# IAM role-based authentication to Amazon Aurora from serverless applications

Src: <https://aws.amazon.com/blogs/database/iam-role-based-authentication-to-amazon-aurora-from-serverless-applications/>

by Mahesh Balumuri | on 13 SEP 2019 | in [Amazon Aurora](https://aws.amazon.com/blogs/database/category/database/amazon-aurora/), [AWS Identity and Access Management (IAM)](https://aws.amazon.com/blogs/database/category/security-identity-compliance/aws-identity-and-access-management-iam/), [AWS Lambda](https://aws.amazon.com/blogs/database/category/compute/aws-lambda/), [Serverless](https://aws.amazon.com/blogs/database/category/serverless/) | [Permalink](https://aws.amazon.com/blogs/database/iam-role-based-authentication-to-amazon-aurora-from-serverless-applications/) | [Comments](https://aws.amazon.com/blogs/database/iam-role-based-authentication-to-amazon-aurora-from-serverless-applications/#Comments) | [Share](https://aws.amazon.com/blogs/database/iam-role-based-authentication-to-amazon-aurora-from-serverless-applications/)

*January 2024: This post was reviewed and updated for accuracy.*

Storing user names and passwords directly in applications is not a best practice. Saving credentials as plaintext should never occur in a secure application. As a solution, [AWS Identity and Access Management](https://docs.aws.amazon.com/iam/) (IAM) policies can assign permissions that determine who is allowed to manage [Amazon Aurora](https://aws.amazon.com/rds/aurora/) resources. For example, you can use IAM to determine who is allowed to create, describe, modify, and delete DB clusters, tag resources, or security groups. In Amazon Aurora, you can associate the database users with the IAM user and roles.

In this post, I explain how to use IAM for accessing your Amazon Aurora databases using an [AWS Lambda](https://docs.aws.amazon.com/lambda/) function.

**Overview**

You can use [IAM database authentication](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.html) to connect to the DB cluster. Using this method, you can access the database with an authentication token generated instead of storing the password in a configuration file. Amazon Aurora generates an [AWS Signature Version 4](https://docs.aws.amazon.com/general/latest/gr/signature-version-4.html) authentication token that is valid for 15 minutes to create a connection from your application. As authentication is fully managed externally by IAM, you do not need to create credentials in the database.

**Walkthrough**

The following diagram shows the workflow.

Diagram of a software process

Description automatically generated

Lambda is a compute service that runs your code without provisioning or managing servers. Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. When using Lambda, you are responsible only for your code. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources. Lambda executes the Lambda function and returns results.

Here’s how you can create a Lambda function with the console.

* Open the [AWS Lambda console](https://console.aws.amazon.com/lambda/home).
* Choose Create function.
* For Function name, enter demo-function.
* Select your preferred language runtime. For this post, choose Python 3.11.
* For permissions, select Create a new role with basic Lambda permission. This creates an execution role (for example, demo-function-role-ipqlab4h), with permissions to upload logs to [Amazon CloudWatch](https://docs.aws.amazon.com/cloudwatch/)
* When you choose Create function, you see the options shown in the following screenshot.

A screenshot of a computer

Description automatically generated

Designer lets you configure triggers and view permissions.

You can create a function using [the AWS Lambda console editor](https://docs.aws.amazon.com/lambda/latest/dg/code-editor.html), as shown in the following screenshot.

A screenshot of a computer

Description automatically generated

Lambda includes sample code that returns a success response. The code editor in the Lambda console enables you to write, test, and view the execution results of your Lambda function code.

You can edit your function code with the embedded [AWS Cloud9](https://docs.aws.amazon.com/cloud9/latest/user-guide/) editor as long as your source code doesn’t exceed the 3 MB limit. If you must include any libraries and dependencies other than the AWS SDK, you must [create a deployment package](https://docs.aws.amazon.com/lambda/latest/dg/lambda-python-how-to-create-deployment-package.html). You can upload the package directly if it does not exceed 50 MB. Otherwise, you can upload to [Amazon S3](https://docs.aws.amazon.com/s3/) and then upload it to Lambda.

The following screenshot shows a dialog box allowing you to set [Lambda environment variables](https://docs.aws.amazon.com/lambda/latest/dg/env_variables.html).

A screenshot of a computer

Description automatically generated

Environment variables for Lambda functions enable you to dynamically pass settings to your function code and libraries, without making changes to your code.

The following screenshot shows a dialog allowing you to set a Lambda [execution role](https://docs.aws.amazon.com/lambda/latest/dg/lambda-intro-execution-role.html):

A screenshot of a computer

Description automatically generated

A Lambda function’s execution role grants it permission to access AWS services and resources.

