referência até que a sessão seja destruída.

Se você não precisar manter uma referência adicional ao token de autenticação, poderá usar o método `aws_cryptosdk_keyring_release` para liberar o objeto do token de autenticação assim que a sessão for criada. Esse método diminui a contagem de referências para o token de autenticação. A referência da sessão ao token de autenticação é liberada quando você chama `aws_cryptosdk_session_destroy` para destruir a sessão.

```
// The session gets a reference to the keyring.
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_ENCRYPT, keyring);

// After you create a session with a keyring, release the reference to the keyring
 object.
aws_cryptosdk_keyring_release(keyring);
```

Para tarefas mais complexas, como reutilizar um token de autenticação para várias sessões ou especificar um pacote de algoritmos em um CMM, talvez seja necessário manter uma referência independente ao objeto. Se assim for, não chame os métodos de liberação imediatamente. Em vez disso, libere as referências quando você não estiver mais usando os objetos, além de destruir a sessão.

Essa técnica de contagem de referência também funciona quando você está usando alternativas CMMs, como o CMM de cache para armazenamento em cache de chaves de dados. Ao criar um CMM de armazenamento em cache de um cache e de um token de autenticação, o CMM de armazenamento em cache obtém uma referência aos dois objetos. A menos que precise delas para outra tarefa, você pode liberar suas referências independentes para cache e o token de autenticação assim que o CMM de armazenamento em cache for criado. Depois, ao criar uma sessão com o CMM de armazenamento em cache, você pode liberar sua referência para o CMM de armazenamento em cache.

Observe que você só é responsável por liberar referências a objetos que cria explicitamente. Os objetos criados pelos métodos para você, como o CMM padrão que é subjacente ao CMM de armazenamento em cache, são gerenciados pelo método.

```
/ Create the caching CMM from a cache and a keyring.
struct aws_cryptosdk_cmm *caching_cmm =
 aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL, 60,
 AWS_TIMESTAMP_SECS);

// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
```

```
aws_cryptosdk_keyring_release(kms_keyring);

// Create a session with the caching CMM.
struct aws_cryptosdk_session *session = aws_cryptosdk_session_new_from_cmm_2(allocator,
 AWS_CRYPTOSDK_ENCRYPT, caching_cmm);

// Release your references to the caching CMM.
aws_cryptosdk_cmm_release(caching_cmm);

// ...

aws_cryptosdk_session_destroy(session);
```

AWS Encryption SDK for C exemplos

Os exemplos a seguir mostram como usar o para AWS Encryption SDK for C criptografar e descriptografar dados.

Esses exemplos mostram como usar as versões 2.0.x e posteriores do AWS Encryption SDK for C. Para exemplos que usam versões anteriores, encontre sua versão na lista de lançamentos do aws-encryption-sdk-c repositório em. GitHub

Quando você instala e constrói o AWS Encryption SDK for C, o código-fonte desses e de outros exemplos é incluído no `examples` subdiretório e eles são compilados e incorporados ao `build` diretório. Você também pode encontrá-los no subdiretório de exemplos do aws-encryption-sdk-crepositório em. GitHub

Tópicos

- Criptografar e descriptografar strings

Criptografar e descriptografar strings

O exemplo a seguir mostra como usar o para AWS Encryption SDK for C criptografar e descriptografar uma string.

Este exemplo apresenta o AWS KMS chaveiro, um tipo de chaveiro que usa um AWS KMS key in AWS Key Management Service (AWS KMS) para gerar e criptografar chaves de dados. O exemplo inclui código escrito em C++. AWS Encryption SDK for C Isso exige que AWS SDK para C++ você ligue AWS KMS ao usar AWS KMS chaveiros. Se você estiver usando um chaveiro que não interage

com AWS KMS, como um chaveiro AES bruto, um chaveiro RSA bruto ou um chaveiro múltiplo que não inclui um AWS KMS chaveiro, isso não é necessário. AWS SDK para C++

Para obter ajuda na criação de um AWS KMS key, consulte Criação de chaves no Guia do AWS Key Management Service desenvolvedor. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

Consulte o exemplo de código completo: string.cpp

Tópicos

- Criptografar uma string
- Descriptografar uma string

Criptografar uma string

A primeira parte deste exemplo usa um AWS KMS chaveiro com um AWS KMS key para criptografar uma string de texto sem formatação.

Etapa 1. Carregar sequências de erro.

Chame o método `aws_cryptosdk_load_error_strings()` no código C++ ou C++. Ele carrega informações de erro que são muito úteis para depuração.

Você só precisa chamá-lo uma vez, como no método `main`.

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

Etapa 2: estruturar o token de autenticação.

Crie um AWS KMS chaveiro para criptografia. O chaveiro neste exemplo é configurado com um AWS KMS key, mas você pode configurar um AWS KMS chaveiro com vários AWS KMS keys, inclusive AWS KMS keys em contas diferentes Regiões da AWS e diferentes.

Para identificar um AWS KMS key em um chaveiro de criptografia no AWS Encryption SDK for C, especifique o ARN da chave ou o ARN do alias. Em um token de autenticação de descriptografia, é necessário usar um ARN de chave. Para obter detalhes, consulte Identificação AWS KMS keys em um AWS KMS chaveiro.

Identificação AWS KMS keys em um AWS KMS chaveiro

Ao criar um chaveiro com vários AWS KMS keys, você especifica o AWS KMS key usado para gerar e criptografar a chave de dados de texto simples e uma matriz opcional de outros AWS KMS keys que criptografam a mesma chave de dados de texto sem formatação. Nesse caso, você especifica somente o gerador AWS KMS key.

Antes de executar esse código, substitua o ARN da chave de exemplo por um válido.

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

struct aws_cryptosdk_keyring *kms_keyring =
       Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

Etapa 3: Criar uma sessão.

Crie uma sessão usando o alocador, um enumerador de modo e o token de autenticação.

Cada sessão requer um modo: `AWS_CRYPTOSDK_ENCRYPT` para criptografar ou `AWS_CRYPTOSDK_DECRYPT` para descriptografar. Para alterar o modo de uma sessão existente, use o método `aws_cryptosdk_session_reset`.

Depois de criar um sessão com o token de autenticação, você poderá liberar sua referência ao token de autenticação usando o método fornecido pelo SDK. A sessão retém uma referência ao objeto token de autenticação durante sua vida útil. Referências ao token de autenticação e aos objetos de sessão são liberadas quando você destrói a sessão. Essa técnica de contagem de referência ajuda a evitar vazamentos de memória e a evitar que os objetos sejam liberados enquanto estão em uso.

```
struct aws_cryptosdk_session *session =
       aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_ENCRYPT,
 kms_keyring);

/* When you add the keyring to the session, release the keyring object */
aws_cryptosdk_keyring_release(kms_keyring);
```

Etapa 4: Definir o contexto de criptografia.

Um contexto de criptografia são dados autenticados adicionais arbitrários e que não são secretos. Quando você fornece um contexto de criptografia na criptografia, ele vincula AWS Encryption SDK criptograficamente o contexto de criptografia ao texto cifrado, de forma que o mesmo

contexto de criptografia seja necessário para descriptografar os dados. O uso de um contexto de criptografia é opcional, mas o recomendamos como uma melhor prática.

Primeiro, crie uma tabela de hash que inclua as strings de contexto de criptografia.

```
/* Allocate a hash table for the encryption context */
int set_up_enc_ctx(struct aws_allocator *alloc, struct aws_hash_table *my_enc_ctx)

// Create encryption context strings
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key1, "Example");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value1, "String");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key2, "Company");
AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value2, "MyCryptoCorp");

// Put the key-value pairs in the hash table
aws_hash_table_put(my_enc_ctx, enc_ctx_key1, (void *)enc_ctx_value1, &was_created)
aws_hash_table_put(my_enc_ctx, enc_ctx_key2, (void *)enc_ctx_value2, &was_created)
```

Obtenha um ponteiro mutável para o contexto de criptografia na sessão. Depois, use a função `aws_cryptosdk_enc_ctx_clone` para copiar o contexto de criptografia na sessão. Mantemos a cópia em `my_enc_ctx` para que possamos validar o valor depois de descriptografar os dados.

O contexto de criptografia faz parte da sessão, não é um parâmetro transmitido para a função de processo da sessão. Isso garante que o mesmo contexto de criptografia seja usado para todos os segmentos de uma mensagem, mesmo se a função de processo de sessão for chamada várias vezes para criptografar a mensagem inteira.

```
struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);

aws_cryptosdk_enc_ctx_clone(alloc, session_enc_ctx, my_enc_ctx)
```

Etapa 5: Criptografar a string.

Para criptografar a string em texto simples, use o método `aws_cryptosdk_session_process_full` com a sessão no modo de criptografia. Esse método, introduzido nas AWS Encryption SDK versões 1.9. x e 2.2. x, foi projetado para criptografia e decodificação sem streaming. Para lidar com dados de streaming, chame o `aws_cryptosdk_session_process` continuamente.

Na criptografia, os campos em texto simples são campos de entrada; os campos em texto cifrado são campos de saída. Concluído o processamento, o campo `ciphertext_output` conterá a mensagem criptografada, incluindo o texto cifrado real, as chaves de dados criptografadas e o contexto de criptografia. Você pode descriptografar essa mensagem criptografada usando o AWS Encryption SDK para qualquer linguagem de programação compatível.

```
/* Gets the length of the plaintext that the session processed */
size_t ciphertext_len_output;
if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
                                  ciphertext_output,
                                  ciphertext_buf_sz_output,
                                  &ciphertext_len_output,
                                  plaintext_input,
                                  plaintext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 8;
}
```

Passo 6: Limpar a sessão.

A etapa final destroi a sessão, inclusive as referências ao CMM e ao token de autenticação.

Se você preferir, em vez de destruir a sessão, reutilize-a com o mesmo token de autenticação e CMM para descriptografar a string, ou para criptografar ou descriptografar outras mensagens. Para usar a sessão para descriptografia, use o método `aws_cryptosdk_session_reset` para alterar o modo para `AWS_CRYPTOSDK_DECRYPT`.

Descriptografar uma string

A segunda parte deste exemplo descriptografa uma mensagem criptografada que contém o texto cifrado da string original.

Etapa 1: carregar sequências de erro.

Chame o método `aws_cryptosdk_load_error_strings()` no código C++ ou C++. Ele carrega informações de erro que são muito úteis para depuração.

Você só precisa chamá-lo uma vez, como no método `main`.

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

Etapa 2: estruturar o token de autenticação.

Ao descriptografar dados AWS KMS, você passa a mensagem criptografada que a API de criptografia retornou. A API Decrypt não aceita uma AWS KMS key entrada. Em vez disso, AWS KMS usa o mesmo AWS KMS key para descriptografar o texto cifrado usado para criptografá-lo. No entanto, AWS Encryption SDK permite que você especifique um AWS KMS chaveiro sem AWS KMS keys criptografar e descriptografar.

Ao descriptografar, você pode configurar um chaveiro apenas com o AWS KMS keys que deseja usar para descriptografar a mensagem criptografada. Por exemplo, talvez você queira criar um chaveiro apenas com o AWS KMS key que é usado por uma função específica em sua organização. Eles nunca AWS Encryption SDK usarão um, AWS KMS key a menos que apareça no chaveiro de decodificação. Se o SDK não conseguir descriptografar as chaves de dados criptografadas usando o AWS KMS keys chaveiro fornecido por você, seja porque nada do chaveiro foi usado para criptografar nenhuma das AWS KMS keys chaves de dados ou porque o chamador não tem permissão para usar o chaveiro para descriptografar, a chamada de descriptografia falhará. AWS KMS keys

Ao especificar um AWS KMS key para um chaveiro de decodificação, você deve usar o ARN da chave. Os alias ARNs são permitidos somente em chaveiros de criptografia. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulteIdentificação AWS KMS keys em um AWS KMS chaveiro.

Neste exemplo, especificamos um chaveiro configurado com o mesmo AWS KMS key usado para criptografar a string. Antes de executar esse código, substitua o ARN da chave de exemplo por um válido.

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

struct aws_cryptosdk_keyring *kms_keyring =
        Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
```

Etapa 3: Criar uma sessão.

Crie uma sessão usando o alocador e o token de autenticação. Para configurar a sessão para descriptografia, configure a sessão com o modo `AWS_CRYPTOSDK_DECRYPT`.

Depois de criar uma sessão com um token de autenticação, você poderá liberar sua referência ao token de autenticação usando o método fornecido pelo SDK. A sessão mantém uma referência

ao objeto token de autenticação durante sua vida útil, e a sessão e o token de autenticação são liberados quando você destrói a sessão. Essa técnica de contagem de referência ajuda a evitar vazamentos de memória e a evitar que os objetos sejam liberados enquanto estão em uso.

```
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_keyring_2(alloc, AWS_CRYPTOSDK_DECRYPT,
 kms_keyring);

/* When you add the keyring to the session, release the keyring object */
aws_cryptosdk_keyring_release(kms_keyring);
```

Etapa 4: Descriptografar a string.

Para descriptografar a string, use o método `aws_cryptosdk_session_process_full` com a sessão configurada para descriptografia. Esse método, introduzido nas versões 1.9.x e 2.2.x do AWS Encryption SDK , foi projetado para criptografia e descriptografia que não sejam de streaming. Para lidar com dados de streaming, chame o `aws_cryptosdk_session_process` continuamente.

Na descriptografia, os campos em texto cifrado são campos de entrada, e os campos em texto simples são campos de saída. O campo `ciphertext_input` contém a mensagem criptografada retornada pelo método de criptografia. Quando o processamento for concluído, o campo `plaintext_output` conterá a string em texto simples (descriptografada).

```
size_t plaintext_len_output;

if (AWS_OP_SUCCESS != aws_cryptosdk_session_process_full(session,
                                    plaintext_output,
                                    plaintext_buf_sz_output,
                                    &plaintext_len_output,
                                    ciphertext_input,
                                    ciphertext_len_input)) {
    aws_cryptosdk_session_destroy(session);
    return 13;
}
```

Etapa 5: Verificar o contexto de criptografia.

Verifique se o contexto de criptografia real, o que foi usado para descriptografar a mensagem, contém o contexto de criptografia fornecido ao criptografar a mensagem. O contexto de

criptografia real pode incluir pares extras porque o gerenciador de materiais criptográficos (CMM) pode adicionar pares ao contexto de criptografia fornecido antes de criptografar a mensagem.

No AWS Encryption SDK for C, você não precisa fornecer um contexto de criptografia ao descriptografar porque o contexto de criptografia está incluído na mensagem criptografada que o SDK retorna. No entanto, antes que a mensagem em texto simples seja retornada, sua função de descriptografia deve verificar se todos os pares no contexto de criptografia fornecido aparecem no contexto que foi usado para descriptografar a mensagem.

Primeiro, obtenha um ponteiro somente leitura para a tabela de hash na sessão. Essa tabela de hash contém o contexto de criptografia que foi usado para descriptografar a mensagem.

```
const struct aws_hash_table *session_enc_ctx =
 aws_cryptosdk_session_get_enc_ctx_ptr(session);
```

Depois, percorra o contexto na tabela de hash `my_enc_ctx` que você copiou ao efetuar a criptografia. Verifique se cada par na tabela de hash `my_enc_ctx` que foi usada para efetuar a criptografia aparece na tabela de hash `session_enc_ctx` que foi usada para efetuar a descriptografia. Se alguma chave estiver ausente ou se essa chave tiver um valor diferente, interrompa o processamento e escreva uma mensagem de erro.

```
for (struct aws_hash_iter iter = aws_hash_iter_begin(my_enc_ctx); !
aws_hash_iter_done(&iter);
      aws_hash_iter_next(&iter)) {
     struct aws_hash_element *session_enc_ctx_kv_pair;
     aws_hash_table_find(session_enc_ctx, iter.element.key,
 &session_enc_ctx_kv_pair)

    if (!session_enc_ctx_kv_pair ||
        !aws_string_eq(
            (struct aws_string *)iter.element.value, (struct aws_string
 *)session_enc_ctx_kv_pair->value)) {
        fprintf(stderr, "Wrong encryption context!\n");
        abort();
    }
}
```

Passo 6: Limpar a sessão.

Depois de verificar o contexto de criptografia, destrua a sessão ou a reutilize. Se precisar reconfigurá-la, use o método `aws_cryptosdk_session_reset`.

```
aws_cryptosdk_session_destroy(session);
```

AWS Encryption SDK para o.NET

O AWS Encryption SDK for .NET é uma biblioteca de criptografia do lado do cliente para desenvolvedores que estão escrevendo aplicativos em C# e em outras linguagens de programação.NET. É compatível com Windows, macOS e Linux.

> **Note**
>
> A versão 4.0.0 do AWS Encryption SDK para.NET se desvia da Especificação da AWS Encryption SDK Mensagem. Como resultado, as mensagens criptografadas pela versão 4.0.0 só podem ser descriptografadas pela versão 4.0.0 ou posterior do para.NET. AWS Encryption SDK Eles não podem ser descriptografados por nenhuma outra implementação de linguagem de programação.
> A versão 4.0.1 do AWS Encryption SDK para.NET grava mensagens de acordo com a Especificação da AWS Encryption SDK Mensagem e é interoperável com outras implementações de linguagem de programação. Por padrão, a versão 4.0.1 pode ler mensagens criptografadas pela versão 4.0.0. No entanto, se você não quiser descriptografar mensagens criptografadas pela versão 4.0.0, você pode especificar a propriedade `NetV4_0_0_RetryPolicy` para impedir que o cliente leia essas mensagens. Para obter mais informações, consulte as notas de versão v4.0.1 no aws-encryption-sdk repositório em. GitHub

O AWS Encryption SDK for .NET difere de algumas das outras implementações de linguagem de programação do AWS Encryption SDK das seguintes maneiras:

- Não há suporte para armazenamento em cache de chaves de dados

  > **Note**
  >
  > Versão 4. O x of the AWS Encryption SDK for.NET suporta o AWS KMS chaveiro hierárquico, uma solução alternativa de cache de materiais criptográficos.

- Não há suporte para streaming de dados
- Não há registros ou rastreamentos de pilha do AWS Encryption SDK para .NET

- Requer o AWS SDK para .NET

O AWS Encryption SDK para o.NET inclui todos os recursos de segurança introduzidos nas versões 2.0. x e posteriores de outras implementações de linguagem do AWS Encryption SDK. No entanto, se você estiver usando o for.NET AWS Encryption SDK para descriptografar dados que foram criptografados por uma versão pré-2.0. versão x outra implementação de linguagem do AWS Encryption SDK, talvez seja necessário ajustar sua política de compromisso. Para obter detalhes, consulte Como definir sua política de compromisso.

O AWS Encryption SDK for .NET é um produto do AWS Encryption SDK in Dafny, uma linguagem de verificação formal na qual você escreve especificações, o código para implementá-las e as provas para testá-las. O resultado é uma biblioteca que implementa os atributos do AWS Encryption SDK em uma estrutura que garante a correção funcional.

Saiba mais

- Para ver exemplos que mostram como configurar opções no AWS Encryption SDK, como especificar um conjunto alternativo de algoritmos, limitar chaves de dados criptografadas e usar chaves AWS KMS multirregionais, consulte. Configurando o AWS Encryption SDK
- Para obter detalhes sobre a programação com o AWS Encryption SDK para.NET, consulte o `aws-encryption-sdk-net`diretório do aws-encryption-sdk repositório em GitHub.

Tópicos

- Instalando o AWS Encryption SDK para o.NET
- Depurando o para o.NET AWS Encryption SDK
- AWS Encryption SDK para exemplos do.NET

Instalando o AWS Encryption SDK para o.NET

O AWS Encryption SDK para o.NET está disponível como `AWS.Cryptography.EncryptionSDK`pacote em NuGet. Para obter detalhes sobre como instalar e criar o AWS Encryption SDK para.NET, consulte o arquivo README.md no repositório. `aws-encryption-sdk-net`

Versão 3.x

Versão 3. x do AWS Encryption SDK para o.NET oferece suporte a o.NET Framework 4.5.2 — 4.8 somente no Windows. É compatível com o .NET Core 3.0+ e o .NET 5.0 e versões posteriores em todos os sistemas operacionais compatíveis.

Versão 4.x

Versão 4. x do AWS Encryption SDK para o.NET oferece suporte a o.NET 6.0 e o.NET Framework net48 e versões posteriores. Versão 4. x requer o AWS SDK para .NET v3.

Versão 5.x

Versão 5. x do AWS Encryption SDK para o.NET oferece suporte a o.NET 6.0 e o.NET Framework net48 e versões posteriores. Versão 5. x requer a versão 2. x da Material Providers Library (MPL) e do AWS SDK para .NET v4.

O AWS Encryption SDK for .NET exige o SDK para .NET mesmo se você não estiver usando as chaves AWS Key Management Service (AWS KMS). Ele é instalado com o NuGet pacote. No entanto, a menos que você esteja usando AWS KMS chaves, AWS Encryption SDK o.NET não exige AWS credenciais ou interação com nenhum AWS serviço. Conta da AWS Para obter ajuda para configurar uma AWS conta, se necessário, consulteUsar o AWS Encryption SDK por AWS KMS.

Depurando o para o.NET AWS Encryption SDK

O AWS Encryption SDK for.NET não gera nenhum registro. As exceções no AWS Encryption SDK for.NET geram uma mensagem de exceção, mas nenhum rastreamento de pilha.

Para ajudar na depuração, certifique-se de habilitar o login no SDK para .NET. Os registros e mensagens de erro do SDK para .NET podem ajudá-lo a distinguir os erros decorrentes do e os SDK para .NET do para.NET. AWS Encryption SDK 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.)

AWS Encryption SDK para exemplos do.NET

Os exemplos a seguir mostram os padrões básicos de codificação que você usa ao programar com o AWS Encryption SDK para o.NET. Especificamente, você instancia a biblioteca AWS Encryption SDK e os fornecedores de materiais. Em seguida, antes de chamar cada método, você deve instanciar um

objeto que define a entrada para o método. Isso é muito parecido com o padrão de codificação usado no SDK para .NET.

Para ver exemplos que mostram como configurar opções no AWS Encryption SDK, como especificar um conjunto alternativo de algoritmos, limitar chaves de dados criptografadas e usar chaves AWS KMS multirregionais, consulte. Configurando o AWS Encryption SDK

Para obter mais exemplos de programação com o AWS Encryption SDK para.NET, consulte os exemplos no `aws-encryption-sdk-net` diretório do `aws-encryption-sdk` repositório em GitHub.

Criptografando dados no AWS Encryption SDK for.NET

Este exemplo mostra o padrão básico para criptografar dados. Ele criptografa um pequeno arquivo com chaves de dados protegidas por uma chave de AWS KMS empacotamento.

Etapa 1: Instanciar a biblioteca AWS Encryption SDK e a biblioteca dos fornecedores de materiais.

Comece instanciando a biblioteca AWS Encryption SDK e a biblioteca dos fornecedores de materiais. Você usará os métodos do AWS Encryption SDK para criptografar e descriptografar dados. Você usará os métodos na biblioteca de fornecedores de materiais para criar os tokens de autenticação que especificam quais chaves protegem seus dados.

A forma como você instancia a biblioteca AWS Encryption SDK e a biblioteca de fornecedores de materiais difere entre as versões 3. x e 4. x do AWS Encryption SDK para o.NET. Todas as etapas a seguir são as mesmas para ambas as versões 3. x e 4. x do AWS Encryption SDK para o.NET.

Version 3.x

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

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders(
```

Version 4.x

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

Etapa 2: crie um objeto de entrada para o token de autenticação.

Cada método que cria um token de autenticação tem uma classe de objeto de entrada correspondente. Por exemplo, para criar o objeto de entrada para o método `CreateAwsKmsKeyring()`, crie uma instância da classe `CreateAwsKmsKeyringInput`.

Embora a entrada desse token de autenticação não especifique uma chave geradora, a única chave do KMS especificada pelo parâmetro `KmsKeyId` é a chave geradora. Ela gera e criptografa a chave de dados que criptografa os dados.

Esse objeto de entrada requer um AWS KMS cliente para Região da AWS a chave KMS. Para criar um AWS KMS cliente, instancie a `AmazonKeyManagementServiceClient` classe no. SDK para .NET Chamar o construtor `AmazonKeyManagementServiceClient()` sem parâmetros cria um cliente com os valores padrão.

Em um AWS KMS chaveiro usado para criptografar com o.NET, você pode identificar as chaves KMS usando o ID da chave, o ARN da chave, o nome do alias ou o ARN do alias. AWS Encryption SDK Em um AWS KMS chaveiro usado para descriptografia, você deve usar um ARN de chave para identificar cada chave KMS. Se você planeja reutilizar seu token de autenticação de criptografia para descriptografar, use um identificador ARN de chave para todas as chaves KMS.

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

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
```

Etapa 3: criar o token de autenticação.

Para criar o token de autenticação, chame o método do token de autenticação com o objeto de entrada do token de autenticação. Este exemplo usa o método `CreateAwsKmsKeyring()`, que usa apenas uma chave do KMS.

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

Etapa 4: defina um contexto de criptografia.

Um contexto de criptografia é um elemento opcional, mas altamente recomendado, de operações criptográficas no AWS Encryption SDK. Você pode definir um ou mais pares de chave-valor não secretos.

> **Note**
>
> Com a versão 4. No AWS Encryption SDK caso do.NET, você pode exigir um contexto de criptografia em todas as solicitações de criptografia com o contexto de criptografia necessário CMM.

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

Etapa 5: crie o objeto de entrada para a criptografia.

Antes de chamar o método `Encrypt()`, crie uma instância da classe `EncryptInput`.

```
string plaintext = File.ReadAllText("C:\\Documents\\CryptoTest\\TestFile.txt");

// Define the encrypt input
var encryptInput = new EncryptInput
{
    Plaintext = plaintext,
    Keyring = keyring,
    EncryptionContext = encryptionContext
};
```

Etapa 6: criptografe o texto sem formatação.

Use o `Encrypt()` método do AWS Encryption SDK para criptografar o texto simples usando o chaveiro que você definiu.

O `EncryptOutput` que o método `Encrypt()` retorna tem métodos para obter a mensagem criptografada (`Ciphertext`), o contexto de criptografia e o pacote de algoritmos.

```
var encryptOutput = encryptionSdk.Encrypt(encryptInput);
```

Etapa 7: obtenha a mensagem criptografada.

O `Decrypt()` método no AWS Encryption SDK for.NET usa o `Ciphertext` membro da `EncryptOutput` instância.

O membro `Ciphertext` do objeto `EncryptOutput` é a mensagem criptografada, um objeto portátil que inclui dados criptografados, chaves de dados criptografadas e metadados, incluindo o contexto de criptografia. É possível armazenar com segurança a mensagem criptografada por um período prolongado ou enviá-la ao método `Decrypt()` para recuperar o texto sem formatação.

```
var encryptedMessage = encryptOutput.Ciphertext;
```

Descriptografando em modo estrito no para.NET AWS Encryption SDK

As práticas recomendadas indicam que você especifique as chaves usadas para descriptografar dados, uma opção conhecida como modo estrito. O AWS Encryption SDK usa somente as chaves KMS que você especifica em seu chaveiro para descriptografar o texto cifrado. As chaves no token de autenticação de descriptografia devem incluir pelo menos uma das chaves que criptografaram os dados.

Este exemplo mostra o padrão básico de descriptografia no modo estrito com o AWS Encryption SDK para .NET.

Etapa 1: Instanciar a biblioteca AWS Encryption SDK e os fornecedores de materiais.

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

Etapa 2: criar o objeto de entrada para seu token de autenticação.

Para especificar os parâmetros do método do token de autenticação, crie um objeto de entrada. Cada método de chaveiro no AWS Encryption SDK for.NET tem um objeto de entrada correspondente. Como esse exemplo usa o método `CreateAwsKmsKeyring()` para criar o token de autenticação, ele instancia a classe `CreateAwsKmsKeyringInput` para a entrada.

Em um token de autenticação de descriptografia, você deve usar um ARN de chave para identificar chaves do KMS.

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

// Instantiate the keyring input object
var kmsKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
```

Etapa 3: criar o token de autenticação.

Para criar o token de autenticação da descriptografia, este exemplo usa o método `CreateAwsKmsKeyring()` e o objeto de entrada do token de autenticação.

```
var keyring = materialProviders.CreateAwsKmsKeyring(kmsKeyringInput);
```

Etapa 4: crie o objeto de entrada para descriptografar.

Para criar o objeto de entrada para o método `Decrypt()`, instancie a classe `DecryptInput`.

O parâmetro `Ciphertext` do construtor `DecryptInput()` usa o membro `Ciphertext` do objeto `EncryptOutput` que o método `Encrypt()` retornou. A propriedade `Ciphertext` representa a mensagem criptografada, que inclui os dados criptografados, as chaves de dados criptografadas e os metadados que o AWS Encryption SDK necessita para descriptografar a mensagem.

Com a versão 4. No AWS Encryption SDK caso do.NET, você pode usar o *`EncryptionContext`* parâmetro opcional para especificar seu contexto de criptografia no *`Decrypt()`* método.

Use o parâmetro `EncryptionContext` para verificar se o contexto de criptografia usado na criptografia está incluído no contexto de criptografia usado para descriptografar o texto cifrado. AWS Encryption SDK Isso adiciona pares ao contexto de criptografia, incluindo a assinatura digital, se você estiver usando um conjunto de algoritmos com assinatura, como o conjunto de algoritmos padrão.

```
var encryptedMessage = encryptOutput.Ciphertext;
```

```
var decryptInput = new DecryptInput
{
    Ciphertext = encryptedMessage,
    Keyring = keyring,
    EncryptionContext = encryptionContext // OPTIONAL
};
```

Etapa 5: descriptografe o texto cifrado.

```
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

Etapa 6: verifique o contexto de criptografia – versão 3.x

O `Decrypt()` método da versão 3. x do AWS Encryption SDK for.NET não usa um contexto de criptografia. Ele obtém os valores do contexto de criptografia dos metadados na mensagem criptografada. No entanto, antes de retornar ou usar o texto simples, é recomendável verificar se o contexto de criptografia usado para descriptografar o texto cifrado inclui o contexto de criptografia que você forneceu ao criptografar.

Verifique se o contexto de criptografia usado na criptografia está incluído no contexto de criptografia usado para descriptografar o texto cifrado. AWS Encryption SDK Isso adiciona pares ao contexto de criptografia, incluindo a assinatura digital, se você estiver usando um conjunto de algoritmos com assinatura, como o conjunto de algoritmos padrão.

```
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var
 decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

Descriptografando com um chaveiro de descoberta no for.NET AWS Encryption SDK

Em vez de especificar as chaves KMS para a descriptografia, você pode fornecer um token de autenticação de descoberta do AWS KMS , que é um token de autenticação que não especifica

nenhuma chave KMS. Um chaveiro de descoberta permite AWS Encryption SDK descriptografar os dados usando qualquer chave KMS criptografada, desde que o chamador tenha permissão de descriptografia na chave. Para obter as melhores práticas, adicione um filtro de descoberta que limita as chaves KMS que podem ser usadas para aquelas específicas Contas da AWS de uma partição especificada.

O AWS Encryption SDK for.NET fornece um chaveiro de descoberta básico que requer um AWS KMS cliente e um chaveiro de descoberta múltiplo que exige que você especifique um ou mais. Regiões da AWS Tanto o cliente quanto as regiões limitam as chaves do KMS que podem ser usadas para descriptografar a mensagem criptografada. Os objetos de entrada dos dois tokens de autenticação usam o filtro de descoberta recomendado.

O exemplo a seguir mostra o padrão para descriptografar dados com um token de autenticação de descoberta do AWS KMS e um filtro de descoberta.

Etapa 1: Instanciar a biblioteca AWS Encryption SDK e a biblioteca dos fornecedores de materiais.

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

Etapa 2: criar o objeto de entrada para o token de autenticação.

Para especificar os parâmetros do método do token de autenticação, crie um objeto de entrada. Cada método de chaveiro no AWS Encryption SDK for.NET tem um objeto de entrada correspondente. Como esse exemplo usa o método `CreateAwsKmsDiscoveryKeyring()` para criar o token de autenticação, ele instancia a classe `CreateAwsKmsDiscoveryKeyringInput` para a entrada.

```
List<string> accounts = new List<string> { "111122223333" };

var discoveryKeyringInput = new CreateAwsKmsDiscoveryKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    DiscoveryFilter = new DiscoveryFilter()
    {
        AccountIds = accounts,
        Partition = "aws"
    }
};
```

Etapa 3: criar o token de autenticação.

Para criar o token de autenticação da descriptografia, este exemplo usa o método `CreateAwsKmsDiscoveryKeyring()` e o objeto de entrada do token de autenticação.

```
var discoveryKeyring =
 materialProviders.CreateAwsKmsDiscoveryKeyring(discoveryKeyringInput);
```

Etapa 4: crie o objeto de entrada para descriptografar.

Para criar o objeto de entrada para o método `Decrypt()`, instancie a classe `DecryptInput`. O valor do parâmetro `Ciphertext` é o membro `Ciphertext` do objeto `EncryptOutput` que o método `Encrypt()` retorna.

Com a versão 4. No AWS Encryption SDK caso do.NET, você pode usar o *`EncryptionContext`* parâmetro opcional para especificar seu contexto de criptografia no *`Decrypt()`* método.

Use o parâmetro `EncryptionContext` para verificar se o contexto de criptografia usado na criptografia está incluído no contexto de criptografia usado para descriptografar o texto cifrado. AWS Encryption SDK Isso adiciona pares ao contexto de criptografia, incluindo a assinatura digital, se você estiver usando um conjunto de algoritmos com assinatura, como o conjunto de algoritmos padrão.

```
var ciphertext = encryptOutput.Ciphertext;

var decryptInput = new DecryptInput
{
    Ciphertext = ciphertext,
    Keyring = discoveryKeyring,
    EncryptionContext = encryptionContext // OPTIONAL

};
var decryptOutput = encryptionSdk.Decrypt(decryptInput);
```

Etapa 5: verificar o contexto de criptografia - versão 3.x

O `Decrypt()` método da versão 3. x do AWS Encryption SDK for.NET não usa um contexto de criptografia`Decrypt()`. Ele obtém os valores do contexto de criptografia dos metadados na mensagem criptografada. No entanto, antes de retornar ou usar o texto simples, é recomendável

verificar se o contexto de criptografia usado para descriptografar o texto cifrado inclui o contexto de criptografia que você forneceu ao criptografar.

Verifique se o contexto de criptografia usado na criptografia foi incluído no contexto de criptografia usado para descriptografar o texto cifrado. AWS Encryption SDK Isso adiciona pares ao contexto de criptografia, incluindo a assinatura digital, se você estiver usando um conjunto de algoritmos com assinatura, como o conjunto de algoritmos padrão.

```
// Verify the encryption context
string contextKey = "purpose";
string contextValue = "test";

if (!decryptOutput.EncryptionContext.TryGetValue(contextKey, out var
 decryptContextValue)
    || !decryptContextValue.Equals(contextValue))
{
    throw new Exception("Encryption context does not match expected values");
}
```

AWS Encryption SDK para Go

Este tópico explica como instalar e usar o AWS Encryption SDK for Go. Para obter detalhes sobre a programação com o AWS Encryption SDK for Go, consulte o diretório go do repositório aws-encryption-sdk em. GitHub

O AWS Encryption SDK for Go difere de algumas das outras implementações da linguagem de programação das seguintes maneiras: AWS Encryption SDK

- Não há suporte para armazenamento em cache ??? de chaves de dados. No entanto, o AWS Encryption SDK for Go suporta o chaveiro AWS KMS hierárquico, uma solução alternativa de armazenamento em cache de materiais criptográficos.
- Não há suporte para streaming de dados

O AWS Encryption SDK for Go inclui todos os recursos de segurança introduzidos nas versões 2.0. x e versões posteriores de outras implementações linguísticas do AWS Encryption SDK. No entanto, se você estiver usando o for Go AWS Encryption SDK para descriptografar dados que foram criptografados por um pré-2.0. versão x, implementação em outro idioma do AWS Encryption SDK,

talvez seja necessário ajustar sua política de compromisso. Para obter detalhes, consulte Como definir sua política de compromisso.

O AWS Encryption SDK for Go é um produto da AWS Encryption SDK in Dafny, uma linguagem de verificação formal na qual você escreve especificações, o código para implementá-las e as provas para testá-las. O resultado é uma biblioteca que implementa os atributos do AWS Encryption SDK em uma estrutura que garante a correção funcional.

Saiba mais

- Para ver exemplos de como configurar opções no AWS Encryption SDK, como especificar um conjunto de algoritmos alternativo, limitar chaves de dados criptografadas e usar chaves AWS KMS multirregionais, consulte. Configurando o AWS Encryption SDK
- Para ver exemplos que mostram como configurar e usar o AWS Encryption SDK for Go, consulte os exemplos de Go no repositório aws-encryption-sdk em. GitHub

Tópicos

- Pré-requisitos
- Instalação
- AWS Encryption SDK para código de exemplo em Go

Pré-requisitos

Antes de instalar o AWS Encryption SDK for Go, verifique se você tem os seguintes pré-requisitos.

Uma versão compatível do Go

O Go 1.23 ou posterior é exigido pelo AWS Encryption SDK for Go.

Para obter mais informações sobre como baixar e instalar o Go, consulte Instalação do Go.

Instalação

Instale a versão mais recente do AWS Encryption SDK for Go. Para obter detalhes sobre como instalar e criar o AWS Encryption SDK for Go, consulte o diretório README.md in the go do repositório aws-encryption-sdk em. GitHub

Para instalar a versão mais recente

- Instale o AWS Encryption SDK for Go

```
go get github.com/aws/aws-encryption-sdk/releases/go/encryption-sdk@latest
```

- Instale a Biblioteca de Provedores de Material Criptográfico (MPL)

```
go get github.com/aws/aws-cryptographic-material-providers-library/releases/go/mpl
```

AWS Encryption SDK para código de exemplo em Go

Os exemplos a seguir mostram os padrões básicos de codificação que você usa ao programar com o AWS Encryption SDK for Go. Especificamente, você instancia a biblioteca AWS Encryption SDK e a biblioteca de fornecedores de materiais. Então, antes de chamar cada método, você instancia o objeto que define a entrada para o método.

Para ver exemplos que mostram como configurar opções no AWS Encryption SDK, como especificar um conjunto de algoritmos alternativo e limitar as chaves de dados criptografadas, consulte os exemplos de Go no repositório aws-encryption-sdk em. GitHub

Criptografando e descriptografando dados no AWS Encryption SDK para Go

Este exemplo mostra o padrão básico para criptografar e descriptografar dados. Ele criptografa uma pequena quantidade de dados com chaves de dados que são protegidas por uma chave de AWS KMS empacotamento.

Etapa 1: Instancie o. AWS Encryption SDK

Você usará os métodos do AWS Encryption SDK para criptografar e descriptografar dados.

```
import (
    "context"

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

```
    esdktypes "github.com/aws/aws-encryption-sdk/
awscryptographyencryptionsdksmithygeneratedtypes"
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/kms"
)

encryptionClient, err := client.NewClient(esdktypes.AwsEncryptionSdkConfig{})
if err != nil {
    panic(err)
}
```

Etapa 2: Crie um AWS KMS cliente.

```
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})
```

Opcional: crie seu contexto de criptografia.

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

Etapa 3: Instancie a biblioteca de fornecedores de materiais.

Você usará os métodos na biblioteca de fornecedores de materiais para criar os tokens de autenticação que especificam quais chaves protegem seus dados.

```
matProv, err := mpl.NewClient(mpltypes.MaterialProvidersConfig{})
if err != nil {
    panic(err)
}
```

Etapa 4: crie um AWS KMS chaveiro.

Para criar o token de autenticação, chame o método do token de autenticação com o objeto de entrada do token de autenticação. Este exemplo usa o `CreateAwsKmsKeyring` método e especifica uma chave KMS. A `kmsKeyId` variável representa o ARN da chave KMS que você fornece.

```
awsKmsKeyringInput := mpltypes.CreateAwsKmsKeyringInput{
    KmsClient: kmsClient,
    KmsKeyId:  kmsKeyId,
}
awsKmsKeyring, err := matProv.CreateAwsKmsKeyring(context.Background(),
 awsKmsKeyringInput)
if err != nil {
    panic(err)
}
```

Etapa 5: Criptografe o texto sem formatação.

```
res, err := encryptionClient.Encrypt(context.Background(), esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}

ciphertext := res.Ciphertext
```

Etapa 6: Descriptografe seus dados criptografados usando o mesmo chaveiro usado na criptografia.

```
decryptOutput, err := encryptionClient.Decrypt(context.Background(),
 esdktypes.DecryptInput{
    Ciphertext:        ciphertext,
    // Provide the encryption context that was supplied to the encrypt method
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}
```

```
decrypted := decryptOutput.Plaintext
```

AWS Encryption SDK for Java

Este tópico explica como instalar e usar o AWS Encryption SDK for Java. Para obter detalhes sobre a programação com o AWS Encryption SDK for Java, consulte o aws-encryption-sdk-javarepositório em GitHub. Para obter a documentação da API, consulte o Javadoc para AWS Encryption SDK for Java.

Tópicos

- Pré-requisitos
- Instalação
- AWS Encryption SDK for Java exemplos

Pré-requisitos

Antes de instalar o AWS Encryption SDK for Java, verifique se você tem os seguintes pré-requisitos.

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).

Bouncy Castle

AWS Encryption SDK for Java Isso requer o Bouncy Castle.

- AWS Encryption SDK for Java as versões 1.6.1 e posteriores usam o Bouncy Castle para serializar e desserializar objetos criptográficos. Você pode usar o Bouncy Castle ou o Bouncy Castle FIPS para atender a esse requisito. Para obter ajuda na instalação e configuração do Bouncy Castle FIPS, consulte a documentação do BC FIPS, especialmente os Guias do Usuário e a Política de Segurança. PDFs
- As versões anteriores do AWS Encryption SDK for Java usam a API de criptografia do Bouncy Castle para Java. Este requisito só é atendido por não FIPS Bouncy Castle.

Se você não tiver o Bouncy Castle, acesse Baixar Bouncy Castle for Java para baixar o arquivo do provedor que corresponde ao seu JDK. Você também pode usar o Apache Maven para obter o artefato para o provedor padrão do Bouncy Castle (bcprov-ext-jdk15on) ou o artefato para o Bouncy Castle FIPS (bc-fips).

AWS SDK para Java

Versão 3. x of the AWS Encryption SDK for Java requer o AWS SDK for Java 2.x, mesmo se você não usar AWS KMS chaveiros.

Versão 2. x ou anterior do AWS Encryption SDK for Java não requer AWS SDK para Java o. No entanto, AWS SDK para Java é necessário usar AWS Key Management Service(AWS KMS) como provedor de chave mestra. A partir da AWS Encryption SDK for Java versão 2.4.0, o AWS Encryption SDK for Java suporta as versões 1.x e 2.x do. AWS SDK para Java AWS Encryption SDK os códigos para AWS SDK para Java 1.x e 2.x são interoperáveis. Por exemplo, você pode criptografar dados com AWS Encryption SDK código compatível com AWS SDK para Java 1.x e descriptografá-los usando código compatível AWS SDK for Java 2.x (ou vice-versa). As versões AWS Encryption SDK for Java anteriores à 2.4.0 suportam apenas AWS SDK para Java 1.x. Para obter informações sobre como atualizar sua versão do AWS Encryption SDK, consulteMigrando seu AWS Encryption SDK.

Ao atualizar seu AWS Encryption SDK for Java código de AWS SDK para Java 1.x para AWS SDK for Java 2.x, substitua as referências à `AWSKMSinterface` em AWS SDK para Java 1.x por referências à `KmsClientinterface` em. AWS SDK for Java 2.x O AWS Encryption SDK for Java não suporta a `KmsAsyncClientinterface`. Além disso, atualize seu código para usar os objetos relacionados ao AWS KMS no namespace `kmssdkv2`, em vez do namespace `kms`.

Para instalar o AWS SDK para Java, use 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 AWS KMS módulo na AWS SDK para Java versão 1.x, siga as instruções para especificar módulos específicos e defina o. `artifactId` `aws-java-sdk-kms`
- Para criar uma dependência somente para o AWS KMS módulo na AWS SDK para Java versão 2.x, siga as instruções para especificar módulos específicos. Defina o `groupId` como `software.amazon.awssdk` e `artifactId` como `kms`.

Para ver mais mudanças, consulte O que há de diferente entre a AWS SDK para Java versão 1.x e a 2.x no Guia do AWS SDK for Java 2.x desenvolvedor.

Os exemplos de Java no Guia do AWS Encryption SDK Desenvolvedor usam AWS SDK for Java 2.x o.

Instalação

Instalar a versão mais recente do AWS Encryption SDK for Java.

> **Note**
>
> Todas as versões AWS Encryption SDK for Java anteriores à 2.0.0 estão em end-of-supportfase.
> Você pode atualizar com segurança a partir da versão 2.0.x e posteriores até a versão mais recente do AWS Encryption SDK for Java sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.0.x não são compatíveis com versões anteriores. Para atualizar a partir de versões anteriores à 1.7.x até a versão 2.0. x e posteriores, primeiro será necessário atualizar para a versão 1.x mais recente do AWS Encryption SDK. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

Você pode instalar o AWS Encryption SDK for Java das seguintes maneiras.

Manualmente

Para instalar o AWS Encryption SDK for Java, clone ou baixe o aws-encryption-sdk-java GitHubrepositório.

Uso do Apache Maven

O AWS Encryption SDK for Java está disponível por meio do Apache Maven com a seguinte definição de dependência.

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-encryption-sdk-java</artifactId>
  <version>3.0.0</version>
</dependency>
```

Depois de instalar o SDK, comece examinando o exemplo de código Java neste guia e o Javadoc ativado. GitHub

AWS Encryption SDK for Java exemplos

Os exemplos a seguir mostram como usar o para AWS Encryption SDK for Java criptografar e descriptografar dados. Esses exemplos mostram como usar a versão 3. x e posterior do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java requer AWS SDK for Java 2.x o. Versão 3. x do AWS Encryption SDK for Java substitui os fornecedores de chaves mestras por chaveiros. Para exemplos que usam versões anteriores, encontre sua versão na lista de lançamentos do aws-encryption-sdk-javarepositório em GitHub.

Tópicos

- Criptografar e descriptografar strings
- Criptografar e descriptografar streams de bytes
- Criptografando e descriptografando fluxos de bytes com um chaveiro múltiplo

Criptografar e descriptografar strings

O exemplo a seguir mostra como usar a versão 3. x do AWS Encryption SDK for Java para criptografar e descriptografar strings. Antes de usar a string, converta-a em uma matriz de bytes.

Este exemplo usa um AWS KMS chaveiro. Ao criptografar com um AWS KMS chaveiro, você pode usar um ID de chave, ARN de chave, nome de alias ou ARN de alias para identificar as chaves KMS. Ao descriptografar, você deve usar um ARN de chave para identificar as chaves KMS.

Quando você chama o método `encryptData()`, ele retorna uma mensagem criptografada (`CryptoResult`) que inclui o texto cifrado, as chaves de dados criptografadas e o contexto de criptografia. Quando você chama `getResult` no objeto `CryptoResult`, ele retorna uma versão de cadeia codificada em base 64 da mensagem criptografada que você pode passar para o método `decryptData()`.

Da mesma forma, quando você chama`decryptData()`, o `CryptoResult` objeto que ele retorna contém a mensagem de texto sem formatação e um AWS KMS key ID. Antes que seu aplicativo retorne o texto sem formatação, verifique se o AWS KMS key ID e o contexto de criptografia na mensagem criptografada são os que você espera.

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import
 software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;

import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.Collections;
import java.util.Map;

/**
 * Encrypts and then decrypts data using an AWS KMS Keyring.
 *
 * <p>Arguments:
 *
 * <ol>
 *   <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS
 customer master
 *       key (CMK), see 'Viewing Keys' at
 *       http://docs.aws.amazon.com/kms/latest/developerguide/viewing-keys.html
 * </ol>
 */
public class BasicEncryptionKeyringExample {

  private static final byte[] EXAMPLE_DATA = "Hello
 World".getBytes(StandardCharsets.UTF_8);

  public static void main(final String[] args) {
    final String keyArn = args[0];

    encryptAndDecryptWithKeyring(keyArn);
  }

  public static void encryptAndDecryptWithKeyring(final String keyArn) {
    // 1. Instantiate the SDK
    // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
 commitment policy,
```

```
    // which means this client only encrypts using committing algorithm suites and
 enforces
    // that the client will only decrypt encrypted messages that were created with a
 committing
    // algorithm suite.
    // This is the default commitment policy if you build the client with
    // `AwsCrypto.builder().build()`
    // or `AwsCrypto.standard()`.
    final AwsCrypto crypto =
        AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();

    // 2. Create the AWS KMS keyring.
    // This example creates a multi keyring, which automatically creates the KMS
 client.
    final MaterialProviders materialProviders =
        MaterialProviders.builder()
            .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
            .build();
    final CreateAwsKmsMultiKeyringInput keyringInput =
        CreateAwsKmsMultiKeyringInput.builder().generator(keyArn).build();
    final IKeyring kmsKeyring =
 materialProviders.CreateAwsKmsMultiKeyring(keyringInput);

    // 3. Create an encryption context
    // We recommend using an encryption context whenever possible
    // to protect integrity. This sample uses placeholder values.
    // For more information see:
    // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-Integrity-
of-Your-Encrypted-Data-by-Using-AWS-Key-Management
    final Map<String, String> encryptionContext =
        Collections.singletonMap("ExampleContextKey", "ExampleContextValue");

    // 4. Encrypt the data
    final CryptoResult<byte[], ?> encryptResult =
        crypto.encryptData(kmsKeyring, EXAMPLE_DATA, encryptionContext);
    final byte[] ciphertext = encryptResult.getResult();

    // 5. Decrypt the data
    final CryptoResult<byte[], ?> decryptResult =
        crypto.decryptData(
            kmsKeyring,
            ciphertext,
```

```
            // Verify that the encryption context in the result contains the
            // encryption context supplied to the encryptData method
            encryptionContext);

    // 6. Verify that the decrypted plaintext matches the original plaintext
    assert Arrays.equals(decryptResult.getResult(), EXAMPLE_DATA);
  }
}
```

Criptografar e descriptografar streams de bytes

O exemplo a seguir mostra como usar o para AWS Encryption SDK criptografar e descriptografar fluxos de bytes.

Este exemplo usa um chaveiro AES bruto.

Ao criptografar, o método `AwsCrypto.builder() .withEncryptionAlgorithm()` é usado para especificar um conjunto de algoritmos sem assinaturas digitais.
Ao descriptografar, para garantir que o texto cifrado não esteja assinado, este exemplo usa o método `createUnsignedMessageDecryptingStream()`. O `createUnsignedMessageDecryptingStream()` método falhará se encontrar um texto cifrado com uma assinatura digital.

Se você estiver criptografando com o conjunto de algoritmos padrão, que inclui assinaturas digitais, use o método `createDecryptingStream()` em seu lugar, conforme mostrado no próximo exemplo.

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoAlgorithm;
import com.amazonaws.encryptionsdk.CryptoInputStream;
import com.amazonaws.encryptionsdk.jce.JceMasterKey;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import software.amazon.cryptography.materialproviders.model.AesWrappingAlg;
import software.amazon.cryptography.materialproviders.model.CreateRawAesKeyringInput;
```

```
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.security.SecureRandom;
import java.util.Collections;
import java.util.Map;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;


/**
 * <p>
 * Encrypts and then decrypts a file under a random key.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 *
 * <p>
 * This program demonstrates using a standard Java {@link SecretKey} object as a {@link
 IKeyring} to
 * encrypt and decrypt streaming data.
 */
public class FileStreamingKeyringExample {
    private static String srcFile;

    public static void main(String[] args) throws IOException {
        srcFile = args[0];

        // In this example, we generate a random key. In practice,
        // you would get a key from an existing store
        SecretKey cryptoKey = retrieveEncryptionKey();

        // Create a Raw Aes Keyring using the random key and an AES-GCM encryption
 algorithm
        final MaterialProviders materialProviders = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
```

```
        final CreateRawAesKeyringInput keyringInput =
CreateRawAesKeyringInput.builder()
                .wrappingKey(ByteBuffer.wrap(cryptoKey.getEncoded()))
                .keyNamespace("Example")
                .keyName("RandomKey")
                .wrappingAlg(AesWrappingAlg.ALG_AES128_GCM_IV12_TAG16)
                .build();
        IKeyring keyring = materialProviders.CreateRawAesKeyring(keyringInput);

        // Instantiate the SDK.
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
        // which means this client only encrypts using committing algorithm suites and
enforces
        // that the client will only decrypt encrypted messages that were created with
a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        // This example encrypts with an algorithm suite that doesn't include signing
for faster decryption,
        // since this use case assumes that the contexts that encrypt and decrypt are
equally trusted.
        final AwsCrypto crypto = AwsCrypto.builder()
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)

.withEncryptionAlgorithm(CryptoAlgorithm.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY)
                .build();

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

        // Because the file might be too large to load into memory, we stream the data,
instead of
        //loading it all at once.
        FileInputStream in = new FileInputStream(srcFile);
        CryptoInputStream<JceMasterKey> encryptingStream =
crypto.createEncryptingStream(keyring, in, context);

        FileOutputStream out = new FileOutputStream(srcFile + ".encrypted");
        IOUtils.copy(encryptingStream, out);
        encryptingStream.close();
```

```
        out.close();

        // Decrypt the file. Verify the encryption context before returning the
 plaintext.
        // Since the data was encrypted using an unsigned algorithm suite, use the
 recommended
        // createUnsignedMessageDecryptingStream method, which only accepts unsigned
 messages.
        in = new FileInputStream(srcFile + ".encrypted");
        CryptoInputStream<JceMasterKey> decryptingStream =
 crypto.createUnsignedMessageDecryptingStream(keyring, in);
        // Does it contain the expected encryption context?
        if
 (!"FileStreaming".equals(decryptingStream.getCryptoResult().getEncryptionContext().get("Exampl
 {
            throw new IllegalStateException("Bad encryption context");
        }

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

    /**
     * In practice, this key would be saved in a secure location.
     * For this demo, we generate a new random key for each operation.
     */
    private static SecretKey retrieveEncryptionKey() {
        SecureRandom rnd = new SecureRandom();
        byte[] rawKey = new byte[16]; // 128 bits
        rnd.nextBytes(rawKey);
        return new SecretKeySpec(rawKey, "AES");
    }
}
```

Criptografando e descriptografando fluxos de bytes com um chaveiro múltiplo

O exemplo a seguir mostra como usar o AWS Encryption SDK com um chaveiro múltiplo. 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. Este exemplo usa um AWS KMS chaveiro e um chaveiro RSA bruto como chaveiros secundários.

Este exemplo criptografa com o pacote de algoritmos padrão, que inclui uma assinatura digital. Durante o streaming, ele AWS Encryption SDK libera texto sem formatação após as verificações de integridade, mas antes de verificar a assinatura digital. Para evitar o uso do texto simples até que a assinatura seja verificada, este exemplo armazena o texto simples em buffer e o grava no disco somente após a conclusão da descriptografia e da verificação.

