## Credential Auto Rotation task

There are lot of components in this Lambda function and so we will go step by step for them

### **Steps to create a secret in Secret Manager to rotate RDS credentials for 90 days**

### **Steps:**

#### **1. Open AWS Secrets Manager**

1. **Sign in to the AWS Management Console**.
2. **Navigate to Secrets Manager**:
   * From the AWS Management Console, navigate to the Secrets Manager service by typing "Secrets Manager" in the search bar and selecting it.

#### **2. Create a New Secret**

1. **Click on "Store a new secret"**:
   * On the Secrets Manager dashboard, click on the "Store a new secret" button.
2. **Select the Secret Type**:
   * Choose "Credentials for RDS database".
3. **Enter Database Credentials**:
   * Provide the username and password for your RDS database instance.
4. **Select the RDS Database**:
   * Choose the RDS database you want to rotate the credentials for from the dropdown list.
5. **Configure Encryption Settings**:
   * By default, Secrets Manager will use the AWS managed key (aws/secretsmanager). You can specify a different KMS key if needed.

#### **3. Configure the Secret**

1. **Secret Name and Description**:
   * Enter a unique name for your secret (e.g., myRDSSecret).
   * Optionally, provide a description for better identification.
2. **Tags (Optional)**:
   * You can add tags to manage and identify your secret.

#### **4. Enable Automatic Rotation**

1. **Enable Automatic Rotation**:
   * Toggle the "Enable automatic rotation" switch.
2. **Set Rotation Schedule**:
   * Choose "Custom rotation period".
   * Enter 90 days for the rotation period.
3. **Select Lambda Function**:
   * If you already have a Lambda function for rotation, select it.
   * If not, choose "Create a new Lambda function".

#### **5. Create a Lambda Function for Rotation (If not existing)**

1. **Create a New Lambda Function**:
   * Follow the prompt to create a new Lambda function.
   * Select the appropriate IAM role for the Lambda function that has permissions to manage the RDS credentials.
2. **Lambda Function Template**:
   * AWS provides a template for rotating RDS credentials. Use this template to create your Lambda function.
3. **Review and Finish**:
   * Review the configuration of your Lambda function and complete the setup.

#### **6. Finalize and Store the Secret**

1. **Review Secret Configuration**:
   * Review all the settings you have configured for your secret.
2. **Store the Secret**:
   * Click "Next" and then "Store" to save the secret.

### Steps to verify your mail in SNS AWS service

### **Steps:**

#### **1. Open the AWS Management Console**

1. **Sign in to the AWS Management Console**.
2. **Navigate to SNS**:
   * From the AWS Management Console, type "SNS" in the search bar and select "Simple Notification Service" from the dropdown.

#### **2. Navigate to Email Subscriptions**

1. **Go to Email Subscriptions**:
   * In the SNS dashboard, click on "Subscriptions" in the left-hand menu.
   * Click on the "Create subscription" button.

#### **3. Create a New Subscription**

1. **Select Topic**:
   * Choose the topic to which you want to subscribe your email address. If you don't have a topic yet, you need to create one first.
   * Click "Create topic", provide a name for your topic, and create it.
2. **Configure Subscription**:
   * In the "Create subscription" form, select your topic ARN.
   * For "Protocol", select "Email".
   * For "Endpoint", enter the email address you want to verify.
   * Click on "Create subscription".

#### **4. Verify Your Email Address**

1. **Check Your Email**:
   * After you create the subscription, AWS SNS will send a verification email to the email address you provided.
2. **Open the Verification Email**:
   * Go to your email inbox and open the email from AWS Notifications.
   * The email will have a subject like "AWS Notification - Subscription Confirmation".
3. **Confirm Subscription**:
   * Click the "Confirm subscription" link in the email.
   * You will be redirected to a confirmation page indicating that your subscription has been confirmed.

#### **5. Verify the Subscription in the SNS Console**

1. **Return to the SNS Console**:
   * Go back to the SNS dashboard in the AWS Management Console.
2. **Check Subscription Status**:
   * Navigate to the "Subscriptions" section.
   * You should see your new subscription with the status "Confirmed".

### **Steps to create a Lambda function in AWS**

### **Steps:**

#### **1. Open the AWS Management Console**

1. **Sign in to the AWS Management Console**.
2. **Navigate to Lambda**:
   * In the AWS Management Console, type "Lambda" in the search bar and select "Lambda" from the dropdown.

