AWS Secret Manager Configuration through Python SDK

Scope: This document details the basic concept of how to integrate secret manager that holds the RDS credentials and rotates the password every 30 days.

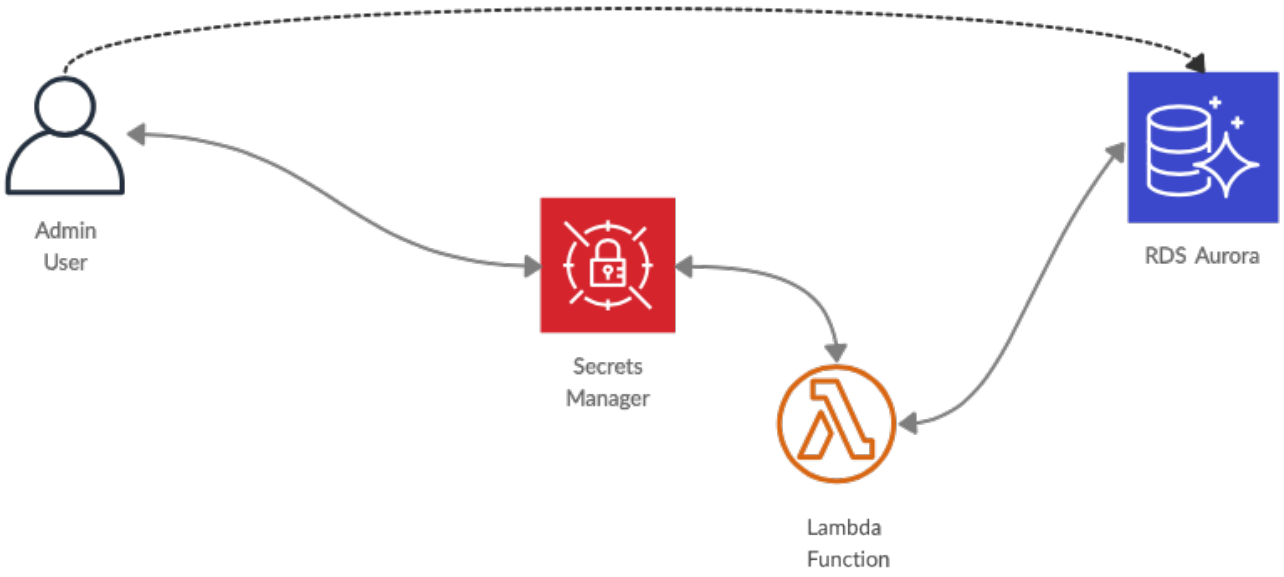
Requirements overview

1. Access to AWS Services
2. Python3 installed
3. Created a RDS Aurora Database and Lambda Function

What is a AWS Secret Manager?

AWS Secrets Manager helps you protect secrets needed to access your applications, services, and IT resources. The service enables you to easily rotate, manage, and retrieve database credentials, API keys, and other secrets throughout their lifecycle.

Flow Diagram



1. Admin User has RDS Aurora Database with username and password
2. Create a secret in AWS Secret Manager using the username and password of RDS Aurora
3. Enable rotation by linking Lambda function to secret
4. The password is immediately rotated and will rotate again after the specified days. (**Note**: You cannot determine the exact time of when the next rotation takes place. You are only promised that after the specified day and within 24 hour, the rotation has happen.)
5. If Admin User wants to access his RDS Aurora, get new password using AWS Secret Manager

Data Objects

These are the required fields needed from RDS. You will need parameter name of the DBClusterIdentifier inside describe_db_clusters():

The minimum object needed are engine, host, port and DBClusterIdentifier.

name	description
engine	Must always be registered as 'mysql' even if the RDS Database is Aurora
host	This is the Endpoint name for the Database cluster. The Endpoint name will be under DB cluster Connectivity & security Endpoints Type: Writer
port	Make port always 3306

The only required fields needed from Lambda is the FunctionArn. You will need the parameter name of the Lambda function inside get_function():

The minimum object needed are LambdaARN.

name	description
FunctionArn	The function's Amazon Resource Name (ARN).

Storing Secret using API

Step 1: Get RDS credentials:

The only required parameter for describe_db_clusters() function is DBClusterIdentifier or the name of the Database Cluster.