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.keyrings;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoOutputStream;
import com.amazonaws.util.IOUtils;
import software.amazon.cryptography.materialproviders.IKeyring;
import software.amazon.cryptography.materialproviders.MaterialProviders;
import
 software.amazon.cryptography.materialproviders.model.CreateAwsKmsMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateMultiKeyringInput;
import software.amazon.cryptography.materialproviders.model.CreateRawRsaKeyringInput;
import software.amazon.cryptography.materialproviders.model.MaterialProvidersConfig;
import software.amazon.cryptography.materialproviders.model.PaddingScheme;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.nio.ByteBuffer;
import java.security.GeneralSecurityException;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.util.Collections;

/**
 * <p>
 * Encrypts a file using both AWS KMS Key and an asymmetric key pair.
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>Key ARN: For help finding the Amazon Resource Name (ARN) of your AWS KMS key,
```

```
 *   see 'Viewing Keys' at http://docs.aws.amazon.com/kms/latest/developerguide/
viewing-keys.html
 *
 * <li>Name of file containing plaintext data to encrypt
 * </ol>
 * <p>
 * You might use AWS Key Management Service (AWS KMS) for most encryption and
 decryption operations, but
 * still want the option of decrypting your data offline independently of AWS KMS. This
 sample
 * demonstrates one way to do this.
 * <p>
 * The sample encrypts data under both an AWS KMS key and an "escrowed" RSA key pair
 * so that either key alone can decrypt it. You might commonly use the AWS KMS key for
 decryption. However,
 * at any time, you can use the private RSA key to decrypt the ciphertext independent
 of AWS KMS.
 * <p>
 * This sample uses the RawRsaKeyring to generate a RSA public-private key pair
 * and saves the key pair in memory. In practice, you would store the private key in a
 secure offline
 * location, such as an offline HSM, and distribute the public key to your development
 team.
 */
public class EscrowedEncryptKeyringExample {
    private static ByteBuffer publicEscrowKey;
    private static ByteBuffer privateEscrowKey;

    public static void main(final String[] args) throws Exception {
        // This sample generates a new random key for each operation.
        // In practice, you would distribute the public key and save the private key in
 secure
        // storage.
        generateEscrowKeyPair();

        final String kmsArn = args[0];
        final String fileName = args[1];

        standardEncrypt(kmsArn, fileName);
        standardDecrypt(kmsArn, fileName);

        escrowDecrypt(fileName);
    }
```

```
   private static void standardEncrypt(final String kmsArn, final String fileName)
throws Exception {
       // Encrypt with the KMS key and the escrowed public key
       // 1. Instantiate the SDK
       // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
       // which means this client only encrypts using committing algorithm suites and
enforces
       // that the client will only decrypt encrypted messages that were created with
a committing
       // algorithm suite.
       // This is the default commitment policy if you build the client with
       // `AwsCrypto.builder().build()`
       // or `AwsCrypto.standard()`.
       final AwsCrypto crypto = AwsCrypto.builder()
               .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
               .build();

       // 2. Create the AWS KMS keyring.
       // This example creates a multi keyring, which automatically creates the KMS
client.
       final MaterialProviders matProv = MaterialProviders.builder()
               .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
               .build();
       final CreateAwsKmsMultiKeyringInput keyringInput =
CreateAwsKmsMultiKeyringInput.builder()
               .generator(kmsArn)
               .build();
       IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

       // 3. Create the Raw Rsa Keyring with Public Key.
       final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
               .keyName("Escrow")
               .keyNamespace("Escrow")
               .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
               .publicKey(publicEscrowKey)
               .build();
       IKeyring rsaPublicKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);

       // 4. Create the multi-keyring.
       final CreateMultiKeyringInput createMultiKeyringInput =
CreateMultiKeyringInput.builder()
```

```
                .generator(kmsKeyring)
                .childKeyrings(Collections.singletonList(rsaPublicKeyring))
                .build();
        IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

        // 5. Encrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName);
        final FileOutputStream out = new FileOutputStream(fileName + ".encrypted");
        final CryptoOutputStream<?> encryptingStream =
crypto.createEncryptingStream(multiKeyring, out);

        IOUtils.copy(in, encryptingStream);
        in.close();
        encryptingStream.close();
    }

    private static void standardDecrypt(final String kmsArn, final String fileName)
throws Exception {
        // Decrypt with the AWS KMS key and the escrow public key.

        // 1. Instantiate the SDK.
        // This builds the AwsCrypto client with the RequireEncryptRequireDecrypt
commitment policy,
        // which means this client only encrypts using committing algorithm suites and
enforces
        // that the client will only decrypt encrypted messages that were created with
a committing
        // algorithm suite.
        // This is the default commitment policy if you build the client with
        // `AwsCrypto.builder().build()`
        // or `AwsCrypto.standard()`.
        final AwsCrypto crypto = AwsCrypto.builder()
                .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
                .build();

        // 2. Create the AWS KMS keyring.
        // This example creates a multi keyring, which automatically creates the KMS
client.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
```

```
        final CreateAwsKmsMultiKeyringInput keyringInput =
CreateAwsKmsMultiKeyringInput.builder()
                .generator(kmsArn)
                .build();
        IKeyring kmsKeyring = matProv.CreateAwsKmsMultiKeyring(keyringInput);

        // 3. Create the Raw Rsa Keyring with Public Key.
        final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .build();
        IKeyring rsaPublicKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);

        // 4. Create the multi-keyring.
        final CreateMultiKeyringInput createMultiKeyringInput =
CreateMultiKeyringInput.builder()
                .generator(kmsKeyring)
                .childKeyrings(Collections.singletonList(rsaPublicKeyring))
                .build();
        IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);

        // 5. Decrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".decrypted");
        // Since we are using a signing algorithm suite, we avoid streaming decryption
directly to the output file,
        // to ensure that the trailing signature is verified before writing any
untrusted plaintext to disk.
        final ByteArrayOutputStream plaintextBuffer = new ByteArrayOutputStream();
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(multiKeyring, plaintextBuffer);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();
        final ByteArrayInputStream plaintextReader = new
ByteArrayInputStream(plaintextBuffer.toByteArray());
        IOUtils.copy(plaintextReader, out);
```

```
        out.close();
    }

    private static void escrowDecrypt(final String fileName) throws Exception {
        // You can decrypt the stream using only the private key.
        // This method does not call AWS KMS.

        // 1. Instantiate the SDK
        final AwsCrypto crypto = AwsCrypto.standard();

        // 2. Create the Raw Rsa Keyring with Private Key.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateRawRsaKeyringInput encryptingKeyringInput =
CreateRawRsaKeyringInput.builder()
                .keyName("Escrow")
                .keyNamespace("Escrow")
                .paddingScheme(PaddingScheme.OAEP_SHA512_MGF1)
                .publicKey(publicEscrowKey)
                .privateKey(privateEscrowKey)
                .build();
        IKeyring escrowPrivateKeyring =
matProv.CreateRawRsaKeyring(encryptingKeyringInput);


        // 3. Decrypt the file
        // To simplify this code example, we omit the encryption context. Production
code should always
        // use an encryption context.
        final FileInputStream in = new FileInputStream(fileName + ".encrypted");
        final FileOutputStream out = new FileOutputStream(fileName + ".deescrowed");
        final CryptoOutputStream<?> decryptingStream =
crypto.createDecryptingStream(escrowPrivateKeyring, out);
        IOUtils.copy(in, decryptingStream);
        in.close();
        decryptingStream.close();

    }

    private static void generateEscrowKeyPair() throws GeneralSecurityException {
        final KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA");
        kg.initialize(4096); // Escrow keys should be very strong
        final KeyPair keyPair = kg.generateKeyPair();
```

```
        publicEscrowKey = RawRsaKeyringExample.getPEMPublicKey(keyPair.getPublic());
        privateEscrowKey = RawRsaKeyringExample.getPEMPrivateKey(keyPair.getPrivate());

    }
}
```

AWS Encryption SDK para JavaScript

O foi AWS Encryption SDK para JavaScript projetado para fornecer uma biblioteca de criptografia do lado do cliente para desenvolvedores que estão escrevendo aplicativos de navegador da Web JavaScript ou aplicativos de servidor Web em Node.js.

Como todas as implementações do AWS Encryption SDK, o AWS Encryption SDK para JavaScript oferece recursos avançados de proteção de dados. Os recursos incluem criptografia de envelope, AAD (additional authenticated data - dados autenticados adicionais) e pacotes de algoritmos de chave simétrica segura e autenticada, como o AES-GCM de 256 bits com derivação de chave e assinatura.

Todas as implementações específicas do AWS Encryption SDK idioma foram projetadas para serem interoperáveis, sujeitas às restrições da linguagem. Para obter detalhes sobre as restrições de linguagem para JavaScript, consulte. the section called “Compatibilidade”

Saiba mais

- Para obter detalhes sobre a programação com o AWS Encryption SDK para JavaScript, consulte o aws-encryption-sdk-javascriptrepositório em GitHub.
- Para exemplos de programação, consulte os the section called “Exemplos” módulos example-browser e example-node no repositório. aws-encryption-sdk-javascript
- Para ver um exemplo real do uso do AWS Encryption SDK para JavaScript para criptografar dados em um aplicativo da Web, consulte Como habilitar a criptografia em um navegador com o AWS Encryption SDK para JavaScript e o Node.js no blog de segurança. AWS

Tópicos

- Compatibilidade do AWS Encryption SDK para JavaScript
- Instalando o AWS Encryption SDK para JavaScript
- Módulos no AWS Encryption SDK para JavaScript

- AWS Encryption SDK para JavaScript exemplos

Compatibilidade do AWS Encryption SDK para JavaScript

O foi AWS Encryption SDK para JavaScript projetado para ser interoperável com outras implementações de linguagem do. AWS Encryption SDKNa maioria dos casos, você pode criptografar dados com o AWS Encryption SDK para JavaScript e descriptografá-los com qualquer outra implementação de linguagem, incluindo a AWS Encryption SDK interface de linha de comando. E você pode usar o AWS Encryption SDK para JavaScript para descriptografar mensagens criptografadas produzidas por outras implementações de linguagem do. AWS Encryption SDK

No entanto, ao usar o AWS Encryption SDK para JavaScript, você precisa estar ciente de alguns problemas de compatibilidade na implementação da JavaScript linguagem e nos navegadores da Web.

Além disso, ao usar implementações de linguagem diferentes, configure provedores de chaves mestras, chaves mestras e tokens de autenticação compatíveis. Para obter detalhes, consulte Compatibilidade dos tokens de autenticação.

AWS Encryption SDK para JavaScript compatibilidade

A JavaScript implementação do AWS Encryption SDK difere das implementações de outras linguagens das seguintes maneiras:

- A operação de criptografia do AWS Encryption SDK para JavaScript não retorna texto cifrado sem moldura. No entanto, o AWS Encryption SDK para JavaScript decifrará o texto cifrado emoldurado e não emoldurado retornado por outras implementações de linguagem do. AWS Encryption SDK
- Começando com o Node.js versão 12.9.0, o Node.js é compatível com as seguintes opções de empacotamento de chave RSA:
  - OAEP com SHA1, SHA256, ou SHA384 SHA512
  - OAEP com e com SHA1 MGF1 SHA1
  - PKCS1v15
- Antes da versão 12.9.0, o Node.js era compatível apenas com as seguintes opções de empacotamento de chave RSA:
  - OAEP com e com SHA1 MGF1 SHA1
  - PKCS1v15

Compatibilidade do navegador

Alguns navegadores da Web não são compatíveis com operações de criptografia básicas exigidas pelo AWS Encryption SDK para JavaScript . Você pode compensar algumas das operações ausentes configurando um substituto para a WebCrypto API que o navegador implementa.

Limitações de navegador da Web

As seguintes limitações são comuns a todos os navegadores da Web:

- A WebCrypto API não oferece suporte ao encapsulamento de PKCS1v15 chaves.
- Os navegadores não são compatíveis com chaves de 192 bits.

Operações de criptografia necessárias

AWS Encryption SDK para JavaScript Isso requer as seguintes operações em navegadores da web. Se um navegador não for compatível com estas operações, ele será incompatível com o AWS Encryption SDK para JavaScript.

- O navegador deve incluir `crypto.getRandomValues()`, que é um método para gerar valores de criptografia aleatórios. Para obter informações sobre as versões do navegador da Web compatíveis`crypto.getRandomValues()`, consulte Posso usar criptografia. getRandomValues()? .

Fallback necessário

O AWS Encryption SDK para JavaScript requer as seguintes bibliotecas e operações em navegadores da web. Se você oferecer suporte a um navegador da Web que não atenda a esses requisitos, deverá configurar um fallback. Caso contrário, as tentativas de usar o AWS Encryption SDK para JavaScript com o navegador falharão.

- A WebCrypto API, que executa operações criptográficas básicas em aplicativos da web, não está disponível para todos os navegadores. Para obter informações sobre as versões do navegador da Web compatíveis com a criptografia da Web, consulte Posso usar criptografia da Web?.
- As versões modernas do navegador Safari não oferecem suporte à criptografia AES-GCM de zero bytes, o que é necessário. AWS Encryption SDK Se o navegador implementa a WebCrypto API, mas não consegue usar o AES-GCM para criptografar zero bytes, ele AWS Encryption SDK para JavaScript usa a biblioteca de fallback somente para criptografia de zero bytes. Ele usa a WebCrypto API para todas as outras operações.

Para configurar um fallback para qualquer limitação, adicione as seguintes instruções ao seu código. Na função configureFallback especifique uma biblioteca que seja compatível com os recursos ausentes. O exemplo a seguir usa a Microsoft Research JavaScript Cryptography Library (`msrcrypto`), mas você pode substituí-la por uma biblioteca compatível. Para obter um exemplo completo, consulte fallback.ts.

```
import { configureFallback } from '@aws-crypto/client-browser'
configureFallback(msrCrypto)
```

Instalando o AWS Encryption SDK para JavaScript

O AWS Encryption SDK para JavaScript consiste em uma coleção de módulos interdependentes. Vários dos módulos são apenas coleções de módulos projetados para funcionar em conjunto. Alguns módulos são projetados para funcionar de forma independente. Alguns módulos são necessários para todas as implementações; alguns outros são necessários apenas para casos especiais. Para obter informações sobre os módulos no AWS Encryption SDK formulário JavaScript, consulte Módulos no AWS Encryption SDK para JavaScript e o `README.md` arquivo em cada um dos módulos no aws-encryption-sdk-javascriptrepositório em GitHub.

> **Note**
>
> Todas as versões AWS Encryption SDK para JavaScript anteriores à 2.0.0 estão em end-of-supportfase.
> Você pode atualizar com segurança a partir da versão 2.0.x e posteriores até a versão mais recente do AWS Encryption SDK para JavaScript sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.0.x não são compatíveis com versões anteriores. Para atualizar a partir de versões anteriores à 1.7.x até a versão 2.0. x e posteriores, primeiro será necessário atualizar para a versão 1.x mais recente do AWS Encryption SDK para JavaScript. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

Para instalar os módulos, use o gerenciador de pacotes npm.

Por exemplo, para instalar o `client-node` módulo, que inclui todos os módulos que você precisa programar com o AWS Encryption SDK para JavaScript no Node.js, use o comando a seguir.

```
npm install @aws-crypto/client-node
```

Para instalar o `client-browser` módulo, que inclui todos os módulos que você precisa programar com o AWS Encryption SDK para JavaScript no navegador, use o comando a seguir.

```
npm install @aws-crypto/client-browser
```

Para exemplos práticos de como usar o AWS Encryption SDK para JavaScript, consulte os exemplos nos `example-browser` módulos `example-node` e no aws-encryption-sdk-javascriptrepositório em GitHub.

Módulos no AWS Encryption SDK para JavaScript

Os módulos do AWS Encryption SDK para JavaScript facilitam a instalação do código necessário para seus projetos.

Módulos para JavaScript Node.js

nó do cliente

Inclui todos os módulos que você precisa programar com o AWS Encryption SDK para JavaScript em Node.js.

caching-materials-manager-node

Exporta funções que oferecem suporte ao recurso de cache de chaves de dados AWS Encryption SDK para JavaScript no Node.js.

decrypt-node

Exporta funções que descriptografam e verificam mensagens criptografadas que representam dados e streams de dados. Incluído no módulo `client-node`.

encrypt-node

Exporta funções que criptografam e assinam diferentes tipos de dados. Incluído no módulo `client-node`.

example-node

Exporta exemplos funcionais de programação com o AWS Encryption SDK para JavaScript em Node.js. Inclui exemplos de diferentes tipos de tokens de autenticação e diferentes tipos de dados.

hkdf-node

Exporta uma função de derivação de chave (HKDF) baseada em HMAC que o Node.js usa AWS Encryption SDK para JavaScript em conjuntos de algoritmos específicos. O AWS Encryption SDK para JavaScript no navegador usa a função HKDF nativa na WebCrypto API.

integration-node

Define testes que verificam se o AWS Encryption SDK para JavaScript em Node.js é compatível com outras implementações de linguagem do AWS Encryption SDK.

kms-keyring-node

Exporta funções que oferecem suporte a AWS KMS chaveiros no Node.js.

raw-aes-keyring-node

Exporta funções que são compatíveis com tokens de autenticação brutos do AES no Node.js.

raw-rsa-keyring-node

Exporta funções compatíveis com tokens de autenticação brutos do RSA em Node.js.

Módulos para JavaScript navegador

client-browser

Inclui todos os módulos que você precisa programar com o AWS Encryption SDK para JavaScript no navegador.

caching-materials-manager-browser

Exporta funções que oferecem suporte ao recurso de cache de chave de dados para JavaScript o navegador.

decrypt-browser

Exporta funções que descriptografam e verificam mensagens criptografadas que representam dados e streams de dados.

encrypt-browser

Exporta funções que criptografam e assinam diferentes tipos de dados.

example-browser

Exemplos práticos de programação com o AWS Encryption SDK para JavaScript no navegador. Inclui exemplos de diferentes tipos de tokens de autenticação e diferentes tipos de dados.

integration-browser

Define testes que verificam se o AWS Encryption SDK for Java script no navegador é compatível com outras implementações de linguagem do AWS Encryption SDK.

kms-keyring-browser

Exporta funções compatíveis com tokens de autenticação do AWS KMS no navegador.

raw-aes-keyring-browser

Exporta funções compatíveis com tokens de autenticação brutos do AES no navegador.

raw-rsa-keyring-browser

Exporta funções compatíveis com tokens de autenticação brutos do RSA no navegador.

Módulos para todas as implementações

cache-material

É compatível com o recurso de armazenamento em cache de chaves de dados. Fornece código para montar o material de criptografia que é armazenado em cache com cada chave de dados.

kms-keyring

Exporta funções compatíveis com tokens de autenticação do KMS.

material-management

Implementa o gerenciador de material de criptografia (CMM).

raw-keyring

Exporta funções necessárias para tokens de autenticação brutos do AES e do RSA.

serialize

Exporta funções que o SDK usa para serializar sua saída.

web-crypto-backend

Exporta funções que usam a WebCrypto API AWS Encryption SDK para JavaScript no navegador.

AWS Encryption SDK para JavaScript exemplos

Os exemplos a seguir mostram como usar o AWS Encryption SDK para JavaScript para criptografar e descriptografar dados.

Você pode encontrar mais exemplos de uso do AWS Encryption SDK para JavaScript nos módulos example-node e example-browser no repositório em. aws-encryption-sdk-javascript GitHub Esses módulos de exemplo não são instalados quando você instala os módulos `client-browser` ou `client-node`.

Consulte os exemplos de código completos: nó: kms_simple.ts, navegador: kms_simple.ts

Tópicos

- Criptografando dados com um chaveiro AWS KMS
- Descriptografando dados com um chaveiro AWS KMS

Criptografando dados com um chaveiro AWS KMS

O exemplo a seguir mostra como usar o para AWS Encryption SDK para JavaScript criptografar e descriptografar uma string curta ou uma matriz de bytes.

Este exemplo apresenta um AWS KMS chaveiro, um tipo de chaveiro que usa um AWS KMS key para gerar e criptografar chaves de dados. Para obter ajuda na criação de um AWS KMS key, consulte Criação de chaves no Guia do AWS Key Management Service desenvolvedor. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulte Identificação AWS KMS keys em um AWS KMS chaveiro

Etapa 1: defina a política de compromisso.

> A partir da versão 1.7. x do AWS Encryption SDK para JavaScript, você pode definir a política de compromisso ao chamar a nova `buildClient` função que instancia um AWS Encryption SDK cliente. A função `buildClient` assume um valor enumerado que representa sua política de compromisso. Ela retorna as funções `encrypt` e `decrypt` atualizadas, que reforçam sua política de compromisso quando você criptografa e descriptografa.
>
> Os exemplos a seguir usam a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called “Limitar as chaves de dados criptografadas”.

JavaScript Browser

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

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

JavaScript Node.js

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

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

Etapa 2: estruturar o token de autenticação.

Crie um AWS KMS chaveiro para criptografia.

Ao criptografar com um AWS KMS chaveiro, você deve especificar uma chave geradora, ou seja, uma AWS KMS key que seja usada para gerar a chave de dados em texto simples e criptografá-la. Você também pode especificar zero ou mais chaves adicionais que criptografam a mesma chave de dados de texto simples. O chaveiro retorna a chave de dados em texto simples e uma cópia criptografada dessa chave de dados para cada um AWS KMS key no chaveiro, incluindo a chave do gerador. Para descriptografar os dados, você precisa descriptografar qualquer uma das chaves de dados criptografadas.

Para especificar o AWS KMS keys para um chaveiro de criptografia no AWS Encryption SDK para JavaScript, você pode usar qualquer identificador de AWS KMS chave compatível. Este exemplo usa uma chave geradora, que é identificada por seu ARN de alias, e uma chave adicional, que é identificada por um ARN de chave.

> **Note**
>
> Se você planeja reutilizar seu AWS KMS chaveiro para descriptografar, você deve usar a chave para identificar o que está no ARNs chaveiro. AWS KMS keys

Antes de executar esse código, substitua os identificadores de exemplo por AWS KMS key identificadores válidos. Você deve ter as permissões necessárias para usar as AWS KMS keys no token de autenticação.

JavaScript Browser

Comece fornecendo suas credenciais para o navegador. Os AWS Encryption SDK para JavaScript exemplos usam o webpack. DefinePlugin, que substitui as constantes de credenciais por suas credenciais reais. Mas você pode usar qualquer método para fornecer suas credenciais. Em seguida, use as credenciais para criar um AWS KMS cliente.

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

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

Em seguida, especifique AWS KMS keys a chave do gerador e a chave adicional. Em seguida, crie um AWS KMS chaveiro usando o AWS KMS cliente e o. AWS KMS keys

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

const keyring = new KmsKeyringBrowser({ clientProvider, generatorKeyId, keyIds })
```

JavaScript Node.js

```
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/EncryptDecrypt'
```

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

const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })
```

Etapa 3: defina o contexto de criptografia.

Um contexto de criptografia são dados autenticados adicionais arbitrários e que não são secretos. Quando você fornece um contexto de criptografia na criptografia, ele vincula AWS Encryption SDK criptograficamente o contexto de criptografia ao texto cifrado, de forma que o mesmo contexto de criptografia seja necessário para descriptografar os dados. O uso de um contexto de criptografia é opcional, mas o recomendamos como uma melhor prática.

Crie um objeto simples que inclua os pares de contexto de criptografia. A chave e o valor em cada par devem ser uma string.

JavaScript Browser

```
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

JavaScript Node.js

```
const context = {
  stage: 'demo',
  purpose: 'simple demonstration app',
  origin: 'us-west-2'
}
```

Etapa 4: criptografar os dados.

Para criptografar os dados de texto simples, chame a função `encrypt`. Passe o AWS KMS chaveiro, os dados em texto simples e o contexto de criptografia.

A função `encrypt` retorna uma mensagem criptografada (`result`) que contém os dados criptografados, as chaves de dados criptografadas e metadados importantes, incluindo o contexto de criptografia e a assinatura.

Você pode descriptografar essa mensagem criptografada usando o AWS Encryption SDK para qualquer linguagem de programação compatível.

JavaScript Browser

```
const plaintext = new Uint8Array([1, 2, 3, 4, 5])

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

JavaScript Node.js

```
const plaintext = 'asdf'

const { result } = await encrypt(keyring, plaintext, { encryptionContext:
 context })
```

Descriptografando dados com um chaveiro AWS KMS

Você pode usar o AWS Encryption SDK para JavaScript para descriptografar a mensagem criptografada e recuperar os dados originais.

Neste exemplo, descriptografamos os dados que criptografamos no exemplo the section called "Criptografando dados com um chaveiro AWS KMS".

Etapa 1: defina a política de compromisso.

A partir da versão 1.7. x do AWS Encryption SDK para JavaScript, você pode definir a política de compromisso ao chamar a nova `buildClient` função que instancia um AWS Encryption SDK cliente. A função `buildClient` assume um valor enumerado que representa sua política de compromisso. Ela retorna as funções `encrypt` e `decrypt` atualizadas, que reforçam sua política de compromisso quando você criptografa e descriptografa.

Os exemplos a seguir usam a `buildClient` função para especificar a política de compromisso padrão,`REQUIRE_ENCRYPT_REQUIRE_DECRYPT`. Você também pode usar o `buildClient` para limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Para obter mais informações, consulte the section called "Limitar as chaves de dados criptografadas".

JavaScript Browser

```
import {
```

```
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
} from '@aws-crypto/client-browser'

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

JavaScript Node.js

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

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

Etapa 2: estruturar o token de autenticação.

Para descriptografar os dados, passe a mensagem criptografada (`result`) que a função `encrypt` retornou. A mensagem criptografada inclui os dados criptografados, as chaves de dados criptografadas e metadados importantes, incluindo o contexto de criptografia e a assinatura.

Você também deve especificar um token de autenticação do AWS KMS ao descriptografar. Você pode usar o mesmo token de autenticação usado para criptografar os dados ou um token de autenticação diferente. Para ter sucesso, pelo menos um AWS KMS key no chaveiro de decodificação deve ser capaz de descriptografar uma das chaves de dados criptografadas na mensagem criptografada. Como nenhuma chave de dados é gerada, você não precisa especificar uma chave geradora em um token de autenticação de descriptografia. Se você fizer isso, a chave geradora e as chaves adicionais serão tratadas da mesma maneira.

Para especificar um AWS KMS key para um chaveiro de decodificação no AWS Encryption SDK para JavaScript, você deve usar a chave ARN. Caso contrário, AWS KMS key o não será

reconhecido. Para obter ajuda para identificar o AWS KMS keys em um AWS KMS chaveiro, consulte Identificação AWS KMS keys em um AWS KMS chaveiro

> **Note**
>
> Se você usar o mesmo chaveiro para criptografar e descriptografar, use a chave ARNs para identificar o que está no chaveiro. AWS KMS keys

Neste exemplo, criamos um chaveiro que inclui apenas um dos do AWS KMS keys chaveiro de criptografia. Antes de executar esse código, substitua o ARN da chave de exemplo por um válido. Você deve ter a permissão `kms:Decrypt` na AWS KMS key.

JavaScript Browser

Comece fornecendo suas credenciais para o navegador. Os AWS Encryption SDK para JavaScript exemplos usam o webpack. DefinePlugin, que substitui as constantes de credenciais por suas credenciais reais. Mas você pode usar qualquer método para fornecer suas credenciais. Em seguida, use as credenciais para criar um AWS KMS cliente.

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

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

Em seguida, crie um AWS KMS chaveiro usando o AWS KMS cliente. Este exemplo usa apenas um dos AWS KMS keys do chaveiro de criptografia.

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

const keyring = new KmsKeyringBrowser({ clientProvider, keyIds })
```

JavaScript Node.js

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

const keyring = new KmsKeyringNode({ keyIds })
```

Etapa 3: decifrar os dados.

Chame a função `decrypt`. Passe o token de autenticação de descriptografia que você acabou de criar (`keyring`) e a mensagem criptografada que a função `encrypt` retornou (`result`). O AWS Encryption SDK usa o chaveiro para descriptografar uma das chaves de dados criptografadas. Ele usa a chave de dados de texto simples para descriptografar os dados.

Se a chamada for bem-sucedida, o campo `plaintext` conterá os dados de texto simples (descriptografados). O campo `messageHeader` contém metadados sobre o processo de descriptografia, incluindo o contexto de criptografia usado para descriptografar os dados.

JavaScript Browser

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

JavaScript Node.js

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

Etapa 4: Verifique o contexto de criptografia.

O contexto de criptografia que foi usado para descriptografar os dados é incluído no cabeçalho da mensagem (`messageHeader`) que a função `decrypt` retorna. Antes do aplicativo retornar os dados de texto simples, verifique se o contexto de criptografia fornecido durante a criptografia está incluído no contexto de criptografia usado ao descriptografar. Uma incompatibilidade pode indicar que os dados foram adulterados ou que você não descriptografou o texto cifrado correto.

Ao verificar o contexto de criptografia, não exija uma correspondência exata. Ao usar um algoritmo de criptografia com a assinatura, o gerenciador de material de criptografia (CMM) adiciona a chave de assinatura pública ao contexto de criptografia antes de criptografar a mensagem. Mas todos os pares de contexto de criptografia que você enviou devem ser incluídos no contexto de criptografia que foi retornado.

Primeiro, obtenha o contexto de criptografia do cabeçalho da mensagem. Depois, verifique se cada par de chave-valor no contexto de criptografia original (`context`) corresponde a um par de chave-valor no contexto de criptografia retornado (`encryptionContext`).

JavaScript Browser

```
const { encryptionContext } = messageHeader

Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context
 does not match expected values')
})
```

JavaScript Node.js

```
const { encryptionContext } = messageHeader

Object
  .entries(context)
  .forEach(([key, value]) => {
    if (encryptionContext[key] !== value) throw new Error('Encryption Context
 does not match expected values')
})
```

Se a verificação de contexto de criptografia for bem-sucedida, você poderá retornar os dados de texto simples.

AWS Encryption SDK for Python

Este tópico explica como instalar e usar o AWS Encryption SDK for Python. Para obter detalhes sobre a programação com o AWS Encryption SDK for Python, consulte o aws-encryption-sdk-pythonrepositório em GitHub. Para obter a documentação da API, consulte Ler os documentos.

Tópicos

- Pré-requisitos
- Instalação
- AWS Encryption SDK for Python código de exemplo

Pré-requisitos

Antes de instalar o AWS Encryption SDK for Python, verifique se você tem os seguintes pré-requisitos.

Uma versão compatível do Python

O Python 3.8 ou posterior é exigido pelas AWS Encryption SDK for Python versões 3.2.0 e posteriores.

> **Note**
>
> A Biblioteca de Provedores de Material AWS Criptográfico (MPL) é uma dependência opcional para a AWS Encryption SDK for Python introduzida na versão 4. x. Se você pretende instalar o MPL, você deve usar o Python 3.11 ou posterior.

As versões anteriores do AWS Encryption SDK oferecem suporte ao Python 2.7 e ao Python 3.4 e posteriores, mas recomendamos que você use a versão mais recente do. AWS Encryption SDK

Para fazer download do Python, consulte Downloads do Python.

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 Instalação na documentação do `pip`.

Instalação

Instalar a versão mais recente do AWS Encryption SDK for Python.

> **Note**
>
> Todas as versões AWS Encryption SDK for Python anteriores à 3.0.0 estão em end-of-supportfase.
> Você pode atualizar com segurança a partir da versão 2.0.x e posteriores até a versão mais recente do AWS Encryption SDK sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.0.x não são compatíveis

com versões anteriores. Para atualizar a partir de versões anteriores à 1.7.x até a versão 2.0.x e posteriores, primeiro será necessário atualizar para a versão 1.x mais recente do AWS Encryption SDK. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

Use `pip` para instalar o AWS Encryption SDK for Python, conforme mostrado nos exemplos a seguir.

Para instalar a versão mais recente

```
pip install "aws-encryption-sdk[MPL]"
```

O `[MPL]` sufixo instala a Biblioteca de Provedores de Material AWS Criptográfico (MPL). O MPL contém construções para criptografar e descriptografar seus dados. O MPL é uma dependência opcional para o AWS Encryption SDK for Python introduzido na versão 4. x. É altamente recomendável instalar o MPL. No entanto, se você não pretende usar o MPL, pode omitir o `[MPL]` sufixo.

Para obter mais detalhes sobre o uso do pip para instalar e atualizar pacotes, consulte Instalação de pacotes.

AWS Encryption SDK for Python Isso requer a biblioteca de criptografia (pyca/cryptography) em todas as plataformas. Todas as versões do `pip` instalam e criam automaticamente a biblioteca `cryptography` no Windows. O `pip` 8.1 e versões posteriores instala e compila `cryptography` automaticamente no Linux. Se você usar uma versão anterior do `pip` e seu ambiente Linux não tiver as ferramentas necessárias para criar a biblioteca `cryptography`, será necessário instalá-las. Para obter mais informações, consulte Building Cryptography on Linux.

As versões 1.10.0 e 2.5.0 do AWS Encryption SDK for Python fixam a dependência criptográfica entre 2.5.0 e 3.3.2. Outras versões do AWS Encryption SDK for Python instalam a versão mais recente da criptografia. Se você precisar de uma versão do cryptography posterior à 3.3.2, recomendamos que use a versão principal mais recente do AWS Encryption SDK for Python.

Para obter a versão de desenvolvimento mais recente do AWS Encryption SDK for Python, acesse o aws-encryption-sdk-pythonrepositório em GitHub.

Depois de instalar o AWS Encryption SDK for Python, comece examinando o código de exemplo do Python neste guia.

AWS Encryption SDK for Python código de exemplo

Os exemplos a seguir mostram como usar o para AWS Encryption SDK for Python criptografar e descriptografar dados.

Os exemplos nesta seção mostram como usar a versão 4. x do AWS Encryption SDK for Python com a dependência opcional da Biblioteca de Provedores de Material Criptográfico ()`aws-cryptographic-material-providers`. Para ver exemplos que usam versões anteriores ou instalações sem a biblioteca de fornecedores de materiais (MPL), encontre sua versão na lista de lançamentos do aws-encryption-sdk-pythonrepositório em. GitHub

Quando você usa a versão 4. x do AWS Encryption SDK for Python com o MPL, ele usa chaveiros para realizar a criptografia de envelopes. AWS Encryption SDK Fornece chaveiros compatíveis com os fornecedores de chaves mestras que você usou nas versões anteriores. Para obter mais informações, consulte the section called “Compatibilidade dos tokens de autenticação”. Para exemplos de migração de provedores de chaves mestras para chaveiros, consulte Exemplos de migração no `aws-encryption-sdk-python` repositório em; GitHub

Tópicos

- Criptografar e descriptografar strings
- Criptografar e descriptografar streams de bytes

Criptografar e descriptografar strings

O exemplo a seguir mostra como usar o para criptografar e AWS Encryption SDK descriptografar cadeias de caracteres. Este exemplo usa um AWS KMS chaveiro com uma chave KMS de criptografia simétrica.

Este exemplo instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. `REQUIRE_ENCRYPT_REQUIRE_DECRYPT` Para obter mais informações, consulte the section called “Como definir sua política de compromisso”.

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example sets up the KMS Keyring

The AWS KMS keyring uses symmetric encryption KMS keys to generate, encrypt and
```

```
decrypt data keys. This example creates a KMS Keyring and then encrypts a custom input
 EXAMPLE_DATA
with an encryption context. This example also includes some sanity checks for
 demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

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

"""

import boto3
from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import CreateAwsKmsKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy

EXAMPLE_DATA: bytes = b"Hello World"


def encrypt_and_decrypt_with_keyring(
    kms_key_id: str
):
    """Demonstrate an encrypt/decrypt cycle using an AWS KMS keyring.

    Usage: encrypt_and_decrypt_with_keyring(kms_key_id)
    :param kms_key_id: KMS Key identifier for the KMS key you want to use for
 encryption and
    decryption of your data keys.
    :type kms_key_id: string

    """
    # 1. Instantiate the encryption SDK client.
    # This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
```

```
    # which enforces that this client only encrypts using committing algorithm suites
and enforces
    # that this client will only decrypt encrypted messages that were created with a
committing
    # algorithm suite.
    # This is the default commitment policy if you were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. Create a boto3 client for KMS.
    kms_client = boto3.client('kms', region_name="us-west-2")

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "encryption": "context",
        "is not": "secret",
        "but adds": "useful metadata",
        "that can help you": "be confident that",
        "the data you are handling": "is what you think it is",
    }

    # 4. Create your keyring
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateAwsKmsKeyringInput = CreateAwsKmsKeyringInput(
        kms_key_id=kms_key_id,
        kms_client=kms_client
    )

    kms_keyring: IKeyring = mat_prov.create_aws_kms_keyring(
        input=keyring_input
    )

    # 5. Encrypt the data with the encryptionContext.
    ciphertext, _ = client.encrypt(
        source=EXAMPLE_DATA,
        keyring=kms_keyring,
        encryption_context=encryption_context
    )
```

```
    # 6. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert ciphertext != EXAMPLE_DATA, \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 7. Decrypt your encrypted data using the same keyring you used on encrypt.
    plaintext_bytes, _ = client.decrypt(
        source=ciphertext,
        keyring=kms_keyring,
        # Provide the encryption context that was supplied to the encrypt method
        encryption_context=encryption_context,
    )

    # 8. Demonstrate that the decrypted plaintext is identical to the original
plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert plaintext_bytes == EXAMPLE_DATA, \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
decryption"
```

Criptografar e descriptografar streams de bytes

O exemplo a seguir mostra como usar o para AWS Encryption SDK criptografar e descriptografar fluxos de bytes. Este exemplo usa um chaveiro AES bruto.

Este exemplo instancia o AWS Encryption SDK cliente com a política de compromisso padrão,. `REQUIRE_ENCRYPT_REQUIRE_DECRYPT` Para obter mais informações, consulte the section called "Como definir sua política de compromisso".

```
# Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
This example demonstrates file streaming for encryption and decryption.

File streaming is useful when the plaintext or ciphertext file/data is too large to
 load into
memory. Therefore, the AWS Encryption SDK allows users to stream the data, instead of
 loading it
all at once in memory. In this example, we demonstrate file streaming for encryption
 and decryption
```

```
using a Raw AES keyring. However, you can use any keyring with streaming.

This example creates a Raw AES Keyring and then encrypts an input stream from the file
`plaintext_filename` with an encryption context to an output (encrypted) file
 `ciphertext_filename`.
It then decrypts the ciphertext from `ciphertext_filename` to a new file
 `decrypted_filename`.
This example also includes some sanity checks for demonstration:
1. Ciphertext and plaintext data are not the same
2. Encryption context is correct in the decrypted message header
3. Decrypted plaintext value matches EXAMPLE_DATA
These sanity checks are for demonstration in the example only. You do not need these in
 your code.

See raw_aes_keyring_example.py in the same directory for another raw AES keyring
 example
in the AWS Encryption SDK for Python.
"""
import filecmp
import secrets

from aws_cryptographic_material_providers.mpl import AwsCryptographicMaterialProviders
from aws_cryptographic_material_providers.mpl.config import MaterialProvidersConfig
from aws_cryptographic_material_providers.mpl.models import AesWrappingAlg,
 CreateRawAesKeyringInput
from aws_cryptographic_material_providers.mpl.references import IKeyring
from typing import Dict  # noqa pylint: disable=wrong-import-order

import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_and_decrypt_with_keyring(
    plaintext_filename: str,
    ciphertext_filename: str,
    decrypted_filename: str
):
    """Demonstrate a streaming encrypt/decrypt cycle.

    Usage: encrypt_and_decrypt_with_keyring(plaintext_filename
                                            ciphertext_filename
                                            decrypted_filename)
    :param plaintext_filename: filename of the plaintext data
    :type plaintext_filename: string
```

```
    :param ciphertext_filename: filename of the ciphertext data
    :type ciphertext_filename: string
    :param decrypted_filename: filename of the decrypted data
    :type decrypted_filename: string
    """
    # 1. Instantiate the encryption SDK client.
    # This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
policy,
    # which enforces that this client only encrypts using committing algorithm suites
and enforces
    # that this client will only decrypt encrypted messages that were created with a
committing
    # algorithm suite.
    # This is the default commitment policy if you were to build the client as
    # `client = aws_encryption_sdk.EncryptionSDKClient()`.
    client = aws_encryption_sdk.EncryptionSDKClient(
        commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
    )

    # 2. The key namespace and key name are defined by you.
    # and are used by the Raw AES keyring to determine
    # whether it should attempt to decrypt an encrypted data key.
    key_name_space = "Some managed raw keys"
    key_name = "My 256-bit AES wrapping key"

    # 3. Optional: create encryption context.
    # Remember that your encryption context is NOT SECRET.
    encryption_context: Dict[str, str] = {
        "encryption": "context",
        "is not": "secret",
        "but adds": "useful metadata",
        "that can help you": "be confident that",
        "the data you are handling": "is what you think it is",
    }

    # 4. Generate a 256-bit AES key to use with your keyring.
    # In practice, you should get this key from a secure key management system such as
an HSM.

    # Here, the input to secrets.token_bytes() = 32 bytes = 256 bits
    static_key = secrets.token_bytes(32)

    # 5. Create a Raw AES keyring
    # We choose to use a raw AES keyring, but any keyring can be used with streaming.
```

```
    mat_prov: AwsCryptographicMaterialProviders = AwsCryptographicMaterialProviders(
        config=MaterialProvidersConfig()
    )

    keyring_input: CreateRawAesKeyringInput = CreateRawAesKeyringInput(
        key_namespace=key_name_space,
        key_name=key_name,
        wrapping_key=static_key,
        wrapping_alg=AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
    )

    raw_aes_keyring: IKeyring = mat_prov.create_raw_aes_keyring(
        input=keyring_input
    )

    # 6. Encrypt the data stream with the encryptionContext
    with open(plaintext_filename, 'rb') as pt_file, open(ciphertext_filename, 'wb') as
ct_file:
        with client.stream(
            mode='e',
            source=pt_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as encryptor:
            for chunk in encryptor:
                ct_file.write(chunk)

    # 7. Demonstrate that the ciphertext and plaintext are different.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert not filecmp.cmp(plaintext_filename, ciphertext_filename), \
        "Ciphertext and plaintext data are the same. Invalid encryption"

    # 8. Decrypt your encrypted data stream using the same keyring you used on
encrypt.
    with open(ciphertext_filename, 'rb') as ct_file, open(decrypted_filename, 'wb') as
pt_file:
        with client.stream(
            mode='d',
            source=ct_file,
            keyring=raw_aes_keyring,
            encryption_context=encryption_context
        ) as decryptor:
            for chunk in decryptor:
```

```
                pt_file.write(chunk)

    # 10. Demonstrate that the decrypted plaintext is identical to the original
plaintext.
    # (This is an example for demonstration; you do not need to do this in your own
code.)
    assert filecmp.cmp(plaintext_filename, decrypted_filename), \
        "Decrypted plaintext should be identical to the original plaintext. Invalid
decryption"
```

AWS Encryption SDK para Rust

Este tópico explica como instalar e usar o AWS Encryption SDK for Rust. Para obter detalhes sobre a programação com o AWS Encryption SDK for Rust, consulte o diretório Rust do aws-encryption-sdk repositório em. GitHub

O AWS Encryption SDK for Rust difere de algumas das outras implementações de linguagem de programação do das seguintes AWS Encryption SDK maneiras:

- Não há suporte para armazenamento em cache de chaves de dados. No entanto, o AWS Encryption SDK for Rust suporta o AWS KMS chaveiro hierárquico, uma solução alternativa de cache de materiais criptográficos.
- Não há suporte para streaming de dados

O AWS Encryption SDK for Rust inclui todos os recursos de segurança introduzidos nas versões 2.0. x e posteriores de outras implementações de linguagem do AWS Encryption SDK. No entanto, se você estiver usando o for Rust AWS Encryption SDK para descriptografar dados que foram criptografados por um pré-2.0. versão x outra implementação de linguagem do AWS Encryption SDK, talvez seja necessário ajustar sua política de compromisso. Para obter detalhes, consulte Como definir sua política de compromisso.

O AWS Encryption SDK for Rust é um produto do AWS Encryption SDK in Dafny, uma linguagem de verificação formal na qual você escreve especificações, o código para implementá-las e as provas para testá-las. O resultado é uma biblioteca que implementa os atributos do AWS Encryption SDK em uma estrutura que garante a correção funcional.

Saiba mais

- Para ver exemplos que mostram como configurar opções no AWS Encryption SDK, como especificar um conjunto alternativo de algoritmos, limitar chaves de dados criptografadas e usar chaves AWS KMS multirregionais, consulte. Configurando o AWS Encryption SDK
- Para exemplos que mostram como configurar e usar o AWS Encryption SDK for Rust, consulte os exemplos do Rust no aws-encryption-sdk repositório em. GitHub

Tópicos

- Pré-requisitos
- Instalação
- AWS Encryption SDK para código de exemplo de Rust

Pré-requisitos

Antes de instalar o AWS Encryption SDK for Rust, 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

O AWS Encryption SDK for Rust está disponível como caixa em `aws-esdk`Crates.io. Para obter detalhes sobre como instalar e construir o AWS Encryption SDK para Rust, consulte o README.md no repositório em. aws-encryption-sdk GitHub

Você pode instalar o AWS Encryption SDK for Rust das seguintes maneiras.

Manualmente

Para instalar o AWS Encryption SDK for Rust, clone ou baixe o aws-encryption-sdk GitHub repositório.

Usando Crates.io

Execute o seguinte comando Cargo no diretório do seu projeto:

```
cargo add aws-esdk
```

Ou adicione a seguinte linha ao seu Cargo.toml:

```
aws-esdk = "<version>"
```

AWS Encryption SDK para código de exemplo de Rust

Os exemplos a seguir mostram os padrões básicos de codificação que você usa ao programar com o AWS Encryption SDK for Rust. Especificamente, você instancia a biblioteca AWS Encryption SDK e os fornecedores de materiais. Em seguida, antes de chamar cada método, você instancia o objeto que define a entrada para o método.

Para exemplos que mostram como configurar opções no AWS Encryption SDK, como especificar um conjunto alternativo de algoritmos e limitar chaves de dados criptografadas, consulte os exemplos de Rust no aws-encryption-sdk repositório em. GitHub

Criptografando e descriptografando dados no for Rust AWS Encryption SDK

Este exemplo mostra o padrão básico para criptografar e descriptografar dados. Ele criptografa um pequeno arquivo com chaves de dados protegidas por uma chave de AWS KMS empacotamento.

Etapa 1: Instancie o. AWS Encryption SDK

Você usará os métodos do AWS Encryption SDK para criptografar e descriptografar dados.

```
let esdk_config = AwsEncryptionSdkConfig::builder().build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;
```

Etapa 2: Crie um AWS KMS cliente.

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_client = aws_sdk_kms::Client::new(&sdk_config);
```

Opcional: crie seu contexto de criptografia.

```
let encryption_context = HashMap::from([
```

```
    ("encryption".to_string(), "context".to_string()),
    ("is not".to_string(), "secret".to_string()),
    ("but adds".to_string(), "useful metadata".to_string()),
    ("that can help you".to_string(), "be confident that".to_string()),
    ("the data you are handling".to_string(), "is what you think it
 is".to_string()),
]);
```

Etapa 3: Instanciar a biblioteca de fornecedores de materiais.

Você usará os métodos na biblioteca de fornecedores de materiais para criar os tokens de autenticação que especificam quais chaves protegem seus dados.

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

Etapa 4: Crie um AWS KMS chaveiro.

Para criar o token de autenticação, chame o método do token de autenticação com o objeto de entrada do token de autenticação. Este exemplo usa o `create_aws_kms_keyring()` método e especifica uma chave KMS.

```
let kms_keyring = mpl
    .create_aws_kms_keyring()
    .kms_key_id(kms_key_id)
    .kms_client(kms_client)
    .send()
    .await?;
```

Etapa 5: criptografar o texto sem formatação.

```
let plaintext = example_data.as_bytes();

let encryption_response = esdk_client.encrypt()
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

let ciphertext = encryption_response
                    .ciphertext
```

```
                    .expect("Unable to unwrap ciphertext from encryption response");
```

Etapa 6: descriptografe seus dados criptografados usando o mesmo chaveiro que você usou na criptografia.

```
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
    .send()
    .await?;

let decrypted_plaintext = decryption_response
                            .plaintext
                            .expect("Unable to unwrap plaintext from decryption
 response");
```

AWS Encryption SDK interface de linha de comando

A interface de linha de AWS Encryption SDK comando (CLI de AWS criptografia) permite que você use o para AWS Encryption SDK criptografar e descriptografar dados interativamente na linha de comando e em scripts. Você não precisa ter competência em criptografia ou em programação.

> **Note**
>
> Versões da CLI de AWS criptografia anteriores à 4.0.0 estão em fase. end-of-support
> Você pode atualizar com segurança a partir da versão 2.1.x e posteriores até a versão mais recente da CLI de criptografia da AWS sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.1.x não são compatíveis com versões anteriores. Para atualizar a partir da versão 1.7. x ou anterior, você deve primeiro atualizar para a última 1. versão x da CLI AWS de criptografia. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.
> Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

Como todas as implementações do AWS Encryption SDK, o AWS Encryption CLI oferece recursos avançados de proteção de dados. Os atributos incluem criptografia envelopada, dados autenticados adicionais (AAD) e pacotes de algoritmos de chave simétrica segura e autenticada, como o AES-GCM de 256 bits com derivação de chave, confirmação de chave e assinatura.

A CLI de AWS criptografia é baseada no AWS Encryption SDK for Pythone é compatível com Linux, macOS e Windows. Você pode executar comandos e scripts para criptografar e descriptografar seus dados no shell de sua preferência no Linux ou macOS, em uma janela do prompt de comando (cmd.exe) no Windows e em um console em qualquer sistema. PowerShell

Todas as implementações específicas de linguagem do AWS Encryption SDK, incluindo a AWS CLI de criptografia, são interoperáveis. Por exemplo, você pode criptografar dados com o AWS Encryption SDK for Javae descriptografá-los com a CLI de criptografia. AWS

Este tópico apresenta a CLI de AWS criptografia, explica como instalá-la e usá-la e fornece vários exemplos para ajudar você a começar. Para começar rapidamente, consulte Como criptografar e descriptografar seus dados com a AWS CLI de criptografia no blog de segurança. AWS Para obter informações mais detalhadas, consulte Leia os documentos e junte-se a nós no desenvolvimento da CLI de AWS criptografia aws-encryption-sdk-clino repositório em. GitHub

desempenho

A CLI de AWS criptografia é baseada no. AWS Encryption SDK for Python Cada vez que executa a CLI, você inicia uma nova instância do runtime do Python. Para melhorar o desempenho, sempre que possível, use um único comando em vez de uma série de comandos independentes. Por exemplo, execute um comando que processe os arquivos em um diretório de forma recursiva, em vez de executar comandos separados para cada arquivo.

Tópicos

- Instalar o AWS Encryption SDK Interface da linha de comando
- Como usar a CLI AWS de criptografia
- Exemplos da CLI AWS de criptografia
- AWS Encryption SDK Referência de sintaxe e parâmetros da CLI
- Versões da CLI AWS de criptografia

Instalar o AWS Encryption SDK Interface da linha de comando

Este tópico explica como instalar a CLI AWS de criptografia. Para obter informações detalhadas, consulte o repositório https://github.com/aws/aws-encryption-sdk-cli/ aws-encryption-sdk-cli em e leia a documentação. GitHub https://aws-encryption-sdk-cli.readthedocs.io/en/latest/

Tópicos

- Instalar os pré-requisitos
- Instalando e atualizando o AWS CLI de criptografia

Instalar os pré-requisitos

A CLI de AWS criptografia é baseada no AWS Encryption SDK for Python. Para instalar o AWS Encryption CLI, você precisa do Python e da ferramenta de `pip` gerenciamento de pacotes Python. O Python e o `pip` estão disponíveis em todas as plataformas compatíveis.

Instale os seguintes pré-requisitos antes de instalar a CLI de AWS criptografia,

Python

O Python 3.8 ou posterior é exigido pelas versões 4.2.0 e posteriores do AWS Encryption CLI.

As versões anteriores da CLI de AWS criptografia oferecem suporte ao Python 2.7 e 3.4 e versões posteriores, mas recomendamos que você use a versão mais recente da CLI de criptografia. AWS

O Python está incluído na maioria das instalações do Linux e do macOS, mas é necessário atualizar para o Python 3.6 ou versões posteriores. É recomendável usar a versão mais recente do Python. No Windows, você precisa instalar o Python: ele não é instalado por padrão. Para fazer download do Python, consulte Downloads do Python.

Para determinar se o Python está instalado, na linha de comando, digite:

```
python
```

Para verificar a versão do Python, use o parâmetro `-V` (V maiúsculo).

```
python -V
```

No Windows, depois de instalar o Python, adicione o caminho para o arquivo `Python.exe` ao valor da variável de ambiente Path.

Por padrão, o Python é instalado em todos os diretórios de usuário ou em um diretório de perfil de usuário (`$home` ou `%userprofile%`) no subdiretório `AppData\Local\Programs\Python`. Para encontrar o local do arquivo `Python.exe` no sistema, verifique uma das seguintes chaves de registro. Você pode usar PowerShell para pesquisar no registro.

```
PS C:\> dir HKLM:\Software\Python\PythonCore\version\InstallPath
# -or-
PS C:\> dir HKCU:\Software\Python\PythonCore\version\InstallPath
```

pip

`pip` é o gerenciador de pacotes do Python. Para instalar a CLI de AWS criptografia e suas dependências, você precisa da `pip` versão 8.1 ou posterior. Para obter ajuda para instalar ou atualizar o `pip`, consulte Instalação na documentação do `pip`.

Em instalações Linux, as versões `pip` anteriores à 8.1 não podem criar a biblioteca de criptografia exigida pela CLI de AWS criptografia. Se você optar por não atualizar sua versão do `pip`, poderá instalar as ferramentas de compilação separadamente. Para obter mais informações, consulte Criação de criptografia no Linux.

AWS Command Line Interface

O AWS Command Line Interface (AWS CLI) é necessário somente se você estiver usando AWS KMS keys in AWS Key Management Service (AWS KMS) com a CLI de AWS criptografia. Se você estiver usando um provedor de chave mestra diferente, AWS CLI isso não é necessário.

Para usar AWS KMS keys com a CLI de AWS criptografia, você precisa instalar e configurar o. AWS CLI A configuração disponibiliza as credenciais que você usa para se autenticar na AWS KMS CLI de AWS criptografia.

Instalando e atualizando o AWS CLI de criptografia

Instale a versão mais recente da CLI AWS de criptografia. Quando você usa `pip` para instalar a CLI de AWS criptografia, ela instala automaticamente as bibliotecas de que a CLI precisa, incluindo a AWS Encryption SDK for Python biblioteca de criptografia Python e a. AWS SDK para Python (Boto3)

> **Note**
>
> As versões do AWS Encryption CLI anteriores à 4.0.0 estão na fase de fim do suporte. Você pode atualizar com segurança a partir da versão 2.1.x e posteriores até a versão mais recente da CLI de criptografia da AWS sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.1.x não são compatíveis com versões anteriores. Para atualizar a partir da versão 1.7. x ou anterior, você deve primeiro atualizar para o 1 mais recente. versão x da CLI AWS de criptografia. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.
> Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão 1.8 do AWS Encryption CLI. x substitui a versão 1.7. x e AWS Encryption CLI 2.1. x substitui 2.0. x. Para obter detalhes, consulte o comunicado de segurança relevante no repositório https://github.com/aws/aws-encryption-sdk-cli/ aws-encryption-sdk-cli em. GitHub

Para instalar a versão mais recente da CLI AWS de criptografia

```
pip install aws-encryption-sdk-cli
```

Para atualizar para a versão mais recente da CLI de AWS criptografia

```
pip install --upgrade aws-encryption-sdk-cli
```

Para encontrar os números de versão da sua CLI de AWS criptografia e AWS Encryption SDK

```
aws-encryption-cli --version
```

A saída lista os números de versão de ambas as bibliotecas.

```
aws-encryption-sdk-cli/2.1.0 aws-encryption-sdk/2.0.0
```

Para atualizar para a versão mais recente da CLI de AWS criptografia

```
pip install --upgrade aws-encryption-sdk-cli
```

A instalação do AWS Encryption CLI também instala a versão mais recente do AWS SDK para Python (Boto3), se ainda não estiver instalada. Se o Boto3 estiver instalado, o instalador verifica a versão do Boto3 e a atualiza, se necessário.

Para encontrar sua versão instalada do Boto3

```
pip show boto3
```

Para atualizar para a versão mais recente do Boto3

```
pip install --upgrade boto3
```

Para instalar a versão do AWS Encryption CLI atualmente em desenvolvimento, consulte o repositório https://github.com/aws/aws-encryption-sdk-cli/ aws-encryption-sdk-cli em. GitHub

Para obter mais detalhes sobre o uso do `pip` para instalar e atualizar pacotes do Python, consulte a documentação do pip.