#### **2. Create a New Lambda Function**

1. **Click on "Create function"**:
   * On the Lambda dashboard, click the "Create function" button.
2. **Choose a Function Creation Method**:
   * Select "Author from scratch".
3. **Configure Basic Settings**:
   * **Function name**: Enter a unique name for your Lambda function.
   * **Runtime**: Select the runtime for your Lambda function (e.g., Node.js, Python, Java, etc.).
   * **Role**: Select "Create a new role with basic Lambda permissions" to allow Lambda to write logs to CloudWatch. Alternatively, you can choose an existing role or create a new one with custom permissions.
4. **Advanced Settings (Optional)**:
   * You can configure advanced settings such as VPC, execution timeout, and environment variables.
5. **Click on "Create function"**:
   * After entering the basic settings, click the "Create function" button to create your Lambda function.

#### **3. Configure the Lambda Function**

1. **Add Code**:
   * In the "Function code" section, you can write or upload the code for your Lambda function. You can either:
     + Write code directly in the built-in code editor.
     + Upload a .zip file with your function code and dependencies.
     + Use an S3 bucket to upload your function code.
2. **Set Up the Handler**:
   * The handler is the entry point for your Lambda function. The default handler is typically index.handler, where index is the name of your code file, and handler is the name of the function within your code.
3. **Configure Environment Variables (Optional)**:
   * You can set environment variables to pass configuration settings to your function code.
4. **Set Memory and Timeout**:
   * Adjust the memory allocation and timeout settings as needed for your function. The default timeout is 3 seconds.

#### **4. Add Triggers**

1. **Add a Trigger**:
   * In the "Designer" section, click on the "Add trigger" button.
   * Select a service that will trigger your Lambda function (e.g., API Gateway, S3, DynamoDB, etc.).
   * Configure the trigger settings according to the service you selected.
2. **Click on "Add"**:
   * After configuring the trigger, click the "Add" button to associate it with your Lambda function.

#### **5. Test the Lambda Function**

1. **Create a Test Event**:
   * Click on the "Test" button in the top-right corner of the Lambda function page.
   * Configure a new test event by selecting a template or creating a custom event.
2. **Run the Test**:
   * After configuring the test event, click the "Test" button to run your Lambda function with the test input.
3. **Review the Results**:
   * Check the execution results and logs to ensure your Lambda function runs as expected. Logs are available in CloudWatch Logs.

### **AWS steps distribution to implement the autorotation of credential code**

1. Create a secret to store RDS credentials for 90 days and rotate the secrets accordingly
2. Make the lambda function with the code that i have explained and given below
3. Attach the appropriate roles to the IAM policy
4. Verify the mails in SNS
5. Deploy and test the AWS Lambda function

### **Compiling them into the Lambda function**

This code now performs the steps in the order specified:

1. Generates new credentials and updates the RDS instance with the new password.
2. Stores the new credentials in Secrets Manager.
3. Send an email using SES with the updated credentials.
4. Stores each secret field in the Systems Manager Parameter Store.

### **Deploying the Lambda Function**

1. **Open Cloud9 and Create a New Directory**:  
     
   1. mkdir lambda-secrets-to-ssm

cd lambda-secrets-to-ssm

**2. Create index.js**:  
  
touch index.js

**3. Open index.js in the Cloud9 editor** and paste the combined code.

**Initialize Node.js Project**:  
  
npm init -y

**4. Install AWS SDK Packages**:  
  
npm install @aws-sdk/client-secrets-manager @aws-sdk/client-rds @aws-sdk/client-ses @aws-sdk/client-ssm

**5. Zip the Project Files**:  
  
cd ..

zip -r lambda-secrets-to-ssm.zip lambda-secrets-to-ssm

1. **Deploy the Lambda Function**:
   * Open the AWS Lambda console.
   * Create a new Lambda function.
   * Upload the lambda-secrets-to-ssm.zip file.
   * Set the handler to index.handler.
   * Configure the function's execution role to allow access to Secrets Manager, Systems Manager, RDS, and SES.
2. **Set Environment Variables**:
   * In the Lambda function configuration, set the necessary environment variables (e.g., EMAIL\_LIST, SES\_SOURCE\_EMAIL).
3. **Test the Lambda Function**:
   * Create a test event in the Lambda console and invoke the function to ensure it works as expected.

**b) Zip file conversion**

To ensure that the Lambda function can correctly find and use index.js, you need to create the zip file in a way that includes the index.js and node\_modules directory at the root of the zip file. Here are the steps:

**Navigate to the Project Directory**: Ensure you are in the project directory where index.js and node\_modules are located.  
  
cd ~/environment/lambda-secrets-to-ssm

### **Complete Steps**

Here’s the complete sequence of commands starting from the project directory:

bash

Copy code

# Navigate to the project directory

cd ~/environment/lambda-secrets-to-ssm

# Create the zip file, ensuring it's in the parent directory

zip -r ../lambda-secrets-to-ssm.zip .