Note: The output of the describe_db_clusters() function provides a large dictionary that contains all the credentials about the RDS Database. You need to further extract only the required information of Engine, Endpoint, Port and DBClusterIdentifier from the RDS and add them to a String. The specified format of the String is under the variable 'myrdsString' in the code below.

```
""" Get credentials engine, host, port, dbClusterIdentifier from RDS """
        import boto3

    def get_rds_info():
       cluster_name = 'MyDB-cluster'
                client = boto3.client('rds')

        try:
            rds_response = client.describe_db_clusters(
                DBClusterIdentifier=cluster_name
            )
            Log.info('Successfully accessed RDS DBClusterIdentifier: ' + cluster_name)
        except ClientError as e:
            if e.response['Error']['Code'] == 'ResourceNotFoundException':
                Log.error('The requested RDS name was not found')
            elif e.response['Error']['Code'] == 'InvalidRequestException':
                Log.error('The request was invalid due to:', e)
            elif e.response['Error']['Code'] == 'InvalidParameterException':
                Log.error('The request had invalid params:', e)

        try:
            engine = rds_response['DBClusters'][0]['Engine'] # This extracts only the Engine value the output
of rds_response
        except:
            Log.error('Error while accessing Engine inside list= DBClusters')

        if engine != 'mysql':
            engine = 'mysql'

        try:
            endpoint = rds_response['DBClusters'][0]['Endpoint'] # This extracts only the Endpoint value (host)
the output of rds_response
        except:
            Log.error('Error while accessing Endpoint inside list= DBClusters')

        try:
            port = rds_response['DBClusters'][0]['Port'] # This extracts only the Port value the output of
rds_response
        except:
            Log.error('Error while accessing Port inside list= DBClusters')

        Log.info('Successfully acquired all the credentials of RDS')

        myrdsString = '"engine":"%s","host":"%s","port":"%s","dbClusterIdentifier":"%s"' % (
            engine, endpoint, port, cluster_name)

      return myrdsString
```

Step 2: Get Lambda credentials:

The only required parameter for get_function() is the FunctionName or the name of function.

Note: The output of the get_function() provides a large dictionary that contains all the credentials about the Lambda function. You need to further extract only the required information of FunctionArn from the Lambda function and add them to a String.

```
""" Get credential Lambda ARN from Lambda"""
        import boto3


    def get_lambda_arn():
                lambda_function_name = 'SecretsManagersecret-rotator'
        client = boto3.client(service_name='lambda')

        try:
            lambda_response = client.get_function(FunctionName=lambda_function_name)
            Log.info("Successfully fetching Lambda function: " + lambda_function_name)
        except ClientError as e:
            if e.response['Error']['Code'] == 'ResourceNotFoundException':
                Log.error('The requested Lambda name was not found')
            elif e.response['Error']['Code'] == 'InvalidRequestException':
                Log.error('The request was invalid due to:', e)
            elif e.response['Error']['Code'] == 'InvalidParameterException':
                Log.error('The request had invalid params:', e)

        try:
            lambda_arn = lambda_response['Configuration']["FunctionArn"] # This extracts only the FunctionARD
from the output of lambda_response
            Log.info('Successfully acquired Lambda ARN: ' + lambda_arn)
        except:
            Log.error('Error accessing the dictionary in Lambda function:' + lambda_function_name)
        return lambda_arn
```

Step 3: Creating Secret and Rotate:

After you receive all the credentials from RDS and Lambda, it is time to call the AWS Secret Manager service through client. There are 3 stages in create a secret and rotating the given user password.

1. Create a secret by using create_secret() function and give it a name. This is your SeceretID or the name you have given your secret
2. Call put_secret_value() function with SeceretID, username, password and your RDS credentials. Now your SeceretID contains all the login information for the RDS.
3. Call rotate_secret() function with the SeceretID, Lambda credential and add in the RotationRules={ 'AutomaticallyAfterDays': 30 } in order to specify the number of days needed for the next rotation.

Once executed, the first rotation takes place immediately and your password would have changed.

Note: You cannot determine the exact time of when the next rotation takes place. You are only promised that after the specified day and within 24 hour, the rotation has happen.

Create Secret

- create_secret() function only requires a String name for you secret.
- The output for calling this function contains a dictionary that has credentials for the secret's ARN, Name and VersionID.

```
""" Creating a Secret"""
import boto3
client = boto3.client(service_name='secretsmanager')

def create_new_secret():
        secret_name = 'MySecret'

        try:
        creating_secret = client.create_secret(
                Name=secret_name
        )
        Log.info('Success in creating secret: ' + secret_name)
    except:
        Log.error("Error in creating the secret: create_secret()")
```

