**Lab Guide for Encryption and Decryption Using AWS KMS**

This lab guide will walk you through the steps required to perform encryption and decryption using AWS Key Management Service (KMS) on a Linux, Mac OS, or Windows machine. The guide covers the usage of AWS CLI to encrypt and decrypt files using AWS KMS and the handling of base64 encoding/decoding.

**Prerequisites**

* **AWS CLI installed** and configured with appropriate permissions to access KMS.
* **AWS account with an active KMS key**. You can create a key in the AWS KMS console if you don't have one.
* **Key alias** or **Key ID** available (we will use alias/tutorial in this guide as an example).
* A **text file** that contains the secret or data you want to encrypt.
* Access to **Linux, Mac OS, or Windows** machine.

**Section 1: Encryption Using AWS KMS**

**Step 1: Create a Plaintext File for Encryption**

First, create a file that contains the data you want to encrypt. This can be any text file. For example:

bash

Copy code

echo "This is a secret message" > ExampleSecretFile.txt

**Step 2: Encrypt the File with AWS KMS**

Use AWS KMS to encrypt the contents of the file using the following AWS CLI command. This command uses the encrypt action of KMS, the key alias alias/tutorial, and the plaintext file as input. The command will output a base64-encoded ciphertext that you can store securely.

bash

Copy code

aws kms encrypt --key-id alias/tutorial \

--plaintext fileb://ExampleSecretFile.txt \

--output text --query CiphertextBlob \

--region eu-west-2 > ExampleSecretFileEncrypted.base64

Here’s a breakdown of the command:

* --key-id alias/tutorial: Specifies the KMS key alias or ID to be used for encryption.
* --plaintext fileb://ExampleSecretFile.txt: Points to the plaintext file for encryption.
* --output text --query CiphertextBlob: Outputs the encrypted ciphertext blob in base64 format.
* --region eu-west-2: The region where your KMS key is located.
* ExampleSecretFileEncrypted.base64: File to store the base64-encoded encrypted data.

**Step 3: Decode the Base64-Encoded File (Optional)**

The AWS CLI encrypt command outputs the encrypted file in base64 format. If you need to store or work with the binary format (non-base64), decode it:

For **Linux or Mac OS**:

bash

Copy code

cat ExampleSecretFileEncrypted.base64 | base64 --decode > ExampleSecretFileEncrypted

For **Windows**:

bash

Copy code

certutil -decode .\ExampleSecretFileEncrypted.base64 .\ExampleSecretFileEncrypted

This will produce a binary file ExampleSecretFileEncrypted that contains the encrypted data.

**Section 2: Decryption Using AWS KMS**

**Step 1: Decrypt the Encrypted File**

Once the file has been encrypted, you can decrypt it using the AWS KMS decrypt command. This command requires the binary file ExampleSecretFileEncrypted as input, decrypts it, and outputs a base64-encoded version of the original plaintext.

bash

Copy code

aws kms decrypt --ciphertext-blob fileb://ExampleSecretFileEncrypted \

--output text --query Plaintext \

--region eu-west-2 > ExampleFileDecrypted.base64

Here’s a breakdown of the command:

* --ciphertext-blob fileb://ExampleSecretFileEncrypted: Points to the encrypted file.
* --output text --query Plaintext: Outputs the decrypted file in base64 format.
* --region eu-west-2: The region where your KMS key is located.
* ExampleFileDecrypted.base64: File to store the base64-encoded decrypted data.

**Step 2: Decode the Decrypted Base64 File**

The decrypt command outputs the plaintext in base64 format. If you need the original plaintext file, you can decode it:

For **Linux or Mac OS**:

bash

Copy code

cat ExampleFileDecrypted.base64 | base64 --decode > ExampleFileDecrypted.txt

For **Windows**:

bash

Copy code

certutil -decode .\ExampleFileDecrypted.base64 .\ExampleFileDecrypted.txt

This will produce a file ExampleFileDecrypted.txt that contains the original data (the same as in ExampleSecretFile.txt).