# Verify the contents of the zip file

unzip -l ../lambda-secrets-to-ssm.zip

**c) After creating the zip file, you can upload it to AWS Lambda:**

1. **Open the AWS Lambda console**.
2. **Create a new Lambda function** or select an existing one.
3. **Upload the lambda-secrets-to-ssm.zip file**.
4. **Set the handler to index.handler**.
5. **Configure the function’s execution role** to allow access to Secrets Manager, Systems Manager, RDS, and SES.
6. **Set environment variables** like EMAIL\_LIST and SES\_SOURCE\_EMAIL in the Lambda configuration.
7. **Test the Lambda function** with a test event to ensure it works as expected.

### **The code and explanation ( we can use cloud9 for writing the same)**

**Code**

const { SecretsManagerClient, GetSecretValueCommand, PutSecretValueCommand } = require("@aws-sdk/client-secrets-manager");

const { RDSClient, ModifyDBInstanceCommand, DescribeDBInstancesCommand } = require("@aws-sdk/client-rds");

const { SESClient, SendEmailCommand } = require("@aws-sdk/client-ses");

const { SSMClient, PutParameterCommand } = require("@aws-sdk/client-ssm");

const https = require('https');

const secretsManagerClient = new SecretsManagerClient();

const rdsClient = new RDSClient();

const sesClient = new SESClient();

const ssmClient = new SSMClient();

exports.handler = async (event) => {

try {

// Step 1: Generate new credentials and update RDS instance with new password

const secretName = 'AppBeta';

const secretValue = await getSecretValue(secretName);

const newPassword = generatePassword();

const newSecretValue = { ...secretValue, password: newPassword };

await updateRDSInstance(newSecretValue);

// Step 2: Store the new credentials in Secrets Manager

await storeNewSecret(secretName, newSecretValue);

// Step 3: Send an email using SES of the updated credentials

const emailListString = process.env.EMAIL\_LIST;

const emailList = emailListString.split(',').map(email => email.trim().replace(/"/g, ''));

const sesSourceEmail = process.env.SES\_SOURCE\_EMAIL.trim().replace(/"/g, '');

const subject = "New RDS Credentials";

const body = `New credentials for RDS instance: ${JSON.stringify(newSecretValue)}`;

for (const email of emailList) {

await sendEmail(sesSourceEmail, email, subject, body);

}

// Step 4: Store each secret field in Systems Manager Parameter Store

const parameterNames = {

username: '/project/aws-node/db\_username',

password: '/project/aws-node/db\_password',

engine: '/project/aws-node/db\_engine',

host: '/project/aws-node/db\_host',

port: '/project/aws-node/db\_port',

dbInstanceIdentifier: '/project/aws-node/db\_dbInstanceIdentifier'

};

for (const [key, parameterName] of Object.entries(parameterNames)) {

if (newSecretValue[key]) {

const value = String(newSecretValue[key]);

console.log(`Storing ${parameterName} with value: ${value}`);

await ssmClient.send(new PutParameterCommand({

Name: parameterName,

Value: value,

Type: 'SecureString',

Overwrite: true

}));

}

}

// Trigger GitHub Actions workflow

await triggerGitHubWorkflow();

return {

statusCode: 200,

body: JSON.stringify({ message: 'Operation completed successfully' }),

};

} catch (error) {

console.error('Error:', error);

return {

statusCode: 500,

body: JSON.stringify({ message: 'Internal Server Error', error: error.message }),

};

}

};

const getSecretValue = async (secretName) => {

const command = new GetSecretValueCommand({ SecretId: secretName });

const response = await secretsManagerClient.send(command);

return JSON.parse(response.SecretString);

};

const generatePassword = (length = 16) => {

const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!#$%^&\*()\_+';

return Array.from({ length }, () => characters.charAt(Math.floor(Math.random() \* characters.length))).join('');

};

const updateRDSInstance = async (newSecretValue, retries = 3) => {

const { dbInstanceIdentifier, username, password } = newSecretValue;

for (let attempt = 0; attempt < retries; attempt++) {

try {

const dbInstanceStatus = await getRDSInstanceStatus(dbInstanceIdentifier);

if (dbInstanceStatus !== 'available') {

throw new Error(`DBInstance ${dbInstanceIdentifier} is not in available state. Current state: ${dbInstanceStatus}`);

}

const command = new ModifyDBInstanceCommand({

DBInstanceIdentifier: dbInstanceIdentifier,

MasterUserPassword: password

});

await rdsClient.send(command);

return;