Put Value in Secret

- put_secret_value() function works only after a secret of the specified name or SecretID has already been created.
- you need all the RDS credentials in a String in the specified format under Get RDS Credentials and also the Masterusername and Masterpassword of said RDS Database
- put_secret_value() function requires parameters of SecretId and SecretString that contains all the above RDS credentials
- The above step is every important because this links the secret manager to the RDS, otherwise it will be stored as a regular text file.
- The output for calling this function contains a dictionary of Strings that has credentials for the secret's ARN, Name, VersionID and VersionStages.

```
""" Putting the actual values to your Secret"""
import boto3
client = boto3.client(service_name='secretsmanager')

rds_string = get_rds_info()

def create_new_secret():
        secret_name = 'MySecret'
        username = 'admin_login'
        password = '1234'

        try:
               adding_secret = client.put_secret_value(
                SecretId=secret_name,
            SecretString='{"username":"%s","password":"%s",%s}' % (username, password, rds_string)
        )
        Log.info('Success in putting the password in secret')
    except:
        Log.error('Error in putting secret to the SecretID: put_secret_value()')
```

Rotate Secret

- rotate_secret() function works only after a secret of the specified name or SecretID has already been created.
- you need all the Lambda credentials in a String under Get Lambda Credentials.
- rotate_secret() function requires parameters of SecretId, RotationLambdaARN from Lambda credentials and RotationRules to specify the number of days for the next rotation.
- Lambda will use the SecretId and its credentials to connect with the RDS in order to rotate the secret
- The output for calling this function contains a dictionary of Strings that has credentials for the secret's ARN, Name and VersionID

```
""" Rotating your Secret"""
import boto3
client = boto3.client(service_name='secretsmanager')

lambda_arn = get_lambda_arn()

def rotate_my_secret():
        secret_name = 'MySecret'

        try:
        rotate_this_secret = client.rotate_secret(
                SecretId=secret_name,
            RotationLambdaARN=lambda_arn,
            RotationRules={
                    'AutomaticallyAfterDays': 30
            }
               )
        Log.info('Successfully rotating secret')
    except:
        Log.error('Error in rotation of secret : rotate_secret()')
```

Example of adding all the three above function together:

```
session = boto3.session.Session()
client = session.client(service_name='secretsmanager')

"""Creates a secret in the AWS Seceret Manager
        Adds the secret value in the seceret
        Adds rotations to the secret"""

    @staticmethod
    def add_new_secret(username, password, sec_name, l_arn, rds_cred):
        global rotate_this_secret
        try:
            client.create_secret(
                Name=sec_name
            )
            Log.info('Success in creating secret: ' + sec_name)
        except:
            Log.error("Error in creating the secret: create_secret()")

        try:
            client.put_secret_value(
                SecretId=sec_name,
                SecretString='{"username":"%s","password":"%s",%s}' % (username, password, rds_cred)
            )
            Log.info('Success in putting the password in secret')
        except:
            Log.error('Error in putting secret to the SecretID: put_secret_value()')

        try:
            rotate_this_secret = client.rotate_secret(
                SecretId=sec_name,
                RotationLambdaARN=l_arn,
                RotationRules={
                    'AutomaticallyAfterDays': 1
                }
            )
            Log.info('Successfully rotating secret')
        except:
            Log.error('Error in rotation of secret : rotate_secret()')
```

Step 4: Get Secret function:

- get_secret_value() function works only after a secret of the specified name or SecretID has already been created.
- get_secret_value() function requires the parameter of SecretId
- The output for calling this function contains a dictionary of Strings that has credentials for the secret's ARN, Name, VersionID, SecretString, VersionStages and CreatedDate.
- The username and the updated password will be under the SecretString

```
import boto3
from botocore.exceptions import ClientError


def get_secret():
    secret_name = "MySecretName"
    region_name = "us-west-2"

    session = boto3.session.Session()
    client = session.client(
        service_name='secretsmanager',
        region_name=region_name,
    )

    try:
        get_secret_value_response = client.get_secret_value(
            SecretId=secret_name
        )
    except ClientError as e:
        if e.response['Error']['Code'] == 'ResourceNotFoundException':
            print("The requested secret " + secret_name + " was not found")
        elif e.response['Error']['Code'] == 'InvalidRequestException':
            print("The request was invalid due to:", e)
        elif e.response['Error']['Code'] == 'InvalidParameterException':
            print("The request had invalid params:", e)
    else:
        # Secrets Manager decrypts the secret value using the associated KMS CMK
        # Depending on whether the secret was a string or binary, only one of these fields will be populated
        if 'SecretString' in get_secret_value_response:
            text_secret_data = get_secret_value_response['SecretString']
        else:
            binary_secret_data = get_secret_value_response['SecretBinary']

        # Your code goes here.
```

Troubleshooting Connectivity Issues

- Check you AWS credentials and make sure you have all the necessary access
- Check the VPC security groups and see if all the subnet groups of the Lambda exactly match the RDS.