Como usar a CLI AWS de criptografia

Este tópico explica como usar os parâmetros na CLI AWS de criptografia. Para obter exemplos, consulte Exemplos da CLI AWS de criptografia. Para obter a documentação completa, consulte Leia os documentos. A sintaxe mostrada nesses exemplos é para a versão 2.1 do AWS Encryption CLI. x e mais tarde.

> **Note**
>
> Versões da CLI de AWS criptografia anteriores à 4.0.0 estão em fase. end-of-support
> Você pode atualizar com segurança a partir da versão 2.1.x e posteriores até a versão mais recente da CLI de criptografia da AWS sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.1.x não são compatíveis com versões anteriores. Para atualizar a partir da versão 1.7. x ou anterior, você deve primeiro atualizar para a última 1. versão x da CLI AWS de criptografia. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.
> Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a

versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

Para obter um exemplo de como usar o atributo de segurança que limita as chaves de dados criptografadas, consulte Limitar as chaves de dados criptografadas.

Para ver um exemplo de como usar chaves AWS KMS multirregionais, consulteUsando várias regiões AWS KMS keys.

Tópicos

- Como criptografar e descriptografar dados
- Como especificar chaves de encapsulamento
- Como fornecer entrada
- Como especificar o local de saída
- Como usar um contexto de criptografia
- Como especificar uma política de compromisso
- Como armazenar parâmetros em um arquivo de configuração

Como criptografar e descriptografar dados

A CLI de AWS criptografia usa os recursos do AWS Encryption SDK para facilitar a criptografia e a descriptografia de dados com segurança.

> **Note**
>
> O parâmetro `--master-keys` foi descontinuado na versão 1.8. x da CLI de criptografia da AWS e removido na versão 2.1.x.. Em vez disso, use o parâmetro `--wrapping-keys`. A partir da versão 2.1.x, o parâmetro `--wrapping-keys` passou a ser necessário ao criptografar e descriptografar. Para obter detalhes, consulte AWS Encryption SDK Referência de sintaxe e parâmetros da CLI.

- Ao criptografar dados na CLI de AWS criptografia, você especifica seus dados de texto sem formatação e uma chave de encapsulamento (ou chave mestra), como in (). AWS KMS key AWS Key Management Service AWS KMS Se estiver usando um provedor de chaves mestras

personalizado, você também precisará especificar o provedor. Você também especifica locais de saída para a mensagem criptografada e para os metadados sobre a operação de criptografia. Um contexto de criptografia é opcional, mas recomendado.

Na versão 1.8.x, o parâmetro `--commitment-policy` é obrigatório quando você usar o parâmetro `--wrapping-keys`; caso contrário ele não será válido. A partir da versão 2.1x, o parâmetro `--commitment-policy` passou a ser opcional, mas é recomendado.

```
aws-encryption-cli --encrypt --input myPlaintextData \
                   --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                   --output myEncryptedMessage \
                   --metadata-output ~/metadata \
                   --encryption-context purpose=test \
                   --commitment-policy require-encrypt-require-decrypt
```

A CLI de AWS criptografia criptografa seus dados com uma chave de dados exclusiva. Em seguida, ele criptografa cada chave de dados sob a chave de encapsulamento especificada. Ela retorna uma mensagem criptografada e os metadados sobre a operação. A mensagem criptografada contém os dados criptografados (texto cifrado) e uma cópia criptografada da chave de dados. Você não precisa se preocupar com o armazenamento, o gerenciamento ou a perda da chave de dados.

- Ao descriptografar os dados, você passa sua mensagem criptografada, o contexto de criptografia opcional e o local para a saída do texto não criptografado e os metadados. Você também especifica as chaves de encapsulamento que a CLI de AWS criptografia pode usar para descriptografar a mensagem ou informa à CLI de AWS criptografia que ela pode usar qualquer chave de encapsulamento que criptografe a mensagem.

  A partir da versão 1.8.x, o parâmetro `--wrapping-keys` passou a ser opcional, mas é recomendado. A partir da versão 2.1.x, o parâmetro `--wrapping-keys` passou a ser necessário ao criptografar e descriptografar.

  Ao descriptografar, você pode usar o atributo key do parâmetro `--wrapping-keys` para especificar as chaves de encapsulamento que descriptografam seus dados. Especificar uma chave de AWS KMS encapsulamento ao descriptografar é opcional, mas é uma prática recomendada que impede que você use uma chave que você não pretendia usar. Se estiver usando um provedor de chaves mestras personalizado, você deverá especificar o provedor.

Se você não usar o atributo de chave, deverá definir o atributo de descoberta do `--wrapping-keys` parâmetro como`true`, o que permite que a CLI de AWS criptografia seja descriptografada usando qualquer chave de encapsulamento que criptografou a mensagem.

É uma prática recomendada usar o parâmetro `--max-encrypted-data-keys`, para evitar a descriptografia de uma mensagem malformada com um número excessivo de chaves de dados criptografadas. Especifique o número esperado de chaves de dados criptografadas (um para cada chave de encapsulamento usada na criptografia) ou uma quantidade máxima razoável (como 5). Para obter detalhes, consulte Limitar as chaves de dados criptografadas.

O parâmetro `--buffer` retorna texto simples somente após o processamento de todas as entradas, incluindo a verificação da assinatura digital, se houver uma.

O parâmetro `--decrypt-unsigned` descriptografa o texto cifrado e garante que as mensagens não sejam assinadas antes de serem descriptografadas. Use esse parâmetro se você usou o parâmetro `--algorithm` e selecionou um pacote de algoritmos sem assinatura digital para criptografar dados. Se o texto cifrado for assinado, a descriptografia falhará.

Você pode usar `--decrypt` ou `--decrypt-unsigned` para fazer a descriptografia, mas não ambos.

```
aws-encryption-cli --decrypt --input myEncryptedMessage \
                   --wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab \
                   --output myPlaintextData \
                   --metadata-output ~/metadata \
                   --max-encrypted-data-keys 1 \
                   --buffer \
                   --encryption-context purpose=test \
                   --commitment-policy require-encrypt-require-decrypt
```

A CLI de AWS criptografia usa a chave de empacotamento para descriptografar a chave de dados na mensagem criptografada. Em seguida, ela usa a chave de dados para descriptografar os dados. Ela retorna os dados em texto não criptografado e os metadados sobre a operação.

Como especificar chaves de encapsulamento

Ao criptografar dados na CLI de AWS criptografia, você precisa especificar pelo menos uma chave de encapsulamento (ou chave mestra). Você pode usar AWS KMS keys in AWS Key Management

Service (AWS KMS), agrupar chaves de um provedor de chave mestra personalizado ou ambos. O provedor de chaves mestras personalizado pode ser qualquer provedor de chaves mestras compatível com o Python.

Para especificar as chaves de encapsulamento nas versões 1.8.x e posteriores, use o parâmetro `--wrapping-keys` (`-w`). O valor deste parâmetro é uma coleção de atributos com o formato `attribute=value`. Os atributos que você usa dependem do provedor de chaves mestras e do comando.

- AWS KMS. Em comandos encrypt, você deve especificar um parâmetro `--wrapping-keys` com um atributo key. A partir da versão 2.1.x, o parâmetro `--wrapping-keys` também passou a ser necessário em comandos de descriptografia. Ao descriptografar, o parâmetro `--wrapping-keys` deve ter um atributo key ou um atributo discovery (mas não os dois) com o valor de `true`. Todos os outros atributos são opcionais.
- Provedor de chaves mestres personalizado. Você deve especificar um parâmetro `--wrapping-keys` em cada comando. O valor do parâmetro deve ter os atributos key e provider.

Você pode incluir vários parâmetros `--wrapping-keys` e vários atributos key no mesmo comando.

Encapsulando os atributos dos parâmetros de chave

O valor do parâmetro `--wrapping-keys` consiste nos seguintes atributos e seus valores. Um parâmetro `--wrapping-keys` (ou parâmetro `--master-keys`) é necessário em todos os comandos de criptografia. A partir da versão 2.1.x, o parâmetro `--wrapping-keys` também passou a ser necessário em comandos de descriptografia.

Se um nome ou valor de atributo incluir espaços ou caracteres especiais, coloque o nome e o valor entre aspas. Por exemplo, `.--wrapping-keys key=12345 "provider=my cool provider"`

Chave: especifique uma chave de encapsulamento

Use o atributo key para identificar uma chave de encapsulamento. Ao criptografar, o valor pode ser qualquer identificador de chave que o provedor de chaves mestras reconhece.

```
--wrapping-keys key=1234abcd-12ab-34cd-56ef-1234567890ab
```

Em um comando encrypt, cada valor do parâmetro você deve incluir pelo menos um atributo key e um valor. Para criptografar sua chave de dados em várias chaves de encapsulamento, use vários atributos key.

```
aws-encryption-cli --encrypt --wrapping-keys
 key=1234abcd-12ab-34cd-56ef-1234567890ab key=1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d
```

Nos comandos de criptografia usados AWS KMS keys, o valor da chave pode ser o ID da chave, o ARN da chave, um nome de alias ou o ARN do alias. Por exemplo, este comando encrypt usa um ARN do alias no valor do atributo key. Para obter detalhes sobre os identificadores de chave de um AWS KMS key, consulte Identificadores de chave no Guia do AWS Key Management Service desenvolvedor.

```
aws-encryption-cli --encrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:alias/ExampleAlias
```

Em comandos decrypt que usam um provedor de chaves mestres personalizado, os atributos key e provider são necessários.

```
\\ Custom master key provider
aws-encryption-cli --decrypt --wrapping-keys provider='myProvider' key='100101'
```

Nos comandos de descriptografia usados AWS KMS, você pode usar o atributo chave para especificar o a ser usado AWS KMS keys para descriptografia ou o atributo de descoberta com um valor de, `true` que permite que a AWS CLI de criptografia use qualquer AWS KMS key um que tenha sido usado para criptografar a mensagem. Se você especificar um AWS KMS key, ele deverá ser uma das chaves de encapsulamento usadas para criptografar a mensagem.

A especificação da chave de encapsulamento é uma AWS Encryption SDK prática recomendada. Isso garante que você use o AWS KMS key que pretende usar.

Em um comando decrypt, o valor do atributo key deve ser um ARN de chave.

```
\\ AWS KMS key
aws-encryption-cli --decrypt --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

Descoberta: use qualquer um AWS KMS key ao descriptografar

Se você não precisar limitar o uso AWS KMS keys ao descriptografar, você pode usar o atributo de descoberta com um valor de. `true` Um valor de `true` permite que a CLI de AWS criptografia decodifique usando qualquer AWS KMS key uma que criptografe a mensagem. Se

você não especificar um atributo discovery, a descoberta será `false` (padrão). O atributo de descoberta é válido somente em comandos de descriptografia e somente quando a mensagem foi criptografada com. AWS KMS keys

O atributo discovery com o valor definido como `true` é uma alternativa ao uso do atributo key para especificar uma AWS KMS keys. Ao descriptografar uma mensagem criptografada com AWS KMS keys, cada `--wrapping-keys` parâmetro deve ter um atributo-chave ou um atributo de descoberta com um valor de`true`, mas não ambos.

Quando a descoberta é verdadeira, é uma prática recomendada usar os atributos discovery-partition e discovery-account para limitar o AWS KMS keys uso aos atributos especificados por você. Contas da AWS No exemplo a seguir, os atributos de descoberta permitem que a CLI de AWS criptografia use qualquer um dos atributos AWS KMS key especificados. Contas da AWS

```
aws-encryption-cli --decrypt --wrapping-keys \
    discovery=true \
    discovery-partition=aws \
    discovery-account=111122223333 \
    discovery-account=444455556666
```

Provider: especifique o provedor de chaves mestras

O atributo provider identifica o provedor de chaves mestres. O valor padrão é `aws-kms` que representa o AWS KMS. Se estiver usando outro provedor de chaves mestres, o atributo provider será necessário.

```
--wrapping-keys key=12345 provider=my_custom_provider
```

Para obter mais informações sobre o uso de provedores de chaves mestras personalizadas (que não sejam AWS KMS), consulte o tópico Configuração avançada no arquivo README do repositório da CLI e criptografia da AWS.

Região: Especifique uma Região da AWS

Use o atributo de região para especificar o Região da AWS de um AWS KMS key. Esse atributo é válido apenas em comandos encrypt e somente quando o provedor de chaves mestras é o AWS KMS.

```
--encrypt --wrapping-keys key=alias/primary-key region=us-east-2
```

AWS Os comandos CLI de criptografia usam o Região da AWS que é especificado no valor do atributo chave se incluir uma região, como um ARN. Se o valor da chave especificar um Região da AWS, o atributo região será ignorado.

O atributo region tem precedência sobre outras especificações de região. Se você não usar um atributo de região, os comandos da CLI de AWS criptografia usarão o Região da AWS especificado em seu perfil AWS CLI nomeado, se houver, ou em seu perfil padrão.

Profile: especifique um perfil nomeado

Use o atributo profile para especificar um AWS CLI perfil nomeado da . Os perfis nomeados podem incluir credenciais e uma Região da AWS. Esse atributo é válido somente quando o provedor de chaves mestras é o AWS KMS.

```
--wrapping-keys key=alias/primary-key profile=admin-1
```

Você pode usar o atributo profile para especificar credenciais alternativas em comandos encrypt e decrypt. Em um comando encrypt, a CLI de AWS criptografia usa Região da AWS o no perfil nomeado somente quando o valor da chave não inclui uma região e não há nenhum atributo de região. Em um comando decrypt, o perfil Região da AWS in the name é ignorado.

Como especificar várias chaves mestras

Você pode especificar várias chaves de encapsulamento (ou chaves mestras) em cada comando.

Se você especificar mais de uma chave de encapsulamento, a primeira chave de encapsulamento gerará (e criptografará) a chave de dados usada para criptografar seus dados. As outras chaves de encapsulamento criptografam a mesma chave de dados. A mensagem criptografada resultante contém os dados criptografados ("texto cifrado") e uma coleção de chaves de dados criptografadas, criptografadas por cada chave de encapsulamento. Qualquer uma das chaves de encapsulamento podem descriptografar uma chave de dados e descriptografar os dados.

Há duas maneiras de especificar várias chaves de encapsulamento:

- Incluir vários atributos key no valor do parâmetro `--wrapping-keys`.

```
$key_oregon=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
$key_ohio=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
```

```
--wrapping-keys key=$key_oregon key=$key_ohio
```

- Incluir vários parâmetros `--wrapping-keys` no mesmo comando. Use essa sintaxe quando os valores dos atributos que você especificar não se aplicarem a todas as chaves de encapsulamento no comando.

```
--wrapping-keys region=us-east-2 key=alias/test_key \
--wrapping-keys region=us-west-1 key=alias/test_key
```

O atributo de descoberta com um valor de `true` permite que a CLI de AWS criptografia use qualquer um AWS KMS key que criptografe a mensagem. Se você usar vários parâmetros `--wrapping-keys` no mesmo comando, o uso de `discovery=true` em qualquer parâmetro `--wrapping-keys` substituirá efetivamente os limites do atributo key em outros parâmetros `--wrapping-keys`.

Por exemplo, no comando a seguir, o atributo chave no primeiro `--wrapping-keys` parâmetro limita a CLI de AWS criptografia ao especificado. AWS KMS key No entanto, o atributo de descoberta no segundo `--wrapping-keys` parâmetro permite que a CLI de AWS criptografia use qualquer AWS KMS key uma das contas especificadas para descriptografar a mensagem.

```
aws-encryption-cli --decrypt \
    --wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab \
    --wrapping-keys discovery=true \
                    discovery-partition=aws \
                    discovery-account=111122223333 \
                    discovery-account=444455556666
```

Como fornecer entrada

A operação de AWS criptografia na CLI de criptografia usa dados de texto simples como entrada e retorna uma mensagem criptografada. A operação de descriptografia usa uma mensagem criptografada como entrada e retorna dados de texto não criptografado.

O `--input` parâmetro (`-i`), que informa à CLI de AWS criptografia onde encontrar a entrada, é obrigatório em todos os comandos da AWS CLI de criptografia.

Você pode fornecer entrada de qualquer uma das seguintes formas:

- Usar um arquivo.

```
--input myData.txt
```

- Usar um padrão de nome de arquivo.

```
--input testdir/*.xml
```

- Use um diretório ou um padrão de nome de diretório. Quando a entrada é um diretório, o parâmetro `--recursive` (`-r`, `-R`) é necessário.

```
--input testdir --recursive
```

- Redirecionar a entrada para o comando (stdin). Use um valor de `-` para o parâmetro `--input`. (O parâmetro `--input` sempre é necessário.)

```
echo 'Hello World' | aws-encryption-cli --encrypt --input -
```

Como especificar o local de saída

O `--output` parâmetro informa à CLI de AWS criptografia onde gravar os resultados da operação de criptografia ou descriptografia. É necessário em todos os comandos da CLI de AWS criptografia. A CLI de criptografia da AWS cria um novo arquivo de saída para cada arquivo de entrada na operação.

Se um arquivo de saída já existir, por padrão, a CLI de AWS criptografia imprime um aviso e, em seguida, sobrescreve o arquivo. Para evitar a substituição, use o parâmetro `--interactive`, que solicita sua confirmação antes de substituir, ou `--no-overwrite`, que ignora a entrada se a saída puder provocar uma substituição. Para suprimir o aviso de substituição, use `--quiet`. Para capturar erros e avisos da CLI de AWS criptografia, use `2>&1` o operador de redirecionamento para gravá-los no fluxo de saída.

> **Note**
>
> Os comandos que substituem arquivos de saída começam excluindo o arquivo de saída. Se o comando falhar, o arquivo de saída talvez já tenha sido excluído.

Você pode definir o local da saída de várias maneiras.

- Especificar um nome de arquivo. Se você especificar um caminho para o arquivo, todos os diretórios no caminho devem existir antes do comando ser executado.

```
--output myEncryptedData.txt
```

- Especificar um diretório. O diretório de saída deve existir antes do comando ser executado.

  Se a entrada contiver subdiretórios, o comando reproduzirá os subdiretórios no diretório especificado.

```
--output Test
```

  Quando o local de saída é um diretório (sem nomes de arquivo), a CLI de AWS criptografia cria nomes de arquivos de saída com base nos nomes dos arquivos de entrada mais um sufixo. As operações de criptografia acrescentam `.encrypted` ao nome do arquivo de entrada e as operações de descriptografia acrescentam `.decrypted`. Para alterar o sufixo, use o parâmetro `--suffix`.

  Por exemplo, se você criptografar `file.txt`, o comando encrypt criará `file.txt.encrypted`. Se você descriptografar `file.txt.encrypted`, o comando decrypt criará `file.txt.encrypted.decrypted`.

- Gravar na linha de comando (stdout). Insira um valor de `-` para o parâmetro `--output`. Você pode usar `--output -` para redirecionar a saída em outro comando ou programa.

```
--output -
```

Como usar um contexto de criptografia

A CLI de AWS criptografia permite que você forneça um contexto de criptografia nos comandos de criptografia e descriptografia. Ele não é necessário, mas é uma melhor prática criptográfica que recomendamos.

Um contexto de criptografia é um tipo de dados autenticados adicionais arbitrários e que não são segredos. Na CLI de criptografia da AWS , o contexto de criptografia consiste em uma coleção de pares `name=value`. Você pode usar qualquer conteúdo nos pares, incluindo informações sobre os

arquivos; dados que o ajudam a encontrar a operação de criptografia em logs; ou dados que suas concessões ou políticas exigem.

Em um comando encrypt

O contexto de criptografia que você especifica em um comando encrypt, junto com qualquer par que o CMM adicionar, é associado de maneira criptográfica aos dados criptografados. Ele também é incluído (em não criptografado) na mensagem criptografada que o comando retorna. Se você estiver usando um AWS KMS key, o contexto de criptografia também poderá aparecer em texto simples em registros e registros de auditoria, como. AWS CloudTrail

O exemplo a seguir mostra um contexto de criptografia com três pares `name=value`.

```
--encryption-context purpose=test dept=IT class=confidential
```

Em um comando decrypt

Em um comando decrypt, o contexto de criptografia ajuda a confirmar se você está descriptografando a mensagem criptografada correta.

Não é necessário fornecer um contexto de criptografia em um comando decrypt, mesmo que um contexto de criptografia tenha sido usado na criptografia. No entanto, se você fizer isso, a CLI de AWS criptografia verificará se cada elemento no contexto de criptografia do comando decrypt corresponde a um elemento no contexto de criptografia da mensagem criptografada. Se um elemento não corresponder, o comando decrypt falhará.

Por exemplo, o comando a seguir descriptografa a mensagem criptografada somente se o contexto de criptografia incluir `dept=IT`.

```
aws-encryption-cli --decrypt --encryption-context dept=IT ...
```

Um contexto de criptografia é uma parte importante de sua estratégia de segurança. No entanto, ao escolher um contexto de criptografia, lembre-se de que seus valores não são secretos. Não inclua dados confidenciais no contexto de criptografia.

Como especificar um contexto de criptografia

- Em um comando encrypt, use o parâmetro `--encryption-context` com um ou mais pares `name=value`. Use um espaço para separar cada par.

```
--encryption-context name=value [name=value] ...
```

- Em um comando decrypt, o valor do parâmetro `--encryption-context` pode incluir pares `name=value`, elementos `name` (sem valores) ou uma combinação de ambos.

```
--encryption-context name[=value] [name] [name=value] ...
```

Se o `name` ou o `value` em um par de `name=value` incluir espaços ou caracteres especiais, coloque o par inteiro entre aspas.

```
--encryption-context "department=software engineering" "Região da AWS=us-west-2"
```

Por exemplo, este comando encrypt inclui um contexto de criptografia com dois pares, `purpose=test` e `dept=23`.

```
aws-encryption-cli --encrypt --encryption-context purpose=test dept=23 ...
```

Esse comando decrypt tem êxito. O contexto de criptografia em cada comando é um subconjunto do contexto de criptografia original.

```
\\ Any one or both of the encryption context pairs
aws-encryption-cli --decrypt --encryption-context dept=23 ...

\\ Any one or both of the encryption context names
aws-encryption-cli --decrypt --encryption-context purpose ...

\\ Any combination of names and pairs
aws-encryption-cli --decrypt --encryption-context dept purpose=test ...
```

No entanto, esses comandos decrypt falharão. O contexto de criptografia na mensagem criptografada não contém os elementos específicos.

```
aws-encryption-cli --decrypt --encryption-context dept=Finance ...
aws-encryption-cli --decrypt --encryption-context scope ...
```

Como especificar uma política de compromisso

Para definir a política de compromisso para o comando, use o parâmetro `--commitment-policy`. Esse parâmetro foi apresentado na versão 1.8.x.. Ele 'é válido em comandos de criptografia e descriptografia. A política de compromisso que você definir será válida somente para o comando no qual ela aparece. Se você não definir uma política de compromisso para um comando, a CLI de AWS criptografia usará o valor padrão.

Por exemplo, o valor do parâmetro a seguir define a política de compromisso como `require-encrypt-allow-decrypt`, que sempre criptografa com o confirmação de chave, mas descriptografa um texto cifrado criptografado com ou sem confirmação de chave.

```
--commitment-policy require-encrypt-allow-decrypt
```

Como armazenar parâmetros em um arquivo de configuração

Você pode economizar tempo e evitar erros de digitação salvando os parâmetros e valores da CLI de AWS criptografia usados com frequência nos arquivos de configuração.

Um arquivo de configuração é um arquivo de texto que contém parâmetros e valores para um comando CLI de AWS criptografia. Ao fazer referência a um arquivo de configuração em um comando da CLI de criptografia da AWS , a referência é substituída pelos parâmetros e valores no arquivo de configuração. O efeito será o mesmo como se você tivesse digitado o conteúdo do arquivo na linha de comando. Um arquivo de configuração pode ter qualquer nome e pode ser localizado em qualquer diretório que o usuário atual pode acessar.

O arquivo de configuração de exemplo a seguir, `key.conf`, especifica duas AWS KMS keys em diferentes regiões.

```
--wrapping-keys key=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
--wrapping-keys key=arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
```

Para usar o arquivo de configuração em um comando, prefixe o nome do arquivo com uma arroba (@). Em um PowerShell console, use um caractere de crase para escapar do sinal arroba (`@).

Este comando de exemplo usa o arquivo `key.conf` em um comando encrypt.

Bash

```
$ aws-encryption-cli -e @key.conf -i hello.txt -o testdir
```

PowerShell

```
PS C:\> aws-encryption-cli -e `@key.conf -i .\Hello.txt -o .\TestDir
```

Regras do arquivo de configuração

As regras para uso de arquivos de configuração são:

- Você pode incluir vários parâmetros em cada arquivo de configuração e listá-los em qualquer ordem. Liste cada parâmetro com seus valores (se houver) em uma linha separada.
- Use # para adicionar um comentário a toda ou a parte de uma linha.
- Você pode incluir referências a outros arquivos de configuração. Não use uma craqueta para escapar da @ placa, mesmo dentro PowerShell.
- Se você usar aspas em um arquivo de configuração, o texto entre aspas não pode abranger várias linhas.

Por exemplo, este é o conteúdo de um arquivo `encrypt.conf` de exemplo.

```
# Archive Files
--encrypt
--output /archive/logs
--recursive
--interactive
--encryption-context class=unclassified dept=IT
--suffix  # No suffix
--metadata-output ~/metadata
@caching.conf  # Use limited caching
```

Você também pode incluir vários arquivos de configuração em um comando. Este comando de exemplo usa os arquivos de configuração `encrypt.conf` e `master-keys.conf`.

Bash

```
$  aws-encryption-cli -i /usr/logs @encrypt.conf @master-keys.conf
```

PowerShell

```
PS C:\> aws-encryption-cli -i $home\Test\*.log `@encrypt.conf `@master-keys.conf
```

Próximo: Experimente os exemplos da CLI de criptografia da AWS

Exemplos da CLI AWS de criptografia

Use os exemplos a seguir para testar a CLI de AWS criptografia na plataforma de sua preferência. Para obter ajuda com chaves mestras e outros parâmetros, consulte Como usar a CLI AWS de criptografia. Para obter uma referência rápida, consulte AWS Encryption SDK Referência de sintaxe e parâmetros da CLI.

> **Note**
>
> Os exemplos a seguir usam a sintaxe da AWS Encryption CLI versão 2.1. x.
> Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

Para obter um exemplo de como usar o atributo de segurança que limita as chaves de dados criptografadas, consulte Limitar as chaves de dados criptografadas.

Para ver um exemplo de como usar chaves AWS KMS multirregionais, consulteUsando várias regiões AWS KMS keys.

Tópicos

- Criptografar um arquivo
- Descriptografar um arquivo
- Criptografar todos os arquivos em um diretório
- Descriptografar todos os arquivos em um diretório
- Criptografar e descriptografar na linha de comando
- Uso de várias chaves mestras
- Criptografar e descriptografar em scripts

- Usar o armazenamento em cache de chaves de dados

Criptografar um arquivo

Este exemplo usa a CLI de AWS criptografia para criptografar o conteúdo do `hello.txt` arquivo, que contém uma string “Hello World”.

Quando você executa um comando de criptografia em um arquivo, a CLI de AWS criptografia obtém o conteúdo do arquivo, gera uma chave de dados exclusiva, criptografa o conteúdo do arquivo sob a chave de dados e, em seguida, grava a mensagem criptografada em um novo arquivo.

O primeiro comando salva a chave ARN de an AWS KMS key na `$keyArn` variável. Ao criptografar com um AWS KMS key, você pode identificá-lo usando um ID de chave, ARN da chave, nome do alias ou ARN do alias. Para obter detalhes sobre os identificadores de chave de um AWS KMS key, consulte Identificadores de chave no Guia do AWS Key Management Service desenvolvedor.

O segundo comando criptografa o conteúdo do arquivo. O comando usa o parâmetro `--encrypt` para especificar a operação, e o parâmetro `--input` para indicar o arquivo a ser criptografado. O `--wrapping-keys`parâmetro e seu atributo de chave obrigatório fazem com que o comando use o AWS KMS key representado pela chave ARN.

O comando usa o parâmetro `--metadata-output` para especificar um arquivo de texto para os metadados sobre a operação de criptografia. Como prática recomendada, o comando usa o parâmetro `--encryption-context` para especificar um contexto de criptografia.

Esse comando também usa o parâmetro `--commitment-policy` para definir explicitamente a política de compromisso. Na versão 1.8. x, ele é necessário quando você usa o parâmetro `--wrapping-keys`. A partir da versão 2.1x, o parâmetro `--commitment-policy` passou a ser opcional, mas é recomendado.

O valor do parâmetro `--output`, um ponto (.), informa o comando para gravar o arquivo de saída no diretório atual.

Bash

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

$ aws-encryption-cli --encrypt \
```

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

PowerShell

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

PS C:\> aws-encryption-cli --encrypt `
                           --input Hello.txt `
                           --wrapping-keys key=$keyArn `
                           --metadata-output $home\Metadata.txt `
                           --commitment-policy require-encrypt-require-decrypt `
                           --encryption-context purpose=test `
                           --output .
```

Quando o comando encrypt é bem-sucedido, ele não retorna nenhuma saída. Para determinar se o comando foi bem-sucedido, verifique o valor booliano na variável `$?`. Quando o comando é bem-sucedido, o valor de `$?` é `0` (Bash) ou `True` (PowerShell). Quando o comando falha, o valor de `$?` é diferente de zero (Bash) ou `False` (PowerShell).

Bash

```
$ echo $?
0
```

PowerShell

```
PS C:\> $?
True
```

Você também pode usar um comando de listagem de diretório para ver se o comando encrypt criou um novo arquivo, `hello.txt.encrypted`. Como o comando encrypt não especificou um nome de arquivo para a saída, a CLI de AWS criptografia gravou a saída em um arquivo com o mesmo nome

do arquivo de entrada mais `.encrypted` um sufixo. Para usar outro sufixo ou suprimir o sufixo, use o parâmetro `--suffix`.

O arquivo `hello.txt.encrypted` contém uma mensagem criptografada que inclui o texto cifrado do arquivo `hello.txt`, uma cópia criptografada da chave de dados e metadados adicionais incluindo o contexto de criptografia.

Bash

```
$  ls
hello.txt  hello.txt.encrypted
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
```

Descriptografar um arquivo

Este exemplo usa a CLI de AWS criptografia para descriptografar o conteúdo do `Hello.txt.encrypted` arquivo que foi criptografado no exemplo anterior.

O comando decrypt usa o parâmetro `--decrypt` para indicar a operação, e o parâmetro `--input` para identificar o arquivo a ser descriptografado. O valor do parâmetro `--output` é um ponto que representa o diretório atual.

O parâmetro `--wrapping-keys` com um atributo key especifica a chave de encapsulamento usada para descriptografar a mensagem criptografada. Em comandos de descriptografia com AWS KMS keys, o valor do atributo chave deve ser um ARN de chave. O parâmetro `--wrapping-keys` é obrigatório em comandos encrypt. Se você usar AWS KMS keys, poderá usar o atributo key prara especificar AWS KMS keys para descritografar ou o atributo discovery com um valor definido como`true` (mas não ambos). Se estiver usando outro provedor de chaves mestras, os atributos key e provider serão necessários.

A partir da versão 2.1x, o parâmetro *--commitment-policy* passou a ser opcional, mas é recomendado. Usá-lo explicitamente deixa clara sua intenção, mesmo se você especificar o valor padrão, `require-encrypt-require-decrypt`.

O parâmetro `--encryption-context` é opcional no comando decrypt, mesmo quando um contexto de criptografia é fornecido no comando encrypt. Nesse caso, o comando decrypt usa o mesmo contexto de criptografia que foi fornecido no comando encrypt. Antes de descriptografar, a AWS CLI de criptografia verifica se o contexto de criptografia na mensagem criptografada inclui um par. `purpose=test` Caso contrário, o comando decrypt falhará.

O parâmetro `--metadata-output` especifica um arquivo de metadados sobre a operação de descriptografia. O valor do parâmetro `--output`, um ponto (.), grava o arquivo de saída no diretório atual.

É uma prática recomendada usar o parâmetro `--max-encrypted-data-keys`, para evitar a descriptografia de uma mensagem malformada com um número excessivo de chaves de dados criptografadas. Especifique o número esperado de chaves de dados criptografadas (um para cada chave de encapsulamento usada na criptografia) ou uma quantidade máxima razoável (como 5). Para obter detalhes, consulte Limitar as chaves de dados criptografadas.

Ele `--buffer` retorna texto sem formatação somente após o processamento de todas as entradas, incluindo a verificação da assinatura digital, se houver uma.

Bash

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

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

PowerShell

```
\\ To run this example, replace the fictitious key ARN with a valid value.
```

```
PS C:\> $keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --decrypt `
                           --input Hello.txt.encrypted `
                           --wrapping-keys key=$keyArn `
                           --commitment-policy require-encrypt-require-decrypt `
                           --encryption-context purpose=test `
                           --metadata-output $home\Metadata.txt `
                           --max-encrypted-data-keys 1 `
                           --buffer `
                           --output .
```

Quando um comando decrypt é bem-sucedido, ele não retorna nenhuma saída. Para determinar se o comando foi bem-sucedido, obtenha o valor da variável `$?`. Você também pode usar um comando de listagem de diretório para ver se o comando criou um novo arquivo com um sufixo `.decrypted`. Para ver o conteúdo de texto não criptografado, use um comando para obter o conteúdo do arquivo, como `cat` ou Get-Content.

Bash

```
$  ls
hello.txt  hello.txt.encrypted  hello.txt.encrypted.decrypted

$  cat hello.txt.encrypted.decrypted
Hello World
```

PowerShell

```
PS C:\> dir

    Directory: C:\TestCLI

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/17/2017   1:01 PM             11 Hello.txt
-a----        9/17/2017   1:06 PM            585 Hello.txt.encrypted
-a----        9/17/2017   1:08 PM             11 Hello.txt.encrypted.decrypted


PS C:\> Get-Content Hello.txt.encrypted.decrypted
```

```
Hello World
```

Criptografar todos os arquivos em um diretório

Este exemplo usa a CLI de AWS criptografia para criptografar o conteúdo de todos os arquivos em um diretório.

Quando um comando afeta vários arquivos, a CLI de AWS criptografia processa cada arquivo individualmente. Ela obtém o conteúdo do arquivo, obtém uma chave de dados exclusiva para o arquivo da chave mestre, criptografa o conteúdo do arquivo sob a chave de dados e grava os resultados em um novo arquivo no diretório de saída. Como resultado, você pode descriptografar os arquivos de saída de maneira independente.

Essa listagem do diretório `TestDir` mostra os arquivos de texto não criptografado que desejamos criptografar.

Bash

```
$  ls testdir
cool-new-thing.py  hello.txt  employees.csv
```

PowerShell

```
PS C:\> dir C:\TestDir

    Directory: C:\TestDir

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/12/2017   3:11 PM           2139 cool-new-thing.py
-a----        9/15/2017   5:57 PM             11 Hello.txt
-a----        9/17/2017   1:44 PM             46 Employees.csv
```

O primeiro comando salva o Amazon Resource Name (ARN) de an AWS KMS key na `$keyArn` variável.

O segundo comando criptografa o conteúdo dos arquivos no diretório `TestDir` e grava os arquivos de conteúdo criptografado no `TestEnc`. Se o diretório `TestEnc` não existir, o comando falhará. Como o local de entrada é um diretório, o parâmetro `--recursive` é obrigatório.

O parâmetro `--wrapping-keys` e seu atributo-chave obrigatório especificam a chave de encapsulamento a ser usada. O comando encrypt inclui um contexto de criptografia, `dept=IT`. Quando você especifica um contexto de criptografia em um comando que criptografa vários arquivos, o mesmo contexto de criptografia é usado para todos os arquivos.

O comando também tem um `--metadata-output` parâmetro para informar à CLI de AWS criptografia onde gravar os metadados sobre as operações de criptografia. A CLI de AWS criptografia grava um registro de metadados para cada arquivo criptografado.

A partir da versão 2.1.x, o `--commitment-policy parameter` passou a ser opcional, mas é recomendado. Se o comando ou script falhar porque não consegue decifrar um texto cifrado, a configuração explícita da política de compromisso pode ajudar a detectar o problema rapidamente.

Quando o comando é concluído, a CLI de AWS criptografia grava os arquivos criptografados `TestEnc` no diretório, mas não retorna nenhuma saída.

O último comando lista os arquivos no diretório `TestEnc`. Há um arquivo de saída de conteúdo criptografado para cada arquivo de entrada de conteúdo de texto não criptografado. Como o comando não especificou um sufixo alternativo, o comando encrypt acrescentou `.encrypted` a cada um dos nomes de arquivos de entrada.

Bash

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

$ aws-encryption-cli --encrypt \
                     --input testdir --recursive\
                     --wrapping-keys key=$keyArn \
                     --encryption-context dept=IT \
                     --commitment-policy require-encrypt-require-decrypt \
                     --metadata-output ~/metadata \
                     --output testenc

$ ls testenc
cool-new-thing.py.encrypted  employees.csv.encrypted  hello.txt.encrypted
```

PowerShell

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

PS C:\> aws-encryption-cli --encrypt `
                           --input .\TestDir --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output .\Metadata\Metadata.txt `
                           --output .\TestEnc

PS C:\> dir .\TestEnc

    Directory: C:\TestEnc

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

Descriptografar todos os arquivos em um diretório

Este exemplo descriptografa todos os arquivos em um diretório. Ele começa com os arquivos no diretório `TestEnc` que foram criptografados no exemplo anterior.

Bash

```
$  ls testenc
cool-new-thing.py.encrypted  hello.txt.encrypted  employees.csv.encrypted
```

PowerShell

```
PS C:\> dir C:\TestEnc

    Directory: C:\TestEnc

Mode                 LastWriteTime         Length Name
```

```
----                 -------------         ------ ----
-a----        9/17/2017   2:32 PM           2713 cool-new-thing.py.encrypted
-a----        9/17/2017   2:32 PM            620 Hello.txt.encrypted
-a----        9/17/2017   2:32 PM            585 Employees.csv.encrypted
```

Esse comando decrypt descriptografa todos os arquivos no TestEnc diretório e grava os arquivos de texto simples no diretório. TestDec O `--wrapping-keys` parâmetro com um atributo de chave e um valor de ARN de chave informa à AWS CLI de criptografia qual usar AWS KMS keys para descriptografar os arquivos. O comando usa o `--interactive` parâmetro para fazer com que a CLI de AWS criptografia avise você antes de sobrescrever um arquivo com o mesmo nome.

Esse comando também usa o contexto de criptografia que foi fornecido quando os arquivos foram criptografados. Ao descriptografar vários arquivos, a AWS CLI de criptografia verifica o contexto de criptografia de cada arquivo. Se a verificação do contexto de criptografia em qualquer arquivo falhar, a CLI de AWS criptografia rejeitará o arquivo, gravará um aviso, registrará a falha nos metadados e continuará verificando os arquivos restantes. Se a CLI de AWS criptografia falhar ao descriptografar um arquivo por qualquer outro motivo, todo o comando decrypt falhará imediatamente.

Neste exemplo, as mensagens criptografadas em todos os arquivos de entrada contêm o elemento do contexto de criptografia de `dept=IT`. No entanto, se você estiver descriptografando mensagens com diferentes contextos de criptografia, você ainda poderá verificar parte do contexto de criptografia. Por exemplo, se algumas mensagens tiverem um contexto de criptografia de `dept=finance` e outras tiverem `dept=IT`, você poderá verificar se o contexto de criptografia sempre contém um nome `dept` sem especificar o valor. Se desejar ser mais específico, você poderá descriptografar os arquivos em comandos separados.

O comando decrypt não retorna nenhuma saída, mas você pode usar um comando de listagem de diretórios para ver se ele criou novos arquivos com o sufixo `.decrypted`. Para ver o conteúdo de texto não criptografado, use um comando para obter o conteúdo do arquivo.

Bash

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

$ aws-encryption-cli --decrypt \
                     --input testenc --recursive \
                     --wrapping-keys key=$keyArn \
```

```
                    --encryption-context dept=IT \
                    --commitment-policy require-encrypt-require-decrypt \
                    --metadata-output ~/metadata \
                    --max-encrypted-data-keys 1 \
                    --buffer \
                    --output testdec --interactive

$ ls testdec
cool-new-thing.py.encrypted.decrypted  hello.txt.encrypted.decrypted
 employees.csv.encrypted.decrypted
```

PowerShell

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

PS C:\> aws-encryption-cli --decrypt `
                           --input C:\TestEnc --recursive `
                           --wrapping-keys key=$keyArn `
                           --encryption-context dept=IT `
                           --commitment-policy require-encrypt-require-decrypt `
                           --metadata-output $home\Metadata.txt `
                           --max-encrypted-data-keys 1 `
                           --buffer `
                           --output C:\TestDec --interactive

PS C:\> dir .\TestDec


    Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        10/8/2017   4:57 PM           2139 cool-new-
thing.py.encrypted.decrypted
-a----        10/8/2017   4:57 PM             46 Employees.csv.encrypted.decrypted
-a----        10/8/2017   4:57 PM             11 Hello.txt.encrypted.decrypted
```

Criptografar e descriptografar na linha de comando

Estes exemplos mostram como redirecionar a entrada para comandos (stdin) e gravar a saída na linha de comando (stdout). Eles explicam como representar stdin e stdout em um comando e como

usar ferramentas de codificação Base64 internas para impedir que o shell interprete caracteres não ASCII incorretamente.

Este exemplo redireciona uma string de texto não criptografado para um comando encrypt e salva a mensagem criptografada em uma variável. Em seguida, ele redireciona a mensagem criptografada na variável para um comando decrypt, que grava sua saída no pipeline (stdout).

O exemplo consiste em três comandos:

- O primeiro comando salva a chave ARN de an AWS KMS key na `$keyArn` variável.

  Bash

  ```
  $  keyArn=arn:aws:kms:us-
  west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
  ```

  PowerShell

  ```
  PS C:\> $keyArn = 'arn:aws:kms:us-
  west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
  ```

- O segundo comando redireciona a string `Hello World` para o comando encrypt e salva o resultado na variável `$encrypted`.

  Os parâmetros `--input` e `--output` são obrigatórios em todos os comandos da CLI de criptografia da AWS . Para indicar que a entrada está sendo redirecionada para o comando (stdin), use um hífen (`-`) para o valor do parâmetro `--input`. Para enviar a saída para a linha de comando (stdout), use um hífen para o valor do parâmetro `--output`.

  O parâmetro `--encode` codifica a saída em Base64 antes de retorná-la. Isso evita que o shell interprete incorretamente os caracteres não ASCII na mensagem criptografada.

  Como esse comando é apenas uma prova de conceito, omitimos o contexto de criptografia e suprimimos os metadados (`-S`).

  Bash

  ```
  $ encrypted=$(echo 'Hello World' | aws-encryption-cli --encrypt -S \
                                                     --input - --output - --
  encode \
  ```

```
                                                        --wrapping-keys key=
$keyArn )
```

PowerShell

```
PS C:\> $encrypted = 'Hello World' | aws-encryption-cli --encrypt -S `
                                                        --input - --output - --
encode `
                                                        --wrapping-keys key=
$keyArn
```

- O terceiro comando redireciona a mensagem criptografada na variável `$encrypted` para descriptografar o comando.

  Esse comando decrypt usa `--input -` para indicar que a entrada é proveniente do pipeline (stdin) e do `--output -` para enviar a saída para o pipeline (stdout). (O parâmetro de entrada usa o local da entrada, não os bytes reais da entrada. Portanto, você não pode usar a variável `$encrypted` como o valor do parâmetro `--input`.)

  Este exemplo usa o atributo de descoberta do `--wrapping-keys` parâmetro para permitir que a CLI de AWS criptografia use qualquer um para AWS KMS key descriptografar os dados. Ele não especifica uma política de compromisso, portanto, usa o valor padrão para a versão 2.1.x e posteriores, `require-encrypt-require-decrypt`.

  Como a saída foi criptografada e, em seguida, codificada, o comando decrypt usa o parâmetro `--decode` para decodificar a entrada codificada em Base64 antes de descriptografá-la. Você também pode usar o parâmetro `--decode` para decodificar a entrada codificada em Base64 antes de criptografá-la.

  Novamente, o comando omite o contexto de criptografia e suprime os metadados (-S).

  Bash

```
$  echo $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=true
 --input - --output - --decode --buffer -S
Hello World
```

PowerShell

```
PS C:\> $encrypted | aws-encryption-cli --decrypt --wrapping-keys discovery=$true
 --input - --output - --decode --buffer -S
Hello World
```

Você também pode executar operações de criptografia e descriptografia em um único comando sem a variável de intervenção.

Como no exemplo anterior, os parâmetros `--input` e `--output` têm um valor `-` e o comando usa o parâmetro `--encode` para codificar a saída, e o parâmetro `--decode` para decodificar a entrada.

Bash

```
$  keyArn=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$  echo 'Hello World' |
          aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --
output - --encode -S |
          aws-encryption-cli --decrypt --wrapping-keys discovery=true --input - --
output - --decode -S
Hello World
```

PowerShell

```
PS C:\> $keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> 'Hello World' |
               aws-encryption-cli --encrypt --wrapping-keys key=$keyArn --input - --
output - --encode -S |
               aws-encryption-cli --decrypt --wrapping-keys discovery=$true --input
 - --output - --decode -S
Hello World
```

Uso de várias chaves mestras

Este exemplo mostra como usar várias chaves mestras ao criptografar e descriptografar dados na CLI de criptografia. AWS

Quando você usa várias chaves mestras para criptografar dados, qualquer uma das chaves mestras pode ser usada para descriptografar os dados. Essa estratégia garante que você possa descriptografar os dados mesmo que uma das chaves mestras esteja indisponível. Se você estiver armazenando os dados criptografados em vários Regiões da AWS, essa estratégia permite usar uma chave mestra na mesma região para descriptografar os dados.

Quando você criptografa com várias chaves mestras, a primeira chave mestra desempenha uma função especial. Ela gera a chave de dados que é usada para criptografar os dados. As demais chaves mestras criptografam a chave de dados de texto não criptografado. A mensagem criptografada resultante inclui os dados criptografados e uma coleção de chaves de dados criptografadas, uma para cada chave mestre. Embora a primeira chave mestra tenha gerado a chave de dados, qualquer uma das chaves mestras poderá descriptografar uma das chaves de dados, que pode ser usada para descriptografar os dados.

Criptografia com três chaves mestres

Este comando de exemplo usa três chaves de encapsulamento para criptografar o arquivo `Finance.log`, uma em cada uma das três Regiões da AWS.

Ele grava a mensagem criptografada no diretório `Archive`. O comando usa o parâmetro `--suffix` sem nenhum valor para suprimir o sufixo. Portanto, os nomes dos arquivos de entrada e saída serão o mesmos.

O comando usa o parâmetro `--wrapping-keys` com três atributos key. Você também pode usar vários parâmetros `--wrapping-keys` no mesmo comando.

Para criptografar o arquivo de log, a CLI de AWS criptografia solicita que a primeira chave de encapsulamento na lista`$key1`,, gere a chave de dados que ela usa para criptografar os dados. Em seguida, ela usa cada uma das outras chaves de encapsulamento para criptografar uma cópia de texto não criptografado da mesma chave de dados. A mensagem criptografada no arquivo de saída inclui todas as três chaves de dados criptografadas.

Bash

```
$ key1=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab
```

```
$ key2=arn:aws:kms:us-east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef
$ key3=arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d

$ aws-encryption-cli --encrypt --input /logs/finance.log \
                               --output /archive --suffix \
                               --encryption-context class=log \
                               --metadata-output ~/metadata \
                               --wrapping-keys key=$key1 key=$key2 key=$key3
```

PowerShell

```
PS C:\> $key1 = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
PS C:\> $key2 = 'arn:aws:kms:us-
east-2:111122223333:key/0987ab65-43cd-21ef-09ab-87654321cdef'
PS C:\> $key3 = 'arn:aws:kms:ap-
southeast-1:111122223333:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d'

PS C:\> aws-encryption-cli --encrypt --input D:\Logs\Finance.log `
                           --output D:\Archive --suffix `
                           --encryption-context class=log `
                           --metadata-output $home\Metadata.txt `
                           --wrapping-keys key=$key1 key=$key2 key=$key3
```

Este comando descriptografa a cópia criptografada do arquivo `Finance.log` e grava-o em um arquivo `Finance.log.clear` no diretório `Finance`. Para descriptografar dados criptografados abaixo de três AWS KMS keys, você pode especificar os mesmos três AWS KMS keys ou qualquer subconjunto deles. Este exemplo especifica somente um dos AWS KMS keys.

Para informar à CLI de AWS criptografia qual usar AWS KMS keys para descriptografar seus dados, use o atributo chave do parâmetro. `--wrapping-keys` Ao descriptografar com AWS KMS keys, o valor do atributo chave deve ser um ARN da chave.

Você deve ter permissão para chamar a API Decrypt no que você especificar. AWS KMS keys Para obter mais informações, consulte Autenticação e controle de acesso do AWS KMS.

Como prática recomendada, estes exemplos usam o parâmetro `--max-encrypted-data-keys` para evitar a descriptografia de uma mensagem malformada com um número excessivo de chaves de dados criptografadas. Embora o exemplo use somente uma chave de encapsulamento para decodificação, a mensagem criptografada tem três (3) chaves de dados criptografadas; uma

para cada uma das três chaves de encapsulamento usadas na criptografia. Especifique o número esperado de chaves de dados criptografadas ou um valor máximo razoável, como 5. Se especificar um valor máximo menor que 3, o comando falhará. Para obter detalhes, consulte Limitar as chaves de dados criptografadas.

Bash

```
$ aws-encryption-cli --decrypt --input /archive/finance.log \
                     --wrapping-keys key=$key1 \
                     --output /finance --suffix '.clear' \
                     --metadata-output ~/metadata \
                     --max-encrypted-data-keys 3 \
                     --buffer \
                     --encryption-context class=log
```

PowerShell

```
PS C:\> aws-encryption-cli --decrypt `
                           --input D:\Archive\Finance.log `
                           --wrapping-keys key=$key1 `
                           --output D:\Finance --suffix '.clear' `
                           --metadata-output .\Metadata\Metadata.txt `
                           --max-encrypted-data-keys 3 `
                           --buffer `
                           --encryption-context class=log
```

Criptografar e descriptografar em scripts

Este exemplo mostra como usar a CLI AWS de criptografia em scripts. Você pode escrever scripts que apenas criptografam e descriptografam dados ou scripts que criptografam ou descriptografam como parte de um processo de gerenciamento de dados.

Neste exemplo, o script obtém uma coleção de arquivos de log, compacta-os, criptografa-os e, em seguida, copia os arquivos criptografados em um bucket do Amazon S3. Esse script processa cada arquivo separadamente, para que você possa descriptografá-los e expandi-los de maneira independente.

Ao compactar e criptografar arquivos, certifique-se de compactar antes de criptografar. Dados criptografados corretamente não podem ser compactados.

> **Warning**
>
> Tenha cuidado ao compactar dados que incluam segredos e dados que possam ser controlados por um ator mal-intencionado. O tamanho final dos dados compactados pode revelar inadvertidamente informações confidenciais sobre seu conteúdo.

Bash

```
# Continue running even if an operation fails.
set +e

dir=$1
encryptionContext=$2
s3bucket=$3
s3folder=$4
masterKeyProvider="aws-kms"
metadataOutput="/tmp/metadata-$(date +%s)"

compress(){
    gzip -qf $1
}

encrypt(){
    # -e encrypt
    # -i input
    # -o output
    # --metadata-output unique file for metadata
    # -m masterKey read from environment variable
    # -c encryption context read from the second argument.
    # -v be verbose
    aws-encryption-cli -e -i ${1} -o $(dirname ${1}) --metadata-output
 ${metadataOutput} -m key="${masterKey}" provider="${masterKeyProvider}" -c
 "${encryptionContext}" -v
}


s3put (){
    # copy file argument 1 to s3 location passed into the script.
    aws s3 cp ${1} ${s3bucket}/${s3folder}
}
```

```
# Validate all required arguments are present.
if [ "${dir}" ] && [ "${encryptionContext}" ] && [ "${s3bucket}" ] &&
 [ "${s3folder}" ] && [ "${masterKey}" ]; then

# Is $dir a valid directory?
test -d "${dir}"
if [ $? -ne 0 ]; then
    echo "Input is not a directory; exiting"
    exit 1
fi

# Iterate over all the files in the directory, except *gz and *encrypted (in case of
 a re-run).
for f in $(find ${dir} -type f \( -name "*" ! -name \*.gz ! -name \*encrypted \) );
 do
    echo "Working on $f"
    compress ${f}
    encrypt ${f}.gz
    rm -f ${f}.gz
    s3put ${f}.gz.encrypted
done;
else
    echo "Arguments: <Directory> <encryption context> <s3://bucketname> <s3 folder>"
    echo " and ENV var \$masterKey must be set"
    exit 255
fi
```

PowerShell

```
#Requires -Modules AWSPowerShell, Microsoft.PowerShell.Archive
Param
(
    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String[]]
    $FilePath,

    [Parameter()]
    [Switch]
    $Recurse,

    [Parameter(Mandatory=$true)]
    [String]
```

```
    $wrappingKeyID,

    [Parameter()]
    [String]
    $masterKeyProvider = 'aws-kms',

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $ZipDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $EncryptDirectory,

    [Parameter()]
    [String]
    $EncryptionContext,

    [Parameter(Mandatory)]
    [ValidateScript({Test-Path $_})]
    [String]
    $MetadataDirectory,

    [Parameter(Mandatory)]
    [ValidateScript({Test-S3Bucket -BucketName $_})]
    [String]
    $S3Bucket,

    [Parameter()]
    [String]
    $S3BucketFolder
)

BEGIN {}
PROCESS {
    if ($files = dir $FilePath -Recurse:$Recurse)
    {

        # Step 1: Compress
        foreach ($file in $files)
        {
            $fileName = $file.Name
```

```
            try
            {
                Microsoft.PowerShell.Archive\Compress-Archive -Path $file.FullName -
DestinationPath $ZipDirectory\$filename.zip
            }
            catch
            {
                Write-Error "Zip failed on $file.FullName"
            }

            # Step 2: Encrypt
            if (-not (Test-Path "$ZipDirectory\$filename.zip"))
            {
                Write-Error "Cannot find zipped file: $ZipDirectory\$filename.zip"
            }
            else
            {
                # 2>&1 captures command output
                $err = (aws-encryption-cli -e -i "$ZipDirectory\$filename.zip" `
                                           -o $EncryptDirectory `
                                           -m key=$wrappingKeyID provider=
$masterKeyProvider `
                                           -c $EncryptionContext `
                                           --metadata-output $MetadataDirectory `
                                           -v) 2>&1

                # Check error status
                if ($? -eq $false)
                {
                    # Write the error
                    $err
                }
                elseif (Test-Path "$EncryptDirectory\$fileName.zip.encrypted")
                {
                    # Step 3: Write to S3 bucket
                    if ($S3BucketFolder)
                    {
                        Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted" -Key "$S3BucketFolder/
$fileName.zip.encrypted"

                    }
                    else
                    {
```

```
                        Write-S3Object -BucketName $S3Bucket -File
 "$EncryptDirectory\$fileName.zip.encrypted"
                    }
                }
            }
        }
    }
}
```

Usar o armazenamento em cache de chaves de dados

Este exemplo usa o armazenamento em cache de chaves de dados em um comando que criptografa um grande número de arquivos.

Por padrão, a CLI de AWS criptografia (e outras versões da AWS Encryption SDK) gera uma chave de dados exclusiva para cada arquivo criptografado. Embora o uso de uma chave de dados exclusiva para cada operação seja uma prática recomendada de criptografia, a reutilização limitada de chaves de dados é aceitável em algumas situações. Se você estiver considerando o armazenamento em cache de chaves de dados, consulte um engenheiro de segurança para compreender os requisitos de segurança do seu aplicativo e determinar os limites de segurança apropriados para você.