**Summary of Key Commands**

| **Operation** | **Command (Linux/Mac)** | **Command (Windows)** |
| --- | --- | --- |
| Encrypt the file | aws kms encrypt --key-id alias/tutorial --plaintext fileb://ExampleSecretFile.txt --output text --query CiphertextBlob --region eu-west-2 > ExampleSecretFileEncrypted.base64 | Same |
| Decode base64 (Optional) | cat ExampleSecretFileEncrypted.base64 | base64 --decode > ExampleSecretFileEncrypted | certutil -decode .\ExampleSecretFileEncrypted.base64 .\ExampleSecretFileEncrypted |
| Decrypt the file | aws kms decrypt --ciphertext-blob fileb://ExampleSecretFileEncrypted --output text --query Plaintext --region eu-west-2 > ExampleFileDecrypted.base64 | Same |
| Decode decrypted file | cat ExampleFileDecrypted.base64 | base64 --decode > ExampleFileDecrypted.txt | certutil -decode .\ExampleFileDecrypted.base64 .\ExampleFileDecrypted.txt |

**Conclusion**

This lab guide demonstrated how to perform encryption and decryption using AWS KMS through the AWS CLI. You now know how to:

* Encrypt a plaintext file using a KMS key.
* Handle base64 encoding and decoding of encrypted and decrypted files.
* Decrypt the file to recover the original plaintext.

AWS KMS encryption is critical for securing sensitive data both at rest and in transit, ensuring your secrets are protected.

**Detailed Explanation of AWS DynamoDB Concepts and Performance**

This guide breaks down key concepts and performance aspects of AWS DynamoDB, particularly focusing on consistency, read capacity units (RCUs), primary keys, and partitioning.

**1. Relational vs. Non-Relational Databases**

* **Relational Data (RDS)**: Structured data stored in **Relational Database Systems (RDS)** like MySQL, PostgreSQL, Oracle, etc. RDS in AWS is a **managed service**, meaning AWS handles routine database tasks such as backups, scaling, and updates.
* **NoSQL Data**: Unstructured or semi-structured data stored in databases that are optimized for fast read/write performance, high availability, and scalability. Examples include:
  + **DynamoDB** (Key-Value store)
  + **DocumentDB** (Document-based NoSQL, MongoDB-compatible)
  + **Neptune** (Graph database, optimized for complex relationships)

**2. DynamoDB Overview**

* **DynamoDB**: A **fully managed NoSQL database** offered by AWS. It supports a **key-value** data model, making it suitable for high-throughput applications that require consistent, low-latency reads and writes at scale.
* **Data Model**:
  + **Key-Value**: DynamoDB stores data as key-value pairs.
  + **Document**: DynamoDB also allows the storage of JSON-like documents.
* **Managed**: AWS handles scaling, availability, and performance tuning automatically without user intervention.

**Kinesis Data Streams and DynamoDB Use Case**

* **AWS Server Logs** → **Kinesis Data Streams** → **DynamoDB**:
  + Logs are ingested from servers and sent to **Kinesis Data Streams** for real-time data streaming.
  + The stream data is processed and then stored in **DynamoDB** for long-term storage and querying.

**3. Consistent Reads and Read Capacity Units (RCUs)**

DynamoDB supports two types of read consistency models:

**1. Strongly Consistent Reads**

* Provides the most up-to-date data.
* Every read returns the most recent write for the queried item.
* **Cost**: 1 **Read Capacity Unit (RCU)** is required for **one strongly consistent read** of **up to 4 KB** of data per second.

**2. Eventually Consistent Reads**

* Eventually consistent reads may not always reflect the most recent write but achieve consistency within milliseconds.
* Consumes half the resources of a strongly consistent read.
* **Cost**: 1 **RCU** provides **two eventually consistent reads** of **up to 4 KB** of data per second.

**Example of Read Capacity Calculation:**

* **1 Strongly Consistent Read** per second for an item of **up to 4 KB** requires **1 RCU**.
* **2 Eventually Consistent Reads** per second for an item of **up to 4 KB** also requires **1 RCU** (since eventually consistent reads use half the resources).
* **Larger Items**: If you need to read items larger than 4 KB, you'll need more RCUs. For example:
  + **12 KB item** with **Eventually Consistent Reads**:
    - 1 RCU covers 2 eventually consistent reads of 4 KB each. A 12 KB item is 3 times larger than 4 KB.
    - To read a 12 KB item with eventually consistent reads, the calculation is as follows:
      * 1 Eventually Consistent Read per second for a 12 KB item: **1/2 RCU**.
      * You would need **16 RCUs** to perform 16 eventually consistent reads for 12 KB items:

text

Copy code

16 RCUs \* 1/2 RCU per read = 24 RCUs total.

**4. Partitioning in DynamoDB**

DynamoDB uses partitions to scale reads and writes. Partitioning is based on the **Primary Key**, which consists of either:

**1. Partition Key (Hash Key)**

* A **single attribute** that DynamoDB uses to determine the partition to store data in.
* **Partition Key** alone uniquely identifies an item.