The following screenshot shows a dialog box allowing you to choose a network:

A screenshot of a computer

Description automatically generated

By default, [a Lambda function has access to your public internet address and public AWS APIs](https://docs.aws.amazon.com/lambda/latest/dg/welcome.html). To access the private resources like Amazon RDS located in private subnets, you must enable your Lambda function for VPC. It’s a best practice to [configure the Lambda function in the same VPC](https://docs.aws.amazon.com/lambda/latest/dg/vpc.html) where the database instance resides. Additionally, ensure that the private subnets have a [NAT gateway](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html) added in their route tables to grant the Lambda function internet access.

Here’s how you can access Amazon Aurora from a Lambda function using IAM authentication:

* You need the [AWS Command Line Interface (AWS CLI)](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html) installed and configured on your machine before moving to the next steps.
* [Enable IAM database authentication](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.Enabling.html) in the DB cluster using the AWS CLI as follows. You could also use the [AWS Management Console or Amazon RDS API](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.Enabling.html).

aws rds modify-db-cluster --db-cluster-identifier <cluster-identifier> --enable-iam-database-authentication --apply-immediately

* + cluster-identifier: name of your DB cluster.  
    Example:

aws rds modify-db-cluster --db-cluster-identifier mysql-demo --enable-iam-database-authentication --apply-immediately

* Connect to the DB cluster, and create a user with login privileges and grant IAM role access to the user:
  + *PostgreSQL*: Grant rds\_iam privilege to the user.
  + CREATE USER <db\_user\_name> WITH LOGIN;

GRANT rds\_iam TO <db\_user\_name>;

SQL

Example:

CREATE USER demouser WITH LOGIN;

GRANT rds\_iam TO demouser;

SQL

* + *MySQL*: Grant privileges as follows.
  + CREATE USER < db\_user\_name> IDENTIFIED WITH AWSAuthenticationPlugin as 'RDS';

GRANT ALL PRIVILEGES ON <dbname>.\* TO '<db\_user\_name>’ @'%';

SQL

Example:

CREATE USER demouser IDENTIFIED WITH AWSAuthenticationPlugin as 'RDS';

GRANT ALL PRIVILEGES ON demodb.\* TO 'demouser’ @'%';

FLUSH PRIVILEGES;

SQL

* Now, create the policy which allows the user created in the previous step to access the database using IAM database authentication. For more information, see [IAM Policy and IAM Database Access.](https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/UsingWithRDS.IAMDBAuth.IAMPolicy.html)  
  Policy Document:
* {
* "Version": "2012-10-17",
* "Statement": [
* {
* "Effect": "Allow",
* "Action": [
* "rds-db:connect"
* ],
* "Resource": [
* "arn:aws:rds-db:<region>:<account-id>:dbuser:<DbiResourceId>/<db\_user\_name>"
* ]
* }
* ]

}

JSON

Use the following settings.

* + region: The region of the database instance.
  + account-id: The AWS account number of the database instance.
  + DbiResourceId: The DB instance identifier, or \* to allow all the database instances in the region of the account.
  + db\_user\_name: The user that has been created in the database. Mention a specific user, or \* to allow IAM authentication for all users in the database.

Follow this example to create the policy using AWS CLI.

aws iam create-policy --policy-name my-policy --policy-document '{

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

"Statement": [

{

"Effect": "Allow",

"Action": [

"rds-db:connect"

],

"Resource": [

"arn:aws:rds-db:us-east-1:123456789012:dbuser:\*/demouser"

]

}

]

}'

* Now attach this policy to the IAM role associated with the Lambda function:

aws iam attach-role-policy --role-name <role\_name> --policy-arn <policy\_arn>

* + role\_name: IAM role associated with the Lambda function (for example, demo-function-role-ipqlab4h).
  + policy\_arn: ARN of IAM policy created for IAM database authentication (for example  “arn:aws:iam::123456789012:policy/my-policy”).
* Set the following environment variables in Lambda.
  + DBEndPoint: The DB instance endpoint (for example, ab9c3efgh31k.us-east-1.rds.amazonaws.com).
  + DatabaseName: The name of the database.
  + DBUserName: The user name created in the previous steps (for example, demouser).

To encrypt the network traffic, [use the RDS-provided root SSL certificate](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/UsingWithRDS.SSL.html), such as [this example](https://truststore.pki.rds.amazonaws.com/us-east-1/us-east-1-bundle.pem). You must include this SSL certificate in your deployment package.

To access the Amazon Aurora PostgreSQL database from your Lambda function, you must include the [pg8000 (PostgreSQL Driver)](https://pypi.org/project/pg8000/) library as part of your deployment package.

Here is the Lambda function for Amazon Aurora PostgreSQL.

import boto3

import pg8000.dbapi

import os

import ssl

import logging

logger = logging.getLogger()

# declare the global connection object to use during warm starting

# to reuse connections that were established during a previous invocation.

connection = None

def get\_connection():

"""

Method to establish the connection.