Neste exemplo, o armazenamento em cache de chaves de dados acelera a operação de criptografia reduzindo a frequência de solicitações ao provedor de chaves mestras.

O comando neste exemplo criptografa um diretório grande com vários subdiretórios que contêm um total de aproximadamente 800 pequenos arquivos de log. O primeiro comando salva o ARN da AWS KMS key em uma variável `keyARN`. O segundo comando criptografa todos os arquivos no diretório de entrada (recursivamente) e os grava em um diretório de arquivo morto. O comando usa o parâmetro `--suffix` para especificar o sufixo `.archive`.

O parâmetro `--caching` permite o armazenamento em cache da chave de dados. O atributo capacity, que limita o número de chaves de dados no cache, está definido como 1, porque o processamento de arquivos seriais nunca usa mais de uma chave de dados de cada vez. O atributo max_age, que determina por quanto tempo a chave de dados armazenada em cache pode ser usada, está definido como 10 segundos.

O atributo opcional max_messages_encrypted está definido como 10 mensagens, portanto, uma única chave de dados nunca é usada para criptografar mais de 10 arquivos. A limitação do número

de arquivos criptografados por cada chave de dados reduz o número de arquivos que devem ser afetados no caso improvável de uma chave de dados estar comprometida.

Para executar esse comando em arquivos de log gerados pelo sistema operacional, você pode precisar de permissões de administrador (`sudo` no Linux; Run as Administrator (Executar como administrador) no Windows).

Bash

```
$  keyArn=arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

$  aws-encryption-cli --encrypt \
                      --input /var/log/httpd --recursive \
                      --output ~/archive --suffix .archive \
                      --wrapping-keys key=$keyArn \
                      --encryption-context class=log \
                      --suppress-metadata \
                      --caching capacity=1 max_age=10 max_messages_encrypted=10
```

PowerShell

```
PS C:\> $keyARN = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

PS C:\> aws-encryption-cli --encrypt `
                           --input C:\Windows\Logs --recursive `
                           --output $home\Archive --suffix '.archive' `
                           --wrapping-keys key=$keyARN `
                           --encryption-context class=log `
                           --suppress-metadata `
                           --caching capacity=1 max_age=10
 max_messages_encrypted=10
```

Para testar o efeito do armazenamento em cache da chave de dados, este exemplo usa o cmdlet Measure-Command em. PowerShell Ao executar esse exemplo sem o armazenamento em cache da chave de dados, ele demora cerca de 25 segundos para ser concluído. Esse processo gera uma nova chave de dados para cada arquivo no diretório.

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
```

```
                                          --input C:\Windows\Logs --recursive `
                                          --output $home\Archive  --suffix '.archive'
 `
                                          --wrapping-keys key=$keyARN `
                                          --encryption-context class=log `
                                          --suppress-metadata }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 25
Milliseconds      : 453
Ticks             : 254531202
TotalDays         : 0.000294596298611111
TotalHours        : 0.00707031116666667
TotalMinutes      : 0.42421867
TotalSeconds      : 25.4531202
TotalMilliseconds : 25453.1202
```

O armazenamento em cache da chave de dados acelera o processo, mesmo quando você limita cada chave de dados para um máximo de 10 arquivos. O comando agora demora menos de 12 segundos para ser concluído e reduz o número de chamadas ao provedor de chaves mestras para 1/10 do valor original.

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                          --input C:\Windows\Logs --recursive `
                                          --output $home\Archive  --suffix '.archive'
 `
                                          --wrapping-keys key=$keyARN `
                                          --encryption-context class=log `
                                          --suppress-metadata `
                                          --caching capacity=1 max_age=10
 max_messages_encrypted=10}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 11
Milliseconds      : 813
Ticks             : 118132640
TotalDays         : 0.000136727592592593
```

```
TotalHours        : 0.00328146222222222
TotalMinutes      : 0.196887733333333
TotalSeconds      : 11.813264
TotalMilliseconds : 11813.264
```

Se você eliminar a restrição `max_messages_encrypted`, todos os arquivos serão criptografados com a mesma chave de dados. Essa alteração aumenta o risco de reutilização de chaves de dados sem tornar o processo muito mais rápido. No entanto, ela reduz o número de chamadas ao provedor de chaves mestras para 1.

```
PS C:\> Measure-Command {aws-encryption-cli --encrypt `
                                            --input C:\Windows\Logs --recursive `
                                            --output $home\Archive  --suffix '.archive'
 `
                                            --wrapping-keys key=$keyARN `
                                            --encryption-context class=log `
                                            --suppress-metadata `
                                            --caching capacity=1 max_age=10}


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 10
Milliseconds      : 252
Ticks             : 102523367
TotalDays         : 0.000118661304398148
TotalHours        : 0.00284787130555556
TotalMinutes      : 0.170872278333333
TotalSeconds      : 10.2523367
TotalMilliseconds : 10252.3367
```

AWS Encryption SDK Referência de sintaxe e parâmetros da CLI

Este tópico fornece diagramas da sintaxe e breves descrições dos parâmetros para ajudá-lo a usar a interface da linha de comando (CLI) do AWS Encryption SDK . Para obter ajuda com chaves mestras e outros parâmetros, consulte Como usar a CLI AWS de criptografia. Para obter exemplos, consulte Exemplos da CLI AWS de criptografia. Para obter a documentação completa, consulte Leia os documentos.

Tópicos

- AWS Sintaxe da CLI de criptografia
- AWS Parâmetros de linha de comando da CLI de criptografia
- Parâmetros avançados

AWS Sintaxe da CLI de criptografia

Esses diagramas de sintaxe da CLI de AWS criptografia mostram a sintaxe de cada tarefa que você executa com a CLI de criptografia. AWS Eles representam a sintaxe recomendada na versão 2.1 do AWS Encryption CLI. x e mais tarde.

Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

> **Note**
>
> A menos que indicado na descrição do parâmetro, cada parâmetro ou atributo pode ser usado apenas uma vez em cada comando.
> Se você usar um atributo que um parâmetro não suporta, a CLI de AWS criptografia ignora esse atributo não suportado sem um aviso ou erro.

Obter ajuda

Para obter a sintaxe completa da CLI de AWS criptografia com descrições de parâmetros, use ou. `--help -h`

```
aws-encryption-cli (--help | -h)
```

Obter a versão

Para obter o número da versão da sua instalação do AWS Encryption CLI, use. `--version` Certifique-se de incluir a versão ao fazer perguntas, relatar problemas ou compartilhar dicas sobre como usar a CLI de AWS criptografia.

```
aws-encryption-cli --version
```

Criptografar dados

O diagrama da sintaxe a seguir mostra os parâmetros usados por um comando encrypt.

```
aws-encryption-cli --encrypt
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite] [--suffix
 [<suffix>]] [--encode]
                   --wrapping-keys  [--wrapping-keys] ...
                       key=<keyID> [key=<keyID>] ...
                       [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                   --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
                   [--commitment-policy <commitment-policy>]
                   [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                   [--max-encrypted-data-keys <integer>]
                   [--algorithm <algorithm_suite>]
                   [--caching <attributes>]
                   [--frame-length <length>]
                   [-v | -vv | -vvv | -vvvv]
                   [--quiet]
```

Descriptografar dados

O diagrama da sintaxe a seguir mostra os parâmetros usados por um comando decrypt.

Na versão 1.8.x, o parâmetro `--wrapping-keys` é opcional ao descriptografar, mas é recomendado. A partir da versão 2.1.x, o parâmetro `--wrapping-keys` passou a ser necessário ao criptografar e descriptografar. Para AWS KMS keys, você pode usar o atributo key para especificar chaves de encapsulamento (prática recomendada) ou definir o atributo discovery como`true`, o que não limita as chaves de encapsulamento que podem ser usadas pela CLI de criptografia da AWS .

```
aws-encryption-cli --decrypt (or [--decrypt-unsigned])
                   --input <input> [--recursive] [--decode]
                   --output <output> [--interactive] [--no-overwrite]  [--suffix
 [<suffix>]] [--encode]
                   --wrapping-keys  [--wrapping-keys] ...
                       [key=<keyID>] [key=<keyID>] ...
                       [discovery={true|false}] [discovery-partition=<aws-partition-
name> discovery-account=<aws-account-ID> [discovery-account=<aws-account-ID>] ...]
```

```
                        [provider=<provider-name>] [region=<aws-region>]
 [profile=<aws-profile>]
                    --metadata-output <location> [--overwrite-metadata] | --suppress-
metadata]
                    [--commitment-policy <commitment-policy>]
                    [--encryption-context <encryption_context> [<encryption_context>
 ...]]
                    [--buffer]
                    [--max-encrypted-data-keys <integer>]
                    [--caching <attributes>]
                    [--max-length <length>]
                    [-v | -vv | -vvv | -vvvv]
                    [--quiet]
```

Usar arquivos de configuração

Você pode fazer referência a arquivos de configuração que contêm parâmetros e seus valores. Isso é equivalente a digitar os parâmetros e os valores no comando. Para ver um exemplo, consulte Como armazenar parâmetros em um arquivo de configuração.

```
aws-encryption-cli @<configuration_file>

# In a PowerShell console, use a backtick to escape the @.
aws-encryption-cli `@<configuration_file>
```

AWS Parâmetros de linha de comando da CLI de criptografia

Essa lista fornece uma descrição básica dos parâmetros do comando AWS Encryption CLI. Para obter uma descrição completa, consulte a aws-encryption-sdk-clidocumentação.

--encrypt (-e)

Criptografa os dados de entrada. Cada comando deve ter um parâmetro `--encrypt`, `--decrypt` ou `--decrypt-unsigned`.

--decrypt (-d)

Descriptografa os dados de entrada. Cada comando deve ter um parâmetro `--encrypt`, `--decrypt` ou `--decrypt-unsigned`.

--decrypt-unsigned [Introduzido nas versões 1.9.x e 2.2.x]

O parâmetro `--decrypt-unsigned` descriptografa o texto cifrado e garante que as mensagens não sejam assinadas antes de serem descriptografadas. Use esse parâmetro se você usou o parâmetro `--algorithm` e selecionou um pacote de algoritmos sem assinatura digital para criptografar dados. Se o texto cifrado for assinado, a descriptografia falhará.

Você pode usar `--decrypt` ou `--decrypt-unsigned` para fazer a descriptografia, mas não ambos.

--wrapping-keys (-w) [Introduzido na versão 1.8.x]

Especifica as chaves de encapsulamento (ou chaves mestras) usadas em operações de criptografia e descriptografia. Você pode usar vários parâmetros de `--wrapping-keys` em cada comando.

A partir da versão 2.1.x, o parâmetro `--wrapping-keys` passou a ser necessário ao criptografar e descriptografar comandos. Na versão 1.8. x, os comandos encrypt requerem um parâmetro `--wrapping-keys` ou `--master-keys`. Nos comandos decrypt fs versão 1.8. x um parâmetro `--wrapping-keys` é opcional, mas recomendado.

Ao usar usam um provedor de chaves mestres personalizado, os comandos encrypt e decrypt exigem os atributos key e provider. Ao usar AWS KMS keys, os comandos de criptografia exigem um atributo chave. Os comandos decrypt exigem que um atributokey ou um atributo discovery sejam definidos com um valor de `true` (mas não ambos). Usar o atributo key ao descriptografar é uma prática recomendada do AWS Encryption SDK. Ela particularmente importante se você estiver descriptografando lotes de mensagens desconhecidas, como aquelas em um bucket do Amazon S3 ou em uma fila do Amazon SQS.

Para ver um exemplo de como usar chaves AWS KMS multirregionais como chaves de agrupamento, consulte. Usando várias regiões AWS KMS keys

Attributes: o valor do parâmetro `--wrapping-keys` consiste nos seguintes atributos. O formato é `attribute_name=value`.

key

Identifica a chave de encapsulamento usada na operação. O formato é um par de key=ID. Você pode especificar vários atributos key em cada valor do parâmetro `--wrapping-keys`.

- Comandos encrypt: todos os comandos encrypt exigem o atributo key. Quando você usa um comando AWS KMS key in a encrypt, o valor do atributo chave pode ser um ID de chave,

ARN de chave, nome de alias ou ARN de alias. Para obter descrições dos identificadores de AWS KMS chave, consulte Identificadores de chave no Guia do AWS Key Management Service desenvolvedor.

- Comandos decrypt: ao descriptografar com as AWS KMS keys, o parâmetro `--wrapping-keys` exige que o valor de um atributokey seja definido como um ARN de chave ou que o valor de um atributo discovery seja definido como `true` (mas não ambos). Usar o atributo key é uma prática recomendada do AWS Encryption SDK. Ao descriptografar com um provedor de chave mestra personalizado, o atributo key é obrigatório.

  > **Note**
  >
  > Para especificar uma chave de AWS KMS encapsulamento em um comando de descriptografia, o valor do atributo chave deve ser um ARN de chave. Se você usar um ID de chave, nome de alias ou ARN de alias, a AWS CLI de criptografia não reconhecerá a chave de empacotamento.

Você pode especificar vários atributos key em cada valor do parâmetro `--wrapping-keys`. No entanto, qualquer atributo provider, region e profile em um parâmetro `--wrapping-keys` será aplicável a todas chaves de encapsulamento no valor desse parâmetro. Para especificar chaves de encapsulamento com diferentes valores de atributos, use vários parâmetros `--wrapping-keys` no comando.

discovery

Permite que a CLI de AWS criptografia use qualquer uma para AWS KMS key descriptografar a mensagem. O valor de discovery pode ser `true` ou`false`. O valor padrão é `false`. O atributo discovery é válido apenas em comandos decrypt e somente quando o provedor de chaves mestras for do AWS KMS.

Ao descriptografar com AWS KMS keys, o `--wrapping-keys` parâmetro requer um atributo-chave ou um atributo de descoberta com um valor de `true` (mas não ambos). Se você usar o atributo key, poderá usar um atributo de discovery com um valor definido como `false` para rejeitar explicitamente a descoberta.

- `False`(padrão) — Quando o atributo de descoberta não é especificado ou seu valor é`false`, a CLI de AWS criptografia descriptografa a mensagem usando somente o AWS KMS keys especificado pelo atributo-chave do parâmetro. `--wrapping-keys` Se você não especificar um atributo key quando discovery for `false`, o comando decrypt falhará. Esse valor oferece suporte a uma prática AWS recomendada de CLI de criptografia.

- `True`— Quando o valor do atributo de descoberta é`true`, a CLI de AWS criptografia obtém os metadados AWS KMS keys da mensagem criptografada e os usa AWS KMS keys para descriptografar a mensagem. O atributo de descoberta com um valor de `true` se comporta como as versões da AWS CLI de criptografia antes da versão 1.8. x que não permitia que você especificasse uma chave de empacotamento ao descriptografar. No entanto, sua intenção de usar qualquer um AWS KMS key é explícita. Se você especificar um atributo key quando discovery for `true`, o comando decrypt falhará.

  O `true` valor pode fazer com que a CLI de AWS criptografia seja usada AWS KMS keys em diferentes Contas da AWS regiões ou tente usar algo AWS KMS keys que o usuário não esteja autorizado a usar.

Quando a descoberta é`true`, é uma prática recomendada usar os atributos discovery-partition e discovery-account para limitar o AWS KMS keys uso aos atributos especificados por você. Contas da AWS

discovery-account

Limita o AWS KMS keys usado para descriptografia aos especificados. Conta da AWS O único valor válido para esse atributo é um ID de Conta da AWS.

Esse atributo é opcional e válido somente em comandos de descriptografia com os AWS KMS keys quais o atributo de descoberta está definido `true` e o atributo de partição de descoberta é especificado.

Cada atributo da conta descoberta usa apenas uma Conta da AWS ID, mas você pode especificar vários atributos da conta descoberta no mesmo parâmetro. `--wrapping-keys` Todas as contas especificadas em um determinado parâmetro `--wrapping-keys` devem estar na partição da AWS especificada.

discovery-partition

Especifica a AWS partição das contas no atributo discovery-account. Seu valor deve ser uma AWS partiçãoaws, comoaws-cn, ouaws-gov-cloud. Para obter mais informações, consulte Nomes de atributo da Amazon no Referência geral da AWS.

Esse atributo é obrigatório quando você usa o atributo discovery-account. Você pode especificar somente um atributo discovery-partition em cada parâmetro `--wrapping keys`. Para especificar Contas da AWS em várias partições, use um `--wrapping-keys` parâmetro adicional.

provider

Identifica o provedor de chaves mestres. O formato é um par de provider=ID. O valor padrão, aws-kms, representa. AWS KMS Esse atributo é necessário somente quando o provedor da chave mestra não é AWS KMS.

region

Identifica o Região da AWS de um AWS KMS key. Esse atributo é válido somente para AWS KMS keys. É usado apenas quando o identificador da chave não especifica uma região; caso contrário, é ignorado. Quando usado, ele substitui a região padrão no perfil chamado AWS CLI.

perfil

Identifica um perfil AWS CLI nomeado. Esse atributo é válido somente para AWS KMS keys. A região no perfil é usada apenas quando o identificador da chave não especifica uma região e não há nenhum atributo region no comando.

--input (-i)

Especifica o local dos dados a serem criptografados ou descriptografados. Esse parâmetro é obrigatório. O valor pode ser um caminho para um arquivo ou diretório ou um nome de arquivo padrão. Se você estiver redirecionando a entrada para o comando (stdin), use `-`.

Se a entrada não existir, o comando é concluído com êxito sem erro ou aviso.

--recursive (-r, -R)

Executa a operação nos arquivos no diretório de entrada e em seus subdiretórios. Esse parâmetro é necessário quando o valor de `--input` é um diretório.

--decode

Decodifica entrada codificada em Base64.

Se estiver descriptografando uma mensagem que foi criptografada e, em seguida, codificado, você deverá decodificar a mensagem antes de descriptografá-la. Esse parâmetro faz isso para você.

Por exemplo, se você tiver usado o parâmetro `--encode` em um comando encrypt, use o parâmetro `--decode` no comando decrypt correspondente. Você também pode usar esse parâmetro para decodificar a entrada codificada em Base64 antes de criptografá-la.

--output (-o)

Especifica um destino para a saída. Esse parâmetro é obrigatório. O valor pode ser um nome de arquivo, um diretório existente ou `-`, que grava a saída na linha de comando (stdout).

Se o diretório de saída especificado não existir, o comando falhará. Se a entrada contiver subdiretórios, a AWS CLI de criptografia reproduzirá os subdiretórios no diretório de saída que você especificar.

Por padrão, a CLI AWS de criptografia sobrescreve arquivos com o mesmo nome. Para alterar esse comportamento, use os parâmetros `--interactive` ou `--no-overwrite`. Para suprimir o aviso de substituição, use o parâmetro `--quiet`.

> **Note**
>
> Se um comando que deve substituir um arquivo de saída falhar, o arquivo de saída será excluído.

--interactive

Solicita antes de substituir o arquivo.

--no-overwrite

Não substitui arquivos. Em vez disso, se o arquivo de saída existir, a CLI de AWS criptografia ignora a entrada correspondente.

--sufixo

Especifica um sufixo de nome de arquivo personalizado para arquivos criados pela AWS CLI de criptografia. Para indicar nenhum sufixo, use o parâmetro sem um valor (`--suffix`).

Por padrão, quando o parâmetro `--output` não especifica um nome de arquivo, o nome do arquivo de saída tem o mesmo nome que o nome do arquivo de entrada mais o sufixo. O sufixo para comandos encrypt é `.encrypted`. O sufixo para comandos decrypt é `.decrypted`.

--encode

Aplica codificação de Base64 (de binário para texto) à saída. A codificação impede que o programa shell do host interprete incorretamente caracteres não ASCII no texto de saída.

Use esse parâmetro ao gravar uma saída criptografada em stdout (`--output -`), especialmente em um PowerShell console, mesmo quando estiver canalizando a saída para outro comando ou salvando-a em uma variável.

--metadata-output

Especifica um local para metadados sobre as operações de criptografia. Insira um caminho e um nome de arquivo. Se o diretório não existir, o comando falhará. Para gravar os metadados na linha de comando (stdout), use `-`.

Você não pode gravar a saída do comando (`--output`) e a saída dos metadados (`--metadata-output`) em stdout no mesmo comando. Além disso, quando o valor de `--input` ou `--output` for um diretório (sem nomes de arquivos), você não poderá gravar a saída de metadados no mesmo diretório ou em qualquer subdiretório desse diretório.

Se você especificar um arquivo existente, por padrão, a CLI de AWS criptografia anexará novos registros de metadados a qualquer conteúdo do arquivo. Esse recurso permite que você crie um único arquivo que contém os metadados de todas as suas operações de criptografia. Para substituir o conteúdo em um arquivo existente, use o parâmetro `--overwrite-metadata`.

A CLI de AWS criptografia retorna um registro de metadados formatado em JSON para cada operação de criptografia ou descriptografia que o comando executa. Cada registro de metadados inclui os caminhos completos para os arquivos de entrada e de saída, o contexto de criptografia, o pacote de algoritmos e outras informações valiosas que você pode usar para rever a operação e verificar se ela atende a seus padrões de segurança.

--overwrite-metadata

Substitui o conteúdo no arquivo de saída de metadados. Por padrão, o parâmetro `--metadata-output` acrescenta metadados a qualquer conteúdo existente no arquivo.

--suppress-metadata (-S)

Suprime os metadados sobre a operação de criptografia ou de descriptografia.

--commitment-policy

Especifica a política de compromisso para comandos encrypt e decrypt. A política de compromisso determina se sua mensagem será criptografada e descriptografada com o atributo de segurança confirmação de chave.

O parâmetro `--commitment-policy` foi introduzido na versão 1.8.x.. Ele 'é válido em comandos de criptografia e descriptografia.

Na versão 1.8. x, a CLI de AWS criptografia usa a política de `forbid-encrypt-allow-decrypt` compromisso para todas as operações de criptografia e descriptografia. Quando você usa o parâmetro `--wrapping-keys` em um comando encrypt ou decrypt, é obrigatório que um parâmetro `--commitment-policy` seja definido com o valor `forbid-encrypt-allow-decrypt`. Se você não usar o parâmetro`--wrapping-keys`, o parâmetro `--commitment-policy` será inválido. Definir uma política de compromisso explicitamente impede que sua política de compromisso seja alterada automaticamente para `require-encrypt-require-decrypt` quando você atualizar para a versão 2.1.x

A partir da versão 2.1.x, todos os valores da política de compromisso são compatíveis. O parâmetro `--commitment-policy` é opcional e o valor padrão é `require-encrypt-require-decrypt`.

Esse parâmetro tem os valores a seguir:

- `forbid-encrypt-allow-decrypt`: não é possível criptografar com confirmação de chave. Ele pode descriptografar textos cifrados criptografados com ou sem confirmação de chave.

  Na versão 1.8.x, esse é o único valor válido. A CLI de AWS criptografia usa a política de `forbid-encrypt-allow-decrypt` compromisso para todas as operações de criptografia e descriptografia.
- `require-encrypt-allow-decrypt`: criptografa somente com confirmação de chave. Descriptografa com e sem compromisso chave. Esse valor foi introduzido na versão 2.1.x..
- `require-encrypt-require-decrypt` (padrão): criptografa e descriptografa somente com confirmação de chave. Esse valor foi introduzido na versão 2.1.x.. É o valor padrão em versões 2.1.x. e posteriores. Com esse valor, a CLI de AWS criptografia não descriptografará nenhum texto cifrado que tenha sido criptografado com versões anteriores do. AWS Encryption SDK

Para obter informações detalhadas sobre como definir sua política de compromisso, consulte Migrando seu AWS Encryption SDK.

--encryption-context (-c)

Especifica um contexto de criptografia para a operação. Esse parâmetro não é obrigatório, mas é recomendado.

- Em um comando `--encrypt`, insira um ou mais pares de `name=value`. Use espaços para separar os pares.
- Em um comando `--decrypt`, insira pares de `name=value`, elementos `name` sem valores ou ambos.

Se o `name` ou o `value` em um par de `name=value` incluir espaços ou caracteres especiais, coloque o par inteiro entre aspas. Por exemplo, `.--encryption-context "department=software development"`

--buffer (-b) [Introduzido nas versões 1.9.x e 2.2.x]

Retorna texto simples somente após o processamento de todas as entradas, incluindo a verificação da assinatura digital, se houver uma.

-- max-encrypted-data-keys [Introduzido nas versões 1.9. x e 2.2. x]

Especifica o número máximo de chaves de dados criptografadas em uma mensagem criptografada. Esse parâmetro é opcional.

Os valores válidos são 1–65.535. Se você omitir esse parâmetro, a CLI de AWS criptografia não impõe nenhum máximo. Uma mensagem criptografada pode conter até 65.535 (2^16 - 1) chaves de dados criptografadas.

Você pode usar esse parâmetro em comandos encrypt para evitar a malformação de uma mensagem. Você pode usá-lo em comandos decrypt para detectar mensagens maliciosas e evitar descriptografar mensagens com várias chaves de dados criptografadas que você não pode descriptografar. Para obter detalhes e um exemplo, consulte Limitar as chaves de dados criptografadas.

--help (-h)

Imprime o uso e a sintaxe na linha de comando.

--version

Obtém a versão da CLI AWS de criptografia.

-v | -vv | -vvv | -vvvv

Exibe informações, avisos e mensagens de depuração detalhados. Os detalhes na saída aumentam com o número de `v`s no parâmetro. A configuração mais detalhada (`-vvvv`) retorna dados em nível de depuração da AWS CLI de criptografia e de todos os componentes que ela usa.

--quiet (-q)

Suprime mensagens de aviso, como a mensagem que aparece quando você substitui um arquivo de saída.

--master-keys (-m) [Descontinuado]

> **Note**
>
> O parâmetro --master-keys foi descontinuado na versão 1.8.x foi removido na versão 2.1.x. Em vez dele, use o parâmetro --wrapping-keys.

Especifica as chaves mestres usadas em operações de criptografia e descriptografia. Você pode usar vários parâmetros de chaves mestras em cada comando.

O parâmetro `--master-keys` é necessário em comandos encrypt. Ele é necessário em comandos decrypt somente quando você estiver usando um provedor de chaves mestras personalizado (que não seja do AWS KMS).

Attributes: o valor do parâmetro `--master-keys` consiste nos seguintes atributos. O formato é `attribute_name=value`.

key

Identifica a chave de encapsulamento usada na operação. O formato é um par de key=ID. O atributo key é obrigatório em todos os comandos encrypt.

Quando você usa um comando AWS KMS key in a encrypt, o valor do atributo chave pode ser um ID de chave, ARN de chave, nome de alias ou ARN de alias. Para obter detalhes sobre identificadores de AWS KMS chave, consulte Identificadores de chave no Guia do AWS Key Management Service desenvolvedor.

O atributo key é necessário em comandos decrypt quando o provedor de chaves mestras não for o AWS KMS. O atributo key não é permitido em comandos que descriptografam dados que foram criptografados com uma AWS KMS key.

Você pode especificar vários atributos key em cada valor do parâmetro `--master-keys`. No entanto, qualquer atributo provider, region e profile aplica-se a todas as chaves mestres no valor do parâmetro. Para especificar chaves mestras com diferentes valores de atributos, use vários parâmetros `--master-keys` no comando.

provider

Identifica o provedor de chaves mestres. O formato é um par de provider=ID. O valor padrão, aws-kms, representa. AWS KMS Esse atributo é necessário somente quando o provedor da chave mestra não é AWS KMS.

region

Identifica o Região da AWS de um AWS KMS key. Esse atributo é válido somente para AWS KMS keys. É usado apenas quando o identificador da chave não especifica uma região; caso contrário, é ignorado. Quando usado, ele substitui a região padrão no perfil chamado AWS CLI.

perfil

Identifica um perfil AWS CLI nomeado. Esse atributo é válido somente para AWS KMS keys. A região no perfil é usada apenas quando o identificador da chave não especifica uma região e não há nenhum atributo region no comando.

Parâmetros avançados

--algorithm

Especifica um pacote de algoritmos alternativo. Esse parâmetro é opcional e válido apenas em comandos encrypt.

Se você omitir esse parâmetro, a CLI de AWS criptografia usará um dos conjuntos de algoritmos padrão para AWS Encryption SDK o apresentado na versão 1.8. x. Ambos os algoritmos padrão usam o AES-GCM com um HKDF, uma assinatura ECDSA e uma chave de criptografia de 256 bits. Um usa confirmação de chave; o outro não. A escolha do pacote de algoritmos padrão é determinada pela política de compromisso do comando.

Os pacotes de algoritmo padrão são recomendados para a maioria das operações de criptografia. Para obter uma lista de valores válidos, consulte os valores do parâmetro `algorithm` em Leia os documentos.

--frame-length

Cria uma saída com o tamanho da moldura especificado. Esse parâmetro é opcional e válido apenas em comandos encrypt.

Digite um valor em bytes. Os valores válidos são 0 e 1–2^31-1. Um valor igual a 0 indica dados sem moldura. O padrão é 4.096 (bytes).

> **Note**
>
> Sempre que possível, use dados com moldura. O AWS Encryption SDK suporta dados não emoldurados somente para uso antigo. Algumas implementações de linguagem

do ainda AWS Encryption SDK podem gerar texto cifrado sem moldura. Todas as implementações de linguagem compatíveis podem descriptografar texto cifrado e não emoldurado.

--max-length

Indica o tamanho máximo da moldura (ou o tamanho máximo do conteúdo de mensagens sem moldura) a ser lido em mensagens criptografadas. Esse parâmetro é opcional e válido apenas em comandos decrypt. Ele foi projetado para proteção contra a descriptografia de texto cifrado mal-intencionado extremamente grande.

Digite um valor em bytes. Se você omitir esse parâmetro, o AWS Encryption SDK não limitará o tamanho do quadro ao descriptografar.

--caching

Habilita o recurso de armazenamento em cache de chaves de dados, que reutiliza chaves de dados, em vez de gerar uma nova chave de dados para cada arquivo de entrada. Esse parâmetro é compatível com um cenário avançado. Não deixe de ler a documentação Armazenamento em cache de chaves de dados antes de usar esse recurso.

O parâmetro `--caching` tem os seguintes atributos.

capacity (obrigatório)

Determina o número máximo de entradas no cache.

O valor mínimo é 1. Não há um valor máximo.

max_age (obrigatório)

Determina o tempo em que as entradas do cache são usadas, em segundos, a partir do momento em que são adicionadas ao cache.

Digite um valor maior que 0. Não há um valor máximo.

max_messages_encrypted (opcional)

Determina o número máximo de mensagens que uma entrada armazenada em cache pode criptografar.

Os valores válidos são 1–2^32. O valor padrão é 2^32 (mensagens).

max_bytes_encrypted (opcional)

Determina o número máximo de bytes que uma entrada armazenada em cache pode criptografar.

Os valores válidos são 0 e 1–2^63 - 1. O valor padrão é 2^63 - 1 (mensagens). Um valor de 0 permite usar armazenamento em cache de chaves de dados somente quando você está criptografando strings de mensagem vazias.

Versões da CLI AWS de criptografia

Recomendamos que você use a versão mais recente da CLI de AWS criptografia.

> **Note**
>
> Versões da CLI de AWS criptografia anteriores à 4.0.0 estão em fase. end-of-support
> Você pode atualizar com segurança a partir da versão 2.1.x e posteriores até a versão mais recente da CLI de criptografia da AWS sem realizar alterações no código ou nos dados. No entanto, os novos atributos de segurança introduzidos na versão 2.1.x não são compatíveis com versões anteriores. Para atualizar a partir da versão 1.7. x ou anterior, você deve primeiro atualizar para a última 1. versão x da CLI AWS de criptografia. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.
> Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

Para obter informações sobre versões significativas do AWS Encryption SDK, consulteVersões do AWS Encryption SDK.

Qual versão devo usar?

Se você é novo na CLI AWS de criptografia, use a versão mais recente.

Para descriptografar dados criptografados por uma versão AWS Encryption SDK anterior à 1.7. x, migre primeiro para a versão mais recente da CLI de AWS criptografia. Faça todas as alterações recomendadas antes de atualizar para a versão 2.1.x ou versões posteriores. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

Saiba mais

- Para obter informações detalhadas sobre as alterações e orientações para migrar para essas novas versões, consulte Migrando seu AWS Encryption SDK.
- Para obter descrições dos novos parâmetros e atributos da CLI de AWS criptografia, consulte. AWS Encryption SDK Referência de sintaxe e parâmetros da CLI

As listas a seguir descrevem a alteração na CLI de AWS criptografia nas versões 1.8. x e 2.1. x.

Versão 1.8. x mudanças na CLI AWS de criptografia

- Descontinua o parâmetro `--master-keys`. Em vez disso, use o parâmetro `--wrapping-keys`.
- Adiciona o parâmetro `--wrapping-keys` (`-w`). Compatível com todos os atributos do parâmetro `--master-keys`. Também adiciona os seguintes atributos opcionais, que são válidos somente ao descriptografar com AWS KMS keys.
  - discovery
  - discovery-partition
  - discovery-account

  Para provedores de chaves mestras personalizadas, os comandos `--encrypt` e`--decrypt` exigem um parâmetro `--wrapping-keys` ou um parâmetro `--master-keys` (mas não ambos). Além disso, um `--encrypt` comando com AWS KMS keys requer um `--wrapping-keys` parâmetro ou um `--master-keys` parâmetro (mas não ambos).

  Em um `--decrypt` comando com AWS KMS keys, o `--wrapping-keys` parâmetro é opcional, mas recomendado, pois é obrigatório na versão 2.1. x. Se você usá-lo, deverá especificar o key ou o atributo discovery com um valor definido como `true` (mas não ambos).
- Adiciona o parâmetro `--commitment-policy`. O único valor válido é `forbid-encrypt-allow-decrypt`. A política de compromisso `forbid-encrypt-allow-decrypt` é usada em todos os comandos encrypt e decrypt.

  Na versão 1.8.x, quando você usa o parâmetro `--wrapping-keys`, é necessário definir um parâmetro `--commitment-policy` com o valor `forbid-encrypt-allow-decrypt`. Definir o valor explicitamente impede que sua política de compromisso seja alterada automaticamente para `require-encrypt-require-decrypt` quando você atualizar para a versão 2.1.x.

Versão 2.1. x mudanças na CLI AWS de criptografia

- Remove o parâmetro `--master-keys`. Em vez disso, use o parâmetro `--wrapping-keys`.
- O parâmetro `--wrapping-keys` é obrigatório em comandos encrypt. Você deve especificar o atributo key ou o atributo discovery com um valor definido como `true` (mas não ambos).
- O parâmetro `--commitment-policy` oferece suporte aos seguintes valores: Para obter detalhes, consulte Como definir sua política de compromisso.
  - `forbid-encrypt-allow-decrypt`
  - `require-encrypt-allow-decrypt`
  - `require-encrypt-require decrypt` (padrão)
- O parâmetro `--commitment-policy` é opcional na versão 2.1.x.. O valor padrão é `require-encrypt-require-decrypt`.

Versão 1.9. x e 2.2. x mudanças na CLI AWS de criptografia

- Adiciona o parâmetro `--decrypt-unsigned`. Para obter detalhes, consulte Versão 2.2.x.
- Adiciona o parâmetro `--buffer`. Para obter detalhes, consulte Versão 2.2.x.
- Adiciona o parâmetro `--max-encrypted-data-keys`. Para obter detalhes, consulte Limitar as chaves de dados criptografadas.

Versão 3.0. x mudanças na CLI AWS de criptografia

- Adiciona suporte para chaves AWS KMS multirregionais. Para obter mais detalhes, consulte Usando várias regiões AWS KMS keys.

Armazenamento em cache de chaves de dados

O armazenamento em cache de chaves de dados armazena chaves de dados e o material criptográfico relacionado em um cache. Quando você criptografa ou descriptografa dados, ele AWS Encryption SDK procura uma chave de dados correspondente no cache. Se encontrar uma correspondência, ele usará a chave de dados armazenada em cache em vez de gerar uma nova. O armazenamento em cache de chaves de dados pode melhorar o desempenho, reduzir os custos e ajudar você a manter os limites do serviço à medida que seu aplicativo é escalado.

O aplicativo poderá se beneficiar do armazenamento em cache de chaves de dados se:

- Puder reutilizar chaves de dados.
- Gerar várias chaves de dados.
- As operações de criptografia estiverem inaceitavelmente lentas, caras, limitadas ou usarem recursos de forma intensiva.

O armazenamento em cache pode reduzir o uso de serviços criptográficos, como AWS Key Management Service ()AWS KMS. Se você está atingindo seu AWS KMS requests-per-secondlimite, o armazenamento em cache pode ajudar. Seu aplicativo pode usar chaves em cache para atender a algumas de suas solicitações de chave de dados em vez de chamar AWS KMS. (Você também pode criar um caso no AWS Support Center para aumentar o limite da conta.)

AWS Encryption SDK Isso ajuda você a criar e gerenciar seu cache de chaves de dados. Ele fornece um cache local e um gerenciador de armazenamento em cache de materiais criptográficos (CMM de armazenamento em cache) que interage com o cache e impõe limites de segurança definidos por você. Juntos, esses componentes ajudam você a se beneficiar da eficiência de reutilização de chaves de dados mantendo a segurança do sistema.

O armazenamento em cache da chave de dados é um recurso opcional do AWS Encryption SDK que você deve usar com cautela. Por padrão, AWS Encryption SDK gera uma nova chave de dados para cada operação de criptografia. Essa técnica é compatível com as melhores práticas criptográficas, que desencorajam a reutilização excessiva de chaves de dados. Em geral, use o armazenamento em cache de chaves de dados somente quando ele for necessário para atender às suas metas de desempenho. Em seguida, use os limites de segurança do armazenamento em cache de chaves de dados para garantir que você use a quantidade mínima de armazenamento em cache necessário para atender a suas metas de desempenho e custo.

Versão 3. x of the suporta AWS Encryption SDK for Java apenas o CMM de armazenamento em cache com a interface antiga de provedores de chaves mestras, não a interface de chaveiro. No entanto, versão 4. x e versões posteriores do AWS Encryption SDK para .NET, versão 3. x do AWS Encryption SDK for Java, versão 4. x do AWS Encryption SDK for Python, versão 1. x do AWS Encryption SDK para Rust e versão 0.1. x ou versões posteriores do AWS Encryption SDK for Go suportam o AWS KMS chaveiro hierárquico, uma solução alternativa de cache de materiais criptográficos. O conteúdo criptografado com o AWS KMS chaveiro hierárquico só pode ser descriptografado com o chaveiro hierárquico. AWS KMS

Para ver uma discussão detalhada dessas vantagens e desvantagens de segurança, consulte AWS Encryption SDK: Como decidir se o armazenamento em cache de chaves de dados é ideal para sua aplicação no Blog de segurança da AWS .

Tópicos

- Como usar o armazenamento em cache de chaves de dados
- Definir limites de segurança do cache
- Detalhes do armazenamento em cache de chaves de dados
- Exemplo de armazenamento em cache de chaves de dados

Como usar o armazenamento em cache de chaves de dados

Este tópico mostra como usar o armazenamento em cache de chaves de dados em seu aplicativo. Ele fornece uma demonstração passo a passo do processo. Em seguida, ele combina as etapas em um exemplo simples que usa o armazenamento em cache da chave de dados em uma operação para criptografar uma string.

Esses exemplos mostram como usar a versão 2.0.x e versões posteriores do AWS Encryption SDK. Para exemplos que usam versões anteriores, encontre sua versão na lista de Lançamentos do GitHub repositório da sua linguagem de programação.

> **Note**
>
> Se seu aplicativo usa a Biblioteca de Provedores de Material AWS Criptográfico (MPL), o chaveiro AWS KMS hierárquico pode servir como uma alternativa para muitos casos de uso que hoje dependem do armazenamento em cache de chaves de dados. O chaveiro hierárquico não é um CMM de cache de chave de dados, mas ele armazena materiais criptográficos localmente e reduz as chamadas de maneira semelhante. AWS KMS

Considere isso quando você já estiver usando o MPL e quiser minimizar as AWS KMS chamadas.

Para ver exemplos completos e testados do uso do cache de chaves de dados no AWS Encryption SDK, consulte:

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript Navegador: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

O AWS Encryption SDK para .NET não oferece é compatível com o cache de chaves de dados.

Tópicos

- Usando o cache de chaves de dados: Step-by-step
- Armazenamento em cache de chaves de dados de exemplo: criptografar uma string

Usando o cache de chaves de dados: Step-by-step

Estas instruções passo a passo mostram como criar os componentes necessários para implementar o armazenamento em cache de chaves de dados.

- Crie um cache de chave de dados. Nesses exemplos, usamos o cache local que ele AWS Encryption SDK fornece. Limitamos o cache a 10 chaves de dados.

  C

```
// Cache capacity (maximum number of entries) is required
size_t cache_capacity = 10;
struct aws_allocator *allocator = aws_default_allocator();

struct aws_cryptosdk_materials_cache *cache =
 aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);
```

Java

O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java obsoleto o CMM de cache da chave de dados. Com a versão 3. x, você também pode usar o chaveiro AWS KMS hierárquico, uma solução alternativa de armazenamento em cache de materiais criptográficos.

```
// Cache capacity (maximum number of entries) is required
int MAX_CACHE_SIZE = 10;

CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(MAX_CACHE_SIZE);
```

JavaScript Browser

```
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

JavaScript Node.js

```
const capacity = 10

const cache = getLocalCryptographicMaterialsCache(capacity)
```

Python

```
# Cache capacity (maximum number of entries) is required
MAX_CACHE_SIZE = 10

cache = aws_encryption_sdk.LocalCryptoMaterialsCache(MAX_CACHE_SIZE)
```

- Crie um provedor de chave mestra (Java e Python) ou um chaveiro (C e JavaScript). Esses exemplos usam um provedor de chave mestra AWS Key Management Service (AWS KMS) ou um AWS KMS chaveiro compatível.

C

```
// Create an AWS KMS keyring
//   The input is the Amazon Resource Name (ARN)
//   of an AWS KMS key
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_arn);
```

Java

O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java obsoleto o CMM de cache da chave de dados. Com a versão 3. x, você também pode usar o chaveiro AWS KMS hierárquico, uma solução alternativa de armazenamento em cache de materiais criptográficos.

```
// Create an AWS KMS master key provider
//   The input is the Amazon Resource Name (ARN)
//   of an AWS KMS key
MasterKeyProvider<KmsMasterKey> keyProvider =
 KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn);
```

JavaScript Browser

No navegador, você deve injetar suas credenciais com segurança. Este exemplo define credenciais em um webpack (kms.webpack.config) que resolve credenciais no runtime. Ele cria uma instância de provedor de AWS KMS cliente a partir de um AWS KMS cliente e das credenciais. Em seguida, ao criar o chaveiro, ele passa o provedor do cliente para o construtor junto com o AWS KMS key (. `generatorKeyId`)

```
const { accessKeyId, secretAccessKey, sessionToken } = credentials

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

/*  Create an AWS KMS keyring
 *  You must configure the AWS KMS keyring with at least one AWS KMS key
```

```
*  The input is the Amazon Resource Name (ARN)
 */ of an AWS KMS key
const keyring = new KmsKeyringBrowser({
    clientProvider,
    generatorKeyId,
    keyIds,
  })
```

JavaScript Node.js

```
/* Create an AWS KMS keyring
 *   The input is the Amazon Resource Name (ARN)
*/   of an AWS KMS key
const keyring = new KmsKeyringNode({ generatorKeyId })
```

Python

```
# Create an AWS KMS master key provider
#  The input is the Amazon Resource Name (ARN)
#  of an AWS KMS key
key_provider =
 aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])
```

- Crie um gerenciador de armazenamento em cache de materiais criptográficos (CMM de armazenamento em cache).

  Associe o CMM de armazenamento em cache ao seu cache e seu provedor de chaves mestras. Em seguida, defina os limites de segurança do cache no CMM de armazenamento em cache.

  C

  No AWS Encryption SDK for C, você pode criar uma CMM de cache a partir de uma CMM subjacente, como a CMM padrão, ou de um chaveiro. Este exemplo cria o CMM de armazenamento em cache de um token de autenticação.

Depois de criar o CMM de armazenamento em cache, você pode liberar suas referências para o token de autenticação e o cache. Para obter detalhes, consulte the section called “Contagem de referências”.

```
// Create the caching CMM
//   Set the partition ID to NULL.
//   Set the required maximum age value to 60 seconds.
struct aws_cryptosdk_cmm *caching_cmm =
 aws_cryptosdk_caching_cmm_new_from_keyring(allocator, cache, kms_keyring, NULL,
 60, AWS_TIMESTAMP_SECS);

// Add an optional message threshold
//   The cached data key will not be used for more than 10 messages.
aws_status = aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, 10);

// Release your references to the cache and the keyring.
aws_cryptosdk_materials_cache_release(cache);
aws_cryptosdk_keyring_release(kms_keyring);
```

Java

O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x of the AWS Encryption SDK for Java não oferece suporte ao cache de chaves de dados, mas oferece suporte ao chaveiro AWS KMS hierárquico, uma solução alternativa de armazenamento em cache de materiais criptográficos.

```
/*
 * Security thresholds
 *   Max entry age is required.
 *   Max messages (and max bytes) per entry are optional
 */
int MAX_ENTRY_AGE_SECONDS = 60;
int MAX_ENTRY_MSGS = 10;

//Create a caching CMM
CryptoMaterialsManager cachingCmm =
    CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                                 .withCache(cache)
                                 .withMaxAge(MAX_ENTRY_AGE_SECONDS,
 TimeUnit.SECONDS)
                                 .withMessageUseLimit(MAX_ENTRY_MSGS)
```

```
                                        .build();
```

JavaScript Browser

```
/*
 * Security thresholds
 *   Max age (in milliseconds) is required.
 *   Max messages (and max bytes) per entry are optional.
 */
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring  */
const cachingCmm = new WebCryptoCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
})
```

JavaScript Node.js

```
/*
 * Security thresholds
 *   Max age (in milliseconds) is required.
 *   Max messages (and max bytes) per entry are optional.
 */
const maxAge = 1000 * 60
const maxMessagesEncrypted = 10

/* Create a caching CMM from a keyring  */
const cachingCmm = new NodeCachingMaterialsManager({
  backingMaterials: keyring,
  cache,
  maxAge,
  maxMessagesEncrypted
})
```

Python

```
# Security thresholds
#   Max entry age is required.
```

```
#   Max messages (and max bytes) per entry are optional
#
MAX_ENTRY_AGE_SECONDS = 60.0
MAX_ENTRY_MESSAGES = 10

# Create a caching CMM
caching_cmm = CachingCryptoMaterialsManager(
    master_key_provider=key_provider,
    cache=cache,
    max_age=MAX_ENTRY_AGE_SECONDS,
    max_messages_encrypted=MAX_ENTRY_MESSAGES
)
```

Isso é tudo o que você precisa fazer. Em seguida, deixe que AWS Encryption SDK eles gerenciem o cache para você ou adicione sua própria lógica de gerenciamento de cache.

Quando desejar usar o armazenamento em cache de chaves de dados em uma chamada para criptografar ou descriptografar dados, especifique o CMM de armazenamento em cache em vez de especificar um provedor de chaves mestras ou outro CMM.

> **Note**
>
> Se estiver criptografando streamings de dados ou quaisquer dados de tamanho desconhecido, certifique-se de especificar o tamanho dos dados na solicitação. O AWS Encryption SDK não usa cache de chave de dados ao criptografar dados de tamanho desconhecido.

C

No AWS Encryption SDK for C, você cria uma sessão com o CMM de cache e depois processa a sessão.

Por padrão, quando o tamanho da mensagem é desconhecido e ilimitado, o AWS Encryption SDK não armazena em cache as chaves de dados. Para permitir o armazenamento em cache quando não se sabe o tamanho exato dos dados, use o método `aws_cryptosdk_session_set_message_bound` para definir o tamanho máximo da mensagem. Defina o vínculo maior do que o tamanho estimado da mensagem. Se o tamanho real da mensagem exceder o vínculo, ocorrerá uma falha na operação da criptografia.

```
/* Create a session with the caching CMM. Set the session mode to encrypt. */
struct aws_cryptosdk_session *session =
 aws_cryptosdk_session_new_from_cmm_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
 caching_cmm);

/* Set a message bound of 1000 bytes */
aws_status = aws_cryptosdk_session_set_message_bound(session, 1000);

/* Encrypt the message using the session with the caching CMM */
aws_status = aws_cryptosdk_session_process(
             session, output_buffer, output_capacity, &output_produced,
 input_buffer, input_len, &input_consumed);

/* Release your references to the caching CMM and the session. */
aws_cryptosdk_cmm_release(caching_cmm);
aws_cryptosdk_session_destroy(session);
```

Java

O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java obsoleto o CMM de cache da chave de dados. Com a versão 3. x, você também pode usar o chaveiro AWS KMS hierárquico, uma solução alternativa de armazenamento em cache de materiais criptográficos.

```
// When the call to encryptData specifies a caching CMM,
// the encryption operation uses the data key cache
final AwsCrypto encryptionSdk = AwsCrypto.standard();
return encryptionSdk.encryptData(cachingCmm, plaintext_source).getResult();
```

JavaScript Browser

```
const { result } = await encrypt(cachingCmm, plaintext)
```

JavaScript Node.js

Quando você usa a CMM de cache no AWS Encryption SDK para JavaScript for Node.js, o `encrypt` método exige o tamanho do texto sem formatação. Se você não fornecer, a chave de dados não será armazenada em cache. Se você fornecer um tamanho, mas os dados de texto simples fornecidos excederem esse tamanho, a operação de criptografia falhará. Se você não souber o tamanho exato do texto simples, como quando estiver fazendo streaming de dados, forneça o maior valor esperado.