**2. Composite Primary Key (Partition Key + Sort Key)**

* The **partition key** is combined with a **sort key**.
* Together, they form the **primary key**, and the combination must be unique.
* This allows storing multiple items with the same partition key but differentiated by the sort key.

**How Partitioning Works**:

* DynamoDB automatically spreads the data across multiple partitions based on the **partition key**.
* It uses the **hash value of the partition key** to determine the partition location.
* DynamoDB partitions are necessary to scale performance and storage as your dataset grows.

**Putting it All Together: An Example**

Let’s say you are using DynamoDB to store server logs coming from **Kinesis Data Streams**. You want to retrieve data frequently and need to calculate the number of RCUs required for your use case:

1. **Data Model**:
   * The server logs are stored with a **partition key** server\_id and a **sort key** timestamp to query logs from different servers over time.
2. **Reads**:
   * You perform **eventually consistent reads**.
   * Each log entry is 12 KB in size, and you need to read 16 logs per second.
3. **RCU Calculation**:
   * For **one eventually consistent read** of a **12 KB item**: **1/2 RCU**.
   * For **16 eventually consistent reads** of **12 KB items per second**:

text

Copy code

(16 reads) \* (1/2 RCU per read) = 24 RCUs.

1. **DynamoDB Partitioning**:
   * The logs are partitioned based on the server\_id, ensuring that each server’s logs are stored together.
   * The **sort key** (e.g., timestamp) allows efficient querying of logs over time for each server.

**Conclusion**

* **DynamoDB** is a highly scalable NoSQL database that provides consistent performance by managing partitions automatically.
* It offers flexible read consistency models (strong and eventual) that allow you to optimize cost and performance based on your application needs.
* Understanding **RCU calculations** and **partitioning** is critical to effectively scaling your DynamoDB workloads.

**Lab Guide: Streaming Server Logs to DynamoDB Using Kinesis, Lambda, and AWS DynamoDB**

This lab will demonstrate how to create an end-to-end pipeline where server logs are sent from EC2 to **Kinesis Data Streams**, processed by a **Lambda function**, and stored in **DynamoDB**. The pipeline will leverage **Kinesis Agent** for data ingestion, **Lambda** for processing, and **DynamoDB** as the final data storage.

**Prerequisites**

1. **AWS CLI** installed and configured.
2. **EC2 instance** (Amazon Linux) with access to the internet and configured with the necessary IAM role to use Kinesis and DynamoDB.
3. **DynamoDB**, **Kinesis**, and **Lambda** IAM roles with the appropriate permissions.
4. A **Kinesis Agent** installed on the EC2 instance for log ingestion.

**Architecture Overview**

1. **Server Logs** → **Kinesis Data Streams** → **Lambda** → **DynamoDB**.
2. The logs will be pushed to a Kinesis Data Stream.
3. A Lambda function will be triggered to process the logs and store the records in DynamoDB.

**Section 1: Setup Kinesis Data Stream**

**Step 1.1: Create Kinesis Data Stream**

1. Go to **AWS Management Console** > **Kinesis** > **Create Data Stream**.
2. Name the stream **CadabraOrders**.
3. Set the **capacity mode** to **On-demand** (this mode scales automatically).
4. Create the stream. This might take a few minutes.

**Section 2: Configure Kinesis Agent on EC2**

**Step 2.1: Log into the EC2 Instance**

* Log in to the EC2 instance (Amazon Linux) using SSH or AWS Systems Manager.

**Step 2.2: Edit the Kinesis Agent Configuration File**

1. Open the **Kinesis Agent** configuration file for editing:

bash

Copy code

sudo vi /etc/aws-kinesis/agent.json

1. Replace the content of the configuration file with the following:

json

Copy code

{

"cloudwatch.emitMetrics": true,

"kinesis.endpoint": "",

"firehose.endpoint": "",

"flows": [

{

"filePattern": "/var/log/cadabra/\*.log",

"kinesisStream": "CadabraOrders",

"partitionKeyOption": "RANDOM",

"dataProcessingOptions": [

{

"optionName": "CSVTOJSON",

"customFieldNames": ["InvoiceNo", "StockCode", "Description", "Quantity", "InvoiceDate", "UnitPrice", "Customer", "Country"]

}

]

}

]

}

* **Note**: Ensure that if you add more than one flow, a comma separates them.