} catch (error) {

if (attempt < retries - 1) {

console.log(`Retrying updateRDSInstance (${attempt + 1}/${retries})...`);

await new Promise(resolve => setTimeout(resolve, 5000));

} else {

throw error;

}

}

}

};

const getRDSInstanceStatus = async (dbInstanceIdentifier) => {

const command = new DescribeDBInstancesCommand({ DBInstanceIdentifier: dbInstanceIdentifier });

const response = await rdsClient.send(command);

return response.DBInstances[0].DBInstanceStatus;

};

const storeNewSecret = async (secretName, newSecretValue) => {

const command = new PutSecretValueCommand({

SecretId: secretName,

SecretString: JSON.stringify(newSecretValue)

});

await secretsManagerClient.send(command);

};

const sendEmail = async (sourceEmail, destinationEmail, subject, body) => {

const command = new SendEmailCommand({

Source: sourceEmail,

Destination: { ToAddresses: [destinationEmail] },

Message: {

Subject: { Data: subject },

Body: { Text: { Data: body } }

}

});

await sesClient.send(command);

};

const triggerGitHubWorkflow = async () => {

const data = JSON.stringify({

event\_type: "trigger-ci-cd"

});

const options = {

hostname: 'api.github.com',

path: `/repos/NundiniKhanna/Plantix-Internship/actions/workflows/deploy.yml/dispatches`,

method: 'POST',

headers: {

'Authorization': 'token github\_pat\_11AXGRJJA0ejZIhX7LTkM7\_vAe2Jy24JDmDHCk5dMG782bmNGkMBBsf2j5vDHQmjiDBIFQI5Q7zmg8zh',

'User-Agent': 'node.js',

'Accept': 'application/vnd.github.v3+json',

'Content-Type': 'application/json',

'Content-Length': data.length

}

};

return new Promise((resolve, reject) => {

const req = https.request(options, (res) => {

let responseData = '';

res.on('data', (chunk) => {

responseData += chunk;

});

res.on('end', () => {

resolve(responseData);

});

});

req.on('error', (e) => {

reject(e);

});

req.write(data);

req.end();

});

};

**Explanation**

It automates several tasks, including:

**1. Rotating RDS Credentials:**

* It retrieves secrets from AWS Secrets Manager using a specific secret name (AppBeta).
* It generates a new, random password.
* It updates the password for the RDS instance using the RDS client.
* It stores the updated secret value (including the new password) back into Secrets Manager.

**2. Sending Email Notification:**

* It retrieves an email list and source email from environment variables.
* It constructs an email with the updated credentials and sends it to the email list using the SES client.

**3. Storing Secrets in Parameter Store:**

* It iterates through predefined secret fields (username, password, etc.).
* It retrieves the corresponding value from the updated secret.
* It stores each secret field securely in Systems Manager Parameter Store using the SSM client.