```
const { result } = await encrypt(cachingCmm, plaintext, { plaintextLength:
 plaintext.length })
```

Python

```
# Set up an encryption client
client = aws_encryption_sdk.EncryptionSDKClient()

# When the call to encrypt specifies a caching CMM,
# the encryption operation uses the data key cache
#
encrypted_message, header = client.encrypt(
    source=plaintext_source,
    materials_manager=caching_cmm
)
```

Armazenamento em cache de chaves de dados de exemplo: criptografar uma string

Este código de exemplo simples usa o armazenamento em cache de chaves de dados ao criptografar uma string. Ela combina o código do procedimento passo a passo no código de teste que você pode executar.

O exemplo cria um cache local e um provedor de chave mestra ou token de autenticação para uma AWS KMS key. Em seguida, ele usa o cache local e o provedor de chaves mestras ou o token de autenticação para criar um CMM de armazenamento em cache com os limites de segurança adequados. Em Java e em Python, a solicitação de criptografia especifica o CMM de armazenamento em cache, os dados de texto simples a serem criptografados e um contexto de criptografia. Em C, o CMM de armazenamento em cache é especificado na sessão, e a sessão é fornecida para a solicitação de criptografia.

Para executar estes exemplos, você precisa fornecer o nome do atributo da Amazon (ARN) de uma AWS KMS key. Verifique se você tem permissão para usar a AWS KMS key para gerar uma chave de dados.

Para obter exemplos reais mais detalhados de como criar e usar um cache de chave mestra, consulte Exemplo de código de armazenamento em cache de chaves de dados.

C

```
/*
 * Copyright 2019 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use
 * this file except in compliance with the License. A copy of the License is
 * located at
 *
 *     http://aws.amazon.com/apache2.0/
 *
 * or in the "license" file accompanying this file. This file is distributed on an
 * "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 * implied. See the License for the specific language governing permissions and
 * limitations under the License.
 */

#include <aws/cryptosdk/cache.h>
#include <aws/cryptosdk/cpp/kms_keyring.h>
#include <aws/cryptosdk/session.h>

void encrypt_with_caching(
    uint8_t *ciphertext,     // output will go here (assumes ciphertext_capacity
 bytes already allocated)
    size_t *ciphertext_len,  // length of output will go here
    size_t ciphertext_capacity,
    const char *kms_key_arn,
    int max_entry_age,
    int cache_capacity) {
    const uint64_t MAX_ENTRY_MSGS = 100;

    struct aws_allocator *allocator = aws_default_allocator();

    // Load error strings for debugging
    aws_cryptosdk_load_error_strings();

    // Create a keyring
    struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(kms_key_arn);

    // Create a cache
    struct aws_cryptosdk_materials_cache *cache =
 aws_cryptosdk_materials_cache_local_new(allocator, cache_capacity);
```

```
   // Create a caching CMM
   struct aws_cryptosdk_cmm *caching_cmm =
aws_cryptosdk_caching_cmm_new_from_keyring(
       allocator, cache, kms_keyring, NULL, max_entry_age, AWS_TIMESTAMP_SECS);
   if (!caching_cmm) abort();

   if (aws_cryptosdk_caching_cmm_set_limit_messages(caching_cmm, MAX_ENTRY_MSGS))
abort();

   // Create a session
   struct aws_cryptosdk_session *session =
       aws_cryptosdk_session_new_from_cmm_2(allocator, AWS_CRYPTOSDK_ENCRYPT,
caching_cmm);
   if (!session) abort();

   // Encryption context
   struct aws_hash_table *enc_ctx =
aws_cryptosdk_session_get_enc_ctx_ptr_mut(session);
   if (!enc_ctx) abort();
   AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_key, "purpose");
   AWS_STATIC_STRING_FROM_LITERAL(enc_ctx_value, "test");
   if (aws_hash_table_put(enc_ctx, enc_ctx_key, (void *)enc_ctx_value, NULL))
abort();

   // Plaintext data to be encrypted
   const char *my_data = "My plaintext data";
   size_t my_data_len  = strlen(my_data);
   if (aws_cryptosdk_session_set_message_size(session, my_data_len)) abort();

   // When the session uses a caching CMM, the encryption operation uses the data
key cache
   // specified in the caching CMM.
   size_t bytes_read;
   if (aws_cryptosdk_session_process(
           session,
           ciphertext,
           ciphertext_capacity,
           ciphertext_len,
           (const uint8_t *)my_data,
           my_data_len,
           &bytes_read))
       abort();
   if (!aws_cryptosdk_session_is_done(session) || bytes_read != my_data_len)
abort();
```

```
    aws_cryptosdk_session_destroy(session);
    aws_cryptosdk_cmm_release(caching_cmm);
    aws_cryptosdk_materials_cache_release(cache);
    aws_cryptosdk_keyring_release(kms_keyring);
}
```

Java

O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java obsoleto o CMM de cache da chave de dados. Com a versão 3. x, você também pode usar o chaveiro AWS KMS hierárquico, uma solução alternativa de armazenamento em cache de materiais criptográficos.

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

package com.amazonaws.crypto.examples;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CryptoMaterialsManager;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.CryptoMaterialsCache;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import java.nio.charset.StandardCharsets;
import java.util.Collections;
import java.util.Map;
import java.util.concurrent.TimeUnit;

/**
 * <p>
 * Encrypts a string using an &KMS; key and data key caching
 *
 * <p>
 * Arguments:
 * <ol>
 * <li>KMS Key ARN: To find the Amazon Resource Name of your &KMS; key,
 *     see 'Find the key ID and ARN' at https://docs.aws.amazon.com/kms/latest/
developerguide/find-cmk-id-arn.html
 * <li>Max entry age: Maximum time (in seconds) that a cached entry can be used
```

```
 * <li>Cache capacity: Maximum number of entries in the cache
 * </ol>
 */
public class SimpleDataKeyCachingExample {

    /*
     * Security thresholds
     *   Max entry age is required.
     *   Max messages (and max bytes) per data key are optional
     */
    private static final int MAX_ENTRY_MSGS = 100;

    public static byte[] encryptWithCaching(String kmsKeyArn, int maxEntryAge, int
 cacheCapacity) {
        // Plaintext data to be encrypted
        byte[] myData = "My plaintext data".getBytes(StandardCharsets.UTF_8);

        // Encryption context
        // Most encrypted data should have an associated encryption context
        // to protect integrity. This sample uses placeholder values.
        // For more information see:
        // blogs.aws.amazon.com/security/post/Tx2LZ6WBJJANTNW/How-to-Protect-the-
Integrity-of-Your-Encrypted-Data-by-Using-AWS-Key-Management
        final Map<String, String> encryptionContext =
 Collections.singletonMap("purpose", "test");

        // Create a master key provider
        MasterKeyProvider<KmsMasterKey> keyProvider =
 KmsMasterKeyProvider.builder()
            .buildStrict(kmsKeyArn);

        // Create a cache
        CryptoMaterialsCache cache = new LocalCryptoMaterialsCache(cacheCapacity);

        // Create a caching CMM
        CryptoMaterialsManager cachingCmm =

 CachingCryptoMaterialsManager.newBuilder().withMasterKeyProvider(keyProvider)
                .withCache(cache)
                .withMaxAge(maxEntryAge, TimeUnit.SECONDS)
                .withMessageUseLimit(MAX_ENTRY_MSGS)
                .build();

        // When the call to encryptData specifies a caching CMM,
```

```
        // the encryption operation uses the data key cache
        final AwsCrypto encryptionSdk = AwsCrypto.standard();
        return encryptionSdk.encryptData(cachingCmm, myData,
 encryptionContext).getResult();
    }
}
```

JavaScript Browser

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

/* This is a simple example of using a caching CMM with a KMS keyring
 * to encrypt and decrypt using the AWS Encryption SDK for Javascript in a browser.
 */

import {
  KmsKeyringBrowser,
  KMS,
  getClient,
  buildClient,
  CommitmentPolicy,
  WebCryptoCachingMaterialsManager,
  getLocalCryptographicMaterialsCache,
} from '@aws-crypto/client-browser'
import { toBase64 } from '@aws-sdk/util-base64-browser'

/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
 * which enforces that this client only encrypts using committing algorithm suites
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

/* This is injected by webpack.
 * The webpack.DefinePlugin or @aws-sdk/karma-credential-loader will replace the
 values when bundling.
```

```
 * The credential values are pulled from @aws-sdk/credential-provider-node
 * Use any method you like to get credentials into the browser.
 * See kms.webpack.config
 */
declare const credentials: {
  accessKeyId: string
  secretAccessKey: string
  sessionToken: string
}

/* This is done to facilitate testing. */
export async function testCachingCMMExample() {
  /* This example uses an &KMS; keyring. The generator key in a &KMS; keyring
 generates and encrypts the data key.
   * The caller needs kms:GenerateDataKey permission on the &KMS; key in
 generatorKeyId.
   */
  const generatorKeyId =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* Adding additional KMS keys that can decrypt.
   * The caller must have kms:Encrypt permission for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
   * In `generatorId`, this &KMS; key is identified by its alias ARN.
   * In `keyIds`, this &KMS; key is identified by its key ARN.
   * In practice, you would specify different &KMS; keys,
   * or omit the `keyIds` parameter.
   * This is *only* to demonstrate how the &KMS; key ARNs are configured.
   */
  const keyIds = [
    'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
  ]

  /* Need a client provider that will inject correct credentials.
   * The credentials here are injected by webpack from your environment bundle is
 created
   * The credential values are pulled using @aws-sdk/credential-provider-node.
   * See kms.webpack.config
   * You should inject your credential into the browser in a secure manner
   * that works with your application.
   */
```

```
 const { accessKeyId, secretAccessKey, sessionToken } = credentials

 /* getClient takes a KMS client constructor
  * and optional configuration values.
  * The credentials can be injected here,
  * because browsers do not have a standard credential discovery process the way
Node.js does.
  */
 const clientProvider = getClient(KMS, {
   credentials: {
     accessKeyId,
     secretAccessKey,
     sessionToken,
   },
 })

 /* You must configure the KMS keyring with your &KMS; keys */
 const keyring = new KmsKeyringBrowser({
   clientProvider,
   generatorKeyId,
   keyIds,
 })

 /* Create a cache to hold the data keys (and related cryptographic material).
  * This example uses the local cache provided by the Encryption SDK.
  * The `capacity` value represents the maximum number of entries
  * that the cache can hold.
  * To make room for an additional entry,
  * the cache evicts the oldest cached entry.
  * Both encrypt and decrypt requests count independently towards this threshold.
  * Entries that exceed any cache threshold are actively removed from the cache.
  * By default, the SDK checks one item in the cache every 60 seconds (60,000
milliseconds).
  * To change this frequency, pass in a `proactiveFrequency` value
  * as the second parameter. This value is in milliseconds.
  */
 const capacity = 100
 const cache = getLocalCryptographicMaterialsCache(capacity)

 /* The partition name lets multiple caching CMMs share the same local
cryptographic cache.
  * By default, the entries for each CMM are cached separately. However, if you
want these CMMs to share the cache,
  * use the same partition name for both caching CMMs.
```

```
  * If you don't supply a partition name, the Encryption SDK generates a random
name for each caching CMM.
  * As a result, sharing elements in the cache MUST be an intentional operation.
  */
 const partition = 'local partition name'

 /* maxAge is the time in milliseconds that an entry will be cached.
  * Elements are actively removed from the cache.
  */
 const maxAge = 1000 * 60

 /* The maximum number of bytes that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest practical value.
  */
 const maxBytesEncrypted = 100

 /* The maximum number of messages that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest practical value.
  */
 const maxMessagesEncrypted = 10

 const cachingCMM = new WebCryptoCachingMaterialsManager({
   backingMaterials: keyring,
   cache,
   partition,
   maxAge,
   maxBytesEncrypted,
   maxMessagesEncrypted,
 })

 /* Encryption context is a *very* powerful tool for controlling
  * and managing access.
  * When you pass an encryption context to the encrypt function,
  * the encryption context is cryptographically bound to the ciphertext.
  * If you don't pass in the same encryption context when decrypting,
  * the decrypt function fails.
  * The encryption context is ***not*** secret!
  * Encrypted data is opaque.
  * You can use an encryption context to assert things about the encrypted data.
  * The encryption context helps you to determine
  * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
  * For example, if you are are only expecting data from 'us-west-2',
```

```
   * the appearance of a different AWS Region in the encryption context can indicate
 malicious interference.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/
concepts.html#encryption-context
   *
   * Also, cached data keys are reused ***only*** when the encryption contexts
 passed into the functions are an exact case-sensitive match.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
caching-details.html#caching-encryption-context
   */
  const encryptionContext = {
    stage: 'demo',
    purpose: 'simple demonstration app',
    origin: 'us-west-2',
  }

  /* Find data to encrypt. */
  const plainText = new Uint8Array([1, 2, 3, 4, 5])

  /* Encrypt the data.
   * The caching CMM only reuses data keys
   * when it know the length (or an estimate) of the plaintext.
   * However, in the browser,
   * you must provide all of the plaintext to the encrypt function.
   * Therefore, the encrypt function in the browser knows the length of the
 plaintext
   * and does not accept a plaintextLength option.
   */
  const { result } = await encrypt(cachingCMM, plainText, { encryptionContext })

  /* Log the plain text
   * only for testing and to show that it works.
   */
  console.log('plainText:', plainText)
  document.write('</br>plainText:' + plainText + '</br>')

  /* Log the base64-encoded result
   * so that you can try decrypting it with another AWS Encryption SDK
 implementation.
   */
  const resultBase64 = toBase64(result)
  console.log(resultBase64)
  document.write(resultBase64)
```

```
  /* Decrypt the data.
   * NOTE: This decrypt request will not use the data key
   * that was cached during the encrypt operation.
   * Data keys for encrypt and decrypt operations are cached separately.
   */
  const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

  /* Grab the encryption context so you can verify it. */
  const { encryptionContext: decryptedContext } = messageHeader

  /* Verify the encryption context.
   * If you use an algorithm suite with signing,
   * the Encryption SDK adds a name-value pair to the encryption context that
 contains the public key.
   * Because the encryption context might contain additional key-value pairs,
   * do not include a test that requires that all key-value pairs match.
   * Instead, verify that the key-value pairs that you supplied to the `encrypt`
 function are included in the encryption context that the `decrypt` function
 returns.
   */
  Object.entries(encryptionContext).forEach(([key, value]) => {
    if (decryptedContext[key] !== value)
      throw new Error('Encryption Context does not match expected values')
  })

  /* Log the clear message
   * only for testing and to show that it works.
   */
  document.write('</br>Decrypted:' + plaintext)
  console.log(plaintext)

  /* Return the values to make testing easy. */
  return { plainText, plaintext }
}
```

JavaScript Node.js

```
// Copyright Amazon.com Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import {
  KmsKeyringNode,
  buildClient,
```

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

/* This builds the client with the REQUIRE_ENCRYPT_REQUIRE_DECRYPT commitment
 policy,
 * which enforces that this client only encrypts using committing algorithm suites
 * and enforces that this client
 * will only decrypt encrypted messages
 * that were created with a committing algorithm suite.
 * This is the default commitment policy
 * if you build the client with `buildClient()`.
 */
const { encrypt, decrypt } = buildClient(
  CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT
)

export async function cachingCMMNodeSimpleTest() {
  /* An &KMS; key is required to generate the data key.
   * You need kms:GenerateDataKey permission on the &KMS; key in generatorKeyId.
   */
  const generatorKeyId =
    'arn:aws:kms:us-west-2:658956600833:alias/EncryptDecrypt'

  /* Adding alternate &KMS; keys that can decrypt.
   * Access to kms:Encrypt is required for every &KMS; key in keyIds.
   * You might list several keys in different AWS Regions.
   * This allows you to decrypt the data in any of the represented Regions.
   * In this example, the generator key
   * and the additional key are actually the same &KMS; key.
   * In `generatorId`, this &KMS; key is identified by its alias ARN.
   * In `keyIds`, this &KMS; key is identified by its key ARN.
   * In practice, you would specify different &KMS; keys,
   * or omit the `keyIds` parameter.
   * This is *only* to demonstrate how the &KMS; key ARNs are configured.
   */
  const keyIds = [
    'arn:aws:kms:us-west-2:658956600833:key/b3537ef1-d8dc-4780-9f5a-55776cbb2f7f',
  ]

  /* The &KMS; keyring must be configured with the desired &KMS; keys
   * This example passes the keyring to the caching CMM
   * instead of using it directly.
```

```
  */
 const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

 /* Create a cache to hold the data keys (and related cryptographic material).
  * This example uses the local cache provided by the Encryption SDK.
  * The `capacity` value represents the maximum number of entries
  * that the cache can hold.
  * To make room for an additional entry,
  * the cache evicts the oldest cached entry.
  * Both encrypt and decrypt requests count independently towards this threshold.
  * Entries that exceed any cache threshold are actively removed from the cache.
  * By default, the SDK checks one item in the cache every 60 seconds (60,000
milliseconds).
  * To change this frequency, pass in a `proactiveFrequency` value
  * as the second parameter. This value is in milliseconds.
  */
 const capacity = 100
 const cache = getLocalCryptographicMaterialsCache(capacity)

 /* The partition name lets multiple caching CMMs share the same local
cryptographic cache.
  * By default, the entries for each CMM are cached separately. However, if you
want these CMMs to share the cache,
  * use the same partition name for both caching CMMs.
  * If you don't supply a partition name, the Encryption SDK generates a random
name for each caching CMM.
  * As a result, sharing elements in the cache MUST be an intentional operation.
  */
 const partition = 'local partition name'

 /* maxAge is the time in milliseconds that an entry will be cached.
  * Elements are actively removed from the cache.
  */
 const maxAge = 1000 * 60

 /* The maximum amount of bytes that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest value possible.
  */
 const maxBytesEncrypted = 100

 /* The maximum number of messages that will be encrypted under a single data key.
  * This value is optional,
  * but you should configure the lowest value possible.
```

```
   */
  const maxMessagesEncrypted = 10

  const cachingCMM = new NodeCachingMaterialsManager({
    backingMaterials: keyring,
    cache,
    partition,
    maxAge,
    maxBytesEncrypted,
    maxMessagesEncrypted,
  })

  /* Encryption context is a *very* powerful tool for controlling
   * and managing access.
   * When you pass an encryption context to the encrypt function,
   * the encryption context is cryptographically bound to the ciphertext.
   * If you don't pass in the same encryption context when decrypting,
   * the decrypt function fails.
   * The encryption context is ***not*** secret!
   * Encrypted data is opaque.
   * You can use an encryption context to assert things about the encrypted data.
   * The encryption context helps you to determine
   * whether the ciphertext you retrieved is the ciphertext you expect to decrypt.
   * For example, if you are are only expecting data from 'us-west-2',
   * the appearance of a different AWS Region in the encryption context can indicate
 malicious interference.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/
concepts.html#encryption-context
   *
   * Also, cached data keys are reused ***only*** when the encryption contexts
 passed into the functions are an exact case-sensitive match.
   * See: https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/data-
caching-details.html#caching-encryption-context
   */
  const encryptionContext = {
    stage: 'demo',
    purpose: 'simple demonstration app',
    origin: 'us-west-2',
  }

  /* Find data to encrypt.  A simple string. */
  const cleartext = 'asdf'

  /* Encrypt the data.
```

```
  * The caching CMM only reuses data keys
  * when it know the length (or an estimate) of the plaintext.
  * If you do not know the length,
  * because the data is a stream
  * provide an estimate of the largest expected value.
  *
  * If your estimate is smaller than the actual plaintext length
  * the AWS Encryption SDK will throw an exception.
  *
  * If the plaintext is not a stream,
  * the AWS Encryption SDK uses the actual plaintext length
  * instead of any length you provide.
  */
 const { result } = await encrypt(cachingCMM, cleartext, {
   encryptionContext,
   plaintextLength: 4,
 })

 /* Decrypt the data.
  * NOTE: This decrypt request will not use the data key
  * that was cached during the encrypt operation.
  * Data keys for encrypt and decrypt operations are cached separately.
  */
 const { plaintext, messageHeader } = await decrypt(cachingCMM, result)

 /* Grab the encryption context so you can verify it. */
 const { encryptionContext: decryptedContext } = messageHeader

 /* Verify the encryption context.
  * If you use an algorithm suite with signing,
  * the Encryption SDK adds a name-value pair to the encryption context that
contains the public key.
  * Because the encryption context might contain additional key-value pairs,
  * do not include a test that requires that all key-value pairs match.
  * Instead, verify that the key-value pairs that you supplied to the `encrypt`
function are included in the encryption context that the `decrypt` function
returns.
  */
 Object.entries(encryptionContext).forEach(([key, value]) => {
   if (decryptedContext[key] !== value)
     throw new Error('Encryption Context does not match expected values')
 })

 /* Return the values so the code can be tested. */
```

```
  return { plaintext, result, cleartext, messageHeader }
}
```

Python

```
# Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License"). You
# may not use this file except in compliance with the License. A copy of
# the License is located at
#
# http://aws.amazon.com/apache2.0/
#
# or in the "license" file accompanying this file. This file is
# distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF
# ANY KIND, either express or implied. See the License for the specific
# language governing permissions and limitations under the License.
"""Example of encryption with data key caching."""
import aws_encryption_sdk
from aws_encryption_sdk import CommitmentPolicy


def encrypt_with_caching(kms_key_arn, max_age_in_cache, cache_capacity):
    """Encrypts a string using an &KMS; key and data key caching.

    :param str kms_key_arn: Amazon Resource Name (ARN) of the &KMS; key
    :param float max_age_in_cache: Maximum time in seconds that a cached entry can
 be used
    :param int cache_capacity: Maximum number of entries to retain in cache at once
    """
    # Data to be encrypted
    my_data = "My plaintext data"

    # Security thresholds
    #   Max messages (or max bytes per) data key are optional
    MAX_ENTRY_MESSAGES = 100

    # Create an encryption context
    encryption_context = {"purpose": "test"}

    # Set up an encryption client with an explicit commitment policy. Note that if
 you do not explicitly choose a
    # commitment policy, REQUIRE_ENCRYPT_REQUIRE_DECRYPT is used by default.
```

```
    client =
aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.REQUIRE_ENCRYPT_R

    # Create a master key provider for the &KMS; key
    key_provider =
aws_encryption_sdk.StrictAwsKmsMasterKeyProvider(key_ids=[kms_key_arn])

    # Create a local cache
    cache = aws_encryption_sdk.LocalCryptoMaterialsCache(cache_capacity)

    # Create a caching CMM
    caching_cmm = aws_encryption_sdk.CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=max_age_in_cache,
        max_messages_encrypted=MAX_ENTRY_MESSAGES,
    )

    # When the call to encrypt data specifies a caching CMM,
    # the encryption operation uses the data key cache specified
    # in the caching CMM
    encrypted_message, _header = client.encrypt(
        source=my_data, materials_manager=caching_cmm,
encryption_context=encryption_context
    )

    return encrypted_message
```

Definir limites de segurança do cache

Quando você implementa o armazenamento em cache de chave de dados, precisa configurar os limites de segurança impostos pelo CMM de armazenamento em cache.

Os limites de segurança ajudam a limitar duração do uso de cada chave de dados e o volume de dados protegido em cada chave de dados. O CMM de armazenamento em cache retorna as chaves de dados armazenadas em cache somente quando a entrada do cache estiver em conformidade com todos os limites de segurança. Se a entrada do cache exceder o limite, ela não será usada para a operação atual e será removida do cache assim que possível. O primeiro uso de cada chave de dados (antes do armazenamento em cache) é isento desses limites.

Como regra, use a quantidade mínima de armazenamento em cache necessária para atender a suas metas de custos e de desempenho.

O AWS Encryption SDK único armazena em cache as chaves de dados que são criptografadas usando uma função de derivação de chave. Além disso, ele estabelece limites máximos para alguns dos valores de limites. Essas restrições garantem que as chaves de dados não sejam reutilizadas além dos limites criptográficos. No entanto, como as chaves de dados de texto sem criptografia são armazenadas em cache (na memória, por padrão), tente minimizar o tempo em que as chaves são salvas. Além disso, tente limitar os dados que poderão ser expostos se uma chave estiver comprometida.

Para obter exemplos de como definir limites de segurança de cache, consulte AWS Encryption SDK: Como decidir se o armazenamento em cache de chaves de dados é adequado para seu aplicativo no blog de AWS segurança.

> **Note**
>
> O CMM do armazenamento em cache impõe todos os limites a seguir. Se você não especificar um valor opcional, o CMM de armazenamento em cache usará o valor padrão. Para desativar temporariamente o armazenamento em cache de chaves de dados, as implementações de Java e Python do AWS Encryption SDK fornecem um cache de materiais criptográficos nulo (cache nulo). O cache nulo retorna um erro para cada solicitação `GET` e não responde a solicitações `PUT`. Recomendamos usar o cache nulo em vez de definir a capacidade do cache ou os limites de segurança como 0. Para obter mais informações, consulte o cache nulo em Java e Python.

Idade máxima (obrigatório)

Determina por quanto tempo uma entrada armazenada em cache pode ser usada, a partir do momento em que foi adicionada. Este valor é obrigatório. Digite um valor maior que 0. AWS Encryption SDK Isso não limita o valor máximo de idade.

Todas as implementações de linguagem do AWS Encryption SDK definem a idade máxima em segundos, exceto a AWS Encryption SDK para JavaScript, que usa milissegundos.

Use o intervalo mais curto que ainda permita que seu aplicativo se beneficie do cache. Você pode usar o limite máximo de idade como uma política de rotação de chaves. Use-o para limitar a reutilização de chaves de dados, minimizar a exposição de material criptográfico e remover

chaves de dados cujas políticas podem ter sido alteradas enquanto estavam armazenadas em cache.

Número máximo de mensagens criptografadas (opcional)

Especifica o número máximo de mensagens que uma chave de dados armazenada em cache pode criptografar. Este valor é opcional. Digite um valor entre 1 e 2^32 mensagens. O valor padrão é 2^32 mensagens.

Defina o número de mensagens protegidas por cada chave armazenada em cache para que seja grande o suficiente para obter o valor da reutilização, mas pequeno o suficiente para limitar o número de mensagens que podem ser expostas se uma chave for comprometida.

Número máximo de bytes criptografados (opcional)

Especifica o número máximo de bytes que uma chave de dados armazenada em cache pode criptografar. Este valor é opcional. Digite um valor entre 0 e 2^63 - 1. O valor padrão é 2^63 - 1. Um valor de 0 permite usar armazenamento em cache de chaves de dados somente quando você está criptografando strings de mensagem vazias.

Os bytes na solicitação atual são incluídos ao avaliar esse limite. Se os bytes processados, mais os bytes atuais, excederem o limite, a chave de dados armazenada em cache será removida do cache, mesmo que ela tenha sido usada em uma solicitação menor.

Detalhes do armazenamento em cache de chaves de dados

A maioria dos aplicativos pode usar a implementação padrão do armazenamento em cache de chave de dados sem escrever código personalizado. Esta seção descreve a implementação padrão e alguns detalhes sobre as opções.

Tópicos

- Como o armazenamento em cache de chaves de dados funciona
- Criar um cache de material de criptografia
- Criar um gerenciador de material de criptografia de armazenamento em cache
- O que é uma entrada de chave de dados em cache?
- Contexto de criptografia: como selecionar entradas do cache
- Meu aplicativo está usando chaves de dados armazenadas em cache?

Como o armazenamento em cache de chaves de dados funciona

Quando você usa o armazenamento em cache de chave de dados em uma solicitação para criptografar ou descriptografar dados, o AWS Encryption SDK primeiro pesquisa uma chave de dados no cache que corresponde à solicitação. Se localizar uma correspondência válida, ele usa a chave de dados armazenada em cache para criptografar os dados. Caso contrário, ele gerará uma nova chave de dados, da mesma forma como o faria sem o cache.

O armazenamento em cache da chave de dados não é usado para dados de tamanho desconhecido, como streaming de dados. Isso permite que o CMM de armazenamento em cache imponha o limite máximo de bytes corretamente. Para evitar esse comportamento, adicione o tamanho da mensagem à solicitação de criptografia.

Além de um cache, o armazenamento em cache de chaves de dados usa um gerenciador de armazenamento em cache de materiais criptográficos (CMM de armazenamento em cache). O CMM de armazenamento em cache é um gerenciador de materiais criptográficos (CMM) especializado que interage com um cache e um CMM subjacente. (Quando você especifica um provedor de chaves mestra ou um token de autenticação, o AWS Encryption SDK cria um CMM padrão para você.) O CMM de armazenamento em cache armazena em cache as chaves de dados que seu CMM subjacente retorna. Também impõe limites de segurança de cache definidos por você.

Para evitar que a chave de dados incorreta seja selecionada no cache, todos os CMMs de armazenamento em cache compatíveis exigem que as seguintes propriedades dos materiais criptográficos em cache correspondam à solicitação de material.

- Pacote de algoritmos
- Contexto de criptografia (mesmo quando vazio)
- Nome da partição (uma string que identifica o CMM de armazenamento em cache)
- (Somente descriptografia) chaves de dados criptografadas

> **Note**
>
> O AWS Encryption SDK cache das chaves de dados somente quando o conjunto de algoritmos usa uma função de derivação de chave.

Os seguintes fluxos de trabalho mostram como uma solicitação para criptografar dados é processada com e sem armazenamento em cache da chave de dados. Eles mostram como o armazenamento

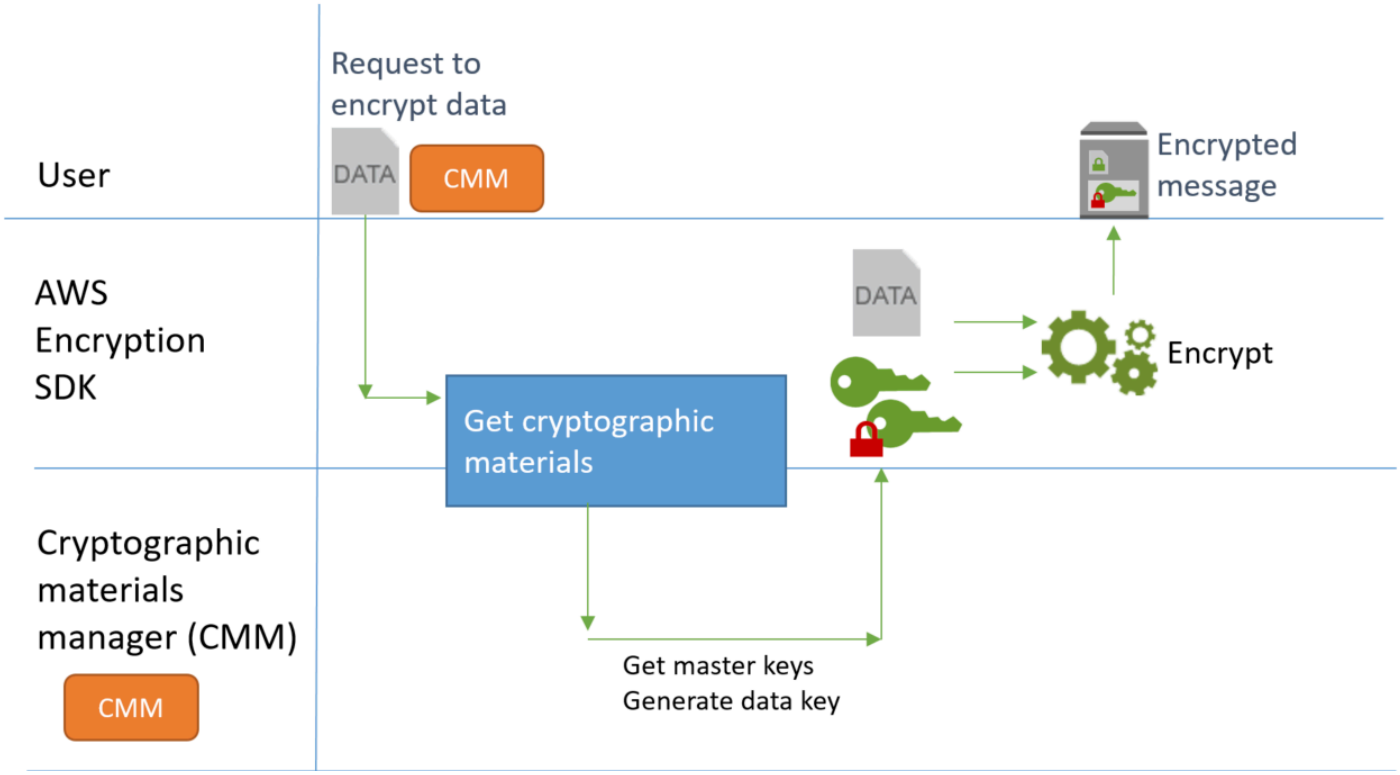
em cache de componentes que você cria, incluindo o cache e o CMM de armazenamento em cache, são usados no processo.

Criptografar dados sem armazenamento em cache

Para obter materiais de criptografia sem armazenamento em cache:

1. Um aplicativo solicita que eles AWS Encryption SDK criptografem os dados.

   A solicitação especifica um provedor de chaves mestres ou um token de autenticação. O AWS Encryption SDK cria um CMM padrão que interage com a chave mestra ou com o token de autenticação.
2. AWS Encryption SDK Ele solicita ao CMM materiais de criptografia (obtenha materiais criptográficos).
3. A CMM solicita materiais criptográficos a seu chaveiro (C e JavaScript) ou provedor de chave mestra (Java e Python). Isso pode envolver uma chamada para um serviço criptográfico, como AWS Key Management Service (AWS KMS). O CMM retorna os materiais de criptografia para o AWS Encryption SDK.
4. O AWS Encryption SDK usa a chave de dados de texto sem formatação para criptografar os dados. Ele armazena os dados criptografados e as chaves de dados criptografadas em uma mensagem criptografada, que ele retorna ao usuário.



Criptografar dados com armazenamento em cache

Para obter materiais de criptografia com armazenamento de chaves de dados em cache:

1. Um aplicativo solicita que eles AWS Encryption SDK criptografem os dados.

   A solicitação especifica um gerenciador de armazenamento em cache materiais criptográficos (CMM de armazenamento em cache) associado a um gerenciador de materiais criptográficos (CMM) subjacente. Quando você especifica um provedor de chaves mestras ou um token de autenticação, o AWS Encryption SDK cria um CMM padrão para você.
2. O SDK solicita ao CMM de armazenamento em cache especificado materiais de criptografia.
3. O CMM de armazenamento em cache solicita materiais de criptografia do cache.
   a. Se encontrar uma correspondência, o cache atualizará a idade e usará os valores da entrada do cache correspondente, retornando os materiais de criptografia armazenados em cache ao CMM de armazenamento em cache.

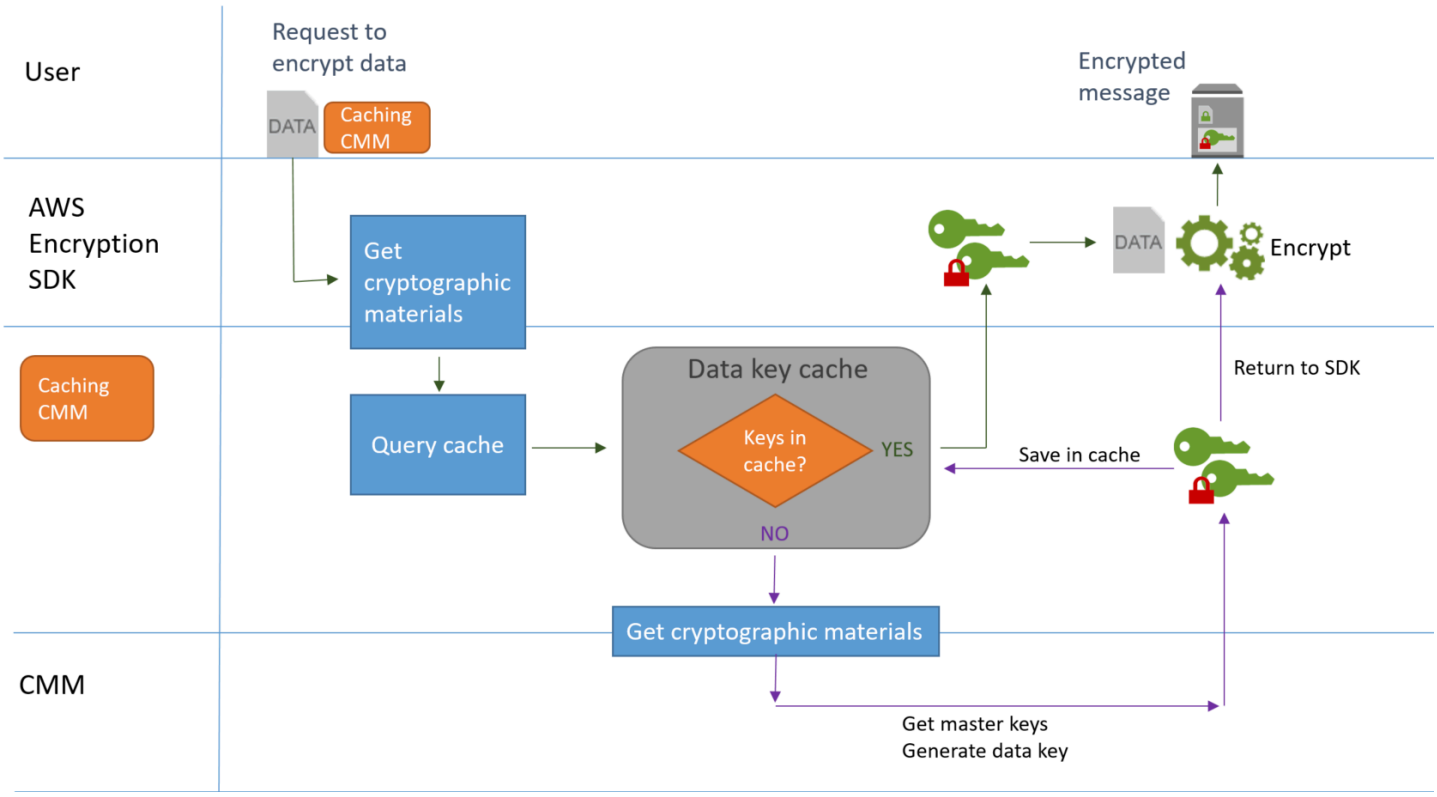
      Se a entrada do cache estiver em conformidade com os limites de segurança, o CMM de armazenamento em cache a retorna ao SDK. Caso contrário, ele instruirá o cache a remover a entrada e prosseguir como se não houvesse correspondência.

b. Se o cache não puder encontrar uma correspondência válida, o CMM de armazenamento em cache solicitará que CMM subjacente gere uma nova chave de dados.

   O CMM subjacente obtém os materiais criptográficos de seu chaveiro (C e JavaScript) ou provedor de chave mestra (Java e Python). Isso pode envolver uma chamada a um serviço criptográfico, como o AWS Key Management Service. O CMM subjacente retorna o texto simples e cópias criptografadas da chave de dados ao CMM de armazenamento em cache.

   O CMM de armazenamento em cache salva os novos materiais de criptografia no cache.

4. O CMM de armazenamento em cache retorna os materiais de criptografia para o AWS Encryption SDK.
5. O AWS Encryption SDK usa a chave de dados de texto sem formatação para criptografar os dados. Ele armazena os dados criptografados e as chaves de dados criptografadas em uma mensagem criptografada, que ele retorna ao usuário.



Criar um cache de material de criptografia

O AWS Encryption SDK define os requisitos para um cache de materiais criptográficos usado no armazenamento em cache de chaves de dados. Também fornece um cache local, que é

um cache least recently used (LRU - menos usado recentemente) configurável e na memória. Para criar uma instância do cache local, use o `LocalCryptoMaterialsCache` construtor em Java e Python, a LocalCryptographicMaterialsCache função get em JavaScript ou o `aws_cryptosdk_materials_cache_local_new` construtor em C.

O cache local contém lógica para gerenciamento básico do cache, incluindo adição, remoção e correspondência de entradas armazenadas em cache e manutenção do cache. Você não precisa escrever nenhuma lógica de gerenciamento de cache personalizado. O cache local pode ser usado como está, ser personalizado ou substituído por cache compatível.

Quando cria um cache local, você define sua capacidade, isto é, o número máximo de entradas que o cache pode conter. Essa configuração ajuda a criar um cache eficiente com reutilização limitada de chaves de dados.

O AWS Encryption SDK for Java e o AWS Encryption SDK for Python também fornecem um cache de materiais criptográficos nulo ()NullCryptoMaterialsCache. As NullCryptoMaterialsCache devoluções são perdidas em todas as `GET` operações e não respondem às `PUT` operações. Você pode usar o NullCryptoMaterialsCache em testes ou para desativar temporariamente o armazenamento em cache em um aplicativo que inclui código de armazenamento em cache.

No AWS Encryption SDK, cada cache de materiais criptográficos é associado a um gerenciador de materiais criptográficos de armazenamento em cache (CMM de armazenamento em cache). O CMM de armazenamento em cache obtém chaves de dados do cache, coloca chaves de dados no cache e impõe os limites de segurança que você define. Ao criar umCMM de armazenamento em cache, você especifica o cache que ele usa e o CMM subjacente ou o provedor de chaves mestras que gera as chaves de dados que ele armazena em cache.

Criar um gerenciador de material de criptografia de armazenamento em cache

Para habilitar o armazenamento em cache da chave de dados, você cria um cache e um gerenciador de armazenamento em cache (CMM de armazenamento em cache). Em seguida, em suas solicitações para criptografar ou descriptografar dados, você especifica um CMM de armazenamento em cache em vez de um gerenciador de materiais criptográficos (CMM) pardão, um provedor de chaves mestras ou um token de autenticação.

Há dois tipos de CMMs. Os dois obtêm chaves de dados (e o material criptográfico relacionado), mas de diferentes maneiras, da seguinte forma:

- Uma CMM está associada a um chaveiro (C ou JavaScript) ou a um provedor de chave mestra (Java e Python). Quando o SDK solicita ao CMM materiais de criptografia ou descriptografia, o CMM obtém os materiais de seu token de autenticação ou do provedor de chaves mestras. Em Java e Python, o CMM usa as chaves mestras para gerar, criptografar ou descriptografar as chaves de dados. Em C e JavaScript, o chaveiro gera, criptografa e retorna os materiais criptográficos.
- Um CMM de armazenamento em cache está associado a um cache, como um cache local e a um CMM subjacente. Quando o SDK solicita materiais criptográficos ao CMM de armazenamento em cache, o CMM de armazenamento em cache tenta obtê-los do cache. Se não conseguir encontrar uma correspondência, o CMM de armazenamento em cache solicitará os materiais ao seu CMM subjacente. Depois, ele armazenará os novos materiais criptográficos antes de retorná-los ao chamador.

O CMM de armazenamento em cache também impõe limites de segurança que você define para cada entrada do cache. Como os limites de segurança são definidos e impostos pelo CMM de armazenamento em cache, você pode usar qualquer cache compatível, mesmo que o cache não esteja projetado para material confidencial.

O que é uma entrada de chave de dados em cache?

O cache de chaves de dados armazena chaves de dados e o material criptográfico relacionado em um cache. Cada entrada inclui os elementos listados a seguir. Há caches separados para criptografia e decodificação, portanto, criptografar uma mensagem não aquece uma entrada de cache para decodificação. Você pode considerar essas informações úteis ao decidir se deseja usar o atributo de armazenamento em cache de chave de dados e ao configurar os limites de segurança em um gerenciador de armazenamento em cache de materiais criptográficos (CMM de armazenamento em cache).

Entradas armazenadas em cache para solicitações de criptografia

As entradas adicionadas a um cache de chave de dados como resultado de uma operação de criptografia incluem os seguintes elementos:

- Chave de dados de texto não criptografado
- Chaves de dados criptografadas (uma ou mais)
- Contexto de criptografia
- Chave de assinatura de mensagem (se uma for usada)

- Pacote de algoritmos
- Metadados, incluindo contadores de uso para impor limites de segurança

Entradas armazenadas em cache para solicitações de descriptografia

As entradas adicionadas a um cache de chave de dados como resultado de uma operação de descriptografia incluem os seguintes elementos:

- Chave de dados de texto não criptografado
- Chave de verificação de assinatura (se uma for usada)
- Metadados, incluindo contadores de uso para impor limites de segurança

Contexto de criptografia: como selecionar entradas do cache

Você pode especificar um contexto de criptografia em qualquer solicitação para criptografar dados. No entanto, o contexto de criptografia desempenha uma função especial no armazenamento em cache de chaves de dados. Ele permite criar subgrupos de chaves de dados em seu cache, mesmo quando as chaves de dados forem originárias do mesmo CMM de armazenamento em cache.

Um contexto de criptografia é um conjunto de pares de chave-valor que contêm dados arbitrários não secretos. Durante a criptografia, o contexto de criptografia é associado de maneira criptográfica aos dados criptografados de forma que o mesmo contexto de criptografia é necessário para descriptografar os dados. No AWS Encryption SDK, o contexto de criptografia é armazenado na mensagem criptografada com os dados criptografados e as chaves de dados.

Ao usar um cache de chave de dados, você também pode usar o contexto de criptografia para selecionar chaves de dados armazenadas em cache específicas para suas operações de criptografia. O contexto de criptografia é salvo na entrada do cache com a chave de dados (ele faz parte do ID de entrada do cache). As chaves de dados armazenadas em cache só são reutilizadas quando os contextos de criptografia correspondem. Se desejar reutilizar determinadas chaves de dados para uma solicitação de criptografia, especifique o mesmo contexto de criptografia. Para evitar essas chaves de dados, especifique outro contexto de criptografia.

O contexto de criptografia é sempre opcional, mas é recomendado. Se você não especificar um contexto de criptografia na solicitação, um contexto de criptografia vazio será incluído no identificador de entrada do cache e correspondido a cada solicitação.

Meu aplicativo está usando chaves de dados armazenadas em cache?

O armazenamento em cache de chaves de dados é uma estratégia de otimização muito eficaz para determinados aplicativos e cargas de trabalho. No entanto, como isso implica algum risco, é importante determinar o quão eficaz é provável que seja para a sua situação e decidir se os benefícios superam os riscos.

Como o armazenamento em cache de chaves de dados reutiliza chaves de dados, o efeito mais óbvio é a redução do número de chamadas para gerar novas chaves de dados. Quando o cache da chave de dados é implementado, AWS Encryption SDK ele chama a AWS KMS `GenerateDataKey` operação somente para criar a chave de dados inicial e quando o cache falha. Mas, o armazenamento em cache melhora o desempenho de forma perceptível somente em aplicativos que geram várias chaves de dados com as mesmas características, incluindo o mesmo contexto de criptografia e pacote de algoritmos.

Para determinar se sua implementação do AWS Encryption SDK está realmente usando chaves de dados do cache, experimente as técnicas a seguir.

- Nos logs da infraestrutura de sua chave mestra, verifique a frequência de chamadas para criar novas chaves de dados. Quando o armazenamento em cache de chaves de dados está efetivo, o número de chamadas para criar novas chaves deve cair de forma perceptível. Por exemplo, se você estiver usando um provedor de chave AWS KMS mestra ou um chaveiro, pesquise GenerateDataKey chamadas nos CloudTrail registros.
- Compare as mensagens criptografadas que o AWS Encryption SDK retorna em resposta a diferentes solicitações de criptografia. Por exemplo, se você estiver usando o AWS Encryption SDK for Java, compare o ParsedCiphertext objeto de diferentes chamadas de criptografia. No AWS Encryption SDK para JavaScript, compare o conteúdo da `encryptedDataKeys` propriedade do MessageHeader. Quando as chaves de dados são reutilizadas, as chaves de dados criptografadas na mensagem criptografada são idênticas.

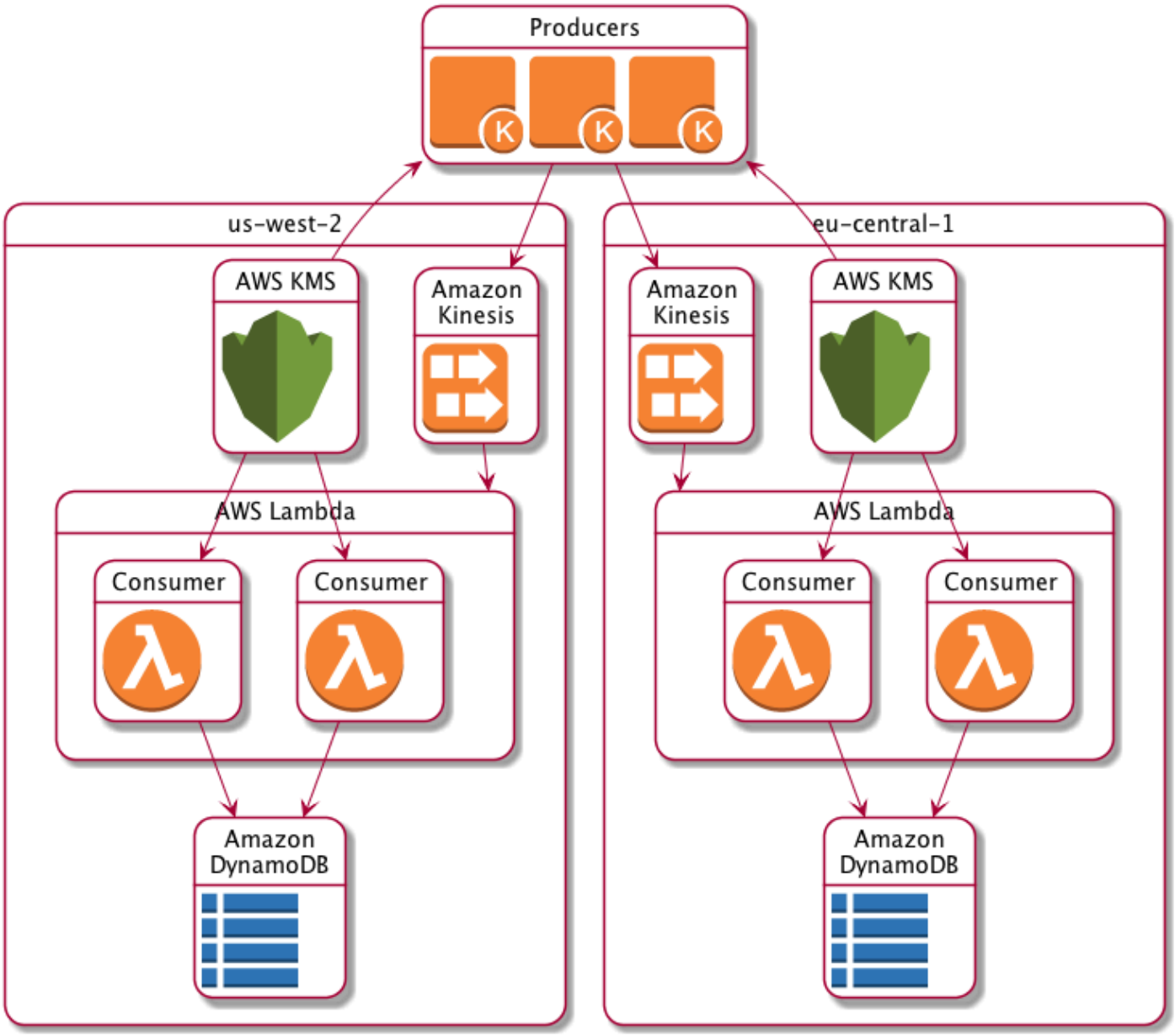
Exemplo de armazenamento em cache de chaves de dados

Este exemplo usa armazenamento em cache de chaves de dados com um cache local para acelerar uma aplicação em que os dados gerados por vários dispositivos são criptografados e armazenados em diferentes regiões.

Nesse cenário, vários produtores de dados geram, criptografam e gravam dados em um stream do Kinesis em cada região. As funções do AWS Lambda (consumidoras) descriptografam os

streams e gravam dados de texto simples em uma tabela do DynamoDB na região. Os produtores e os consumidores de dados usam o AWS Encryption SDK e um AWS KMS provedor de chaves mestras do . Para reduzir as chamadas ao KMS, cada produtor e consumidor tem seu próprio armazenamento em cache local.

Você pode encontrar o código-fonte desses exemplos em Java e Python. A amostra também inclui um CloudFormation modelo que define os recursos para as amostras.



Resultados do cache local

A tabela a seguir mostra que um armazenamento em cache local reduz o total de chamadas ao KMS (por segundo por região) neste exemplo em 1% de seu valor original.

Solicitações de produtores

	Solicitações por segundo por cliente			Clientes por região	Média de solicitações por segundo por região
	Gerar chaves de dados (us-west-2)	Criptografar chave de dados (eu-central-1)	Total (por região)		
Sem cache	1	1	1	500	500
Cache local	1 rps/100 usos	1 rps/100 usos	1 rps/100 usos	500	5

Solicitações de consumidor

	Solicitações por segundo por cliente			Cliente por região	Média de solicitações por segundo por região
	Descriptografar chave de dados	Produtores	Total		
Sem cache	1 rps por produtor	500	500	2	1.000
Cache local	1 rps por produtor/100 usos	500	5	2	10

Exemplo de código de armazenamento em cache de chaves de dados

Este exemplo de código cria uma implementação básica do armazenamento em cache de chaves de dados com um cache local em Java e Python. O código cria duas instâncias de um cache local:

uma para produtores de dados que estão criptografando dados e outra para consumidores de dados (AWS Lambda funções) que estão descriptografando dados. Para obter detalhes sobre a implementação do armazenamento em cache de chaves de dados em cada linguagem, consulte a documentação de Javadoc e Python para o AWS Encryption SDK.

O armazenamento em cache de chaves de dados está disponível para todas as linguagens de programação suportadas AWS Encryption SDK .

Para obter exemplos completos e testados do uso do armazenamento em cache de chaves de dados no AWS Encryption SDK, consulte:

- C/C++: caching_cmm.cpp
- Java: SimpleDataKeyCachingExample.java
- JavaScript Navegador: caching_cmm.ts
- JavaScript Node.js: caching_cmm.ts
- Python: data_key_caching_basic.py

Produtor

O produtor obtém um mapa, o converte em JSON, usa o AWS Encryption SDK para criptografá-lo e envia o registro de texto cifrado para um stream do Kinesis em cada um. Região da AWS

O código define um gerenciador de materiais criptográficos de armazenamento em cache (CMM de armazenamento em cache) e o associa a um cache local e a um provedor de chave mestrado AWS KMS subjacente. O CMM de amazenamento em cache armazena em cache as chaves de dados (e o material criptográfico relacionado) do provedor de chaves mestras. Ele também interage com o cache em nome do SDK e impõe os limites de segurança que você define.

Como a chamada para o método de criptografia especifica um CMM de armazenamento em cache, em vez de um gerenciador de materiais criptográficos (CMM) ou provedor de chave mestra comum, a criptografia usará o cache de chave de dados.

Java

> O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java desaprova o CMM de armazenamento em cache da chave de dados. Com a versão 3. x, você também pode usar o AWS KMS chaveiro hierárquico, uma solução alternativa de cache de materiais criptográficos.

```
/*
 * Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use
 this file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed on an
 "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
 * specific language governing permissions and limitations under the License.
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.MasterKeyProvider;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKey;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import com.amazonaws.encryptionsdk.multi.MultipleProviderFactory;
import com.amazonaws.util.json.Jackson;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.auth.credentials.AwsCredentialsProvider;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.core.SdkBytes;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.kinesis.KinesisClient;
import software.amazon.awssdk.services.kms.KmsClient;

/**
 * Pushes data to Kinesis Streams in multiple Regions.
 */
```

```
public class MultiRegionRecordPusher {

    private static final long MAX_ENTRY_AGE_MILLISECONDS = 300000;
    private static final long MAX_ENTRY_USES = 100;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final String streamName_;
    private final ArrayList<KinesisClient> kinesisClients_;
    private final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;

    /**
     * Creates an instance of this object with Kinesis clients for all target
 Regions and a cached
     * key provider containing KMS master keys in all target Regions.
     */
    public MultiRegionRecordPusher(final Region[] regions, final String
 kmsAliasName,
        final String streamName) {
        streamName_ = streamName;
        crypto_ = AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();
        kinesisClients_ = new ArrayList<>();

        AwsCredentialsProvider credentialsProvider =
 DefaultCredentialsProvider.builder().build();

        // Build KmsMasterKey and AmazonKinesisClient objects for each target region
        List<KmsMasterKey> masterKeys = new ArrayList<>();
        for (Region region : regions) {
            kinesisClients_.add(KinesisClient.builder()
                .credentialsProvider(credentialsProvider)
                .region(region)
                .build());

            KmsMasterKey regionMasterKey = KmsMasterKeyProvider.builder()
                .defaultRegion(region)
                .builderSupplier(() ->
 KmsClient.builder().credentialsProvider(credentialsProvider))
                .buildStrict(kmsAliasName)
                .getMasterKey(kmsAliasName);

            masterKeys.add(regionMasterKey);
        }
```

```
        // Collect KmsMasterKey objects into single provider and add cache
        MasterKeyProvider<?> masterKeyProvider =
MultipleProviderFactory.buildMultiProvider(
            KmsMasterKey.class,
            masterKeys
        );

        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()
            .withMasterKeyProvider(masterKeyProvider)
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .withMessageUseLimit(MAX_ENTRY_USES)
            .build();
    }

    /**
     * JSON serializes and encrypts the received record data and pushes it to all
target streams.
     */
    public void putRecord(final Map<Object, Object> data) {
        String partitionKey = UUID.randomUUID().toString();
        Map<String, String> encryptionContext = new HashMap<>();
        encryptionContext.put("stream", streamName_);

        // JSON serialize data
        String jsonData = Jackson.toJsonString(data);

        // Encrypt data
        CryptoResult<byte[], ?> result = crypto_.encryptData(
            cachingMaterialsManager_,
            jsonData.getBytes(),
            encryptionContext
        );
        byte[] encryptedData = result.getResult();

        // Put records to Kinesis stream in all Regions
        for (KinesisClient regionalKinesisClient : kinesisClients_) {
            regionalKinesisClient.putRecord(builder ->
                builder.streamName(streamName_)
                    .data(SdkBytes.fromByteArray(encryptedData))
                    .partitionKey(partitionKey));
        }
    }
```

```
}
```

Python

```
"""
Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License"). You may not use this
 file except
in compliance with the License. A copy of the License is located at

https://aws.amazon.com/apache-2-0/

or in the "license" file accompanying this file. This file is distributed on an "AS
 IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
specific language governing permissions and limitations under the License.
"""
import json
import uuid

from aws_encryption_sdk import EncryptionSDKClient, StrictAwsKmsMasterKeyProvider,
 CachingCryptoMaterialsManager, LocalCryptoMaterialsCache, CommitmentPolicy
from aws_encryption_sdk.key_providers.kms import KMSMasterKey
import boto3


class MultiRegionRecordPusher(object):
    """Pushes data to Kinesis Streams in multiple Regions."""
    CACHE_CAPACITY = 100
    MAX_ENTRY_AGE_SECONDS = 300.0
    MAX_ENTRY_MESSAGES_ENCRYPTED = 100

    def __init__(self, regions, kms_alias_name, stream_name):
        self._kinesis_clients = []
        self._stream_name = stream_name

        # Set up EncryptionSDKClient
        _client =
 EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

        # Set up KMSMasterKeyProvider with cache
```

```
        _key_provider = StrictAwsKmsMasterKeyProvider(kms_alias_name)

        # Add MasterKey and Kinesis client for each Region
        for region in regions:
            self._kinesis_clients.append(boto3.client('kinesis',
region_name=region))
            regional_master_key = KMSMasterKey(
                client=boto3.client('kms', region_name=region),
                key_id=kms_alias_name
            )
            _key_provider.add_master_key_provider(regional_master_key)

        cache = LocalCryptoMaterialsCache(capacity=self.CACHE_CAPACITY)
        self._materials_manager = CachingCryptoMaterialsManager(
            master_key_provider=_key_provider,
            cache=cache,
            max_age=self.MAX_ENTRY_AGE_SECONDS,
            max_messages_encrypted=self.MAX_ENTRY_MESSAGES_ENCRYPTED
        )

    def put_record(self, record_data):
        """JSON serializes and encrypts the received record data and pushes it to
all target streams.

        :param dict record_data: Data to write to stream
        """
        # Kinesis partition key to randomize write load across stream shards
        partition_key = uuid.uuid4().hex

        encryption_context = {'stream': self._stream_name}

        # JSON serialize data
        json_data = json.dumps(record_data)

        # Encrypt data
        encrypted_data, _header = _client.encrypt(
            source=json_data,
            materials_manager=self._materials_manager,
            encryption_context=encryption_context
        )

        # Put records to Kinesis stream in all Regions
        for client in self._kinesis_clients:
            client.put_record(
```

```
                StreamName=self._stream_name,
                Data=encrypted_data,
                PartitionKey=partition_key
            )
```

Consumidor

O consumidor de dados é uma função do AWS Lambda acionada por eventos do Kinesis. Ele descriptografa e desserializa cada registro e grava o registro de texto simples em uma tabela do Amazon DynamoDB na mesma região.

Como o código do produtor, o código do consumidor habilita o armazenamento em cache da chave de dados usando um gerenciador de materiais criptográficos de cache (caching CMM) em chamadas para o método de descriptografia.

O código Java cria um provedor de chave mestra no modo estrito com um especificado AWS KMS key. O modo estrito não é necessário ao descriptografar, mas é uma prática recomendada. O código Python usa o modo de descoberta, que permite AWS Encryption SDK usar qualquer chave de empacotamento que criptografe uma chave de dados para descriptografá-la.

Java

O exemplo a seguir usa a versão 2. x do AWS Encryption SDK for Java. Versão 3. x do AWS Encryption SDK for Java desaprova o CMM de armazenamento em cache da chave de dados. Com a versão 3. x, você também pode usar o AWS KMS chaveiro hierárquico, uma solução alternativa de cache de materiais criptográficos.