**Step 2.3: Restart the Kinesis Agent**

Once you’ve updated the configuration file, restart the Kinesis agent:

bash

Copy code

sudo service aws-kinesis-agent restart

**Step 2.4: Generate Logs for Kinesis Stream**

Generate logs that will be sent to the Kinesis Data Stream:

bash

Copy code

sudo ./LogGenerator.py 100

**Step 2.5: Monitor Kinesis Agent Logs**

You can monitor the logs of the Kinesis Agent to ensure it is working correctly:

bash

Copy code

tail -f /var/log/aws-kinesis-agent/aws-kinesis-agent.log

**Section 3: Create DynamoDB Table**

**Step 3.1: Create the DynamoDB Table**

1. Go to **AWS Management Console** > **DynamoDB** > **Create Table**.
2. Set the **Table Name** to CadabraOrders.
3. Configure the **Partition Key** and **Sort Key**:
   * **Partition Key**: CustomerID (Number).
   * **Sort Key**: OrderID (String).
4. Set the **capacity mode** to **On-demand**.
5. Create the table.

**Section 4: Lambda Setup**

**Step 4.1: Create a Role for Lambda**

1. Go to **IAM Console** > **Roles** > **Create Role**.
2. Select **AWS Service** and choose **Lambda**.
3. Attach the following policies:
   * **AmazonKinesisReadOnlyAccess**.
   * **AmazonDynamoDBFullAccess**.
4. Name the role **CadabraOrders**.

**Step 4.2: Create a Lambda Function**

1. Go to **AWS Lambda Console** > **Create Function**.
2. Choose **Author from Scratch**.
3. Name the function **ProcessOrders**.
4. Select **Python 3.6** as the runtime.
5. Under **Execution Role**, choose **CadabraOrders**.

**Step 4.3: Add Kinesis Trigger to Lambda**

1. In the Lambda function, go to **Triggers** > **Add Trigger**.
2. Choose **Kinesis** and select the **CadabraOrders** stream.
3. Set **Batch Size** to 100.
4. Add the trigger.

**Step 4.4: Add Lambda Code**

1. Navigate to the **Lambda function** > **Code** section.
2. Replace the default code with the following:

python

Copy code

import json

import boto3

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('CadabraOrders')

def lambda\_handler(event, context):

for record in event['Records']:

payload = record['kinesis']['data']

decoded\_record = json.loads(base64.b64decode(payload).decode('utf-8'))

# Extract necessary fields

customer\_id = int(decoded\_record.get('Customer'))

order\_id = decoded\_record.get('InvoiceNo')

# Insert data into DynamoDB

table.put\_item(

Item={

'CustomerID': customer\_id,

'OrderID': order\_id,

'Details': decoded\_record

}

)

return {

'statusCode': 200,

'body': json.dumps('Processed Successfully!')

}

1. **Save** the function.

**Section 5: Test the Setup**

**Step 5.1: Generate Logs in EC2**

Open a new terminal on the same EC2 instance and run the log generator again to simulate incoming logs:

bash

Copy code

sudo ./LogGenerator.py 100

**Step 5.2: Verify DynamoDB Data Insertion**

1. Go to **AWS Management Console** > **DynamoDB**.
2. Check if the **CadabraOrders** table is being populated with data from the logs.
3. Ensure the **CustomerID** and **OrderID** fields are properly filled.

**Section 6: Using Lambda for Scalability**

**Step 6.1: Why Lambda?**

Instead of using a continuously running **Consumer Script**, Lambda functions provide a **serverless** and **scalable** way to process data in real-time from Kinesis Data Streams. This eliminates the need for a dedicated EC2 instance to run consumer code.

**Conclusion**

This lab demonstrated how to set up an end-to-end pipeline using **Kinesis Data Streams**, **Lambda**, and **DynamoDB**. By leveraging **Kinesis Agent** to stream log files and using **Lambda** to process the data in real-time, the system is scalable and can handle large volumes of incoming logs.