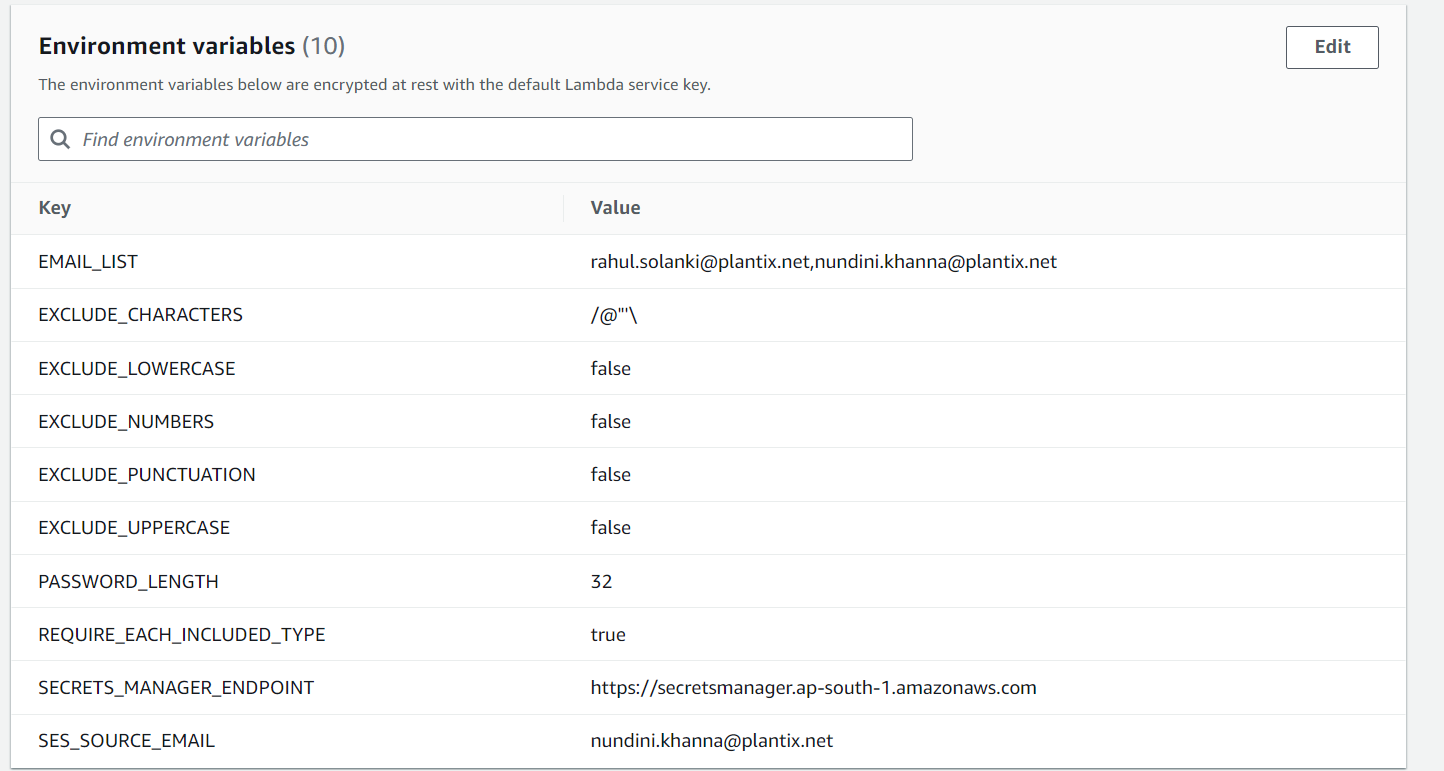
**4. Triggering GitHub Actions Workflow:**

* It constructs a JSON payload with the desired event type ("trigger-ci-cd").
* It sends a POST request to the GitHub API using HTTPS, specifying the workflow name (deploy.yml) in the path.
* It uses an access token stored in the GITHUB\_PAT environment variable for authorization.

**Breakdown of Key Sections:**

1. **Imports:**
   * The code starts by importing necessary libraries for interacting with different AWS services (Secrets Manager, RDS, SES, SSM) and making HTTPS requests.
2. **Client Initialization:**
   * Clients for each AWS service are instantiated, allowing the function to interact with those services through API calls.
3. **exports.handler Function:**
   * This is the main entry point of the Lambda function. It's an asynchronous function (async) that handles the event and potential errors.
4. **a. Try Block:**
   * This block contains the main logic of the function, performing the steps mentioned above (rotating credentials, sending emails, storing secrets, triggering workflow).
5. **b. Catch Block:**
   * This block catches any errors that occur during execution and returns an appropriate error response.
6. **Helper Functions:**
   * These functions are defined outside the main handler and provide specific functionalities for each task (retrieving secrets, generating passwords, updating RDS, sending emails, storing secrets in SSM, triggering workflow).

There are the environment variables:



Add this IAM policy to the Lambda function as an inline policy:

**IAM policy**

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"ses:SendEmail",

"ses:SendRawEmail"

],

"Resource": "arn:aws:ses:ap-south-1:815469512242:identity/nundini.khanna@plantix.net"

},

{

"Effect": "Allow",

"Action": [

"secretsmanager:GetSecretValue",

"secretsmanager:PutSecretValue",

"secretsmanager:UpdateSecretVersionStage",

"secretsmanager:DescribeSecret"

],

"Resource": "\*"

},

{

"Effect": "Allow",

"Action": [

"rds:ModifyDBInstance",

"rds:DescribeDBInstances",

"rds-db:connect"

],

"Resource": "arn:aws:rds:ap-south-1:815469512242:db:database-task-3"

},

{

"Effect": "Allow",

"Action": [

"ssm:PutParameter"

],

"Resource": [

"arn:aws:ssm:ap-south-1:815469512242:parameter/project/aws-node/db\_username",

"arn:aws:ssm:ap-south-1:815469512242:parameter/project/aws-node/db\_password",

"arn:aws:ssm:ap-south-1:815469512242:parameter/project/aws-node/db\_engine",

"arn:aws:ssm:ap-south-1:815469512242:parameter/project/aws-node/db\_host",

"arn:aws:ssm:ap-south-1:815469512242:parameter/project/aws-node/db\_port",

"arn:aws:ssm:ap-south-1:815469512242:parameter/project/aws-node/db\_dbInstanceIdentifier"

]

}

]

}