Esse código cria um provedor de chave mestra para descriptografia no modo estrito. O AWS Encryption SDK pode usar somente o AWS KMS keys que você especificar para descriptografar sua mensagem.

```
/*
 * Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"). You may not use
 this file except
 * in compliance with the License. A copy of the License is located at
 *
 * http://aws.amazon.com/apache2.0
 *
```

```
 * or in the "license" file accompanying this file. This file is distributed on an
 "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
 * specific language governing permissions and limitations under the License.
 */
package com.amazonaws.crypto.examples.kinesisdatakeycaching;

import com.amazonaws.encryptionsdk.AwsCrypto;
import com.amazonaws.encryptionsdk.CommitmentPolicy;
import com.amazonaws.encryptionsdk.CryptoResult;
import com.amazonaws.encryptionsdk.caching.CachingCryptoMaterialsManager;
import com.amazonaws.encryptionsdk.caching.LocalCryptoMaterialsCache;
import com.amazonaws.encryptionsdk.kmssdkv2.KmsMasterKeyProvider;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent.KinesisEventRecord;
import com.amazonaws.util.BinaryUtils;
import java.io.UnsupportedEncodingException;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.TimeUnit;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbEnhancedClient;
import software.amazon.awssdk.enhanced.dynamodb.DynamoDbTable;
import software.amazon.awssdk.enhanced.dynamodb.TableSchema;

/**
 * Decrypts all incoming Kinesis records and writes records to DynamoDB.
 */
public class LambdaDecryptAndWrite {

    private static final long MAX_ENTRY_AGE_MILLISECONDS = 600000;
    private static final int MAX_CACHE_ENTRIES = 100;
    private final CachingCryptoMaterialsManager cachingMaterialsManager_;
    private final AwsCrypto crypto_;
    private final DynamoDbTable<Item> table_;

    /**
     * Because the cache is used only for decryption, the code doesn't set the max
 bytes or max
     * message security thresholds that are enforced only on on data keys used for
 encryption.
     */
    public LambdaDecryptAndWrite() {
```

```
        String kmsKeyArn = System.getenv("CMK_ARN");
        cachingMaterialsManager_ = CachingCryptoMaterialsManager.newBuilder()

 .withMasterKeyProvider(KmsMasterKeyProvider.builder().buildStrict(kmsKeyArn))
            .withCache(new LocalCryptoMaterialsCache(MAX_CACHE_ENTRIES))
            .withMaxAge(MAX_ENTRY_AGE_MILLISECONDS, TimeUnit.MILLISECONDS)
            .build();

        crypto_ = AwsCrypto.builder()
            .withCommitmentPolicy(CommitmentPolicy.RequireEncryptRequireDecrypt)
            .build();

        String tableName = System.getenv("TABLE_NAME");
        DynamoDbEnhancedClient dynamodb = DynamoDbEnhancedClient.builder().build();
        table_ = dynamodb.table(tableName, TableSchema.fromClass(Item.class));
    }

    /**
     * @param event
     * @param context
     */
    public void handleRequest(KinesisEvent event, Context context)
        throws UnsupportedEncodingException {
        for (KinesisEventRecord record : event.getRecords()) {
            ByteBuffer ciphertextBuffer = record.getKinesis().getData();
            byte[] ciphertext = BinaryUtils.copyAllBytesFrom(ciphertextBuffer);

            // Decrypt and unpack record
            CryptoResult<byte[], ?> plaintextResult =
 crypto_.decryptData(cachingMaterialsManager_,
                ciphertext);

            // Verify the encryption context value
            String streamArn = record.getEventSourceARN();
            String streamName = streamArn.substring(streamArn.indexOf("/") + 1);
            if (!
streamName.equals(plaintextResult.getEncryptionContext().get("stream"))) {
                throw new IllegalStateException("Wrong Encryption Context!");
            }

            // Write record to DynamoDB
            String jsonItem = new String(plaintextResult.getResult(),
 StandardCharsets.UTF_8);
            System.out.println(jsonItem);
```

```
            table_.putItem(Item.fromJSON(jsonItem));
        }
    }

    private static class Item {

        static Item fromJSON(String jsonText) {
            // Parse JSON and create new Item
            return new Item();
        }
    }
}
```

Python

Esse código Python é descriptografado com um provedor de chave mestra no modo de descoberta. Ele permite ao AWS Encryption SDK usar qualquer chave de encapsulamento que criptografe uma chave de dados para descriptografá-la. O modo estrito, no qual você especifica as chaves de encapsulamento que podem ser usadas para decodificação, é uma prática recomendada.

```
"""
Copyright 2017 Amazon.com, Inc. or its affiliates. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License"). You may not use this
 file except
in compliance with the License. A copy of the License is located at

https://aws.amazon.com/apache-2-0/

or in the "license" file accompanying this file. This file is distributed on an "AS
 IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 License for the
specific language governing permissions and limitations under the License.
"""
import base64
import json
import logging
import os
```

```
from aws_encryption_sdk import EncryptionSDKClient,
 DiscoveryAwsKmsMasterKeyProvider, CachingCryptoMaterialsManager,
 LocalCryptoMaterialsCache, CommitmentPolicy
import boto3

_LOGGER = logging.getLogger(__name__)
_is_setup = False
CACHE_CAPACITY = 100
MAX_ENTRY_AGE_SECONDS = 600.0

def setup():
    """Sets up clients that should persist across Lambda invocations."""
    global encryption_sdk_client
    encryption_sdk_client =
 EncryptionSDKClient(CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT)

    global materials_manager
    key_provider = DiscoveryAwsKmsMasterKeyProvider()
    cache = LocalCryptoMaterialsCache(capacity=CACHE_CAPACITY)

    #  Because the cache is used only for decryption, the code doesn't set
    #   the max bytes or max message security thresholds that are enforced
    #   only on on data keys used for encryption.
    materials_manager = CachingCryptoMaterialsManager(
        master_key_provider=key_provider,
        cache=cache,
        max_age=MAX_ENTRY_AGE_SECONDS
    )
    global table
    table_name = os.environ.get('TABLE_NAME')
    table = boto3.resource('dynamodb').Table(table_name)
    global _is_setup
    _is_setup = True


def lambda_handler(event, context):
    """Decrypts all incoming Kinesis records and writes records to DynamoDB."""
    _LOGGER.debug('New event:')
    _LOGGER.debug(event)
    if not _is_setup:
        setup()
    with table.batch_writer() as batch:
        for record in event.get('Records', []):
            # Record data base64-encoded by Kinesis
```

```
            ciphertext = base64.b64decode(record['kinesis']['data'])

            # Decrypt and unpack record
            plaintext, header = encryption_sdk_client.decrypt(
                source=ciphertext,
                materials_manager=materials_manager
            )
            item = json.loads(plaintext)

            # Verify the encryption context value
            stream_name = record['eventSourceARN'].split('/', 1)[1]
            if stream_name != header.encryption_context['stream']:
                raise ValueError('Wrong Encryption Context!')

            # Write record to DynamoDB
            batch.put_item(Item=item)
```

Exemplo de armazenamento em cache de chave de dados: modelo CloudFormation

Esse CloudFormation modelo configura todos os AWS recursos necessários para reproduzir o exemplo de armazenamento em cache da chave de dados.

JSON

```
{
    "Parameters": {
        "SourceCodeBucket": {
            "Type": "String",
            "Description": "S3 bucket containing Lambda source code zip files"
        },
        "PythonLambdaS3Key": {
            "Type": "String",
            "Description": "S3 key containing Python Lambda source code zip file"
        },
        "PythonLambdaObjectVersionId": {
            "Type": "String",
            "Description": "S3 version id for S3 key containing Python Lambda source
 code zip file"
        },
```

```
        "JavaLambdaS3Key": {
            "Type": "String",
            "Description": "S3 key containing Python Lambda source code zip file"
        },
        "JavaLambdaObjectVersionId": {
            "Type": "String",
            "Description": "S3 version id for S3 key containing Python Lambda source
 code zip file"
        },
        "KeyAliasSuffix": {
            "Type": "String",
            "Description": "Suffix to use for KMS key Alias (ie: alias/
KeyAliasSuffix)"
        },
        "StreamName": {
            "Type": "String",
            "Description": "Name to use for Kinesis Stream"
        }
    },
    "Resources": {
        "InputStream": {
            "Type": "AWS::Kinesis::Stream",
            "Properties": {
                "Name": {
                    "Ref": "StreamName"
                },
                "ShardCount": 2
            }
        },
        "PythonLambdaOutputTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "id",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
                        "AttributeName": "id",
                        "KeyType": "HASH"
                    }
                ],
```

```
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        },
        "PythonLambdaRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                ],
                "Policies": [
                    {
                        "PolicyName": "PythonLambdaAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:DescribeTable",
                                        "dynamodb:BatchWriteItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}"
                                    }
                                },
                                {
                                    "Effect": "Allow",
```

```
                                    "Action": [
                                        "dynamodb:PutItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "kinesis:GetRecords",
                                        "kinesis:GetShardIterator",
                                        "kinesis:DescribeStream",
                                        "kinesis:ListStreams"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "PythonLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Python consumer",
                "Runtime": "python2.7",
                "MemorySize": 512,
                "Timeout": 90,
                "Role": {
                    "Fn::GetAtt": [
                        "PythonLambdaRole",
                        "Arn"
                    ]
                },
                "Handler":
 "aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler",
                "Code": {
                    "S3Bucket": {
```

```
                        "Ref": "SourceCodeBucket"
                    },
                    "S3Key": {
                        "Ref": "PythonLambdaS3Key"
                    },
                    "S3ObjectVersion": {
                        "Ref": "PythonLambdaObjectVersionId"
                    }
                },
                "Environment": {
                    "Variables": {
                        "TABLE_NAME": {
                            "Ref": "PythonLambdaOutputTable"
                        }
                    }
                }
            }
        },
        "PythonLambdaSourceMapping": {
            "Type": "AWS::Lambda::EventSourceMapping",
            "Properties": {
                "BatchSize": 1,
                "Enabled": true,
                "EventSourceArn": {
                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                },
                "FunctionName": {
                    "Ref": "PythonLambdaFunction"
                },
                "StartingPosition": "TRIM_HORIZON"
            }
        },
        "JavaLambdaOutputTable": {
            "Type": "AWS::DynamoDB::Table",
            "Properties": {
                "AttributeDefinitions": [
                    {
                        "AttributeName": "id",
                        "AttributeType": "S"
                    }
                ],
                "KeySchema": [
                    {
```

```
                        "AttributeName": "id",
                        "KeyType": "HASH"
                    }
                ],
                "ProvisionedThroughput": {
                    "ReadCapacityUnits": 1,
                    "WriteCapacityUnits": 1
                }
            }
        },
        "JavaLambdaRole": {
            "Type": "AWS::IAM::Role",
            "Properties": {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "Service": "lambda.amazonaws.com"
                            },
                            "Action": "sts:AssumeRole"
                        }
                    ]
                },
                "ManagedPolicyArns": [
                    "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
                ],
                "Policies": [
                    {
                        "PolicyName": "JavaLambdaAccess",
                        "PolicyDocument": {
                            "Version": "2012-10-17",
                            "Statement": [
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:DescribeTable",
                                        "dynamodb:BatchWriteItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}"
```

```
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "dynamodb:PutItem"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*"
                                    }
                                },
                                {
                                    "Effect": "Allow",
                                    "Action": [
                                        "kinesis:GetRecords",
                                        "kinesis:GetShardIterator",
                                        "kinesis:DescribeStream",
                                        "kinesis:ListStreams"
                                    ],
                                    "Resource": {
                                        "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                                    }
                                }
                            ]
                        }
                    }
                ]
            }
        },
        "JavaLambdaFunction": {
            "Type": "AWS::Lambda::Function",
            "Properties": {
                "Description": "Java consumer",
                "Runtime": "java8",
                "MemorySize": 512,
                "Timeout": 90,
                "Role": {
                    "Fn::GetAtt": [
                        "JavaLambdaRole",
                        "Arn"
                    ]
                },
```

```
                "Handler":
 "com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest",
                "Code": {
                    "S3Bucket": {
                        "Ref": "SourceCodeBucket"
                    },
                    "S3Key": {
                        "Ref": "JavaLambdaS3Key"
                    },
                    "S3ObjectVersion": {
                        "Ref": "JavaLambdaObjectVersionId"
                    }
                },
                "Environment": {
                    "Variables": {
                        "TABLE_NAME": {
                            "Ref": "JavaLambdaOutputTable"
                        },
                        "CMK_ARN": {
                            "Fn::GetAtt": [
                                "RegionKinesisCMK",
                                "Arn"
                            ]
                        }
                    }
                }
            }
        },
        "JavaLambdaSourceMapping": {
            "Type": "AWS::Lambda::EventSourceMapping",
            "Properties": {
                "BatchSize": 1,
                "Enabled": true,
                "EventSourceArn": {
                    "Fn::Sub": "arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}"
                },
                "FunctionName": {
                    "Ref": "JavaLambdaFunction"
                },
                "StartingPosition": "TRIM_HORIZON"
            }
        },
        "RegionKinesisCMK": {
```

```
            "Type": "AWS::KMS::Key",
            "Properties": {
                "Description": "Used to encrypt data passing through Kinesis Stream
in this region",
                "Enabled": true,
                "KeyPolicy": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "AWS": {
                                    "Fn::Sub": "arn:aws:iam::${AWS::AccountId}:root"
                                }
                            },
                            "Action": [
                                "kms:Encrypt",
                                "kms:GenerateDataKey",
                                "kms:CreateAlias",
                                "kms:DeleteAlias",
                                "kms:DescribeKey",
                                "kms:DisableKey",
                                "kms:EnableKey",
                                "kms:PutKeyPolicy",
                                "kms:ScheduleKeyDeletion",
                                "kms:UpdateAlias",
                                "kms:UpdateKeyDescription"
                            ],
                            "Resource": "*"
                        },
                        {
                            "Effect": "Allow",
                            "Principal": {
                                "AWS": [
                                    {
                                        "Fn::GetAtt": [
                                            "PythonLambdaRole",
                                            "Arn"
                                        ]
                                    },
                                    {
                                        "Fn::GetAtt": [
                                            "JavaLambdaRole",
                                            "Arn"
```

```
                                    ]
                                }
                            ]
                        },
                        "Action": "kms:Decrypt",
                        "Resource": "*"
                    }
                ]
            }
        }
    },
    "RegionKinesisCMKAlias": {
        "Type": "AWS::KMS::Alias",
        "Properties": {
            "AliasName": {
                "Fn::Sub": "alias/${KeyAliasSuffix}"
            },
            "TargetKeyId": {
                "Ref": "RegionKinesisCMK"
            }
        }
    }
}
```

YAML

```
Parameters:
    SourceCodeBucket:
        Type: String
        Description: S3 bucket containing Lambda source code zip files
    PythonLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    PythonLambdaObjectVersionId:
        Type: String
        Description: S3 version id for S3 key containing Python Lambda source code
 zip file
    JavaLambdaS3Key:
        Type: String
        Description: S3 key containing Python Lambda source code zip file
    JavaLambdaObjectVersionId:
        Type: String
```

```
        Description: S3 version id for S3 key containing Python Lambda source code
 zip file
    KeyAliasSuffix:
        Type: String
        Description: 'Suffix to use for KMS CMK Alias (ie: alias/<KeyAliasSuffix>)'
    StreamName:
        Type: String
        Description: Name to use for Kinesis Stream
Resources:
    InputStream:
        Type: AWS::Kinesis::Stream
        Properties:
            Name: !Ref StreamName
            ShardCount: 2
    PythonLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    PythonLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: PythonLambdaAccess
```

```
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${PythonLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    PythonLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Python consumer
            Runtime: python2.7
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt PythonLambdaRole.Arn
            Handler:
 aws_crypto_examples.kinesis_datakey_caching.consumer.lambda_handler
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref PythonLambdaS3Key
                S3ObjectVersion: !Ref PythonLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref PythonLambdaOutputTable
    PythonLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
```

```
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref PythonLambdaFunction
            StartingPosition: TRIM_HORIZON
    JavaLambdaOutputTable:
        Type: AWS::DynamoDB::Table
        Properties:
            AttributeDefinitions:
                -
                    AttributeName: id
                    AttributeType: S
            KeySchema:
                -
                    AttributeName: id
                    KeyType: HASH
            ProvisionedThroughput:
                ReadCapacityUnits: 1
                WriteCapacityUnits: 1
    JavaLambdaRole:
        Type: AWS::IAM::Role
        Properties:
            AssumeRolePolicyDocument:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            Service: lambda.amazonaws.com
                        Action: sts:AssumeRole
            ManagedPolicyArns:
                - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
            Policies:
                -
                    PolicyName: JavaLambdaAccess
                    PolicyDocument:
                        Version: 2012-10-17
                        Statement:
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:DescribeTable
```

```
                                    - dynamodb:BatchWriteItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}
                            -
                                Effect: Allow
                                Action:
                                    - dynamodb:PutItem
                                Resource: !Sub arn:aws:dynamodb:${AWS::Region}:
${AWS::AccountId}:table/${JavaLambdaOutputTable}*
                            -
                                Effect: Allow
                                Action:
                                    - kinesis:GetRecords
                                    - kinesis:GetShardIterator
                                    - kinesis:DescribeStream
                                    - kinesis:ListStreams
                                Resource: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
    JavaLambdaFunction:
        Type: AWS::Lambda::Function
        Properties:
            Description: Java consumer
            Runtime: java8
            MemorySize: 512
            Timeout: 90
            Role: !GetAtt JavaLambdaRole.Arn
            Handler:
 com.amazonaws.crypto.examples.kinesisdatakeycaching.LambdaDecryptAndWrite::handleRequest
            Code:
                S3Bucket: !Ref SourceCodeBucket
                S3Key: !Ref JavaLambdaS3Key
                S3ObjectVersion: !Ref JavaLambdaObjectVersionId
            Environment:
                Variables:
                    TABLE_NAME: !Ref JavaLambdaOutputTable
                    CMK_ARN: !GetAtt RegionKinesisCMK.Arn
    JavaLambdaSourceMapping:
        Type: AWS::Lambda::EventSourceMapping
        Properties:
            BatchSize: 1
            Enabled: true
            EventSourceArn: !Sub arn:aws:kinesis:${AWS::Region}:
${AWS::AccountId}:stream/${InputStream}
            FunctionName: !Ref JavaLambdaFunction
```

```
            StartingPosition: TRIM_HORIZON
    RegionKinesisCMK:
        Type: AWS::KMS::Key
        Properties:
            Description: Used to encrypt data passing through Kinesis Stream in this
region
            Enabled: true
            KeyPolicy:
                Version: 2012-10-17
                Statement:
                    -
                        Effect: Allow
                        Principal:
                            AWS: !Sub arn:aws:iam::${AWS::AccountId}:root
                        Action:
                            # Data plane actions
                            - kms:Encrypt
                            - kms:GenerateDataKey
                            # Control plane actions
                            - kms:CreateAlias
                            - kms:DeleteAlias
                            - kms:DescribeKey
                            - kms:DisableKey
                            - kms:EnableKey
                            - kms:PutKeyPolicy
                            - kms:ScheduleKeyDeletion
                            - kms:UpdateAlias
                            - kms:UpdateKeyDescription
                        Resource: '*'
                    -
                        Effect: Allow
                        Principal:
                            AWS:
                                - !GetAtt PythonLambdaRole.Arn
                                - !GetAtt JavaLambdaRole.Arn
                        Action: kms:Decrypt
                        Resource: '*'
    RegionKinesisCMKAlias:
        Type: AWS::KMS::Alias
        Properties:
            AliasName: !Sub alias/${KeyAliasSuffix}
            TargetKeyId: !Ref RegionKinesisCMK
```

Versões do AWS Encryption SDK

As implementações da AWS Encryption SDK linguagem usam controle de versão semântico para facilitar a identificação da magnitude das mudanças em cada versão. Uma alteração no número da versão principal, como de 1.x. x para 2.x.x, indica uma alteração significativa que provavelmente exigirá alterações no código e uma implantação planejada. Alterações significativas em uma nova versão podem não afetar todos os casos de uso, revise as notas de lançamento para ver se você foi afetado. Uma alteração em uma versão secundária, como de x.1.x para x.2.x, é sempre compatível com versões anteriores, mas pode incluir elementos descontinuados.

Sempre que possível, use a versão mais recente do AWS Encryption SDK na linguagem de programação escolhida. A política de manutenção e suporte para cada versão é diferente para cada implementação de linguagem de programação. Para obter detalhes sobre as versões suportadas em sua linguagem de programação preferida, consulte o `SUPPORT_POLICY.rst` arquivo em seu GitHub repositório.

Quando as atualizações incluem novos atributos que exigem configuração especial para evitar erros de criptografia ou descriptografia, fornecemos uma versão intermediária e instruções detalhadas para usá-la. Por exemplo, as versões 1.7.x e 1.8.x foram projetadas para serem versões transitórias que ajudam você a atualizar de versões anteriores à 1.7.x para as versões 2.0.x e posteriores. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

> **Note**
>
> O x em um número de versão representa qualquer patch da versão principal e secundária. Por exemplo, a versão 1.7.x representa todas as versões que começam com 1.7, incluindo 1.7.1 e 1.7.9.
>
> Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão 1.8 do AWS Encryption CLI. x substitui a versão 1.7. x e AWS Encryption CLI 2.1. x substitui 2.0. x. Para obter detalhes, consulte o comunicado de segurança relevante no repositório https://github.com/aws/aws-encryption-sdk-cli/ aws-encryption-sdk-cli em. GitHub

As tabelas a seguir fornecem uma visão geral das principais diferenças entre as versões suportadas do AWS Encryption SDK para cada linguagem de programação.

C

Para obter uma descrição detalhada de todas as alterações, consulte o repositório https://github.com/aws/aws-encryption-sdk-c/ aws-encryption-sdk-c CHANGELOG.md em. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
1.x	1,0	Versão inicial.	End-of-Support fase
	1,7	Atualizações do AWS Encryption SDK que ajudam os usuários de versões anteriores a fazer o upgrade para a versão 2.0. x e mais tarde. Para obter mais informações, consulte a versão 1.7. x.	
2.x	2,0	Atualizações para AWS Encryption SDK o. Para obter mais informações, consulte a versão 2.0. x.	Disponibilidade geral (GA)
	2.2	Melhorias no processo de decodificação de mensagens.	
	2.3	Adiciona suporte para chaves AWS KMS multirregionais.	

C#/.NET

Para obter uma descrição detalhada de todas as alterações, consulte CHANGELOG.md no aws-encryption-sdk-net repositório em GitHub.

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
3.x	3.1.0	Versão inicial.	End-of-Support A versão 3.x do AWS Encryption SDK for.NET entrou no fim do suporte; atualize para 4.x.
4.x	4,0	Adiciona suporte para o chaveiro AWS KMS hierárquico, o CMM de contexto de criptografia necessári o e chaveiros RSA assimétricos. AWS KMS	Disponibilidade geral (GA)
5.x	5,0	Atualizações para a versão 2. x da Biblioteca de Provedores de Materiais (MPL). Descarta o suporte ao AWS SDK para .NET v3 e exige o AWS SDK para .NET v4.	Disponibilidade geral (GA)

Interface de linha de comando (CLI)

Para obter uma descrição detalhada de todas as alterações, consulte Versões da CLI AWS de criptografia e acesse o CHANGELOG.rst repositório https://github.com/aws/aws-encryption-sdk-cli/ aws-encryption-sdk-cli ativado. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
1.x	1,0	Versão inicial.	End-of-Support fase
	1,7	Atualizações do AWS Encryption SDK que ajudam os usuários de versões anteriores a fazer o upgrade para a versão 2.0. x e mais tarde. Para obter mais informações, consulte a versão 1.7. x.	
2.x	2,0	Atualizações para AWS Encryption SDK o. Para obter mais informações, consulte a versão 2.0. x.	End-of-Support fase
	2.1	Remove o `--discovery` parâmetro e o substitui pelo `discovery` atributo do `--wrapping-keys` parâmetro.	

		A versão 2.1.0 do AWS Encryption CLI é equivalente à versão 2.0 em outras linguagens de programação.	
	2.2	Melhorias no processo de decodificação de mensagens.	
3.x	3.0	Adiciona suporte para chaves AWS KMS multirregionais.	End-of-Support fase
4.x	4,0	A CLI AWS de criptografia não oferece mais suporte ao Python 2 ou ao Python 3.4. A partir da versão principal 4. x da CLI de AWS criptografia, somente Python 3.5 ou posterior é suportado.	Disponibilidade geral (GA)
	4.1	A CLI AWS de criptografia não oferece mais suporte ao Python 3.5. A partir da versão 4.1. x da CLI de AWS criptografia, somente Python 3.6 ou posterior é suportado.	

	4.2	A CLI AWS de criptografia não oferece mais suporte ao Python 3.6. A partir da versão 4.2. x da CLI de AWS criptografia, somente Python 3.7 ou posterior é suportado.

Java

Para obter uma descrição detalhada de todas as alterações, consulte o repositório https://github.com/aws/aws-encryption-sdk-java/ aws-encryption-sdk-java CHANGELOG.rst em. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
1.x	1,0	Versão inicial.	End-of-Support fase
	1.3	Adiciona suporte para gerenciador de materiais criptográficos e armazenamento em cache de chaves de dados. Transferido para a geração IV determinística.	
	1.6.1	Desaprova `AwsCrypto.encryptString()` `AwsCrypto`	

		`.decryptS tring()` e os substitui por e. `AwsCrypto .encryptD ata()` `AwsCrypto .decryptData()`	
	1,7	Atualizações do AWS Encryption SDK que ajudam os usuários de versões anteriores a fazer o upgrade para a versão 2.0. x e mais tarde. Para obter mais informações, consulte a versão 1.7. x.	
2.x	2,0	Atualizações para AWS Encryption SDK o. Para obter mais informações, consulte a versão 2.0. x.	Disponibilidade geral (GA) A versão 2.x do AWS Encryption SDK for Java entrará no modo de manutenção em 2024.
	2.2	Melhorias no processo de decodificação de mensagens.	
	2.3	Adiciona suporte para chaves AWS KMS multirregionais.	
	2.4	Adiciona suporte para AWS SDK for Java 2.x.	

3.x	3.0	Integra-o AWS Encryption SDK for Java com a Bibliotec a de Provedores de Materiais (MPL). Adiciona suporte para chaveiros RSA simétricos e assimétri cos, chaveiros AWS KMS ECDH, AWS KMS chaveiros AWS KMS hierárqui cos, chaveiros Raw AES, chaveiros Raw RSA, chaveiros Raw ECDH e o CMM de contexto de criptogra fia necessário. Multi-keyrings	Disponibilidade geral (GA)

Go

Para obter uma descrição detalhada de todas as alterações, consulte o diretório Go do repositório https://github.com/aws/aws-encryption-sdk/tree/mainline aws-encryption-sdk CHANGELOG.md em. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
0,1. x	0.1.0	Versão inicial.	Disponibilidade geral (GA)

JavaScript

Para obter uma descrição detalhada de todas as alterações, consulte o repositório https://github.com/aws/aws-encryption-sdk-javascript/tree/master/modules aws-encryption-sdk-javascript CHANGELOG.md em. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
1.x	1,0	Versão inicial.	End-of-Support fase
	1,7	Atualizações do AWS Encryption SDK que ajudam os usuários de versões anteriores a fazer o upgrade para a versão 2.0. x e mais tarde. Para obter mais informações, consulte a versão 1.7. x.	
2.x	2,0	Atualizações para AWS Encryption SDKo. Para obter mais informações, consulte a versão 2.0. x.	End-of-Support fase
	2.2	Melhorias no processo de decodificação de mensagens.	
	2.3	Adiciona suporte para chaves AWS KMS multirregionais.	

3.x	3.0	Remove a cobertura de CI para o Node 10. Atualiza as dependências para não suportar mais o Node 8 e o Node 10.	Manutenção O suporte para a versão 3.x do AWS Encryption SDK para JavaScript terminará em 17 de janeiro de 2024.
4.x	4,0	Requer a versão 3 AWS Encryption SDK para JavaScript dos `kms-client` para usar o AWS KMS chaveiro.	Disponibilidade geral (GA)

Python

Para obter uma descrição detalhada de todas as alterações, consulte o repositório https://github.com/aws/aws-encryption-sdk-python/ aws-encryption-sdk-python CHANGELOG.rst em. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
1.x	1,0	Versão inicial.	End-of-Support fase
	1.3	Adiciona suporte para gerenciador de materiais criptográficos e armazenamento em cache de chaves de dados. Transferido para a geração IV determinística.	

	1,7	Atualizações do AWS Encryption SDK que ajudam os usuários de versões anteriores a fazer o upgrade para a versão 2.0. x e mais tarde. Para obter mais informações, consulte a versão 1.7. x.	
2.x	2,0	Atualizações para AWS Encryption SDK o. Para obter mais informações, consulte a versão 2.0. x.	End-of-Support fase
	2.2	Melhorias no processo de decodificação de mensagens.	
	2.3	Adiciona suporte para chaves AWS KMS multirregionais.	
3.x	3.0	O AWS Encryption SDK for Python não oferece mais suporte ao Python 2 ou ao Python 3.4. A partir da versão principal 3. x do AWS Encryption SDK for Python, somente o Python 3.5 ou posterior é suportado.	Disponibilidade geral (GA)

4.x	4,0	Integra-o AWS Encryption SDK for Python com a Biblioteca de Provedores de Materiais (MPL).	Disponibilidade geral (GA)

Rust

Para obter uma descrição detalhada de todas as alterações, consulte o diretório Rust do repositório https://github.com/aws/aws-encryption-sdk/tree/mainline aws-encryption-sdk CHANGELOG.md em. GitHub

Versão principal	Detalhes		Fase do ciclo de vida da versão principal do SDK
1.x	1,0	Versão inicial.	Disponibilidade geral (GA)

Detalhes da versão

A lista a seguir descreve as principais diferenças entre as versões suportadas do AWS Encryption SDK.

Tópicos

- Versões anteriores à 1.7.x
- Versão 1.7.x
- Versão 2.0.x
- Versão 2.2.x
- Versão 2.3.x

Versões anteriores à 1.7.x

> **Note**
>
> Todos os 1. x. x versões do AWS Encryption SDK estão na fase de fim do suporte. Atualize para a versão mais recente disponível do AWS Encryption SDK para sua linguagem de programação assim que possível. Para atualizar de uma AWS Encryption SDK versão anterior à 1.7. x, você deve primeiro atualizar para 1.7. x. Para obter detalhes, consulte Migrando seu AWS Encryption SDK.

Versões AWS Encryption SDK anteriores à 1.7. x fornece recursos de segurança importantes, incluindo criptografia com o algoritmo Advanced Encryption Standard in Galois/Counter Mode (AES-GCM), uma função de derivação de chave HMAC-based de extração e expansão (HKDF), assinatura e uma chave de criptografia de 256 bits. No entanto, elas não são compatíveis com as práticas recomendadas por nós, incluindo confirmação de chave.

Versão 1.7.x

> **Note**
>
> Todos os 1. x. x versões do AWS Encryption SDK estão na fase de fim do suporte.

Versão 1.7. x foi projetado para ajudar os usuários de versões anteriores do a fazer o upgrade AWS Encryption SDK para a versão 2.0. x e mais tarde. Se você não conhece o AWS Encryption SDK, pode pular essa versão e começar com a versão mais recente disponível em sua linguagem de programação.

A versão 1.7.x é totalmente compatível com versões anteriores; ela não introduz nenhuma alteração significativa nem altera o comportamento do AWS Encryption SDK. Também é compatível com versões posteriores. Permite que você atualize seu código para que ele seja compatível com a versão 2.0.x.. Ela inclui novos atributos, mas não os habilita completamente. E requer valores de configuração que evitem que você adote imediatamente todos os novos atributos até que esteja pronto para fazer isso.

A versão 1.7.x inclui as seguintes alterações:

AWS KMS atualizações do provedor de chave mestra (obrigatório)

Versão 1.7. x introduz novos construtores no AWS Encryption SDK for Java e AWS Encryption SDK for Python que criam explicitamente provedores de chaves AWS KMS mestras no modo estrito ou de descoberta. Esta versão adiciona alterações semelhantes à interface de AWS Encryption SDK linha de comando (CLI). Para obter detalhes, consulte Atualizando provedores de chaves AWS KMS mestras.

- No modo estrito, os provedores de chaves mestras do AWS KMS exigem uma lista de chaves de encapsulamento e criptografam e descriptografam somente com as chaves de encapsulamento que você especificar. Essa é uma prática recomendada do AWS Encryption SDK que garante que você use as chaves de encapsulamento que pretende usar.
- No modo de descoberta, os provedores de chaves mestras do AWS KMS não aceitam nenhuma chave de encapsulamento. Você não pode usá-los para criptografar. Ao descriptografar, eles podem usar qualquer chave de encapsulamento para descriptografar uma chave de dados criptografada. No entanto, você pode limitar as chaves de encapsulamento usadas para descriptografia àquelas presentes em Contas da AWS específicas. Esse filtro de descoberta é opcional, mas é uma prática recomendada que incentivamos.

Os construtores que criam versões anteriores dos provedores de chaves AWS KMS mestras foram descontinuados na versão 1.7. x e removido na versão 2.0. x. Esses construtores instanciam provedores de chave mestra que criptografam usando as chaves de encapsulamento que você especificar. No entanto, eles descriptografam chaves de dados criptografadas usando a chave de encapsulamento que as criptografou, independentemente das chaves de encapsulamento especificadas. Os usuários podem descriptografar mensagens sem querer com chaves de embrulho que não pretendem usar, inclusive AWS KMS keys em outras regiões. Contas da AWS

Não há alterações nos construtores das chaves AWS KMS mestras. Ao criptografar e descriptografar, as chaves AWS KMS mestras usam somente o AWS KMS key que você especificar.

AWS KMS atualizações de chaveiros (opcional)

Versão 1.7. x adiciona um novo filtro às AWS Encryption SDK para JavaScript implementações AWS Encryption SDK for C e que limita os chaveiros de AWS KMS descoberta a determinados. Contas da AWS Esse novo filtro de conta é opcional, mas é uma prática recomendada que apoiamos. Para obter detalhes, consulte Atualizando AWS KMS chaveiros.

Não há alterações nos construtores dos AWS KMS chaveiros. AWS KMS Os chaveiros padrão se comportam como fornecedores de chaves mestras no modo estrito. AWS KMS os chaveiros de descoberta são criados explicitamente no modo de descoberta.

Passando um ID de chave para AWS KMS Decrypt

Começando na versão 1.7. x, ao descriptografar chaves de dados criptografadas, AWS Encryption SDK sempre especifica um AWS KMS key em suas chamadas para a AWS KMS operação Decrypt. O AWS Encryption SDK obtém o valor do ID da chave a AWS KMS key partir dos metadados em cada chave de dados criptografada. Esse atributo não requer alterações no código.

Não AWS KMS key é necessário especificar o ID da chave do para descriptografar o texto cifrado que foi criptografado em uma chave KMS de criptografia simétrica, mas é uma prática recomendada. AWS KMS Assim como especificar chaves de encapsulamento em seu provedor de chaves, essa prática garante que AWS KMS só decodifique usando a chave de encapsulamento que você pretende usar.

Decriptografar texto cifrado com confirmação de chave

A versão 1.7x pode descriptografar texto cifrado criptografado com ou sem confirmação de chave. No entanto, ela não pode criptografar texto cifrado com confirmação de chave. Essa propriedade permite que você implante totalmente aplicações que podem descriptografar texto cifrado criptografado com confirmação de chave antes mesmo de encontrem esse tipo de texto cifrado. Como essa versão descriptografa mensagens que são criptografadas sem confirmação de chave, você não precisa recriptografar nenhum texto cifrado.

Para implementar esse comportamento, versão 1.7. x inclui uma nova configuração de política de compromisso que determina se é AWS Encryption SDK possível criptografar ou descriptografar com comprometimento de chave. Na versão 1.7. x, o único valor válido para a política de compromisso, `ForbidEncryptAllowDecrypt`, é usado em todas as operações de criptografia e descriptografia. Esse valor impede que o AWS Encryption SDK criptografe com qualquer um dos novos pacotes de algoritmos que incluem confirmação de chave. Ele permite que você decifre AWS Encryption SDK o texto cifrado com e sem o comprometimento da chave.

Embora haja apenas um valor de política de compromisso válido na versão 1.7.x, exigimos que você defina esse valor explicitamente ao usar as novas APIs introduzidas nesta versão. Definir o valor explicitamente impede que sua política de compromisso seja alterada automaticamente para `require-encrypt-require-decrypt` quando você atualizar para a versão 2.1.x. Em vez disso, você pode migrar sua política de compromisso em etapas.

Pacotes de algoritmos com confirmação de chave fundamental

Versão 1.7. x inclui dois novos pacotes de algoritmos compatíveis com o confirmação de chaves. Um inclui assinatura; o outro, não. Como os conjuntos de algoritmos suportados anteriormente, esses dois novos conjuntos de algoritmos incluem criptografia com AES-GCM, uma chave de criptografia de 256 bits e uma função de derivação de chave HMAC-based de extração e expansão (HKDF).

No entanto, o conjunto de algoritmos padrão usado para criptografia não muda. Esses pacotes de algoritmos foram adicionados à versão 1.7.x para preparar a aplicação para usá-los nas versões 2.0.x e posteriores.

Alterações na implementação do CMM

A versão 1.7.x introduz mudanças na interface Default do gerenciador de materiais criptográficos (CMM) para dar suporte ao o comprometimento chave. Essa alteração lhe afeta somente se você tiver escrito um CMM personalizado. Para obter detalhes, consulte a documentação ou o GitHub repositório da API para sua linguagem ??? de programação.

Versão 2.0.x

Versão 2.0. O x oferece suporte aos novos recursos de segurança oferecidos no AWS Encryption SDK, incluindo chaves de empacotamento especificadas e comprometimento de chaves. A versão 2.0.x inclui alterações significativas em relação a todas as versões anteriores do AWS Encryption SDK. Você pode se preparar para essas mudanças implantando a versão 1.7.x.. A versão 2.0.x inclui todos os novos atributos introduzidos na versão 1.7.x com as seguintes adições e alterações.

> **Note**
>
> Versão 2. x. x do AWS Encryption SDK for Python, AWS Encryption SDK para JavaScript, e a CLI de AWS criptografia estão na fase de fim do suporte.
> Para obter informações sobre suporte e manutenção desta AWS Encryption SDK versão em sua linguagem de programação preferida, consulte o `SUPPORT_POLICY.rst` arquivo em seu GitHub repositório.

AWS KMS fornecedores de chaves mestras

Os construtores originais do provedor de chave AWS KMS mestra que foram descontinuados na versão 1.7. x são removidos na versão 2.0. x. Você deve criar explicitamente provedores de chaves mestras do AWS KMS no modo estrito ou no modo de descoberta.

Criptografe e descriptografe texto cifrado com confirmação de chave

A versão 2.0.x pode descriptografar texto cifrado criptografado com ou sem confirmação de chave. Seu comportamento é determinado pela definição da política de compromisso. Por padrão, ela sempre criptografa com confirmação de chave e só descriptografa texto cifrado criptografado com confirmação de chave. A menos que você altere a política de compromisso, o AWS Encryption SDK não descriptografa textos cifrados criptografados por nenhuma versão anterior do AWS Encryption SDK, incluindo a versão 1.7.x..

> **Important**
>
> Por padrão, a versão 2.0.x não descriptografa nenhum texto cifrado que tenha sido criptografado sem a confirmação de chave. Se a aplicação encontrar um texto cifrado criptografado sem confirmação de chave, defina um valor de política de compromisso como `AllowDecrypt`.

Na versão 2.0.x, a configuração da política de compromisso tem três valores válidos:

- `ForbidEncryptAllowDecrypt`: o AWS Encryption SDK não pode criptografar com confirmação de chave. Ele pode descriptografar textos cifrados criptografados com ou sem confirmação de chave.
- `RequireEncryptAllowDecrypt`: o AWS Encryption SDK deve criptografar com confirmação de chave. Ele pode descriptografar textos cifrados criptografados com ou sem confirmação de chave.
- `RequireEncryptRequireDecrypt`(padrão) — AWS Encryption SDK É necessário criptografar com comprometimento de chave. Ele só descriptografa textos cifrados com confirmação de chave.

Se você estiver migrando de uma versão anterior do AWS Encryption SDK para a versão 2.0. x, defina a política de compromisso com um valor que garanta que você possa descriptografar todos os textos cifrados existentes que seu aplicativo possa encontrar. É provável que você ajuste essa configuração com o tempo.

Versão 2.2.x

Adiciona suporte para assinaturas digitais e limita as chaves de dados criptografadas.

> **Note**
>
> Versão 2. x. x do AWS Encryption SDK for Python, AWS Encryption SDK para JavaScript, e a CLI de AWS criptografia estão na fase de fim do suporte.
> Para obter informações sobre suporte e manutenção desta AWS Encryption SDK versão em sua linguagem de programação preferida, consulte o `SUPPORT_POLICY.rst` arquivo em seu GitHub repositório.

Assinaturas digitais

Para melhorar o tratamento de assinaturas digitais durante a descriptografia, ele AWS Encryption SDK inclui os seguintes recursos:

- Non-streaming modo — retorna texto simples somente após o processamento de todas as entradas, incluindo a verificação da assinatura digital, se presente. Esse atributo impede que você use texto simples antes de verificar a assinatura digital. Use-o sempre que descriptografar dados criptografados com assinaturas digitais (o pacote de algoritmos padrão). Por exemplo, como a CLI de AWS criptografia sempre processa dados no modo de streaming, use o `--buffer` parâmetro ao descriptografar texto cifrado com assinaturas digitais.
- Unsigned-only modo de decodificação — esse recurso decodifica somente texto cifrado não assinado. Se a descriptografia encontrar uma assinatura digital no texto cifrado, a operação falhará. Use esse atributo para evitar o processamento não intencional de texto simples de mensagens assinadas antes de verificar a assinatura.

Limitar as chaves de dados criptografadas

Você pode limitar o número de chaves de dados criptografadas em uma mensagem criptografada. Esse atributo pode ajudar você a detectar um provedor de chave mestra ou um token de autenticação mal configurado ao criptografar ou a identificar um texto cifrado malicioso ao descriptografar.

Você deve limitar as chaves de dados criptografadas ao descriptografar mensagens de uma fonte não confiável. Isso evita chamadas desnecessárias, caras e potencialmente exaustivas para sua infraestrutura principal.

Versão 2.3.x

Adiciona suporte para chaves AWS KMS multirregionais. Para obter detalhes, consulte Usando várias regiões AWS KMS keys.

> **Note**
>
> A CLI AWS de criptografia oferece suporte a chaves multirregionais a partir da versão 3.0. x. Versão 2. x. x do AWS Encryption SDK for Python, AWS Encryption SDK para JavaScript, e a CLI de AWS criptografia estão na fase de fim do suporte.
> Para obter informações sobre suporte e manutenção desta AWS Encryption SDK versão em sua linguagem de programação preferida, consulte o `SUPPORT_POLICY.rst` arquivo em seu GitHub repositório.

Migrando seu AWS Encryption SDK

O AWS Encryption SDK suporta várias implementações de linguagem de programação interoperáveis, cada uma delas desenvolvida em um repositório de código aberto no. GitHub Como prática recomendada, recomendamos que você use a versão mais recente do AWS Encryption SDK para cada idioma.

Você pode atualizar com segurança a partir da versão 2.0. x ou posterior AWS Encryption SDK para a versão mais recente. No entanto, o 2.0. A versão x do AWS Encryption SDK introduz novos recursos de segurança significativos, alguns dos quais são mudanças significativas. Para atualizar de versões anteriores à 1.7.x para a versão 2.0.x e posteriores, primeiro será necessário atualizar para a versão 1.x mais recente. Os tópicos desta seção foram elaborados para ajudar você a entender as alterações, selecionar a versão correta para a aplicação e migrar com segurança e sucesso para as versões mais recentes do AWS Encryption SDK.

Para obter informações sobre versões significativas do AWS Encryption SDK, consulteVersões do AWS Encryption SDK.

Important

Não atualize diretamente de uma versão anterior à 1.7x para a versão 2.0.x ou posterior sem primeiro atualizar para a mais recente versão 1x.. Se você atualizar diretamente para a versão 2.0. x ou posterior e habilite todos os novos recursos imediatamente, eles não AWS Encryption SDK conseguirão descriptografar texto cifrado criptografado em versões mais antigas do. AWS Encryption SDK

Note

A versão mais antiga do AWS Encryption SDK para.NET é a versão 3.0. x. Todas as versões do AWS Encryption SDK para.NET oferecem suporte às melhores práticas de segurança introduzidas na versão 2.0. x do AWS Encryption SDK. É possível atualizar com segurança para a versão mais recente sem fazer alterações no código ou nos dados.
AWS CLI de criptografia: ao ler este guia de migração, use a versão 1.7. x instruções de migração para o AWS Encryption CLI 1.8. x e use o 2.0. x instruções de migração para o AWS Encryption CLI 2.1. x. Para obter detalhes, consulte Versões da CLI AWS de criptografia.

Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

Novos usuários

Se você é novo no AWS Encryption SDK, instale a versão mais recente do AWS Encryption SDK para sua linguagem de programação. Os valores padrão habilitam todos os recursos de segurança do AWS Encryption SDK, incluindo criptografia com assinatura, derivação de chave e comprometimento de chave. AWS Encryption SDK

Usuários atuais

Recomendamos atualizar da versão atual para a versão mais recente disponível assim que possível. Todos os 1. As versões x do AWS Encryption SDK estão em end-of-support fase, assim como as versões posteriores em algumas linguagens de programação. Para obter detalhes sobre o status de suporte e manutenção do AWS Encryption SDK em sua linguagem de programação, consulte Suporte e manutenção.

AWS Encryption SDK versões 2.0. x e versões posteriores fornecem novos recursos de segurança para ajudar a proteger seus dados. No entanto, AWS Encryption SDK a versão 2.0. x inclui alterações significativas que não são compatíveis com versões anteriores. Para garantir uma transição segura, comece migrando da sua versão atual para a mais recente 1.x na sua linguagem de programação. Quando a versão 1.x estiver totalmente implantada e operando com sucesso, você poderá migrar com segurança para as versões 2.0.x e posteriores. Esse processo de duas etapas é essencial, especialmente para aplicações distribuídas.

Para obter mais informações sobre os recursos AWS Encryption SDK de segurança subjacentes a essas mudanças, consulte Criptografia aprimorada do lado do cliente: compromisso explícito KeyIds e fundamental no Blog de Segurança.AWS

Procurando ajuda para usar o AWS Encryption SDK for Java com o AWS SDK for Java 2.x? Consulte Pré-requisitos.

Tópicos

- Como migrar e implantar o AWS Encryption SDK

- Atualizando provedores de chaves AWS KMS mestras
- Atualizando AWS KMS chaveiros
- Como definir sua política de compromisso
- Solução de problemas de migração para as versões mais recentes

Como migrar e implantar o AWS Encryption SDK

Ao migrar de uma AWS Encryption SDK versão anterior à 1.7. x até a versão 2.0. x ou mais tarde, você deve fazer a transição segura para a criptografia com comprometimento de chave. Caso contrário, a aplicação encontrará textos cifrados que não poderá descriptografar. Se você estiver usando provedores de chave AWS KMS mestra, deverá atualizar para novos construtores que criam provedores de chave mestra no modo estrito ou no modo de descoberta.

> **Note**
>
> Este tópico foi desenvolvido para usuários que estão migrando de versões anteriores do AWS Encryption SDK para a versão 2.0. x posterior. Se você é novo no AWS Encryption SDK, pode começar a usar a versão mais recente disponível imediatamente com as configurações padrão.

Para evitar uma situação crítica na qual você não possa descriptografar o texto cifrado que precisa ler, recomendamos que você migre e implante em várias etapas distintas. Verifique se cada etapa está completa e totalmente implantada antes de iniciar a próxima etapa. Isso é particularmente importante para aplicações distribuídas com vários hosts.

Etapa 1: atualize a aplicação para a versão 1.x mais recente

Atualize para a versão 1.x mais recente para sua linguagem de programação. Teste com cuidado, implante suas alterações e confirme se a atualização foi propagada para todos os hosts de destino antes de iniciar a etapa 2.

> **Important**
>
> Verifique se a sua versão 1.x mais recente é a versão 1.7.x ou versão posterior do AWS Encryption SDK.

O mais recente 1. As versões x do AWS Encryption SDK são compatíveis com versões anteriores do AWS Encryption SDK e versões anteriores com as versões 2.0. x e mais tarde. Elas incluem os novos atributos presentes na versão 2.0.x, mas inclui padrões seguros projetados para essa migração. Eles permitem que você atualize seus provedores de chave AWS KMS mestra, se necessário, e implante totalmente pacotes de algoritmos que podem decifrar texto cifrado com comprometimento de chave.

- Substitua elementos descontinuados, incluindo construtores para provedores de chaves metras do AWS KMS herdados. Em Python,ative os avisos de descontinuidade. Elementos de código que foram descontinuados na mais recente versão 1.x foram removidos das versões 2.0. x e posteriores.
- Defina explicitamente sua política de compromisso como `ForbidEncryptAllowDecrypt`. Embora esse seja o único valor válido no último 1. Nas versões x, essa configuração é necessária quando você usa a APIs introduzida nesta versão. Isso impede que a aplicação rejeite texto cifrado criptografado sem confirmação de chave quando você migra para a versão 2.0.x e versões posteriores. Para obter detalhes, consulte the section called “Como definir sua política de compromisso”.
- Se você usa provedores de chave AWS KMS mestra, deve atualizar seus provedores de chave mestra legados para provedores de chave mestra que ofereçam suporte ao modo estrito e ao modo de descoberta. Essa atualização é necessária para o AWS Encryption SDK for Java AWS Encryption SDK for Python, e para a CLI AWS de criptografia. Se você usa provedores de chave mestra no modo de descoberta, recomendamos que implemente o filtro de descoberta que limita as chaves de encapsulamento usadas àquelas presentes em Contas da AWS. Essa atualização é opcional, mas é uma prática recomendada que incentivamos. Para obter detalhes, consulte Atualizando provedores de chaves AWS KMS mestras.
- Se você usa token de autenticação de descoberta do AWS KMS, recomendamos que inclua um filtro de descoberta que limite as chaves de encapsulamento usadas na descriptografia a aquelas em particular. Contas da AWS Essa atualização é opcional, mas é uma prática recomendada que incentivamos. Para obter detalhes, consulte Atualizando AWS KMS chaveiros.

Etapa 2: atualize a aplicação para a versão mais recente

Depois que a implantação da mais recente versão 1.x for bem-sucedida em todos os hosts, você pode atualizar para as versões 2.0. x e posteriores. Versão 2.0. x inclui alterações significativas em todas as versões anteriores do AWS Encryption SDK. No entanto, se você fizer as alterações de código recomendadas na etapa 1, poderá evitar erros ao migrar para a versão mais recente.

Antes de atualizar para a versão mais recente, verifique se sua política de compromisso está consistentemente definida como `ForbidEncryptAllowDecrypt`. Em seguida, dependendo da configuração de dados, você pode migrar no seu próprio ritmo para `RequireEncryptAllowDecrypt` e depois para a configuração padrão, `RequireEncryptRequireDecrypt`. Recomendamos uma série de etapas de transição, como o padrão a seguir.

1. Comece com sua política de compromisso definida como `ForbidEncryptAllowDecrypt`. O AWS Encryption SDK pode descriptografar mensagens com confirmação de chave, mas ainda não descriptografa com confirmação de chave.
2. Quando estiver pronto, atualize a política de compromisso para `RequireEncryptAllowDecrypt`. AWS Encryption SDK Começa a criptografar seus dados com um compromisso fundamental. Ele poderá descriptografar textos cifrados criptografados com ou sem confirmação de chave.

   Antes de atualizar sua política de compromisso para `RequireEncryptAllowDecrypt`, verifique se sua versão 1x mais recente foi implantada em todos os hosts, incluindo os hosts de qualquer aplicação que decodifique o texto cifrado que você produz. Versões AWS Encryption SDK anteriores à versão 1.7. x não pode descriptografar mensagens criptografadas com comprometimento de chave.

   Esse também é um bom momento para adicionar métricas à sua aplicação para medir se você ainda está processando texto cifrado sem confirmação de chave. Isso ajudará você a determinar quando é seguro atualizar sua configuração de política de compromisso para `RequireEncryptRequireDecrypt`. Para algumas aplicações, como aquelas que criptografam mensagens em uma fila do Amazon SQS, isso pode significar esperar tempo suficiente para que todo o texto cifrado criptografado nas versões antigas seja recriptografado ou excluído. Para outras aplicações, como objetos criptografados do S3, talvez seja necessário baixar, recriptografar e recarregar todos os objetos.
3. Quando tiver certeza de que não tem nenhuma mensagem criptografada sem confirmação de chave, você pode atualizar sua política de compromisso para `RequireEncryptRequireDecrypt`. Esse valor garante que seus dados sejam sempre criptografados e descriptografados com o confirmação de chave. Essa configuração é a padrão, então você não precisa defini-la explicitamente, mas recomendamos que faça isso. Uma configuração explícita ajudará na depuração e em quaisquer possíveis reversões que possam ser necessárias se a aplicação encontrar texto cifrado criptografado sem confirmação de chave.

Atualizando provedores de chaves AWS KMS mestras

Para migrar para o mais recente 1. versão x do e AWS Encryption SDK, em seguida, para a versão 2.0. x ou posterior, você deve substituir os provedores de chave AWS KMS mestra legados por provedores de chave mestra criados explicitamente no modo estrito ou no modo de descoberta. Os provedores de chave mestra herdados foram descontinuados na versão 1.7.x e foram removidos na versão 2.0. x. Essa alteração é necessária para aplicações e scripts que usam o AWS Encryption SDK for Java, o AWS Encryption SDK for Python e a CLI de criptografia da AWS. Os exemplos nesta seção mostrarão como atualizar seu código.

> **Note**
>
> Em Python, ative os avisos de obsolescência. Isso ajudará você a identificar as partes do código que precisa atualizar.

Se você estiver usando uma chave AWS KMS mestra (não um provedor de chave mestra), você pode pular esta etapa. AWS KMS as chaves mestras não estão obsoletas nem foram removidas. Elas criptografam e descriptografam somente com as chaves de encapsulamento que você especificar.

Os exemplos nesta seção se concentram nos elementos do seu código precisam ser alterados. Para obter um exemplo completo do código atualizado, consulte a seção Exemplos do GitHub repositório da sua linguagem de programação. Além disso, esses exemplos normalmente usam ARNs a chave para representar AWS KMS keys. Ao criar um provedor de chave mestra para criptografia, você pode usar qualquer identificador de AWS KMS chave válido para representar um AWS KMS key . Ao criar um provedor de chave mestra para descriptografia, você deve usar um ARN de chave.

Saiba mais sobre migração

Para todos os AWS Encryption SDK usuários, saiba como definir sua política de compromisso emthe section called “Como definir sua política de compromisso”.

Para AWS Encryption SDK para JavaScript usuários AWS Encryption SDK for C e usuários, saiba mais sobre uma atualização opcional dos chaveiros emAtualizando AWS KMS chaveiros.

Tópicos

- Migração para o modo estrito

- Migrar para o modo de descoberta

Migração para o modo estrito

Depois de atualizar para o mais recente 1. versão x do AWS Encryption SDK, substitua seus provedores de chave mestra legados por provedores de chave mestra no modo estrito. No modo estrito, você deve especificar as chaves de encapsulamento a serem usadas ao criptografar e descriptografar. O AWS Encryption SDK usa somente as chaves de empacotamento que você especificar. Provedores de chaves mestras obsoletas podem descriptografar dados usando qualquer um AWS KMS key que criptografe uma chave de dados, inclusive em diferentes regiões. AWS KMS keys Contas da AWS

Os provedores de chaves mestras no modo estrito são introduzidos na AWS Encryption SDK versão 1.7. x. Eles substituem os provedores de chaves mestras herdados, que foram suspensos na versão 1.7.x e removidos na versão 2.0.x.. Usar provedores de chave mestra no modo estrito é uma prática AWS Encryption SDK recomendada.

O código a seguir cria um provedor de chave mestra no modo estrito que você pode usar para criptografar e descriptografar.

Java

Este exemplo representa o código em uma aplicação que usa a versão 1.6.2 ou anterior do AWS Encryption SDK for Java.