"""

try:

logger.info ("Connecting to database")

# Create a low-level client with the service name for rds

client = boto3.client("rds")

# Read the environment variables to get DB EndPoint

DBEndPoint = os.environ.get("DBEndPoint")

# Read the environment variables to get the Database name

DatabaseName = os.environ.get("DatabaseName")

# Read the environment variables to get the Database username which has access to database.

DBUserName = os.environ.get("DBUserName")

# ssl context information

ssl\_context = ssl.create\_default\_context()

ssl\_context.verify\_mode = ssl.CERT\_REQUIRED

# update the certificate filename based on the one which is packaged with lambda function.

ssl\_context.load\_verify\_locations('us-east-1-bundle.pem')

# Generates an auth token used to connect to a db with IAM credentials.

password = client.generate\_db\_auth\_token(

DBHostname=DBEndPoint, Port=5432, DBUsername=DBUserName

)

# Establishes the connection with the server using the token generated as password

conn = pg8000.dbapi.connect(

host=DBEndPoint,

user=DBUserName,

database=DatabaseName,

password=password,

ssl\_context=ssl\_context

)

return conn

except Exception as e:

logger.error ("While connecting failed due to :{0}".format(str(e)))

return None

def lambda\_handler(event, context):

global connection

try:

if connection is None:

connection = get\_connection()

if connection is None:

return {"status": "Error", "message": "Failed"}

logger.info ("instantiating the cursor from connection")

# Instantiate the cursor object

cursor = connection.cursor()

# Query to be executed

query = "SELECT CURRENT\_DATABASE()"

# execute the query / command in the connected database

cursor.execute(query)

# Get the column names

columns = [str(desc[0]) for desc in cursor.description]

results = []

# create the rows of dict from the result set.

for res in cursor:

results.append(dict(zip(columns, res)))

cursor.close()

# result to return

response = {"status": "Success", "results": results}

return response

except Exception as e:

try:

connection.close()

except Exception as e:

connection = None

logger.error ("Failed due to :{0}".format(str(e)))

return {"status": "Error", "message": "Something went wrong. Try again"}

Python

To access Amazon Aurora MySQL database from your Lambda function, you must include [mysql-connector-python (MySQL Driver](https://pypi.org/project/mysql-connector-python/" \t "_blank)) library as part of your deployment package.

Here is the Lambda function for Amazon Aurora MySQL.

import os

import boto3

from mysql import connector

from mysql.connector import Error

import logging

logger = logging.getLogger()

# declare the global connection object to use during warm starting

# to reuse connections that were established during a previous invocation.

connection = None

def get\_connection():

"""

Method to establish the connection.

"""

try:

logger.info ("Connecting to database")

# Create a low-level client with the service name for rds

client = boto3.client("rds")

# Read the environment variables to get DB EndPoint

DBEndPoint = os.environ.get("DBEndPoint")

# Read the environment variables to get the Database name

DatabaseName = os.environ.get("DatabaseName")

# Read the environment variables to get the Database username which has access to database.

DBUserName = os.environ.get("DBUserName")

# Generates an auth token used to connect to a db with IAM credentials.

password = client.generate\_db\_auth\_token(

DBHostname=DBEndPoint, Port=3306, DBUsername=DBUserName

)

# Establishes the connection with the server using the token generated as password

conn = connector.connect(

host=DBEndPoint,

database=DatabaseName,

user=DBUserName,

password=password,

auth\_plugin="mysql\_clear\_password",

ssl\_ca="us-east-1-bundle.pem",

)

return conn

except Exception as e:

logger.error ("While connecting failed due to :{0}".format(str(e)))

return None

def lambda\_handler(event, context):

global connection

try:

if connection is None:

connection = get\_connection()

if connection is None:

return {"status": "Error", "message": "Failed"}

logger.info ("instantiating the cursor from connection")

# Instantiate the cursor object

# Instantiate the cursor object with dictionary as true to return the rows as dictionaries

cursor = connection.cursor(dictionary=True)

# execute the query / command in the connected database

cursor.execute("select database() as database\_name")

# Retreive the next row of query result set.

record = cursor.fetchone()

# Return the results

response = {"status": "Success", "results": record}

return response

except Exception as e:

try:

connection.close()

except Exception as e:

connection = None

logger.error ("Failed due to :{0}".format(str(e)))

return {"status": "Error", "message": "Something went wrong. Try again"}

Python

**Conclusion**

Many applications store the database credentials in the plaintext files, which are not encrypted and expose credentials to the users, which is a potential risk. Now you can use IAM to centrally manage access to your database resources, instead of managing access individually on each database instance. As long as your IAM user or role has the permission to access the database, you can generate the authentication token to access the database.