AWS KMS (Key Management Service)

If you are studying for AWS Developer Associate Exam, this guide will help you with quick revision before the exam. it can use as study notes for your preparation.

Dashboard Other Certification Notes

AWS KMS (Key Management Service)

- Easy way to control who can access our data
- AWS will manage all they keys for us
- Fully integrated with IAM for authorization
- Seamlessly integrates into:
  - EBS: encrypted volumes
  - S3: server side encryption
  - Redshift: encryption of data
  - RDS: encryption of data
  - SSM: parameter store
  - etc.
- It can be used with the CLI/SDK

KMS - Customer Master Key (CMK) Types

Symmetric (AES-256 keys)

- First offering of KMS
- Single encryption key that is used to encrypt and decrypt data
- Most of the services integrated with KMS use symmetric keys
- Necessary for envelope encryption
- We never get access to the key unencrypted (must use the KMS API to decrypt data)

Asymmetric (RSA and ECC key pairs)

- Key pair: public key used for encryption. private key used for decryption
- Used for encryption/decryption or sing/verify operations
- Public key can be download, the access for private key is impossible
- Use case: encryption can be done outside of AWS using the public key, decryption can be done only in AWS with the private key

KMS - Key Management System

- We are able to fully manage the keys and policies:
  - Create
  - Rotate
  - Disable
  - Enable
- We are able to audit key usage (using CloudTrail)
- There are 3 type of CMK keys:
  - AWS Managed Service Default CMK: free
  - User Keys created in KMS: 1$/month
  - User Keys imported (must be 256-bit symmetric key): 1$/month
- We pay for API calls: 0.03$ / 10K calls
- We should use KMS anytime when:
  - We need to store DB passwords
  - We need to store credentials for external services
  - We need to store SSL certificates
- The value of KMS is that CMK used used to encrypt data can never be retrieved by the user. CMK can be rotated for extra security
- **We should never store secrets in plain text, especially in code!**
- Secrets should be encrypted and then they can be stored in code/env. variables
- KMS can only help in encrypting up to **4KB** of data per call
- In case of data larger than 4KB, we should use **envelope encryption**
- KMS keys are bound to a specific region

- KMS keys are bound to a specific region

KMS - Key Policies

- Control access to KMS keys, similar to how S3 Bucket policies control access to S3 Buckets
- Difference between S3 Bucket policies and KMS Key policies is that we can not control access without KMS Key policies
- Default KMS Key policy:
    - Is automatically created if we don't prove a specific policy
    - Gives complete access to the key to the root user = entire AWS account
    - We should also give access to IAM policies to the KMS key
- Custom KMS Key policies:
    - We define the user and the roles who can access the KMS key
    - We define who can administer the key
    - It is useful for cross-account access to the KMS key

Cross account copying of snapshots

1. When we create a snapshot, it will be encrypted with our CMK
2. We should attach a KMS key policy to authorize cross-account access
3. We share the encrypted snapshot
4. In the target account we create a copy of the snapshot and we encrypt it with a KMS key from this account
5. We create a volume from the snapshot

KMS CLI

- Encryption

```
aws kms encrypt --key-id alias/tutorial --plaintext fileb://location --output text --query Ciphe
```

- Decryption

```
aws kms decrypt --ciphertext-blob fileb://locaiton --ouput text --query Plaintext > decrypted.ba
```

Envelope Encryption

- KMS *Encrypt API* call has a limit of 4KB
- In order to encrypt more than 4KB data, we need to use **Envelope Encryption**
- Main API that will help encrypt data using envelope encryption is the **GenerateDataKey** API

Envelope encryption flow

1. We call *GenerateDataKey* API
2. KMS checks IAM permissions
3. KMS sends back the plaintext data encryption key (DEK) and the encrypted data encryption key
4. We encrypt the large data locally using the plaintext data key
5. We build an envelope around the encrypted data. We put in the envelop the encrypted data key and the encrypted data creating one final file

Envelope decryption flow

1. We get the envelope file and extract the encrypted data encryption key
2. We decrypt the data encryption key (DEK) by calling the **Decrypt** API from KMS
3. We decrypt the data with the plaintext data encryption key

Encryption SDK

- For envelope encryption we should leverage the SDK provided by AWS
- The Encryption SDK can slo be used from a CLI (separate tool we can install)
- There is SDK implementation for Java, Python, C, JavaScript

Data Key Caching

- We can re-use data encryption keys instead of creating new ones
- This helps with reducing the number of API call to KMS but it has a security trade-off
- If we are using data key caching, we should duse *LocalCryptoMaterialsCache* to indicate how bug this data cache should be. We can define max age, max bytes, max number of messages for our keys

KMS Symmetric API Summary

- **Encrypt**: can encrypt data up to 4KB using KMS

- **GenerateDataKey**: generates an unique symmetric data encryption key (DEK)
    - Returns both plaintext and encrypted versions for the data key
- **GenerateDataKeyWithoutPlainText**: generates a DEK for future use. Must be decrypted when used (additional step)
- **Decrypt**: can decrypt data up to 4KB using KMS
- **GenerateRandom**: returns a random byte string

KMS Request Quotas

- When we exceed a request quota, we get a *ThrottlingException*
- To mitigate this issue, we should use **exponential back-off**
- For each cryptographic operations a single quota is shared per account
    - Including requests made by AWS on our behalf (ex: SSE-KMS)
- To reduce throttling we can consider using DEK caching for envelope encryption
- We can do a request quota increase through API or AWS support (we pay more money)