Esse código usa o `KmsMasterKeyProvider.builder()` método para instanciar um provedor de chave AWS KMS mestra que usa um AWS KMS key como chave de encapsulamento.

```
// Create a master key provider
// Replace the example key ARN with a valid one
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .withKeysForEncryption(awsKmsKey)
    .build();
```

Este exemplo representa o código em uma aplicação que usa a versão 1.7.x ou versões posteriores do AWS Encryption SDK for Java . Para ver um exemplo completo, consulte BasicEncryptionExample.java.

Os métodos `Builder.build()` e `Builder.withKeysForEncryption()` usados no exemplo anterior foram suspensos na versão 1.7.x e removidos da versão 2.0.x..

Para atualizar para um provedor de chave mestra de modo estrito, esse código substitui as chamadas para métodos suspensos por uma chamada para o novo método `Builder.buildStrict()`. Este exemplo especifica uma AWS KMS key como chave de empacotamento, mas o `Builder.buildStrict()` método pode usar uma lista de várias. AWS KMS keys

```
// Create a master key provider in strict mode
// Replace the example key ARN with a valid one from your Conta da AWS.
String awsKmsKey = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);
```

Python

Este exemplo representa o código em uma aplicação que usa a versão 1.4.1 do AWS Encryption SDK for Python. Esse código usa `KMSMasterKeyProvider`, que foi suspenso na versão 1.7.x e removido da versão 2.0.x.. Ao descriptografar, ele usa qualquer uma AWS KMS key que criptografe uma chave de dados sem levar em conta o AWS KMS keys que você especificar.

Observe que `KMSMasterKey` não foi suspenso nem removido. Ao criptografar e descriptografar, ele usa somente o que você especifica. AWS KMS key

```
# Create a master key provider
# Replace the example key ARN with a valid one
key_1 = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"

aws_kms_master_key_provider = KMSMasterKeyProvider(
   key_ids=[key_1, key_2]
)
```

Este exemplo representa o código em uma aplicação que usa a versão 1.7.x do AWS Encryption SDK for Python. Para ver um exemplo completo, consulte basic_encryption.py.

Para atualizar para um provedor de chave mestra de modo estrito, esse código substitui a chamada para `KMSMasterKeyProvider()` com uma chamada para `StrictAwsKmsMasterKeyProvider()`.

```
# Create a master key provider in strict mode
# Replace the example key ARNs with valid values from your Conta da AWS
key_1 = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab"
key_2 = "arn:aws:kms:us-west-2:111122223333:key/0987dcba-09fe-87dc-65ba-
ab0987654321"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[key_1, key_2]
)
```

AWS Encryption CLI

Este exemplo mostra como criptografar e descriptografar usando a versão 1.1.7 ou anterior do Encryption AWS CLI.

Na versão 1.1.7 e anteriores, ao criptografar, você especifica uma ou mais chaves mestras (ou chaves de encapsulamento), como uma AWS KMS key. Ao descriptografar, você não pode especificar nenhuma chave de encapsulamento, a menos que esteja usando um provedor de chave mestra personalizado. A CLI de AWS criptografia pode usar qualquer chave de empacotamento que criptografe uma chave de dados.

```
\\ Replace the example key ARN with a valid one
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --master-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
```

```
                    --output .
```

Este exemplo mostra como criptografar e descriptografar usando a versão 1.7 do Encryption AWS CLI. x ou mais tarde. Para obter exemplos completos, consulte Exemplos da CLI AWS de criptografia.

O parâmetro `--master-keys` foi suspenso na versão 1.7.x e removido na versão 2.0.x.. Ele foi substituído pelo parâmetro `--wrapping-keys`, que é exigido nos comandos de encrypt e decrypt. Esse parâmetro é compatível com o modo estrito e o modo de descoberta. O modo estrito é uma prática AWS Encryption SDK recomendada que garante que você use a chave de encapsulamento desejada.

Para atualizar para o modo estrito, use o atributo key do parâmetro `--wrapping-keys` para especificar uma chave de encapsulamento ao criptografar e descriptografar.

```
\\ Replace the example key ARN with a valid value
$ keyArn=arn:aws:kms:us-west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

Migrar para o modo de descoberta

A partir da versão 1.7. x, é uma prática AWS Encryption SDK recomendada usar o modo estrito para provedores de chaves AWS KMS mestras, ou seja, especificar chaves de agrupamento ao criptografar e descriptografar. Você deve sempre especificar as chaves de encapsulamento ao criptografar. Mas há situações em que especificar a chave ARNs AWS KMS keys para

descriptografar é impraticável. Por exemplo, se você estiver usando aliases para identificar AWS KMS keys ao criptografar, perderá o benefício dos aliases se precisar listar a chave ARNs ao descriptografar. Além disso, como os provedores de chave mestra no modo de descoberta se comportam como os provedores de chave mestra originais, você pode usá-los temporariamente como parte de sua estratégia de migração e, posteriormente, atualizar para provedores de chave mestra no modo estrito.

Em casos como esse, você pode usar provedores de chaves mestras no modo de descoberta. Esses provedores de chaves mestras não permitem que você especifique chaves de encapsulamento, portanto, você não pode usá-los para criptografar. Ao descriptografar, eles podem usar qualquer chave de encapsulamento que criptografe uma chave de dados. Mas, diferentemente dos provedores de chaves mestras herdados, que se comportam da mesma maneira, você cria esses provedores explicitamente no modo de descoberta. Ao usar provedores de chave mestra no modo de descoberta, você pode limitar as chaves de encapsulamento que podem ser usadas para aquelas que estão presentes em Contas da AWS específicas. Esse filtro de descoberta é opcional, mas é uma prática recomendada que incentivamos. Para obter informações sobre partições e contas da AWS , consulte Nomes do atributo da Amazon no Referência geral da AWS.

Os exemplos a seguir criam um provedor de chave AWS KMS mestra no modo estrito para criptografia e um provedor de chave AWS KMS mestra no modo de descoberta para descriptografia. O provedor da chave mestra no modo de descoberta usa um filtro de descoberta para limitar as chaves de encapsulamento usadas para descriptografar à partição `aws` e ao exemplo específico de Contas da AWS. Embora o filtro de conta não seja necessário neste exemplo bastante simples, é uma prática recomendada muito benéfica quando uma aplicação criptografa os dados e outra diferente os descriptografa.

Java

Este exemplo representa o código em uma aplicação que usa a versão 1.7.x ou versões posteriores do AWS Encryption SDK for Java. Para ver um exemplo completo, consulte DiscoveryDecryptionExample.java.

Para instanciar um provedor de chave mestra no modo estrito para criptografar, este exemplo usa o método `Builder.buildStrict()`. Para instanciar um provedor de chave mestra no modo de descoberta para descriptografar ele usa o método `Builder.buildDiscovery()`. O `Builder.buildDiscovery()` método usa um `DiscoveryFilter` que limita o AWS Encryption SDK to AWS KMS keys na AWS partição e nas contas especificadas.

```
// Create a master key provider in strict mode for encrypting
```

```
// Replace the example alias ARN with a valid one from your Conta da AWS.
String awsKmsKey = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias";

KmsMasterKeyProvider encryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

// Create a master key provider in discovery mode for decrypting
// Replace the example account IDs with valid values.
DiscoveryFilter accounts = new DiscoveryFilter("aws", Arrays.asList("111122223333",
 "444455556666"));

KmsMasterKeyProvider decryptingKeyProvider = KmsMasterKeyProvider.builder()
    .buildDiscovery(accounts);
```

Python

Este exemplo representa o código em uma aplicação que usa a versão 1.7.x ou versões posteriores do AWS Encryption SDK for Python . Para obter um exemplo completo, consulte: discovery_kms_provider.py.

Para criar um provedor de chave mestra no modo estrito para criptografar, este exemplo usa o método `StrictAwsKmsMasterKeyProvider`. Para criar um provedor de chave mestra no modo de descoberta para descriptografia, ele usa `DiscoveryAwsKmsMasterKeyProvider` um `DiscoveryFilter` que limita o AWS Encryption SDK to AWS KMS keys na AWS partição e nas contas especificadas.

```
# Create a master key provider in strict mode
# Replace the example key ARN and alias ARNs with valid values from your Conta da
 AWS.
key_1 = "arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias"
key_2 = "arn:aws:kms:us-
west-2:444455556666:key/1a2b3c4d-5e6f-1a2b-3c4d-5e6f1a2b3c4d"

aws_kms_master_key_provider = StrictAwsKmsMasterKeyProvider(
    key_ids=[key_1, key_2]
)

# Create a master key provider in discovery mode for decrypting
# Replace the example account IDs with valid values
accounts = DiscoveryFilter(
    partition="aws",
    account_ids=["111122223333", "444455556666"]
```

```
)
aws_kms_master_key_provider = DiscoveryAwsKmsMasterKeyProvider(
        discovery_filter=accounts
)
```

AWS Encryption CLI

Este exemplo mostra como criptografar e descriptografar usando a versão 1.7 do Encryption AWS CLI. x ou mais tarde. A partir da versão 1.7.x, o parâmetro `--wrapping-keys` passou a ser necessário ao criptografar e descriptografar. O parâmetro `--wrapping-keys` é compatível com o modo estrito e o modo de descoberta. Para obter exemplos completos, consulte the section called “Exemplos”.

Ao criptografar, este exemplo especifica uma chave de encapsulamento, que é obrigatória. Ao descriptografar, ele escolhe explicitamente o modo de descoberta usando o atributo `discovery` do parâmetro `--wrapping-keys` com um valor definido como `true`.

Para limitar as chaves de encapsulamento que AWS Encryption SDK podem ser usadas no modo de descoberta àquelas em particular Contas da AWS, este exemplo usa os `discovery-account` atributos `discovery-partition` e do `--wrapping-keys` parâmetro. Esses atributos opcionais são válidos somente quando o atributo `discovery` for definido como `true`. Você deve usar os atributos `discovery-partition` e `discovery-account` juntos. Nenhum deles é válido sozinho.

```
\\ Replace the example key ARN with a valid value
$ keyAlias=arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias

\\ Encrypt your plaintext data
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyAlias \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext
\\ Replace the example account IDs with valid values
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys discovery=true \
                                     discovery-partition=aws \
```

```
                                discovery-account=111122223333 \
                                discovery-account=444455556666 \
                        --encryption-context purpose=test \
                        --metadata-output ~/metadata \
                        --output .
```

Atualizando AWS KMS chaveiros

Os AWS KMS chaveiros do AWS Encryption SDK for C, do AWS Encryption SDK para o.NET e do AWS Encryption SDK para JavaScriptoferecem suporte às melhores práticas, permitindo que você especifique chaves de agrupamento ao criptografar e descriptografar. Se você criar um token de autenticação de descoberta do AWS KMS, você o fará de maneira explícita.

> **Note**
>
> A versão mais antiga do AWS Encryption SDK para.NET é a versão 3.0. x. Todas as versões do AWS Encryption SDK para.NET oferecem suporte às melhores práticas de segurança introduzidas na versão 2.0. x do AWS Encryption SDK. É possível atualizar com segurança para a versão mais recente sem fazer alterações no código ou nos dados.

Quando você atualiza para o mais recente 1. Na versão x do AWS Encryption SDK, você pode usar um filtro de descoberta para limitar as chaves de agrupamento que um chaveiro de AWS KMS descoberta ou um chaveiro de descoberta AWS KMS regional usa ao descriptografar para aquelas em particular. Contas da AWS Filtrar um chaveiro de descoberta é uma prática AWS Encryption SDK recomendada.

Os exemplos nesta seção mostram como adicionar o filtro de descoberta a um token de autenticação de descoberta regional do AWS KMS .

Saiba mais sobre migração

Para todos os AWS Encryption SDK usuários, saiba como definir sua política de compromisso emthe section called “Como definir sua política de compromisso”.

Para usuários da CLI de AWS criptografia e AWS Encryption SDK for Java AWS Encryption SDK for Python, saiba mais sobre uma atualização necessária para os provedores de chaves mestras em. the section called “Atualizando provedores de chaves AWS KMS mestras”

Você pode ter um código como o seguinte na aplicação. Este exemplo cria um token de autenticação de descoberta regional do AWS KMS que só pode usar chaves de encapsulamento na região Oeste dos EUA (Oregon) (us-west-2). Este exemplo representa o código em AWS Encryption SDK versões anteriores à 1.7. x. No entanto, ele ainda é válido nas versões 1.7x e posteriores.

C

```
struct aws_cryptosdk_keyring *kms_regional_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder()
       .WithKmsClient(create_kms_client(Aws::Region::US_WEST_2)).BuildDiscovery());
```

JavaScript Browser

```
const clientProvider = getClient(KMS, { credentials })

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser({ clientProvider, discovery })
```

JavaScript Node.js

```
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({ clientProvider, discovery })
```

A partir da versão 1.7. x, você pode adicionar um filtro de descoberta a qualquer chaveiro de AWS KMS descoberta. Esse filtro de descoberta limita o AWS KMS keys que eles AWS Encryption SDK podem usar para decodificação àqueles na partição e nas contas especificadas. Antes de usar esse código, altere a partição, se necessário, e substitua a conta IDs de exemplo por outras válidas.

C

Para obter um exemplo completo, consulte: kms_discovery.cpp.

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

```
        .AddAccount("444455556666")
        .Build());

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

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

JavaScript Browser

```
const clientProvider = getClient(KMS, { credentials })

const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringBrowser(clientProvider, {
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

JavaScript Node.js

Para obter um exemplo completo, consulte: kms_filtered_discovery.ts.

```
const discovery = true
const clientProvider = limitRegions(['us-west-2'], getKmsClient)
const keyring = new KmsKeyringNode({
    clientProvider,
    discovery,
    discoveryFilter: { accountIDs: ['111122223333', '444455556666'], partition:
 'aws' }
})
```

Como definir sua política de compromisso

O confirmação de chave assegura que seus dados criptografados sempre sejam descriptografados para o mesmo texto simples. Para fornecer essa propriedade de segurança, começando na versão 1.7. x, o AWS Encryption SDK usa novos conjuntos de algoritmos com comprometimento fundamental. Para determinar se seus dados são criptografados e descriptografados com confirmação de chave, use a definição de configuração da política de compromisso. Criptografar e

descriptografar dados com confirmação de chave é uma prática recomendada do AWS Encryption SDK .

Definir uma política de compromisso é uma parte importante da segunda etapa do processo de migração — migrar da última 1. versões x das AWS Encryption SDK duas versões 2.0. x e mais tarde. Após definir e alterar sua política de compromisso, certifique-se de testar a aplicação minuciosamente antes de implantá-la em produção. Para obter orientação sobre migração, consulte Como migrar e implantar o AWS Encryption SDK.

A configuração da política de compromisso tem três valores válidos nas versões 2.0. x posteriores. Nas versões 1.x mais recentes (a partir da versão 1.7.x), somente `ForbidEncryptAllowDecrypt` é válido.

- `ForbidEncryptAllowDecrypt`— Eles AWS Encryption SDK não podem criptografar com comprometimento chave. Ele pode descriptografar textos cifrados criptografados com ou sem confirmação de chave.

  Na mais recente versão 1.x, esse é o único valor válido. Isso garante que você não criptografe com confirmação de chave até que esteja totalmente preparado para descriptografar com confirmação de chave. Definir o valor explicitamente impede que sua política de compromisso seja alterada automaticamente para `require-encrypt-require-decrypt` quando você atualizar para as versões 2.0.x ou posteriores. Em vez disso, você pode migrar sua política de compromisso em etapas.
- `RequireEncryptAllowDecrypt`— AWS Encryption SDK Sempre criptografa com comprometimento fundamental. Ele pode descriptografar textos cifrados criptografados com ou sem confirmação de chave. Esse valor foi adicionado na versão 2.0.x..
- `RequireEncryptRequireDecrypt`— AWS Encryption SDK Sempre criptografa e descriptografa com comprometimento fundamental. Esse valor foi adicionado na versão 2.0.x.. É o valor padrão em versões 2.0.x. e posteriores.

Na versão 1.x mais recente, o único valor de política de compromisso válido é `ForbidEncryptAllowDecrypt`. Depois de migrar para a versão 2.0. x ou posterior, você pode alterar sua política de compromisso em etapas conforme estiver pronto. Não atualize sua política de compromisso `RequireEncryptRequireDecrypt` até ter certeza de que não tem nenhuma mensagem criptografada sem o confirmação de chave.

Esses exemplos mostram como definir sua política de compromisso na última versão 1.x e nas versões 2.0.x posteriores. A técnica depende da sua linguagem de programação.

Saiba mais sobre migração

Para AWS Encryption SDK for Java, AWS Encryption SDK for Python, e a CLI de AWS criptografia, saiba mais sobre as mudanças necessárias nos provedores de chaves mestras em. the section called “Atualizando provedores de chaves AWS KMS mestras”

Para AWS Encryption SDK for C e AWS Encryption SDK para JavaScript, saiba mais sobre uma atualização opcional dos chaveiros emAtualizando AWS KMS chaveiros.

Como definir sua política de compromisso

A técnica que você usa para definir sua política de compromisso difere um pouco em cada implementação de linguagem. Esses exemplos .mostram a você como fazer isso. Antes de alterar sua política de compromisso, revise a abordagem de vários estágios em Como migrar e implantar.

C

A partir da versão 1.7. x do AWS Encryption SDK for C, você usa a `aws_cryptosdk_session_set_commitment_policy` função para definir a política de compromisso em suas sessões de criptografia e descriptografia. A política de compromisso que você define se aplica a todas as operações de criptografia e descriptografia chamadas na sua sessão.

As funções `aws_cryptosdk_session_new_from_keyring` e `aws_cryptosdk_session_new_from_cmm` foram descontinuadas na versão 1.7.x e foram removidas na versão 2.0.x.. Essas funções foram substituídas pelas funções `aws_cryptosdk_session_new_from_keyring_2` e `aws_cryptosdk_session_new_from_cmm_2` que retornam uma sessão.

Ao usar `aws_cryptosdk_session_new_from_keyring_2` e `aws_cryptosdk_session_new_from_cmm_2` na versão 1.x mais recentes, você deve chamar a função `aws_cryptosdk_session_set_commitment_policy` com o valor da política de compromisso `COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT`. Nas versões 2.0.x e posteriores, chamar essa função é opcional, e ela aceita todos os valores válidos. A política de compromisso padrão para as versões 2.0. x e posteriores é `COMMITMENT_POLICY_REQUIRE_ENCRYPT_REQUIRE_DECRYPT`.

Para obter um exemplo completo, consulte string.cpp.

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

```
/* Create an AWS KMS keyring */
const char * key_arn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);

/* Create an encrypt session with a CommitmentPolicy setting */
struct aws_cryptosdk_session *encrypt_session =
 aws_cryptosdk_session_new_from_keyring_2(
    alloc, AWS_CRYPTOSDK_ENCRYPT, kms_keyring);

aws_cryptosdk_keyring_release(kms_keyring);
aws_cryptosdk_session_set_commitment_policy(encrypt_session,
    COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);

...
/* Encrypt your data */

size_t plaintext_consumed_output;
aws_cryptosdk_session_process(encrypt_session,
                              ciphertext_output,
                              ciphertext_buf_sz_output,
                              ciphertext_len_output,
                              plaintext_input,
                              plaintext_len_input,
                              &plaintext_consumed_output)
...

/* Create a decrypt session with a CommitmentPolicy setting */

struct aws_cryptosdk_keyring *kms_keyring =
 Aws::Cryptosdk::KmsKeyring::Builder().Build(key_arn);
struct aws_cryptosdk_session *decrypt_session =
 *aws_cryptosdk_session_new_from_keyring_2(
        alloc, AWS_CRYPTOSDK_DECRYPT, kms_keyring);
aws_cryptosdk_keyring_release(kms_keyring);
aws_cryptosdk_session_set_commitment_policy(decrypt_session,
        COMMITMENT_POLICY_FORBID_ENCRYPT_ALLOW_DECRYPT);

/* Decrypt your ciphertext */
size_t ciphertext_consumed_output;
aws_cryptosdk_session_process(decrypt_session,
                              plaintext_output,
```

```
                                plaintext_buf_sz_output,
                                plaintext_len_output,
                                ciphertext_input,
                                ciphertext_len_input,
                                &ciphertext_consumed_output)
```

C# / .NET

O `require-encrypt-require-decrypt` valor é a política de compromisso padrão em todas as versões do AWS Encryption SDK para.NET. Você pode definir isso explicitamente como uma prática recomendadas, mas isso não é necessário. No entanto, se você estiver usando o for.NET AWS Encryption SDK para descriptografar texto cifrado que foi criptografado por outra implementação de linguagem do AWS Encryption SDK sem compromisso de chave, você precisará alterar o valor da política de compromisso para ou. `REQUIRE_ENCRYPT_ALLOW_DECRYPT` `FORBID_ENCRYPT_ALLOW_DECRYPT` Caso contrário, ocorrerá uma falha na tentativa de descriptografar o texto cifrado.

No AWS Encryption SDK para.NET, você define a política de compromisso em uma instância do AWS Encryption SDK. Instancie um `AwsEncryptionSdkConfig` objeto com um `CommitmentPolicy` parâmetro e use o objeto de configuração para criar a AWS Encryption SDK instância. Em seguida, chame os `Decrypt()` métodos `Encrypt()` e da AWS Encryption SDK instância configurada.

Este exemplo define a política de compromisso como `require-encrypt-allow-decrypt`.

```
// Instantiate the material providers
var materialProviders =

 AwsCryptographicMaterialProvidersFactory.CreateDefaultAwsCryptographicMaterialProviders();

// Configure the commitment policy on the AWS Encryption SDK instance
var config = new AwsEncryptionSdkConfig
{
    CommitmentPolicy = CommitmentPolicy.REQUIRE_ENCRYPT_ALLOW_DECRYPT
};
var encryptionSdk = AwsEncryptionSdkFactory.CreateAwsEncryptionSdk(config);

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

var encryptionContext = new Dictionary<string, string>()
{
```

```
    {"purpose", "test"}encryptionSdk
};

var createKeyringInput = new CreateAwsKmsKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = keyArn
};
var keyring = materialProviders.CreateAwsKmsKeyring(createKeyringInput);

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

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

AWS Encryption CLI

Para definir uma política de compromisso na CLI de AWS criptografia, use o `--commitment-policy` parâmetro. Esse parâmetro foi apresentado na versão 1.8.x..

No versão 1.x mais recente, quando você usa o parâmetro `--wrapping-keys` em um comando `--encrypt`ou `--decrypt`, é necessário um parâmetro `--commitment-policy` com o valor `forbid-encrypt-allow-decrypt`. Caso contrário, o parâmetro `--commitment-policy` será inválido.

Nas versões 2.1.x e posteriores, o parâmetro `--commitment-policy` é opcional e usa como padrão o valor `require-encrypt-require-decrypt`, que não criptografará nem descriptografará nenhum texto cifrado criptografado sem confirmação de chave. No entanto, recomendamos definir a política de compromisso de forma explícita em todas as chamadas de criptografia e descriptografia, para ajudar na manutenção e na solução de problemas.

Este exemplo define a política de compromisso como . Ele também usa o parâmetro `--wrapping-keys`, que substituiu o parâmetro `--master-keys` a partir da versão 1.8.x.. Para obter detalhes, consulte the section called “Atualizando provedores de chaves AWS KMS mestras”. Para obter exemplos completos, consulte Exemplos da CLI AWS de criptografia.

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

\\ Encrypt your plaintext data - no change to algorithm suite used
$ aws-encryption-cli --encrypt \
                     --input hello.txt \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy forbid-encrypt-allow-decrypt \
                     --metadata-output ~/metadata \
                     --encryption-context purpose=test \
                     --output .

\\ Decrypt your ciphertext - supports key commitment on 1.7 and later
$ aws-encryption-cli --decrypt \
                     --input hello.txt.encrypted \
                     --wrapping-keys key=$keyArn \
                     --commitment-policy forbid-encrypt-allow-decrypt \
                     --encryption-context purpose=test \
                     --metadata-output ~/metadata \
                     --output .
```

Java

A partir da versão 1.7. x do AWS Encryption SDK for Java, você define a política de compromisso em sua instância do `AwsCrypto` objeto que representa o AWS Encryption SDK cliente. Essa definição política de compromisso se aplica a todas as operações de criptografia e descriptografia chamadas nesse cliente.

O `AwsCrypto()` construtor está obsoleto na última versão 1. As versões x do AWS Encryption SDK for Java e são removidas na versão 2.0. x. Ele foi substituído por uma nova classe `Builder`, um método `Builder.withCommitmentPolicy()` e pelo tipo enumerado `CommitmentPolicy`.

Nas versões 1.x mais recentes, a classe `Builder` requer o método `Builder.withCommitmentPolicy()` e o argumento`CommitmentPolicy.ForbidEncryptAllowDecrypt`. A partir da versão

2.0.x, o método `Builder.withCommitmentPolicy()` é opcional. O valor padrão é `CommitmentPolicy.RequireEncryptRequireDecrypt`.

Para ver um exemplo completo, consulte SetCommitmentPolicyExample.java.

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

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

KmsMasterKeyProvider masterKeyProvider = KmsMasterKeyProvider.builder()
    .buildStrict(awsKmsKey);

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

// Decrypt your ciphertext
CryptoResult<byte[], KmsMasterKey> decryptResult = crypto.decryptData(
        masterKeyProvider,
        ciphertext);
byte[] decrypted = decryptResult.getResult();
```

JavaScript

A partir da versão 1.7. x do AWS Encryption SDK para JavaScript, você pode definir a política de compromisso ao chamar a nova `buildClient` função que instancia um AWS Encryption SDK cliente. A função `buildClient` assume um valor enumerado que representa sua política de compromisso. Ela retorna as funções `encrypt` e `decrypt` atualizadas, que reforçam sua política de compromisso quando você criptografa e descriptografa.

Nas versões 1.x mais recentes, a função `buildClient` requer o argumento `CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT`. A partir da versão 2.0.x, o argumento da política de compromisso é opcional, e o valor padrão é `CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT`.

O código para Node.js e o navegador são idênticos para essa finalidade, exceto que o navegador precisa de uma instrução para definir as credenciais.

O exemplo a seguir criptografa os dados com um AWS KMS chaveiro. A nova função `buildClient` define a política de compromisso como`FORBID_ENCRYPT_ALLOW_DECRYPT`, o valor padrão nas versões 1.x. mais recentes. A funções `encrypt` e `decrypt` atualizadas retornadas por `buildClient` reforçam a política de compromisso que você definiu.

```
import { buildClient } from '@aws-crypto/client-node'
const { encrypt, decrypt } =
 buildClient(CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT)

// Create an AWS KMS keyring
const generatorKeyId = 'arn:aws:kms:us-west-2:111122223333:alias/ExampleAlias'
const keyIds = ['arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab']
const keyring = new KmsKeyringNode({ generatorKeyId, keyIds })

// Encrypt your plaintext data
const { ciphertext } = await encrypt(keyring, plaintext, { encryptionContext:
 context })

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

Python

A partir da versão 1.7. x do AWS Encryption SDK for Python, você define a política de compromisso em sua instância de`EncryptionSDKClient`, um novo objeto que representa o AWS Encryption SDK cliente. A política de compromisso que você define se aplica a todas as chamadas `encrypt` e `decrypt` que usam essa instância do cliente.

Nas versões 1.x mais recentes, o construtor `EncryptionSDKClient` requer o valor enumerado `CommitmentPolicy.FORBID_ENCRYPT_ALLOW_DECRYPT`. A partir da versão 2.0.x, o argumento da política de compromisso é opcional, e o valor padrão é `CommitmentPolicy.REQUIRE_ENCRYPT_REQUIRE_DECRYPT`.

Este exemplo usa o novo construtor`EncryptionSDKClient` e define a política de compromisso como o valor padrão de 1.7.x. O construtor instancia um cliente que representa o AWS Encryption SDK. Quando você chama os métodos `encrypt`,`decrypt` ou `stream` desse cliente, eles aplicam a política de compromisso que você definiu. Esse exemplo também usa o novo construtor

da `StrictAwsKmsMasterKeyProvider` classe, que especifica AWS KMS keys quando criptografar e descriptografar.

Para obter um exemplo completo, consulte set_commitment.py.

```
# Instantiate the client
client =
 aws_encryption_sdk.EncryptionSDKClient(commitment_policy=CommitmentPolicy.FORBID_ENCRYPT_AL

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

# Encrypt your plaintext data
ciphertext, encrypt_header = client.encrypt(
        source=source_plaintext,
        encryption_context=encryption_context,
        master_key_provider=aws_kms_strict_master_key_provider
)

# Decrypt your ciphertext
decrypted, decrypt_header = client.decrypt(
        source=ciphertext,
        master_key_provider=aws_kms_strict_master_key_provider
)
```

Rust

O `require-encrypt-require-decrypt` valor é a política de compromisso padrão em todas as versões do AWS Encryption SDK for Rust. Você pode definir isso explicitamente como uma prática recomendadas, mas isso não é necessário. No entanto, se você estiver usando o for Rust AWS Encryption SDK para descriptografar texto cifrado que foi criptografado por outra implementação de linguagem do AWS Encryption SDK sem compromisso de chave, você precisará alterar o valor da política de compromisso para ou. `REQUIRE_ENCRYPT_ALLOW_DECRYPT` `FORBID_ENCRYPT_ALLOW_DECRYPT` Caso contrário, ocorrerá uma falha na tentativa de descriptografar o texto cifrado.

No AWS Encryption SDK for Rust, você define a política de compromisso em uma instância do AWS Encryption SDK. Instancie um `AwsEncryptionSdkConfig` objeto com um

`comitment_policy` parâmetro e use o objeto de configuração para criar a AWS Encryption SDK instância. Em seguida, chame os `Decrypt()` métodos `Encrypt()` e da AWS Encryption SDK instância configurada.

Este exemplo define a política de compromisso como `forbid-encrypt-allow-decrypt`.

```
// Configure the commitment policy on the AWS Encryption SDK instance
let esdk_config = AwsEncryptionSdkConfig::builder()
                    .commitment_policy(ForbidEncryptAllowDecrypt)
                    .build()?;
let esdk_client = esdk_client::Client::from_conf(esdk_config)?;

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

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

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

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

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

let encryption_response = esdk_client.encrypt()
```

```
    .plaintext(plaintext)
    .keyring(kms_keyring.clone())
    .encryption_context(encryption_context.clone())
    .send()
    .await?;

// Decrypt your ciphertext
let decryption_response = esdk_client.decrypt()
    .ciphertext(ciphertext)
    .keyring(kms_keyring)
    // Provide the encryption context that was supplied to the encrypt method
    .encryption_context(encryption_context)
    .send()
    .await?;
```

Go

```
import (
    "context"

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

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

// Create an AWS KMS client
```

```
cfg, err := config.LoadDefaultConfig(context.TODO())
if err != nil {
    panic(err)
}
kmsClient := kms.NewFromConfig(cfg, func(o *kms.Options) {
    o.Region = KmsKeyRegion
})

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

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

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

// Encrypt your plaintext data
res, err := forbidEncryptClient.Encrypt(context.Background(),
 esdktypes.EncryptInput{
    Plaintext:         []byte(exampleText),
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}
```

```
// Decrypt your ciphertext
decryptOutput, err := forbidEncryptClient.Decrypt(context.Background(),
 esdktypes.DecryptInput{
    Ciphertext:        res.Ciphertext,
    EncryptionContext: encryptionContext,
    Keyring:           awsKmsKeyring,
})
if err != nil {
    panic(err)
}
```

Solução de problemas de migração para as versões mais recentes

Antes de atualizar seu aplicativo para a versão 2.0. x ou posterior do AWS Encryption SDK, atualize para a última 1. versão x do AWS Encryption SDK e implante-a completamente. Isso ajudará você a evitar a maioria dos erros que pode encontrar ao atualizar para as versões 2.0.x e posteriores. Para obter orientações detalhadas, incluindo exemplos, consulte Migrando seu AWS Encryption SDK.

Important

Verifique se a sua versão 1.x mais recente é a versão 1.7.x ou versão posterior do AWS Encryption SDK.

Note

AWS CLI de criptografia: referências neste guia à versão 1.7. x do AWS Encryption SDK se aplica à versão 1.8. x da CLI AWS de criptografia. Referências neste guia para a versão 2.0. x do AWS Encryption SDK se aplica a 2.1. x da CLI AWS de criptografia.
Novos recursos de segurança foram lançados originalmente nas versões 1.7 do AWS Encryption CLI. x e 2.0. x. No entanto, a versão AWS 1.8 do Encryption CLI. x substitui a versão 1.7. x e CLI de AWS criptografia 2.1. x substitui 2.0. x. Para obter detalhes, consulte a consultoria de segurança relevante no aws-encryption-sdk-clirepositório em GitHub.

Este tópico foi criado para ajudar você a reconhecer e solucionar os erros mais comuns que pode encontrar.

Tópicos

- Objetos descontinuados ou removidos
- Conflito de configuração: política de compromisso e pacote de algoritmos
- Conflito de configuração: política de compromisso e texto cifrado
- Falha na validação do confirmação de chave
- Outras falhas de criptografia
- Outras falhas de decriptografia
- Considerações sobre reversão

Objetos descontinuados ou removidos

A versão 2.0.x inclui várias alterações importantes, incluindo a remoção de construtores, métodos, funções e classes herdados que foram descontinuados na versão 1.7.x.. Para evitar erros do compilador, erros de importação, erros de sintaxe e erros de símbolo não encontrado (dependendo da sua linguagem de programação), atualize primeiro para a mais recente 1. versão x do AWS Encryption SDK para sua linguagem de programação. (Deve ser a versão 1.7.x ou posterior) Ao usar a versão 1.x mais recente, você pode começar a usar os elementos de substituição antes que os símbolos originais sejam removidos.

Se precisar atualizar para a versão 2.0. x ou posterior imediatamente, consulte o changelog da sua linguagem de programação e substitua os símbolos herdados pelos símbolos recomendados pelo changelog.

Conflito de configuração: política de compromisso e pacote de algoritmos

Se você especificar um pacote de algoritmos que entre em conflito com sua política de compromisso, a chamada para criptografar falhará com um erro de conflito de configuração.

Para evitar esse tipo de erro, não especifique um pacote de algoritmos. Por padrão, o AWS Encryption SDK escolhe o algoritmo mais seguro que seja compatível com sua política de compromisso. No entanto, se você precisar especificar um pacote de algoritmos, como um sem assinatura, certifique-se de escolher um pacote de algoritmos que seja compatível com sua política de compromisso.

Política de compromisso	Pacotes de algoritmos compatíveis
ForbidEncryptAllowDecrypt	Qualquer pacote de algoritmos sem confirmação de chave, como: `AES_256_GCM_IV12_TAG16_HKDF_SHA384_ECDSA_P384` (03 78) (com assinatura) `AES_256_GCM_IV12_TAG16_HKDF_SHA256` (01 78) (sem assinatura)
RequireEncryptAllowDecrypt RequireEncryptRequireDecrypt	Qualquer pacote de algoritmos com confirmação de chave, como: `AES_256_GCM_HKDF_SHA512_COMMIT_KEY_ECDSA_P384` (05 78) (com assinatura) `AES_256_GCM_HKDF_SHA512_COMMIT_KEY` (04 78) (sem assinatura)

Se você encontrar esse erro sem ter especificado um pacote de algoritmos, o pacote de algoritmos conflitante pode ter sido escolhido pelo seu gerenciador de materiais criptográficos (CMM). O CMM padrão não selecionará um pacote de algoritmos conflitantes, mas um CMM personalizado pode fazer isso. Para obter ajuda, consulte a documentação do seu CMM personalizado.

Conflito de configuração: política de compromisso e texto cifrado

A política de compromisso `RequireEncryptRequireDecrypt` não permite que o AWS Encryption SDK descriptografe uma mensagem que foi criptografada sem confirmação de chave. Se você pedir AWS Encryption SDK para decifrar uma mensagem sem o compromisso da chave, ele retornará um erro de conflito de configuração.

Para evitar esse erro, antes de definir a política de compromisso `RequireEncryptRequireDecrypt`, certifique-se de que todos os textos cifrados criptografados sem confirmação de chave sejam descriptografados e recriptografados com confirmação de chave ou sejam manipulados por uma aplicação diferente. Se você encontrar esse erro, poderá retornar

um erro para o texto cifrado conflitante ou alterar temporariamente sua política de compromisso para `RequireEncryptAllowDecrypt`.

Se estiver encontrando esse erro porque você atualizou para a versão 2.0.x ou posterior a partir uma versão anterior à 1.7.x sem ter atualizado primeiro para a versão 1x mais recente (versão 1.7. x ou posterior), considere reverter para a versão 1.x mais recente e implantar essa versão em todos os hosts antes de atualizar para a versão 2.0.x ou posterior. Para obter ajuda, consulte Como migrar e implantar o AWS Encryption SDK.

Falha na validação do confirmação de chave

Ao descriptografar mensagens criptografadas com confirmação de chave, você pode receber uma mensagem de erro de Falha na validação da confirmação de chave. Isso indica que a chamada de descriptografia falhou porque a chave de dados em uma mensagem criptografada não é idêntica à chave de dados exclusiva da mensagem. Ao validar a chave de dados durante a descriptografia, a confirmação de chave protege você de descriptografar uma mensagem que pode resultar em mais de um texto simples.

Esse erro indica que a mensagem criptografada que você estava tentando descriptografar não foi retornada pelo AWS Encryption SDK. Pode ser uma mensagem criada manualmente ou o resultado de dados corrompidos. Se você encontrar esse erro, a aplicação poderá rejeitar a mensagem e prosseguir ou interromper o processamento de novas mensagens.

Outras falhas de criptografia

A criptografia pode falhar por vários motivos. Você não pode usar um token de autenticação de descoberta do AWS KMS ou um provedor de chave mestra em modo de descoberta para criptografar uma mensagem.

Certifique-se de especificar um provedor de token de autenticação ou de chave mestra com chaves de encapsulamento que deem permissão para usar para criptografia. Para obter ajuda com as permissões AWS KMS keys, consulte Visualizar uma política de chaves e Determinar o acesso a uma AWS KMS key no Guia do AWS Key Management Service desenvolvedor.

Outras falhas de decriptografia

Se sua tentativa de descriptografar uma mensagem criptografada falhar, isso significa que o AWS Encryption SDK não conseguiu (ou não pôde) descriptografar alguma das chaves de dados criptografadas na mensagem.

Se você usou um provedor de chaveiro ou chave mestra que especifica chaves de encapsulamento, ele AWS Encryption SDK usa somente as chaves de encapsulamento que você especificar. Verifique se está usando as chaves de encapsulamento pretendidas e se tem permissão `kms:Decrypt` em pelo menos uma das chaves de encapsulamento. Se você estiver usando AWS KMS keys, como alternativa, tente descriptografar a mensagem com um chaveiro de AWS KMS descoberta ou um provedor de chave mestra no modo de descoberta. Se a operação for bem-sucedida, antes de retornar o texto simples, verifique se a chave usada para descriptografar a mensagem é confiável.

Considerações sobre reversão

Se a aplicação não conseguir criptografar ou descriptografar dados, geralmente, é possível resolver o problema atualizando os símbolos de código, os tokens de autenticação, os provedores de chaves mestras ou a política de compromisso. No entanto, em alguns casos, você pode decidir que é melhor reverter a aplicação para uma versão anterior do AWS Encryption SDK.

Se você precisar reverter, faça isso com cuidado. Versões AWS Encryption SDK anteriores à 1.7. x não pode decifrar texto cifrado criptografado com comprometimento de chave.

- Geralmente, é seguro reverter da versão 1.x mais recente para uma versão anterior do AWS Encryption SDK . Talvez seja necessário desfazer as alterações feitas no código para usar símbolos e objetos que não são compatíveis com as versões anteriores.
- Depois de começar a criptografar com confirmação de chave (definindo sua política de compromisso como `RequireEncryptAllowDecrypt`) na versão 2.0.x ou posterior, você poderá reverter para a versão 1.7.x, mas não para alguma versão anterior a ela. Versões AWS Encryption SDK anteriores à 1.7. x não pode decifrar texto cifrado criptografado com comprometimento de chave.

Se você acidentalmente ativar a criptografia com confirmação de chave antes de que todos os hosts possam descriptografar com confirmação de chave, talvez seja melhor continuar com a implantação em vez de revertê-la. Se as mensagens forem transitórias ou puderem ser descartadas com segurança, considere fazer uma reversão com perda de mensagens. Se for necessária uma reversão, considere criar uma ferramenta que descriptografe e recriptografe todas as mensagens.

Perguntas frequentes

Perguntas frequentes

- Como o é AWS Encryption SDK diferente do AWS SDKs?
- Como ele é AWS Encryption SDK diferente do cliente de criptografia Amazon S3?
- Quais algoritmos criptográficos são suportados pelo AWS Encryption SDK e qual é o padrão?
- Como o vetor de inicialização (IV) é gerado e onde é armazenado?
- Como cada chave de dados é gerada, criptografada e descriptografada?
- Como faço para controlar as chaves de dados que foram usadas para criptografar meus dados?
- Como eles AWS Encryption SDK armazenam chaves de dados criptografadas com seus dados criptografados?
- Quanta sobrecarga o formato da AWS Encryption SDK mensagem adiciona aos meus dados criptografados?
- Posso usar meu próprio provedor de chaves mestras?
- Posso criptografar dados com mais de uma chave de encapsulamento?
- Com quais tipos de dados posso criptografar? AWS Encryption SDK
- Como os fluxos AWS Encryption SDK criptografam e descriptografam input/output (E/S)?

Como o é AWS Encryption SDK diferente do AWS SDKs?

Eles AWS SDKsfornecem bibliotecas para interagir com a Amazon Web Services (AWS), incluindo AWS Key Management Service (AWS KMS). Algumas das implementações de linguagem do AWS Encryption SDK, como a AWS Encryption SDK para o.NET, sempre exigem o AWS SDK na mesma linguagem de programação. Outras implementações de linguagem exigem o AWS SDK correspondente somente quando você usa AWS KMS chaves em seus chaveiros ou provedores de chaves mestras. Para obter detalhes, consulte o tópico sobre sua linguagem de programação em AWS Encryption SDK linguagens de programação.

Você pode usar o AWS SDKs para interagir AWS KMS, incluindo criptografar e descriptografar pequenas quantidades de dados (até 4.096 bytes com uma chave de criptografia simétrica) e gerar chaves de dados para criptografia do lado do cliente. No entanto, ao gerar uma chave de dados, você deve gerenciar todo o processo de criptografia e descriptografia, incluindo criptografar seus dados com a chave de dados externa, descartar com segurança a chave de dados em texto simples

AWS KMS, armazenar a chave de dados criptografada e, em seguida, descriptografar a chave de dados e descriptografar seus dados. O AWS Encryption SDK gerencia esse processo para você.

O AWS Encryption SDK fornece uma biblioteca que criptografa e descriptografa dados usando os padrões e as melhores práticas do setor. Ele gera a chave de dados, criptografa-a com as chaves de encapsulamento especificadas e retorna uma mensagem criptografada, um objeto de dados portátil que inclui os dados criptografados e as chaves de dados criptografadas necessárias para descriptografá-los. Na hora de descriptografar, você passa a mensagem criptografada e pelo menos uma das chaves de encapsulamento (opcional) e AWS Encryption SDK retorna seus dados em texto sem formatação.

Você pode usar AWS KMS keys como chaves de empacotamento no AWS Encryption SDK, mas isso não é obrigatório. Você pode usar as chaves de criptografia geradas por você e as do seu gerenciador de chaves ou módulo de segurança de hardware on-premises. Você pode usar o AWS Encryption SDK mesmo se não tiver uma AWS conta.

Como ele é AWS Encryption SDK diferente do cliente de criptografia Amazon S3?

O cliente de criptografia Amazon S3 no AWS SDKs fornece criptografia e descriptografia para dados que você armazena no Amazon Simple Storage Service (Amazon S3). Esses clientes são totalmente acoplados ao Amazon S3 e são destinados para uso apenas com os dados armazenados ali.

AWS Encryption SDK Ele fornece criptografia e decodificação para dados que você pode armazenar em qualquer lugar. O AWS Encryption SDK e o cliente de criptografia Amazon S3 não são compatíveis porque produzem textos cifrados com formatos de dados diferentes.

Quais algoritmos criptográficos são suportados pelo AWS Encryption SDK e qual é o padrão?

O AWS Encryption SDK usa o algoritmo simétrico Advanced Encryption Standard (AES) no Galois/Counter Modo (GCM), conhecido como AES-GCM, para criptografar seus dados. Ele permite que você escolha entre vários algoritmos simétricos e assimétricos para criptografar as chaves de dados que criptografam seus dados.

Para o AES-GCM, o conjunto de algoritmos padrão é o AES-GCM com uma chave de 256 bits, derivação de chave (HKDF), assinaturas digitais e compromisso de chave. AWS Encryption SDK

também oferece suporte a chaves de criptografia de 192 e 128 bits e algoritmos de criptografia sem assinaturas digitais e comprometimento de chaves.

Em todos os casos, o tamanho do vetor de inicialização (IV) é de 12 bytes, e o tamanho da tag de autenticação é de 16 bytes. Por padrão, o SDK usa a chave de dados como entrada para a função de derivação de chave baseada em HMAC (HKDF) para derivar a extract-and-expand chave de criptografia AES-GCM e também adiciona uma assinatura do Algoritmo de Assinatura Digital de Curva Elíptica (ECDSA).

Para obter informações sobre como escolher o algoritmo a ser usado, consulte Pacotes de algoritmos compatíveis.

Para obter detalhes sobre a implementação de algoritmos compatíveis, consulte Referência de algoritmos.

Como o vetor de inicialização (IV) é gerado e onde é armazenado?

O AWS Encryption SDK usa um método determinístico para construir um valor IV diferente para cada quadro. Esse procedimento garante que nunca IVs sejam repetidos em uma mensagem. (Antes da versão 1.3.0 do AWS Encryption SDK for Java e do AWS Encryption SDK for Python, o AWS Encryption SDK gerava aleatoriamente um valor IV exclusivo para cada quadro.)

O IV é armazenado na mensagem criptografada que ele AWS Encryption SDK retorna. Para obter mais informações, consulte o AWS Encryption SDK referência de formato de mensagem.

Como cada chave de dados é gerada, criptografada e descriptografada?

O método depende do token de autenticação ou do provedor de chave mestra que você usa.

Os AWS KMS chaveiros e os provedores de chaves mestras AWS Encryption SDK usam a operação da AWS KMS GenerateDataKeyAPI para gerar cada chave de dados e criptografá-la sob sua chave de encapsulamento. Para criptografar cópias da chave de dados em chaves KMS adicionais, eles usam a operação AWS KMS Criptografar. Para descriptografar as chaves de dados, eles usam a operação Decrypt. AWS KMS Para obter detalhes, consulte AWS KMS chaveiro na AWS Encryption SDK Especificação em GitHub.

Outros tokens de autenticação geram a chave de dados, criptografam e descriptografam usando os métodos das práticas recomendadas para cada linguagem de programação. Para obter detalhes,

consulte a especificação do fornecedor do chaveiro ou da chave mestra na seção Estrutura da AWS Encryption SDK Especificação em GitHub.

Como faço para controlar as chaves de dados que foram usadas para criptografar meus dados?

Ele AWS Encryption SDK faz isso por você. Ao criptografar dados, o SDK criptografa a chave de dados e armazena a chave criptografada junto com os dados criptografados na mensagem criptografada que retorna. Ao descriptografar dados, o AWS Encryption SDK extrai a chave de dados criptografada da mensagem criptografada, descriptografa-a usa-a para descriptografar os dados.

Como eles AWS Encryption SDK armazenam chaves de dados criptografadas com seus dados criptografados?

As operações de criptografia AWS Encryption SDK retornam uma mensagem criptografada, uma estrutura de dados única que contém os dados criptografados e suas chaves de dados criptografadas. O formato da mensagem consiste em pelo menos duas partes: um cabeçalho e um corpo. O cabeçalho da mensagem contém as chaves de dados criptografadas e informações sobre como o corpo da mensagem é formado. O corpo da mensagem contém os dados criptografados. Se o pacote de algoritmos incluir uma assinatura digital, o formato da mensagem incluirá um rodapé que contém a assinatura. Para obter mais informações, consulte AWS Encryption SDK referência de formato de mensagem.

Quanta sobrecarga o formato da AWS Encryption SDK mensagem adiciona aos meus dados criptografados?

A quantidade de sobrecarga adicionada pelo AWS Encryption SDK depende de vários fatores, incluindo os seguintes:

- O tamanho dos dados de texto descriptografado
- Qual dos algoritmos compatíveis é usado
- Se dados autenticados adicionais (AAD) são fornecidos e o tamanho desse AAD
- O número e o tipo de chave de encapsulamento ou chave mestra
- O tamanho da moldura (quando dados com moldura são usados)

Quando você usa o AWS Encryption SDK com sua configuração padrão (uma AWS KMS key como chave de empacotamento (ou chave mestra), sem AAD, dados não emoldurados e um algoritmo de criptografia com assinatura), a sobrecarga é de aproximadamente 600 bytes. Em geral, você pode pressupor de forma razoável que o AWS Encryption SDK adiciona uma sobrecarga de 1 KB ou menos, sem incluir o AAD fornecido. Para obter mais informações, consulte AWS Encryption SDK referência de formato de mensagem.

Posso usar meu próprio provedor de chaves mestras?

Sim. Os detalhes da implementação variam dependendo de qual das linguagens de programação suportadas você usa. No entanto, todas as linguagens suportadas permitem que você defina gerenciadores de materiais criptográficos personalizados (CMMs) Ms), fornecedores de chaves mestras, chaveiros, chaves mestras e chaves de empacotamento.

Posso criptografar dados com mais de uma chave de encapsulamento?

Sim. Você pode criptografar a chave de dados com chaves de encapsulamento (ou chaves mestras) adicionais para adicionar redundância, no caso de uma estar em uma região diferente ou não estar disponível para a descriptografia.

Para criptografar dados com várias chaves de encapsulamento, crie um provedor de tokens de chave ou de chaves mestras com várias chaves de encapsulamento. Ao trabalhar com tokens de autenticação, você pode criar um único token de autenticação com várias chaves de empacotamento ou um multitoken de autenticação.

Quando você criptografa dados com várias chaves de encapsulamento, o AWS Encryption SDK usa uma chave de encapsulamento para gerar uma chave de dados em texto simples. A chave de dados é exclusiva e matematicamente não está relacionada à chave de encapsulamento. A operação retorna a chave de dados em texto simples e uma cópia da chave de dados criptografada pela chave de encapsulamento. Em seguida, o método de criptografia criptografa a chave de dados com as outras chaves de encapsulamento. A mensagem criptografada resultante inclui os dados criptografados e uma chave de dados criptografada para cada chave de encapsulamento.

A mensagem criptografada pode ser descriptografada usando qualquer uma das chaves de encapsulamento usadas na operação de criptografia. O AWS Encryption SDK usa uma chave de empacotamento para descriptografar uma chave de dados criptografada. Em seguida, usa a chave de dados de texto simples para descriptografar os dados.

Com quais tipos de dados posso criptografar? AWS Encryption SDK

A maioria das implementações de linguagem de programação do AWS Encryption SDK pode criptografar bytes brutos (matrizes de bytes), I/O fluxos (fluxos de bytes) e cadeias de caracteres. O AWS Encryption SDK for.NET não oferece suporte a I/O streams. Fornecemos um código de exemplo para cada uma das linguagens de programação compatíveis.

Como os fluxos AWS Encryption SDK criptografam e descriptografam input/output (E/S)?

O AWS Encryption SDK cria um fluxo de criptografia ou descriptografia que envolve um fluxo subjacente. I/O O fluxo de criptografia ou descriptografia executa uma operação de criptografia em uma chamada de leitura ou de gravação. Por exemplo, ele pode ler dados de texto não criptografado no fluxo subjacente e criptografá-los antes de retornar o resultado. Ou pode ler texto cifrado de um fluxo subjacente e descriptografá-lo antes de retornar o resultado. Fornecemos um código de exemplo para criptografar e descriptografar fluxos para cada uma das linguagens de programação compatíveis que oferecem suporte a streaming.

O AWS Encryption SDK for.NET não oferece suporte a I/O streams.

AWS Encryption SDK referência

As informações nesta página são uma referência para criar sua própria biblioteca de criptografia compatível com o AWS Encryption SDK. Se você não estiver criando sua própria biblioteca de criptografia compatível, provavelmente não precisará dessas informações.

Para usar o AWS Encryption SDK em uma das linguagens de programação suportadas, consulteLinguagens de programação.

Para obter a especificação que define os elementos de uma AWS Encryption SDK implementação adequada, consulte a AWS Encryption SDK Especificação em GitHub.

O AWS Encryption SDK usa os algoritmos suportados para retornar uma única estrutura de dados ou mensagem que contém dados criptografados e as chaves de dados criptografadas correspondentes. Os tópicos a seguir explicam os algoritmos e a estrutura de dados. Use essas informações para criar bibliotecas que podem ler e gravar textos cifrados compatíveis com este SDK.

Tópicos

- AWS Encryption SDK referência de formato de mensagem
- AWS Encryption SDK exemplos de formato de mensagem
- Referência de dados autenticados adicionais (AAD) do corpo para o AWS Encryption SDK
- AWS Encryption SDK referência de algoritmos
- AWS Encryption SDK referência vetorial de inicialização
- AWS KMS Detalhes técnicos do chaveiro hierárquico

AWS Encryption SDK referência de formato de mensagem

As informações nesta página são uma referência para criar sua própria biblioteca de criptografia compatível com o AWS Encryption SDK. Se você não estiver criando sua própria biblioteca de criptografia compatível, provavelmente não precisará dessas informações.

Para usar o AWS Encryption SDK em uma das linguagens de programação suportadas, consulteLinguagens de programação.

Para obter a especificação que define os elementos de uma AWS Encryption SDK implementação adequada, consulte a AWS Encryption SDK Especificação em GitHub.

As operações de criptografia no AWS Encryption SDK retornam uma única estrutura de dados ou mensagem criptografada que contém os dados criptografados (texto cifrado) e todas as chaves de dados criptografadas. Para compreender essa estrutura de dados ou para criar bibliotecas que a leem ou gravam nela, você precisa compreender o formato da mensagem.

O formato da mensagem consiste em pelo menos duas partes: um cabeçalho e um corpo. Em alguns casos, o formato da mensagem consiste em uma terceira parte, um rodapé. O formato da mensagem define uma sequência ordenada de bytes em ordem de bytes de rede, também chamado de formato big-endian. O formato da mensagem começa com o cabeçalho, seguido pelo corpo, seguido pelo rodapé (se houver).

Os pacotes de algoritmos suportados pelo AWS Encryption SDK usam uma das duas versões de formato de mensagem. Os pacotes de algoritmos sem confirmação de chave usam formato de mensagem versão 1. Os pacotes de algoritmos com confirmação de chave usam formato de mensagem versão 2.

Tópicos

- Estrutura do cabeçalho
- Estrutura do corpo
- Estrutura do rodapé

Estrutura do cabeçalho

O cabeçalho da mensagem contém a chave de dados criptografada e informações sobre como o corpo da mensagem é formado. A tabela a seguir descreve os campos que formam o cabeçalho nas versões 1 e 2 do formato de mensagem. Os bytes são anexados na ordem mostrada.

O valor Não presente indica que o campo não existe nessa versão do formato de mensagem. O texto em negrito indica valores que são diferentes em cada versão.

> **Note**
>
> Talvez seja necessário rolar horizontalmente ou verticalmente para ver todos os dados nessa tabela.

Estrutura do cabeçalho

Campo	Formato da mensagem versão 1 Tamanho (bytes)	Formato da mensagem versão 2 Tamanho (bytes)
Version	1	1
Type	1	Não está presente
Algorithm ID	2	2
Message ID	16	32
AAD Length	2 Quando o contexto de criptografia está vazio, o valor do tamanho do AAD de 2 bytes é 0.	2 Quando o contexto de criptografia está vazio, o valor do tamanho do AAD de 2 bytes é 0.
AAD	Variável. O tamanho desse campo aparece nos 2 bytes anteriores (campo Tamanho do AAD). Quando o contexto de criptografia está vazio, não há um campo AAD no cabeçalho.	Variável. O tamanho desse campo aparece nos 2 bytes anteriores (campo Tamanho do AAD). Quando o contexto de criptografia está vazio, não há um campo AAD no cabeçalho.
Encrypted Data Key Count	2	2
Encrypted Data Key(s)	Variável. Determinado pelo número de chaves de dados criptografadas e pelo tamanho de cada uma delas.	Variável. Determinado pelo número de chaves de dados criptografadas e pelo tamanho de cada uma delas.
Content Type	1	1
Reserved	4	Não está presente

Campo	Formato da mensagem versão 1 Tamanho (bytes)	Formato da mensagem versão 2 Tamanho (bytes)
IV Length	1	Não está presente
Frame Length	4	4
Algorithm Suite Data	Não está presente	Variável. Determinado pelo algoritmo que gerou a mensagem.
Header Authentication	Variável. Determinado pelo algoritmo que gerou a mensagem.	Variável. Determinado pelo algoritmo que gerou a mensagem.

Versão

A versão do formato desta mensagem. A versão é 1 ou 2 codificado como o byte `01` ou `02` em notação hexadecimal

Tipo

O tipo deste formato de mensagem. O tipo indica o tipo da estrutura. O único tipo suportado é descrito como dados criptografados e autenticados pelo cliente. Seu valor de tipo é 128 bytes, codificado como byte `80` em notação hexadecimal.

Esse campo não está presente na versão 2 do formato de mensagem.

ID do algoritmo

Um identificador para o algoritmo usado. É um valor de 2 bytes interpretado como um inteiro não assinado de 16 bits. Para obter mais informações sobre os algoritmos, consulte AWS Encryption SDK referência de algoritmos.

ID da mensagem

Um valor gerado aleatoriamente que identifica a mensagem. O ID da mensagem:

- Identifica exclusivamente a mensagem criptografada.
- Associa levemente o cabeçalho da mensagem ao corpo da mensagem.

- Fornece um mecanismo para reutilizar uma chave de dados com segurança com várias mensagens criptografadas.
- Protege contra a reutilização acidental de uma chave de dados ou contra o desgaste de chaves no AWS Encryption SDK.

Esse valor é de 128 bits no formato de mensagem versão 1 e 256 bits na versão 2.

Comprimento do AAD

O tamanho dos dados autenticados adicionais (AAD). 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 AAD.

Quando o contexto de criptografia está vazio, o valor do campo de tamanho do AAD é 0.

AAD

Os dados autenticados adicionais. O AAD é uma codificação do contexto de criptografia, uma matriz de pares de valores-chave em que cada chave e valor são uma sequência de caracteres codificados. UTF-8 O contexto de criptografia é convertido em uma sequência de bytes e usado para o valor do AAD. Quando o contexto de criptografia está vazio, não há um campo AAD no cabeçalho.

Quando os algoritmos com assinatura são usados, o contexto de criptografia deve conter o par de chave-valor `{'aws-crypto-public-key', Qtxt}`. Qtxt representa o ponto Q compactado da curva elíptica de acordo com o SEC 1 versão 2.0 e, em seguida, codificado em base64. O contexto de criptografia pode conter valores adicionais, mas o tamanho máximo do construído AAD é 2^16 - 1 bytes.

A tabela a seguir descreve os campos que formam o AAD. Key-value os pares são classificados, por chave, em ordem crescente de acordo com o código do UTF-8 caractere. Os bytes são anexados na ordem mostrada.

Estrutura do AAD

Campo	Tamanho (bytes)
Key-Value Pair Count	2
Key Length	2
Key	Variável. Igual ao valor especificado nos 2 bytes anteriores (tamanho da chave).

Campo	Tamanho (bytes)
Value Length	2
Value	Variável. Igual ao valor especificado nos 2 bytes anteriores (tamanho do valor).

Key-Value Contagem de pares

O número de pares chave-valor no AAD. É um valor de 2 bytes interpretado como um número inteiro não assinado de 16 bits que especifica o número de pares de chave-valor no AAD. O número máximo de pares chave-valor no AAD é 2^16 - 1.

Quando não houver um contexto de criptografia ou o contexto de criptografia estiver vazio, esse campo não estará presente na estrutura do AAD.

Comprimento da chave

O tamanho da chave do par de chave-valor. 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.

Chave

A chave do par de chave-valor. É uma sequência de bytes UTF-8 codificados.

Comprimento do valor

O tamanho do valor do par de chave-valor. 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 valor.

Valor

O valor do par de chave-valor. É uma sequência de bytes UTF-8 codificados.

Contagem de chaves de dados criptografados

O número de chaves de dados criptografadas. É um valor de 2 bytes interpretado como um número inteiro não assinado de 16 bits que especifica o número de chaves de dados criptografadas. O número máximo de chaves de dados criptografadas em cada registro é 65.535 (2^16 - 1).

Chave (s) de dados criptografada (s)

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 (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).
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 chaves

O identificador do provedor de chave. É usado para indicar o provedor da chave de dados criptografada e deve ser extensível.

Comprimento das informações do provedor de chaves

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 o fornecedor principal

As informações do provedor de chave. São determinadas pelo provedor de chaves.

Quando AWS KMS é o provedor da chave mestra ou você está usando um AWS KMS chaveiro, esse valor contém o nome de recurso da Amazon (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.

Tipo de conteúdo

O tipo de conteúdo criptografado, com moldura ou sem moldura.

> **Note**
>
> Sempre que possível, use dados com moldura. O AWS Encryption SDK suporta dados não emoldurados somente para uso legado. Algumas implementações de linguagem do ainda AWS Encryption SDK podem gerar texto cifrado sem moldura. Todas as implementações de linguagem compatíveis podem descriptografar texto cifrado e não emoldurado.

Os dados com moldura são divididos em partes de tamanho igual; cada parte é criptografada separadamente. O conteúdo com moldura é do tipo 2, codificado como o byte `02` em notação hexadecimal.

Os dados não enquadrados não são divididos; são um único blob criptografado. Non-framed o conteúdo é do tipo 1, codificado como byte `01` em notação hexadecimal.

Reservado

Uma sequência reservada de 4 bytes. Esse valor deve ser 0. Ele é codificado como os bytes `00 00 00 00` em notação hexadecimal (ou seja, uma sequência de 4 bytes de um valor inteiro de 32 bits igual a 0).

Esse campo não está presente na versão 2 do formato de mensagem.

Comprimento IV

O tamanho do IV (initialization vector - vetor de inicialização). É um valor de 1 byte interpretado como um número inteiro não assinado de 8 bits que especifica o número de bytes que contém o IV. Esse valor é determinado pelo valor de bytes do IV do algoritmo que gerou a mensagem.

Este campo não está presente no formato de mensagem versão 2, que somente é compatível com pacotes de algoritmos que usam valores IV determinísticos no cabeçalho da mensagem.

Comprimento do quadro

O tamanho de cada moldura do dado com moldura. É um valor de 4 bytes interpretado como um número inteiro não assinado de 32 bits que especifica o número de bytes que forma cada estrutura. Quando o dado não for com moldura, isto é, quando o valor do campo do `Content Type` campo for 1, esse valor deve ser 0.

> **Note**
>
> Sempre que possível, use dados com moldura. O AWS Encryption SDK suporta dados não emoldurados somente para uso legado. Algumas implementações de linguagem do ainda AWS Encryption SDK podem gerar texto cifrado sem moldura. Todas as implementações de linguagem compatíveis podem descriptografar texto cifrado e não emoldurado.

Dados do conjunto de algoritmos

Dados suplementares necessários pelo algoritmo que gerou a mensagem. O tamanho e o conteúdo são determinados pelo algoritmo. Seu tamanho pode ser 0.

Este campo não está presente na versão 1 do formato de mensagem.

autenticação de cabeçalho

A autenticação do cabeçalho é determinada pelo algoritmo que gerou a mensagem. A autenticação do cabeçalho é calculada sobre o cabeçalho inteiro. Consiste em um IV e uma tag de autenticação. Os bytes são anexados na ordem mostrada.

Estrutura da autenticação do cabeçalho

Campo	Tamanho na versão 1.0 (bytes)	Tamanho na versão 2.0 (bytes)
IV	Variável. Determinada pelo valor de bytes do IV do algoritmo que gerou a mensagem.	N/A
Authentication Tag	Variável. Determinada pelo valor dos bytes da tag de autenticação do algoritmo que gerou a mensagem.	Variável. Determinada pelo valor dos bytes da tag de autenticação do algoritmo que gerou a mensagem.

IV

O vetor de inicialização (IV) usado para calcular a tag de autenticação do cabeçalho.

Este campo não está presente na versão 2 do formato de mensagem. Este campo não está presente na versão 2 do formato de mensagem, que somente é compatível com pacotes de algoritmos que usam valores IV determinísticos no cabeçalho da mensagem.

Etiqueta de autenticação

O valor da autenticação do cabeçalho. É usado para autenticar todo o conteúdo do cabeçalho.

Estrutura do corpo

O corpo da mensagem contém os dados criptografados, chamados de texto cifrado. A estrutura do corpo depende do tipo de conteúdo (sem moldura ou com moldura). As seções a seguir descrevem o formato do corpo da mensagem para cada tipo de conteúdo. A estrutura do corpo da mensagem é a mesma nas versões 1 e 2 do formato de mensagem.

Tópicos

- Non-framed dados
- Dados com moldura

Non-framed dados

Non-framed os dados são criptografados em um único blob com um IV e um AAD corporal exclusivos.

> **Note**
>
> Sempre que possível, use dados com moldura. O AWS Encryption SDK suporta dados não emoldurados somente para uso legado. Algumas implementações de linguagem do ainda AWS Encryption SDK podem gerar texto cifrado sem moldura. Todas as implementações de linguagem compatíveis podem descriptografar texto cifrado e não emoldurado.

A tabela a seguir descreve os campos que formam os dados sem moldura. Os bytes são anexados na ordem mostrada.

Non-Framed Estrutura corporal

Campo	Tamanho, em bytes
IV	Variável. Igual ao valor especificado no byte do IV Length do cabeçalho.
Encrypted Content Length	8
Encrypted Content	Variável. Igual ao valor especificado nos 8 bytes anteriores (tamanho do conteúdo criptografado).
Authentication Tag	Variável. Determinado pela implementação do algoritmo usado.

IV

O vetor de inicialização (IV) para uso com o algoritmo de criptografia.

Tamanho do conteúdo criptografado

O tamanho do conteúdo criptografado ou do texto cifrado. É um valor de 8 bytes interpretado como um número inteiro não assinado de 64 bits que especifica o número de bytes que contém o conteúdo criptografado.

Tecnicamente, o valor máximo permitido é 2^63 - 1 ou 8 exbibytes (8 EiB). No entanto, na prática, o valor máximo é 2^36 - 32 ou 64 gibibytes (64 GiB), devido às restrições impostas pelos algoritmos implementados.

> **Note**
>
> A implementação Java deste SDK restringe ainda mais esse valor para 2^31 - 1 ou 2 gibibytes (2 GiB), devido às restrições da linguagem.

Conteúdo criptografado

O conteúdo criptografado (texto cifrado) como retornado pelo algoritmo de criptografia.

Etiqueta de autenticação

O valor da autenticação do corpo. É usado para autenticar o corpo da mensagem.

Dados com moldura

Em dados com moldura, os dados de texto simples são divididos em partes iguais chamadas molduras. O AWS Encryption SDK criptografa cada quadro separadamente com um IV e um AAD corporal exclusivos.

> **Note**
>
> Sempre que possível, use dados com moldura. O AWS Encryption SDK suporta dados não emoldurados somente para uso legado. Algumas implementações de linguagem do ainda AWS Encryption SDK podem gerar texto cifrado sem moldura. Todas as implementações de linguagem compatíveis podem descriptografar texto cifrado e não emoldurado.

O tamanho da moldura, que é o tamanho do conteúdo criptografado na moldura, pode ser diferente para cada mensagem. O número máximo de bytes em uma moldura é 2^32 - 1. O número máximo de molduras em uma mensagem é 2^32 - 1.

Há dois tipos de moldura: normal e final. Cada mensagem deve consistir ou incluir uma moldura final.

Todas as molduras normais em uma mensagem têm o mesmo tamanho de moldura. A moldura final pode ter um tamanho de moldura diferente.

A composição das molduras em dados com molduras varia de acordo com o tamanho do conteúdo criptografado.

- Igual ao tamanho do quadro: quando o tamanho do conteúdo criptografado é igual ao tamanho do quadro das molduras regulares, a mensagem pode consistir em um quadro normal que contém os dados, seguido por um quadro final de tamanho zero (0). Ou, a mensagem pode consistir apenas em um a moldura que contém os dados. Nesse caso, a moldura final tem o mesmo tamanho de moldura que as molduras normais.
- Múltiplo do tamanho do quadro: quando o tamanho do conteúdo criptografado é um múltiplo exato do tamanho do quadro das molduras regulares, a mensagem pode terminar em um quadro regular que contém os dados, seguido por um quadro final de tamanho zero (0). Ou, a mensagem pode terminar em uma moldura final que contém os dados. Nesse caso, a moldura final tem o mesmo tamanho de moldura que as molduras normais.
- Não é múltiplo do tamanho do quadro: quando o tamanho do conteúdo criptografado não é um múltiplo exato do comprimento do quadro das molduras regulares, o quadro final contém os dados restantes. O tamanho da moldura final é menor que o tamanho das molduras normais.
- Menor que o tamanho do quadro: quando o tamanho do conteúdo criptografado é menor que o tamanho do quadro das molduras regulares, a mensagem consiste em um quadro final que contém todos os dados. O tamanho da moldura final é menor que o tamanho das molduras normais.

As tabelas a seguir descrevem os campos que formam as molduras. Os bytes são anexados na ordem mostrada.

Estrutura de corpo com moldura, moldura normal

Campo	Tamanho, em bytes
Sequence Number	4

Campo	Tamanho, em bytes
IV	Variável. Igual ao valor especificado no byte do IV Length do cabeçalho.
Encrypted Content	Variável. Igual ao valor especificado no Frame Length do cabeçalho.
Authentication Tag	Variável. Determinada pelo algoritmo usado, conforme especificado no Algorithm ID do cabeçalho.

Número de sequência

O número sequencial da moldura. É um número do contador incremental da moldura. É um valor de 4 bytes interpretado como um inteiro não assinado de 32 bits.

Os dados com moldura devem começar no número sequencial 1. As molduras subsequentes devem estar em ordem e devem conter um incremento de 1 da moldura anterior. Caso contrário, o processo de descriptografia será interrompido e relatará um erro.

IV

O vetor de inicialização (IV) da moldura. O SDK usa um método determinístico para construir um IV diferente para cada moldura na mensagem. O tamanho é especificado pelo pacote de algoritmos usado.

Conteúdo criptografado

O conteúdo criptografado (texto cifrado) da moldura, conforme retornado pelo algoritmo de criptografia.

Etiqueta de autenticação

O valor da autenticação da moldura. É usado para autenticar a moldura inteira.

Estrutura de corpo com moldura, moldura final

Campo	Tamanho, em bytes
Sequence Number End	4

Campo	Tamanho, em bytes
Sequence Number	4
IV	Variável. Igual ao valor especificado no byte do IV Length do cabeçalho.
Encrypted Content Length	4
Encrypted Content	Variável. Igual ao valor especificado nos 4 bytes anteriores (tamanho do conteúdo criptografado).
Authentication Tag	Variável. Determinada pelo algoritmo usado, conforme especificado no Algorithm ID do cabeçalho.

Fim do número de sequência

Um indicador para a moldura final. O valor é codificado como 4 bytes `FF FF FF FF` em notação hexadecimal.

Número de sequência

O número sequencial da moldura. É um número do contador incremental da moldura. É um valor de 4 bytes interpretado como um inteiro não assinado de 32 bits.

Os dados com moldura devem começar no número sequencial 1. As molduras subsequentes devem estar em ordem e devem conter um incremento de 1 da moldura anterior. Caso contrário, o processo de descriptografia será interrompido e relatará um erro.

IV

O vetor de inicialização (IV) da moldura. O SDK usa um método determinístico para construir um IV diferente para cada moldura na mensagem. O tamanho do IV é especificado pelo pacote de algoritmos.

Tamanho do conteúdo criptografado

O tamanho do conteúdo criptografado. É um valor de 4 bytes interpretado como um número inteiro não assinado de 32 bits que especifica o número de bytes que contém o conteúdo criptografado da moldura.

Conteúdo criptografado

O conteúdo criptografado (texto cifrado) da moldura, conforme retornado pelo algoritmo de criptografia.

Etiqueta de autenticação

O valor da autenticação da moldura. É usado para autenticar a moldura inteira.

Estrutura do rodapé

Quando os algoritmos com assinatura são usados, o formato da mensagem contém um rodapé. O rodapé da mensagem contém uma assinatura digital calculada sobre o cabeçalho e o corpo da mensagem. A tabela a seguir descreve os campos que formam o rodapé. Os bytes são anexados na ordem mostrada. A estrutura do rodapé da mensagem é a mesma nas versões 1 e 2 do formato de mensagem.

Estrutura do rodapé

Campo	Tamanho, em bytes
Signature Length	2
Signature	Variável. Igual ao valor especificado nos 2 bytes anteriores (tamanho da assinatura).

Comprimento da assinatura

O tamanho da assinatura. 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 assinatura.

Assinatura

A assinatura.

AWS Encryption SDK exemplos de formato de mensagem

As informações nesta página são uma referência para criar sua própria biblioteca de criptografia compatível com o AWS Encryption SDK. Se você não estiver criando sua própria biblioteca de criptografia compatível, provavelmente não precisará dessas informações.

Para usar o AWS Encryption SDK em uma das linguagens de programação suportadas, consulteLinguagens de programação.

Para obter a especificação que define os elementos de uma AWS Encryption SDK implementação adequada, consulte a AWS Encryption SDK Especificação em GitHub.

Os tópicos a seguir mostram exemplos do formato da AWS Encryption SDK mensagem. Cada exemplo mostra os bytes brutos, em notação hexadecimal, seguidos por uma descrição do que os bytes representam.

Tópicos

- Dados emoldurados (formato de mensagem versão 1)
- Dados emoldurados (formato de mensagem versão 2)
- Non-framed dados (formato de mensagem versão 1)

Dados emoldurados (formato de mensagem versão 1)

O exemplo a seguir mostra o formato da mensagem para dados com moldura na vesão 1 do formato de mansagem.