Future Development and Integration

- Integrate Secret Manager to phat repository under the rds.py
- Might need to create a new lambda function through the API that directly connects to AWS Secret Manager in order to rotate the secret.
- Add get_seceret() function to receive the secret with new password after the add_new_secret() has been run.

AWS Secrets Managers Architecture in phat Repository

In the phat repository, the AWS Secrets Managers is integrated with the RDS deployment. The code changes that are added to the existing phat repo to integrate AWS Secrets Managers are contained in files: main, config.yml and rds.py. There are two stages to the AWS Secrets Managers code that reflects the architecture of AWS RDS code, one is the creation of the secret after the RDS-cluster is created and the other is updating the config.yml file with the latest version of the secret. (This is the updating the config.yml file with the latest password.)

config.yml

The secretsmanager in *config.yml* script holds three properties:

```
secretsmanager:
  host: phatcluster.cluster-cbimfz1gncny.us-east-1.rds.amazonaws.com
  name: rds_login
  rotatePasswordInDays: 30
```

name - The name of the secret that holds all the RDS credentials

rotatePasswordInDay - The number of days between the rotation of the secret

main

This is the updated AWS RDS deployment in *main*:

```
 elif type == "rds":
            RDS().create_cluster()
            RDS().create_database_instance()
            RDS().deploy_secrets_manager()
```

Firstly, the RDS creates the Aurora Cluster and then the Aurora database instance.

Note: RDS().create_database_instance() only triggers the creation the of the dbinstance and does not wait for the Aurora database instance to finish being created. The actual creation of the Aurora database instance takes place on a separate thread inside create.py. This is important because it is only after the creation of the dbinstance is fully completed, will the password actually rotate.

RDS().deploy_secrets_manager() creates and/or updates the secrets with the existing RDS.

rds.py

This script holds the major additions of how the Secret Manager is handled. There are two stages, when then secret is first created after the RDS is formed and the other stage is when it checks wether the secret should to be updated or not. The Secret Manager feature starts and ends in deploy_secrets_manager() function. Below are the functions inside RDS that deals with creation and updates secret respectively.

Create Secret	Update Secret
deploy_secrets_manager()	deploy_secrets_manager()
is_exists_secret()	is_exists_secret()
get_rds_endpoint()	
get_lambda_arn()	
create_secret()	

Creating Secret:

- Checks whether the secret exists or not
- When not, gets the RDS endpoint/host using get_rds_endpoint()
- Collects all the RDS credentials into secret_value
- Creates secret, adds secret_value and enables rotation while calling get_lambda_arn()
- Waits for the Aurora database instance to finish being created
- Updates config.yml with the latest password

Updating Secret:

- Checks whether the secret exists or not
- When exists, checks to see if the secret is rotated
- Updates config.yml with the latest password

AWS Lambda Integration with AWS Secrets Manger in the phat repository

Since the Secrets Manger is created during the deployment scripts, the Lambda must a dependency must be added to showcase that a lambda function is necessary for secret rotation.

check_for_secret_lambda() is added to the dependency.py to make sure on this check.

Connecting the Lambda to Secret Manger:

During the Lambda deployment, configure:

common.py

configure_secret_to_lambda() : Adds a permission to secretsManager Lambda. This is a function policy that links the Secrets Manager and Lambda together.

secretsManager.yml

The files consist the name of Lambda Function Name and Lambda Function Policy Name.

AWS Lambda Logic

secretsManager.js

The thing that actually changes the password in RDS and updates the Secret Manger is the Lambda function. When you are using Secret Manger threw the AWS console, the AWS Service automatically creates lambda function for you. But since we are doing this through deployment scripts, in this sprint, I had to actually create a lambda function from scratch, link it with my secret and code the function to access the RDS, change password and update secrets Manager. All the lambda in the deployment scripts are written in JavaScript with a an attached YAML file. The same configuration is used for this lambda function as well. The secrets manager invokes the lambda function in 4 stages:

createSecret: This method first checks for the existence of a secret for the passed in token. If one does not exist, it will generate anew secret and put it with the passed in token.

setSecret: This method gets the AWSPENDING secret and modifys with the database with the AWSPENDING secret.

testSecret:This method tries to log into the database with the secrets staged with AWSPENDING.

finishSecret: This method finishes the secret rotation by staging the secret staged AWSPENDING with the AWSCURRENT stage.

Additional Resources

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/secretsmanager.html#SecretsManager.Client.rotate_secret

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/rds.html#RDS.Client.describe_db_clusters

https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/lambda.html

https://aws.amazon.com/secrets-manager/