```
+--------+
| Header |
+--------+
01                                      Version (1.0)
80                                      Type (128, customer authenticated encrypted
 data)
0378                                    Algorithm ID (see Referência de algoritmos)
6E7C0FBD 4DF4A999 717C22A2 DDFE1A27     Message ID (random 128-bit value)
008E                                    AAD Length (142)
0004                                    AAD Key-Value Pair Count (4)
0005                                    AAD Key-Value Pair 1, Key Length (5)
30746869 73                             AAD Key-Value Pair 1, Key ("0This")
0002                                    AAD Key-Value Pair 1, Value Length (2)
6973                                    AAD Key-Value Pair 1, Value ("is")
0003                                    AAD Key-Value Pair 2, Key Length (3)
31616E                                  AAD Key-Value Pair 2, Key ("1an")
000A                                    AAD Key-Value Pair 2, Value Length (10)
656E6372 79774690 6F6E                  AAD Key-Value Pair 2, Value ("encryption")
0008                                    AAD Key-Value Pair 3, Key Length (8)
```

```
32636F6E 74657874                       AAD Key-Value Pair 3, Key ("2context")
0007                                    AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                         AAD Key-Value Pair 3, Value ("example")
0015                                    AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69     AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632D6B65 79
0044                                    AAD Key-Value Pair 4, Value Length (68)
416A4173 7569326F 7430364C 4B77715A     AAD Key-Value Pair 4, Value
 ("AjAsui2ot06LKwqZXDJnU/Aqc2vD+0OkpOZ1cc8Tg2qd7rs5aLTg7lvfUEW/86+/5w==")
58444A6E 552F4171 63327644 2B304F6B
704F5A31 63633854 67327164 37727335
614C5467 376C7666 5545572F 38362B2F
35773D3D
0002                                    EncryptedDataKeyCount (2)
0007                                    Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                    Encrypted Data Key 1, Key Provider
 Information Length (75)
61726E3A 6177733A 6B6D733A 75732D77     Encrypted Data Key 1, Key Provider
 Information ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-
a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                    Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010200 7857A1C1 F7370545 4ECA7C83     Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C3F F02C897B
7A12EB19 8BF2D802 0110803B 24003D1F
A5474FBC 392360B5 CB9997E0 6A17DE4C
A6BD7332 6BF86DAB 60D8CCB8 8295DBE9
4707E356 ADA3735A 7C52D778 B3135A47
9F224BF9 E67E87
0007                                    Encrypted Data Key 2, Key Provider ID Length
 (7)
```

```
6177732D 6B6D73                          Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                     Encrypted Data Key 2, Key Provider
 Information Length (78)
61726E3A 6177733A 6B6D733A 63612D63       Encrypted Data Key 2, Key Provider
 Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-
be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                     Encrypted Data Key 2, Encrypted Data Key
 Length (167)
01010200 78FAFFFB D6DE06AF AC72F79B       Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C36 CD985E12
D218B674 5BBC6102 0110803B 0320E3CD
E470AA27 DEAB660B 3E0CE8E0 8B1A89E4
57DCC69B AAB1294F 21202C01 9A50D323
72EBAAFD E24E3ED8 7168E0FA DB40508F
556FBD58 9E621C
02                                       Content Type (2, framed data)
00000000                                 Reserved
0C                                       IV Length (12)
00000100                                 Frame Length (256)
4ECBD5C0 9899CA65 923D2347               IV
0B896144 0CA27950 CA571201 4DA58029       Authentication Tag
+------+
| Body |
+------+
00000001                                 Frame 1, Sequence Number (1)
6BD3FE9C ADBCB213 5B89E8F1               Frame 1, IV
1F6471E0 A51AF310 10FA9EF6 F0C76EDF       Frame 1, Encrypted Content
F5AFA33C 7D2E8C6C 9C5D5175 A212AF8E
FBD9A0C3 C6E3FB59 C125DBF2 89AC7939
BDEE43A8 0F00F49E ACBBD8B2 1C785089
A90DB923 699A1495 C3B31B50 0A48A830
201E3AD9 1EA6DA14 7F6496DB 6BC104A4
DEB7F372 375ECB28 9BF84B6D 2863889F
CB80A167 9C361C4B 5EC07438 7A4822B4
A7D9D2CC 5150D414 AF75F509 FCE118BD
```

```
6D1E798B AEBA4CDB AD009E5F 1A571B77
0041BC78 3E5F2F41 8AF157FD 461E959A
BB732F27 D83DC36D CC9EBC05 00D87803
57F2BB80 066971C2 DEEA062F 4F36255D
E866C042 E1382369 12E9926B BA40E2FC
A820055F FB47E428 41876F14 3B6261D9
5262DB34 59F5D37E 76E46522 E8213640
04EE3CC5 379732B5 F56751FA 8E5F26AD        Frame 1, Authentication Tag
00000002                                   Frame 2, Sequence Number (2)
F1140984 FF25F943 959BE514                 Frame 2, IV
216C7C6A 2234F395 F0D2D9B9 304670BF        Frame 2, Encrypted Content
A1042608 8A8BCB3F B58CF384 D72EC004
A41455B4 9A78BAC9 36E54E68 2709B7BD
A884C1E1 705FF696 E540D297 446A8285
23DFEE28 E74B225A 732F2C0C 27C6BDA2
7597C901 65EF3502 546575D4 6D5EBF22
1FF787AB 2E38FD77 125D129C 43D44B96
778D7CEE 3C36625F FF3A985C 76F7D320
ED70B1F3 79729B47 E7D9B5FC 02FCE9F5
C8760D55 7779520A 81D54F9B EC45219D
95941F7E 5CBAEAC8 CEC13B62 1464757D
AC65B6EF 08262D74 44670624 A3657F7F
2A57F1FD E7060503 AC37E197 2F297A84
DF1172C2 FA63CF54 E6E2B9B6 A86F582B
3B16F868 1BBC5E4D 0B6919B3 08D5ABCF
FECDC4A4 8577F08B 99D766A1 E5545670
A61F0A3B A3E45A84 4D151493 63ECA38F        Frame 2, Authentication Tag
FFFFFFFF                                   Final Frame, Sequence Number End
00000003                                   Final Frame, Sequence Number (3)
35F74F11 25410F01 DD9E04BF                 Final Frame, IV
0000008E                                   Final Frame, Encrypted Content Length (142)
F7A53D37 2F467237 6FBD0B57 D1DFE830        Final Frame, Encrypted Content
B965AD1F A910AA5F 5EFFFFF4 BC7D431C
BA9FA7C4 B25AF82E 64A04E3A A0915526
88859500 7096FABB 3ACAD32A 75CFED0C
4A4E52A3 8E41484D 270B7A0F ED61810C
3A043180 DF25E5C5 3676E449 0986557F
C051AD55 A437F6BC 139E9E55 6199FD60
6ADC017D BA41CDA4 C9F17A83 3823F9EC
B66B6A5A 80FDB433 8A48D6A4 21CB
811234FD 8D589683 51F6F39A 040B3E3B        Final Frame, Authentication Tag
+--------+
| Footer |
+--------+
```

```
0066                                    Signature Length (102)
30640230 085C1D3C 63424E15 B2244448     Signature
639AED00 F7624854 F8CF2203 D7198A28
758B309F 5EFD9D5D 2E07AD0B 467B8317
5208B133 02301DF7 2DFC877A 66838028
3C6A7D5E 4F8B894E 83D98E7C E350F424
7E06808D 0FE79002 E24422B9 98A0D130
A13762FF 844D
```

Dados emoldurados (formato de mensagem versão 2)

O exemplo a seguir mostra o formato da mensagem para dados com moldura na versão 2 do formato de mensagem.

```
+--------+
| Header |
+--------+
02                                      Version (2.0)
0578                                    Algorithm ID (see Algorithms reference)
122747eb 21dfe39b 38631c61 7fad7340
cc621a30 32a11cc3 216d0204 fd148459     Message ID (random 256-bit value)
008e                                    AAD Length (142)
0004                                    AAD Key-Value Pair Count (4)
0005                                    AAD Key-Value Pair 1, Key Length (5)
30546869 73                             AAD Key-Value Pair 1, Key ("0This")
0002                                    AAD Key-Value Pair 1, Value Length (2)
6973                                    AAD Key-Value Pair 1, Value ("is")
0003                                    AAD Key-Value Pair 2, Key Length (3)
31616e                                  AAD Key-Value Pair 2, Key ("1an")
000a                                    AAD Key-Value Pair 2, Value Length (10)
656e6372 79707469 6f6e                  AAD Key-Value Pair 2, Value ("encryption")
0008                                    AAD Key-Value Pair 3, Key Length (8)
32636f6e 74657874                       AAD Key-Value Pair 3, Key ("2context")
0007                                    AAD Key-Value Pair 3, Value Length (7)
6578616d 706c65                         AAD Key-Value Pair 3, Value ("example")
0015                                    AAD Key-Value Pair 4, Key Length (21)
6177732d 63727970 746f2d70 75626c69     AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632d6b65 79
0044                                    AAD Key-Value Pair 4, Value Length (68)
41746733 72703845 41345161 36706669     AAD Key-Value Pair 4, Value
 ("QXRnM3JwOEVBNFFhNnBmaTk3MUlTNTk3NHpOMnlZWE5vSmtwRHFPc0dIYkVaVDRqME5OMlFkRStmbTFVY01WdThnPT0=
39373149 53353937 347a4e32 7959584e
```

```
6f4a6b70 44714f73 47486245 5a54346a
304e4e32 5164452b 666d3155 634d5675
38673d3d
0001                                    Encrypted Data Key Count (1)
0007                                    Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732d 6b6d73                         Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004b                                    Encrypted Data Key 1, Key Provider
 Information Length (75)
61726e3a 6177733a 6b6d733a 75732d77      Encrypted Data Key 1, Key
 Provider Information ("arn:aws:kms:us-west-2:658956600833:key/b3537ef1-
d8dc-4780-9f5a-55776cbb2f7f")
6573742d 323a3635 38393536 36303038
33333a6b 65792f62 33353337 6566312d
64386463 2d343738 302d3966 35612d35
35373736 63626232 663766
00a7                                    Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010100 7840f38c 275e3109 7416c107      Encrypted Data Key 1, Encrypted Data Key
29515057 1964ada3 ef1c21e9 4c8ba0bd
bc9d0fb4 14000000 7e307c06 092a8648
86f70d01 0706a06f 306d0201 00306806
092a8648 86f70d01 0701301e 06096086
48016503 04012e30 11040c39 32d75294
06063803 f8460802 0110803b 2a46bc23
413196d2 903bf1d7 3ed98fc8 a94ac6ed
e00ee216 74ec1349 12777577 7fa052a5
ba62e9e4 f2ac8df6 bcb1758f 2ce0fb21
cc9ee5c9 7203bb
02                                      Content Type (2, framed data)
00001000                                Frame Length (4096)
05cd035b 29d5499d 4587570b 87502afe      Algorithm Suite Data (key commitment)
634f7b2c c3df2aa9 88a10105 4a2c7687
76cb339f 2536741f 59a1c202 4f2594ab      Authentication Tag
+------+
| Body |
+------+
ffffffff                                Final Frame, Sequence Number End
00000001                                Final Frame, Sequence Number (1)
00000000 00000000 00000001              Final Frame, IV
00000009                                Final Frame, Encrypted Content Length (9)
fa6e39c6 02927399 3e                    Final Frame, Encrypted Content
f683a564 405d68db eeb0656c d57c9eb0      Final Frame, Authentication Tag
```

```
+--------+
| Footer |
+--------+
0067                                    Signature Length (103)
30650230 2a1647ad 98867925 c1712e8f     Signature
ade70b3f 2a2bc3b8 50eb91ef 56cfdd18
967d91d8 42d92baf 357bba48 f636c7a0
869cade2 023100aa ae12d08f 8a0afe85
e5054803 110c9ed8 11b2e08a c4a052a9
074217ea 3b01b660 534ac921 bf091d12
3657e2b0 9368bd
```

Non-framed dados (formato de mensagem versão 1)

O exemplo a seguir mostra o formato da mensagem para dados sem moldura.

> **Note**
>
> Sempre que possível, use dados com moldura. O AWS Encryption SDK suporta dados não emoldurados somente para uso legado. Algumas implementações de linguagem do ainda AWS Encryption SDK podem gerar texto cifrado sem moldura. Todas as implementações de linguagem compatíveis podem descriptografar texto cifrado e não emoldurado.

```
+--------+
| Header |
+--------+
01                                      Version (1.0)
80                                      Type (128, customer authenticated encrypted
 data)
0378                                    Algorithm ID (see Referência de algoritmos)
B8929B01 753D4A45 C0217F39 404F70FF     Message ID (random 128-bit value)
008E                                    AAD Length (142)
0004                                    AAD Key-Value Pair Count (4)
0005                                    AAD Key-Value Pair 1, Key Length (5)
30746869 73                             AAD Key-Value Pair 1, Key ("0This")
0002                                    AAD Key-Value Pair 1, Value Length (2)
6973                                    AAD Key-Value Pair 1, Value ("is")
0003                                    AAD Key-Value Pair 2, Key Length (3)
31616E                                  AAD Key-Value Pair 2, Key ("1an")
000A                                    AAD Key-Value Pair 2, Value Length (10)
```

```
656E6372 79774690 6F6E                    AAD Key-Value Pair 2, Value ("encryption")
0008                                      AAD Key-Value Pair 3, Key Length (8)
32636F6E 74657874                         AAD Key-Value Pair 3, Key ("2context")
0007                                      AAD Key-Value Pair 3, Value Length (7)
6578616D 706C65                           AAD Key-Value Pair 3, Value ("example")
0015                                      AAD Key-Value Pair 4, Key Length (21)
6177732D 63727970 746F2D70 75626C69       AAD Key-Value Pair 4, Key ("aws-crypto-
public-key")
632D6B65 79
0044                                      AAD Key-Value Pair 4, Value Length (68)
41734738 67473949 6E4C5075 3136594B       AAD Key-Value Pair 4, Value
 ("AsG8gG9InLPu16YKlqXTOD+nykG8YqHAhqecj8aXfD2e5B4gtVE73dZkyClA+rAMOQ==")
6C715854 4F442B6E 796B4738 59714841
68716563 6A386158 66443265 35423467
74564537 33645A6B 79436C41 2B72414D
4F513D3D
0002                                      Encrypted Data Key Count (2)
0007                                      Encrypted Data Key 1, Key Provider ID Length
 (7)
6177732D 6B6D73                           Encrypted Data Key 1, Key Provider ID ("aws-
kms")
004B                                      Encrypted Data Key 1, Key Provider
 Information Length (75)
61726E3A 6177733A 6B6D733A 75732D77       Encrypted Data Key 1, Key Provider
 Information ("arn:aws:kms:us-west-2:111122223333:key/715c0818-5825-4245-
a755-138a6d9a11e6")
6573742D 323A3131 31313232 32323333
33333A6B 65792F37 31356330 3831382D
35383235 2D343234 352D6137 35352D31
33386136 64396131 316536
00A7                                      Encrypted Data Key 1, Encrypted Data Key
 Length (167)
01010200 7857A1C1 F7370545 4ECA7C83       Encrypted Data Key 1, Encrypted Data Key
956C4702 23DCE8D7 16C59679 973E3CED
02A4EF29 7F000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040C28 4116449A
0F2A0383 659EF802 0110803B B23A8133
3A33605C 48840656 C38BCB1F 9CCE7369
E9A33EBE 33F46461 0591FECA 947262F3
418E1151 21311A75 E575ECC5 61A286E0
3E2DEBD5 CB005D
```

```
0007                                    Encrypted Data Key 2, Key Provider ID Length
 (7)
6177732D 6B6D73                         Encrypted Data Key 2, Key Provider ID ("aws-
kms")
004E                                    Encrypted Data Key 2, Key Provider
 Information Length (78)
61726E3A 6177733A 6B6D733A 63612D63      Encrypted Data Key 2, Key Provider
 Information ("arn:aws:kms:ca-central-1:111122223333:key/9b13ca4b-afcc-46a8-aa47-
be3435b423ff")
656E7472 616C2D31 3A313131 31323232
32333333 333A6B65 792F3962 31336361
34622D61 6663632D 34366138 2D616134
372D6265 33343335 62343233 6666
00A7                                    Encrypted Data Key 2, Encrypted Data Key
 Length (167)
01010200 78FAFFFB D6DE06AF AC72F79B      Encrypted Data Key 2, Encrypted Data Key
0E57BD87 3F60F4E6 FD196144 5A002C94
AF787150 69000000 7E307C06 092A8648
86F70D01 0706A06F 306D0201 00306806
092A8648 86F70D01 0701301E 06096086
48016503 04012E30 11040CB2 A820D0CC
76616EF2 A6B30D02 0110803B 8073D0F1
FDD01BD9 B0979082 099FDBFC F7B13548
3CC686D7 F3CF7C7A CCC52639 122A1495
71F18A46 80E2C43F A34C0E58 11D05114
2A363C2A E11397
01                                      Content Type (1, nonframed data)
00000000                                Reserved
0C                                      IV Length (12)
00000000                                Frame Length (0, nonframed data)
734C1BBE 032F7025 84CDA9D0              IV
2C82BB23 4CBF4AAB 8F5C6002 622E886C      Authentication Tag
+------+
| Body |
+------+
D39DD3E5 915E0201 77A4AB11              IV
00000000 0000028E                       Encrypted Content Length (654)
E8B6F955 B5F22FE4 FD890224 4E1D5155      Encrypted Content
5871BA4C 93F78436 1085E4F8 D61ECE28
59455BD8 D76479DF C28D2E0B BDB3D5D3
E4159DFE C8A944B6 685643FC EA24122B
6766ECD5 E3F54653 DF205D30 0081D2D8
55FCDA5B 9F5318BC F4265B06 2FE7C741
C7D75BCC 10F05EA5 0E2F2F40 47A60344
```

```
ECE10AA7 559AF633 9DE2C21B 12AC8087
95FE9C58 C65329D1 377C4CD7 EA103EC1
31E4F48A 9B1CC047 EE5A0719 704211E5
B48A2068 8060DF60 B492A737 21B0DB21
C9B21A10 371E6179 78FAFB0B BAAEC3F4
9D86E334 701E1442 EA5DA288 64485077
54C0C231 AD43571A B9071925 609A4E59
B8178484 7EB73A4F AAE46B26 F5B374B8
12B0000C 8429F504 936B2492 AAF47E94
A5BA804F 7F190927 5D2DF651 B59D4C2F
A15D0551 DAEBA4AF 2060D0D5 CB1DA4E6
5E2034DB 4D19E7CD EEA6CF7E 549C86AC
46B2C979 AB84EE12 202FD6DF E7E3C09F
C2394012 AF20A97E 369BCBDA 62459D3E
C6FFB914 FEFD4DE5 88F5AFE1 98488557
1BABBAE4 BE55325E 4FB7E602 C1C04BEE
F3CB6B86 71666C06 6BF74E1B 0F881F31
B731839B CF711F6A 84CA95F5 958D3B44
E3862DF6 338E02B5 C345CFF8 A31D54F3
6920AA76 0BF8E903 552C5A04 917CCD11
D4E5DF5C 491EE86B 20C33FE1 5D21F0AD
6932E67C C64B3A26 B8988B25 CFA33E2B
63490741 3AB79D60 D8AEFBE9 2F48E25A
978A019C FE49EE0A 0E96BF0D D6074DDB
66DFF333 0E10226F 0A1B219C BE54E4C2
2C15100C 6A2AA3F1 88251874 FDC94F6B
9247EF61 3E7B7E0D 29F3AD89 FA14A29C
76E08E9B 9ADCDF8C C886D4FD A69F6CB4
E24FDE26 3044C856 BF08F051 1ADAD329
C4A46A1E B5AB72FE 096041F1 F3F3571B
2EAFD9CB B9EB8B83 AE05885A 8F2D2793
1E3305D9 0C9E2294 E8AD7E3B 8E4DEC96
6276C5F1 A3B7E51E 422D365D E4C0259C
50715406 822D1682 80B0F2E5 5C94
65B2E942 24BEEA6E A513F918 CCEC1DE3      Authentication Tag
+--------+
| Footer |
+--------+
0067                                     Signature Length (103)
30650230 7229DDF5 B86A5B64 54E4D627      Signature
CBE194F1 1CC0F8CF D27B7F8B F50658C0
BE84B355 3CED1721 A0BE2A1B 8E3F449E
1BEB8281 023100B2 0CB323EF 58A4ACE3
1559963B 889F72C3 B15D1700 5FB26E61
```

```
331F3614 BC407CEE B86A66FA CBF74D9E
34CB7E4B 363A38
```

Referência de dados autenticados adicionais (AAD) do corpo para o AWS Encryption SDK

As informações nesta página são uma referência para criar sua própria biblioteca de criptografia compatível com o AWS Encryption SDK. Se você não estiver criando sua própria biblioteca de criptografia compatível, provavelmente não precisará dessas informações.

Para usar o AWS Encryption SDK em uma das linguagens de programação suportadas, consulteLinguagens de programação.

Para obter a especificação que define os elementos de uma AWS Encryption SDK implementação adequada, consulte a AWS Encryption SDK Especificação em GitHub.

Você deve fornecer dados autenticados (AAD) adicionais ao AES-GCM algoritmo para cada operação criptográfica. Isso é verdadeiro para dados de corpo com e sem moldura. Para obter mais informações sobre o AAD e como ele é usado no Galois/Counter Modo (GCM), consulte Recomendações para modos de operações com criptografia de bloco: Galois/Counter modo (GCM) e GMAC.

A tabela a seguir descreve os campos que formam o AAD do corpo. Os bytes são anexados na ordem mostrada.

Estrutura do AAD do corpo

Campo	Tamanho, em bytes
Message ID	16
Body AAD Content	Variável. Consulte Conteúdo do AAD do copo na lista a seguir.
Sequence Number	4
Content Length	8

ID da mensagem

O mesmo valor de Message ID definido no cabeçalho da mensagem.

Conteúdo corporal de AAD

Um valor UTF-8 codificado determinado pelo tipo de dados corporais usados.

Para dados sem moldura, use o valor `AWSKMSEncryptionClient Single Block`.

Para molduras normais em dados com moldura, use o valor `AWSKMSEncryptionClient Frame`.

Para a moldura final nos dados com moldura, use o valor `AWSKMSEncryptionClient Final Frame`.

Número de sequência

Um valor de 4 bytes interpretado como um inteiro não assinado de 32 bits.

Para dados com moldura, esse é o número sequencial da moldura.

Para dados sem moldura, use o valor 1, codificado como os 4 bytes `00 00 00 01` em notação hexadecimal.

Comprimento do conteúdo

O tamanho, em bytes, do texto não criptografado fornecido ao algoritmo para criptografia. É um valor de 8 bytes interpretado como um inteiro não assinado de 64 bits.

AWS Encryption SDK referência de algoritmos

As informações nesta página são uma referência para criar sua própria biblioteca de criptografia compatível com o AWS Encryption SDK. Se você não estiver criando sua própria biblioteca de criptografia compatível, provavelmente não precisará dessas informações.

Para usar o AWS Encryption SDK em uma das linguagens de programação suportadas, consulteLinguagens de programação.

Para obter a especificação que define os elementos de uma AWS Encryption SDK implementação adequada, consulte a AWS Encryption SDK Especificação em GitHub.

Se você estiver criando sua própria biblioteca capaz de ler e escrever textos cifrados compatíveis com o. AWS Encryption SDK, precisará entender como ela AWS Encryption SDK implementa os conjuntos de algoritmos compatíveis para criptografar dados brutos.

O AWS Encryption SDK suporta os seguintes conjuntos de algoritmos. Todos os conjuntos de AES-GCM algoritmos têm um vetor de inicialização de 12 bytes e uma tag de autenticação de 16 AES-GCM bytes. O conjunto de algoritmos padrão varia de acordo com a AWS Encryption SDK versão e a política de comprometimento de chaves selecionada. Para obter detalhes, consulte Política de compromisso e pacote de algoritmos.

AWS Encryption SDK Suítes de algoritmos

ID do algoritmo	Versão do formato de mensagem	Algoritmo de criptografia	Tamanho da chave de dados (bits)	Algoritmo de derivação de chave	Algoritmo de assinatura	Algoritmo de confirmação de chave	Tamanho dos dados do pacote de algoritmos (bytes)
05 78	0x02	AES-GCM	256	HKDF com SHA-512	ECDSA com e P-384 SHA-384	HKDF com SHA-512	32 (confirmação de chave)
04 78	0x02	AES-GCM	256	HKDF com SHA-512	Nenhum	HKDF com SHA-512	32 (confirmação de chave)
03 78	0x01	AES-GCM	256	HKDF com SHA-384	ECDSA com e P-384 SHA-384	Nenhum	N/A
03 46	0x01	AES-GCM	192	HKDF com SHA-384	ECDSA com e P-384 SHA-384	Nenhum	N/A

ID do algoritmo	Versão do formato de mensagem	Algoritmo de criptografia	Tamanho da chave de dados (bits)	Algoritmo de derivação de chave	Algoritmo de assinatura	Algoritmo de confirmação de chave	Tamanho dos dados do pacote de algoritmos (bytes)
02 14	0x01	AES-GCM	128	HKDF com SHA-256	ECDSA com e P-256 SHA-256	Nenhum	N/A
01 78	0x01	AES-GCM	256	HKDF com SHA-256	Nenhum	Nenhum	N/A
01 46	0x01	AES-GCM	192	HKDF com SHA-256	Nenhum	Nenhum	N/A
01 14	0x01	AES-GCM	128	HKDF com SHA-256	Nenhum	Nenhum	N/A
00 78	0x01	AES-GCM	256	Nenhum	Nenhum	Nenhum	N/A
00 46	0x01	AES-GCM	192	Nenhum	Nenhum	Nenhum	N/A
00 14	0x01	AES-GCM	128	Nenhum	Nenhum	Nenhum	N/A

ID do algoritmo

Um valor de 2 bytes hexadecimal que identifica exclusivamente a implementação de um algoritmo. Esse valor é armazenado no cabeçalho da mensagem do texto cifrado.

Versão do formato de mensagem

A versão do formato desta mensagem. Os pacotes de algoritmos com confirmação de chave usam formato de mensagem versão 2 (0x02). Os pacotes de algoritmos sem confirmação de chave usam formato de mensagem versão 1 (0x01).

Tamanho dos dados do pacote de algoritmos

O tamanho em bytes dos dados específicos do pacote de algoritmos. Esse campo é suportado somente no formato de mensagem versão 2 (0x02). No formato de mensagem versão 2 (0x02), esses dados aparecem no campo `Algorithm suite data` do cabeçalho da mensagem. Os conjuntos de algoritmos que compatíveis com o confirmação de chave usam 32 bytes para a cadeia de caracteres de confirmação de chave. Para obter mais informações, consulte Algoritmo de confirmação de chaves nesta lista.

Tamanho da chave de dados

O tamanho da chave de dados em bits. O AWS Encryption SDK é compatível com chaves de 256, 192 e 128 bits. A chave de dados é gerada por um token de autenticação ou chave mestra.

Em algumas implementações, essa chave de dados é usada como entrada para uma função de derivação de chave HMAC-based de extração e expansão (HKDF). A saída da HKDF é usada como a chave de criptografia de dados no algoritmo de criptografia. Para obter mais informações, consulte Algoritmo de derivação de chaves nessa lista.

Algoritmo de criptografia

O nome e o modo do algoritmo de criptografia utilizado. Os conjuntos de algoritmos AWS Encryption SDK usam o algoritmo de criptografia Advanced Encryption Standard (AES) com Galois/Counter Modo (GCM).

Algoritmo de confirmação de chave

O algoritmo usado para calcular a string de confirmação de chave. A saída é armazenada no campo `Algorithm suite data` do cabeçalho da mensagem e é usada para validar a chave de dados para o confirmação de chave.

Para obter uma explicação técnica sobre como adicionar confirmação de chave a um conjunto de algoritmos, consulte Confirmação de chaves AEADs no Cryptology ePrintArchive.

Algoritmo de derivação de chave

A função de derivação de chave HMAC-based de extração e expansão (HKDF) usada para derivar a chave de criptografia de dados. O AWS Encryption SDK usa o HKDF definido na RFC 5869.

Pacotes de algoritmos sem confirmação de chave (ID do algoritmo `01xx` – `03xx`)

- A função hash usada é SHA-384 ou SHA-256, dependendo do conjunto de algoritmos.
- Para a etapa de extração:
  - Nenhum sal é usado. De acordo com a RFC, o sal é definido como uma string de zeros. O comprimento da string é igual ao tamanho da saída da função hash, que é 48 bytes para SHA-384 e 32 bytes para SHA-256.
  - O material de chaveamento de entrada é a chave de dados recebida do provedor de tokens de autenticação ou de chaves mestras.
- Para a etapa de expansão:
  - A chave pseudoaleatória de entrada é a saída da etapa de extração.
  - As informações da entrada são uma concatenação do ID do algoritmo seguido pelo ID da mensagem (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.

Pacotes de algoritmos com confirmação de chave (ID do algoritmo `04xx` e `05xx`)

- A função hash usada é SHA-512.
- Para a etapa de extração:
  - O sal é um valor aleatório criptográfico de 256 bits. No formato de mensagem versão 2 (0x02), esse valor é armazenado no campo `MessageID`.
  - O material de chaveamento inicial é a chave de dados recebida do provedor de tokens de autenticação ou de chaves mestras.
- Para a etapa de expansão:
  - A chave pseudoaleatória de entrada é a saída da etapa de extração.
  - O rótulo da chave são os UTF-8-encoded bytes da `DERIVEKEY` string em grande ordem de bytes 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.

Versão do formato de mensagem

A versão do formato de mensagem usado com o conjunto de algoritmos. Para obter detalhes, consulte Referência do formato de mensagens.

Algoritmo de assinatura

O algoritmo de assinatura usado para gerar umaassinatura digital sobre o cabeçalho e o corpo do texto cifrado. O AWS Encryption SDK usa o Algoritmo de Assinatura Digital de Curva Elíptica (ECDSA) com as seguintes especificações:

- A curva elíptica usada é a P-256 curva P-384 ou, conforme especificado pelo ID do algoritmo. Essas curvas são definidas no Digital Signature Standard (DSS) (FIPS PUB 186-4).
- A função hash usada é SHA-384 (com a P-384 curva) ou SHA-256 (com a P-256 curva).

AWS Encryption SDK referência vetorial de inicialização

As informações nesta página são uma referência para criar sua própria biblioteca de criptografia compatível com o AWS Encryption SDK. Se você não estiver criando sua própria biblioteca de criptografia compatível, provavelmente não precisará dessas informações.

Para usar o AWS Encryption SDK em uma das linguagens de programação suportadas, consulteLinguagens de programação.

Para obter a especificação que define os elementos de uma AWS Encryption SDK implementação adequada, consulte a AWS Encryption SDK Especificação em GitHub.

O AWS Encryption SDK fornece os vetores de inicialização (IVs) que são exigidos por todos os conjuntos de algoritmos suportados. O SDK usa números sequenciais de molduras para construir um IV, de forma que duas molduras na mesma mensagem não podem ter o mesmo IV.

Cada IV de 96 bits (12 bytes) é construído a partir de duas matrizes de bytes big-endian concatenadas na seguinte ordem:

- 64 bits: 0 (reservado para uso futuro)

- 32 bits: o número sequencial da moldura. Para a tag de autenticação de cabeçalho, esse valor é todo de zeros.

Antes da apresentação do armazenamento em cache de chaves de dados, o AWS Encryption SDK sempre usava uma nova chave de dados para criptografar cada mensagem e gerava todos os IVs aleatoriamente. Os IVs gerados aleatoriamente eram seguros criptograficamente porque as chaves de dados nunca eram reutilizadas. Quando o SDK apresentou o armazenamento em cache da chave de dados, que reutiliza intencionalmente as chaves de dados, alteramos a maneira como o SDK gera IVs.

O uso de IVs determinísticos que não podem ser repetidos em uma mensagem aumenta significativamente o número de chamadas que podem ser executadas com segurança sob uma única chave de dados. Além disso, as chaves de dados que são armazenadas em cache sempre usam um pacote de algoritmos com uma função de derivação de chaves. O uso de um IV determinístico com uma função de derivação de chave pseudo-aleatória para derivar chaves de criptografia de uma chave de dados permite criptografar 2^32 mensagens sem AWS Encryption SDK exceder os limites criptográficos.

AWS KMS Detalhes técnicos do chaveiro hierárquico

O chaveiro AWS KMS hierárquico usa uma chave de dados exclusiva para criptografar cada mensagem 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 pseudo-aleatória com HMAC SHA-256 para derivar a chave de encapsulamento de 32 bytes com as seguintes entradas.

- Um sal aleatório de 16 bytes
- A chave de ramificação ativa
- O valor UTF-8 codificado para o identificador do provedor de chaves “aws-kms-hierarchy”

O chaveiro hierárquico usa a chave de empacotamento derivada para criptografar uma cópia da chave de dados de texto sem formatação usando uma tag de autenticação de 16 bytes e AES-GCM-256 as seguintes entradas.

- A chave de empacotamento derivada é usada como chave AES-GCM cifrada
- A chave de dados é usada como AES-GCM mensagem

- Um vetor de inicialização aleatória de 12 bytes (IV) é usado como o IV AES-GCM
- Dados autenticados adicionais (AAD) contendo os seguintes valores serializados.

Valor	Tamanho em bytes	Interpretada como
"aws-kms-hierarchy"	17	UTF-8 codificado
O identificador de chave de ramificação	Variável	UTF-8 codificado
A versão da chave de ramificação	16	UTF-8 codificado
Contexto de criptografia	Variável	UTF-8 pares de valores-chave codificados

Histórico de documentos do AWS Encryption SDK Developer Guide

Este tópico descreve atualizações importantes no Guia do desenvolvedor do AWS Encryption SDK .

Tópicos

- Atualizações recentes
- Atualizações anteriores

Atualizações recentes

A tabela a seguir descreve alterações significativas nesta documentação desde novembro de 2017. Além das principais alterações listadas aqui, 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
AWS Encryption SDK para o.NET versão 5.x	Atualizações para a versão 2.0 da Material Providers Library (MPL). Elimina o suporte ao AWS SDK para.NET v3 e adiciona suporte ao SDK para.NET v4. AWS	25 de março de 2025
Disponibilidade geral	Foi adicionada documentação para o AWS KMS chaveiro ECDH e o chaveiro ECDH bruto.	17 de junho de 2024
AWS Encryption SDK for Java versão 3.x	Integra-o AWS Encryption SDK for Java com a bibliotec a do fornecedor de materiais . Adiciona suporte para	6 de dezembro de 2023

	chaveiros e o contexto de criptografia necessário (CMM).	
AWS Encryption SDK para o.NET versão 4.x	Adiciona suporte ao AWS KMS chaveiro hierárquico, ao contexto de criptografia necessário (CMM) e aos chaveiros RSA assimétricos. AWS KMS	12 de outubro de 2023
Disponibilidade geral	Apresentando o suporte AWS Encryption SDK para o.NET.	17 de maio de 2022
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
Disponibilidade geral	Foi adicionado suporte para AWS Key Management Service. (AWS KMS) Chaves multirregionais. 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
Disponibilidade geral	Adicionada e atualizada a documentação sobre o processo de decodificação de mensagens aprimorado.	11 de maio de 2021

Disponibilidade geral	Documentação adicionada e atualizada para a versão de disponibilidade geral do AWS Encryption CLI versão 1.8. x para substituir a versão 1.7 do AWS Encryption CLI. x e CLI AWS de criptografia 2.1. x para substituir o AWS Encryption CLI 2.0. x.	27 de outubro de 2020
Disponibilidade geral	Adicionada e atualizada a documentação da versão de disponibilidade geral das versões 1.7.x e 2.0.x do AWS Encryption SDK , incluindo um guia de melhores práticas, um guia de migração, conceitos atualizados, tópicos de linguagem de programação atualizados, uma atualizaç ão de referência de pacotes de algoritmos, uma atualizaç ão de referência de formato de mensagem e um novo exemplo de formato de mensagem.	24 de setembro de 2020
Disponibilidade geral	Adicionada e atualizada a documentação para a versão de disponibilidade geral do AWS Encryption SDK para JavaScript.	1 de outubro de 2019
Versão de visualização	Adicionada e atualizada a documentação da versão beta pública do AWS Encryption SDK para JavaScript.	21 de junho de 2019

Disponibilidade geral	Adicionada e atualizada a documentação para a versão de disponibilidade geral do AWS Encryption SDK for C.	16 de maio de 2019
Versão de visualização	Documentação adicionada da versão de pré-visualização do AWS Encryption SDK for C.	5 de fevereiro de 2019
Nova versão	Documentação adicionada da interface de linha de comando para o AWS Encryption SDK.	20 de novembro de 2017

Atualizações anteriores

A tabela a seguir descreve as alterações significativas feitas no Guia do desenvolvedor do AWS Encryption SDK antes de novembro de 2017.

Alteração	Descrição	Data
Nova versão	Adicionado o capítulo Armazenamento em cache de chaves de dados para o novo recurso. Foi adicionado o the section called “Referência do vetor de inicialização” tópico que explica que o SDK mudou da geração aleatória IVs para a construção determinística. IVs Adicionado o tópico the section called “Conceitos” para explicar conceitos, incluindo o novo gerenciador de materiais criptográficos.	31 de julho de 2017

Alteração	Descrição	Data
Atualizar	Expandida a documentação de Referência do formato de mensagens para uma nova seção AWS Encryption SDK referência. Foi adicionada uma seção sobre AWS Encryption SDK Pacotes de algoritmos compatíveis o.	21 de março de 2017
Nova versão	O AWS Encryption SDK agora suporta a linguagem de Python programação, além deJava.	21 de março de 2017
Versão inicial	Versão inicial do AWS Encryption SDK e desta documentação.	22 de março de 2016

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.