Drift DETECTION LAMBDA FUNCTION

[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you’re ready to add your content, just click here and start typing.]

[Document subtitle]

….

1. Purpose of this code

The code is used to detect drifts (permissions, roles, parameters etc.) in RDS PostgreSQL databases to fulfil CIS requirements and to perform regular audits to understand deviations from the information security standards.

1. Code breakup-walkthrough

The code is used via an AWS Lambda Function, or a lambda expression written in Python language to anonymously define single user single/multi use functions.

* AWS Services Used:

The function used multiple AWS services and features. Here's a breakdown of the AWS services and features used in the code:

**Amazon RDS (Relational Database Service)**:

* + Amazon RDS is used for managing and running relational databases. The code connects to RDS instances and performs SQL queries to retrieve information about database objects and permissions.

**AWS Secrets Manager**:

* + AWS Secrets Manager is used for storing and managing sensitive information, such as database connection credentials. The code retrieves database connection secrets from Secrets Manager to connect to RDS instances securely.

**AWS Lambda**:

* + AWS Lambda is not directly used in the code, but it's a serverless compute service that can execute code in response to events. Lambda functions can be used to trigger the code you provided in response to specific events or on a schedule.

**Amazon S3 (Simple Storage Service)**:

* + Amazon S3 is not explicitly mentioned in the provided code, but it is commonly used to store and manage files, including logs, reports, or backups generated by Lambda functions or other AWS services.

**AWS IAM (Identity and Access Management)**:

* + AWS IAM is used to manage access control and permissions for AWS resources. In the code, roles and permissions for accessing AWS services and resources are managed using IAM roles.
* Libraries/Modules Used:

The code uses multiple python libraries and modules. Each of them are explained below.

**import boto3**: This imports the **boto3** library, which is the official AWS SDK for Python. It allows you to interact with various AWS services programmatically.

**import botocore**: This imports the **botocore** library, which is a low-level library for handling AWS interactions. It is often used internally by **boto3**.

**from botocore.exceptions import ClientError**: This imports the **ClientError** class from the **botocore.exceptions** module. **ClientError** is used for handling exceptions related to AWS service clients.

**import postgres\_utils as pgs**: This imports a module named **postgres\_utils** and gives it the alias **pgs**. It suggests that there's a custom Python module called **postgres\_utils** that contains utility functions related to PostgreSQL or database operations.

**from postgres\_utils import pg**: This imports a specific object or function named **pg** from the **postgres\_utils** module. It's used to access a particular function or class from the custom module.

**from csv2pdf import convert**: This imports a function named **convert** from a module named **csv2pdf**. This suggests that there's a custom module called **csv2pdf** that contains a function for converting CSV data to PDF format.

**from PIL import ImageFont**: This imports the **ImageFont** class from the Python Imaging Library (PIL), which is often used for working with images and fonts in image processing tasks.

**import logging**: This imports the **logging** module, which is used for adding logging capabilities to the Python script. It allows you to record and manage log messages.

**from datetime import datetime**: This imports the **datetime** class from the Python **datetime** module. It's used for working with date and time values.

**import csv**: This imports the built-in **csv** module, which provides functionality for working with CSV (Comma-Separated Values) files.

**import os**: This imports the built-in **os** module, which is used for interacting with the operating system. It allows you to perform file and directory operations.

**import pgdb**: This imports the **pgdb** module, which is often used for working with PostgreSQL databases in Python. It provides a database interface for PostgreSQL.

* Functions Used & overview of code blocks:

code defines a set of functions used for interacting with AWS services and handling RDS (Relational Database Service) instances and snapshots. Here's the purpose of each function and an overview of what each block of code does:

**get\_assume\_role\_session(sts\_role)**: This function obtains temporary AWS credentials by assuming an IAM role (sts\_role) using the Security Token Service (STS) client. It returns the temporary credentials.

**run\_query(secrets\_client, secret\_arn, sql)**: This function runs an SQL query on an RDS instance. It first attempts to establish a database connection using **pgs.get\_connection** from a custom module and secret information retrieved from AWS Secrets Manager. If the connection is successful, it executes the SQL query using **pgs.run\_query\_using\_secrets**. It logs status messages and returns a status code.

**update\_table(rds\_list, parameter\_list, table\_name)**: This function is responsible for updating a table (**table\_name**) with data from a list of RDS instances (**rds\_list**) using an SQL **INSERT INTO** statement. It constructs the SQL query dynamically based on the provided data.

**create\_or\_alter\_table(parameter\_list, table\_name)**: This function creates or alters a table (**table\_name**) in an RDS database. It generates SQL statements to create or alter the table structure based on the list of column headers (**parameter\_list**).

**parse\_instances\_info(rds\_client, acct\_id, acct\_ids\_dict, scan\_id)**: This function retrieves and parses information about RDS instances from target AWS accounts. It uses the **rds\_client** to describe RDS instances and clusters and populates the **rds\_list** with instance details. It also handles exceptions and logs errors. After retrieving the instance information, it creates or updates an "instances\_info" table.

**parse\_snapshots\_info(rds\_client, acct\_id, acct\_ids\_dict, scan\_id)**: This function retrieves and parses information about RDS snapshots from target AWS accounts. It uses the **rds\_client** to describe RDS instance snapshots and populates the **rds\_snapshots\_list** with snapshot details. It also handles exceptions and logs errors. After retrieving the snapshot information, it creates or updates a "snapshots\_info" table.

**get\_secrets\_list(secrets\_client)**: This function retrieves a list of secrets from AWS Secrets Manager using the provided **secrets\_client**. It creates a dictionary mapping secret names to their ARNs.

**get\_account\_ids()**: This function retrieves a list of AWS account IDs based on the environment and account IDs specified in environment variables. It uses the **sts\_org\_arn** to assume an IAM role in AWS Organizations to list accounts. It filters accounts based on the environment and specified account IDs and returns a dictionary mapping account IDs to account names.

SQL Based functions:

For each function, the general pattern is to iterate through the result list, convert each row into a dictionary using the each.\_asdict() method (assuming each is a namedtuple), and append the resulting dictionary to the appropriate list with predefined keys. The function then returns the updated list.

**get\_user\_roles\_list**: This function takes four arguments: **scan\_id**, **acct\_id**, **user\_roles\_list**, and **result**. It iterates through the **result** list, which presumably contains namedtuples (or similar objects with named attributes), converts each result row into a dictionary, and appends it to the **user\_roles\_list** as a dictionary with specific keys.

**get\_database\_permissions\_list**: Similar to the first function, this one parses database permission-related results and appends them to the **database\_permissions\_list**.

**get\_schema\_privs\_role\_list**: This function processes schema privilege-related results and adds them to the **schema\_privs\_role\_list**.

**get\_role\_specific\_privs\_list**: Handles role-specific privilege results and appends them to the **role\_specific\_privs\_list**.

**get\_role\_priv\_tables\_list** and **get\_role\_specific\_priv\_tables\_list**: These functions deal with privileges related to tables and specific tables, respectively, and store the data in **role\_priv\_tables\_list** and **role\_specific\_priv\_tables\_list**.

**get\_views\_ownership\_usage\_privs\_list**: Parses views, ownership, and usage privilege results, and stores them in **views\_ownership\_usage\_privs\_list**.

**get\_view\_privs\_role\_list**: Handles view privilege-related results and adds them to **view\_privs\_role\_list**.

**get\_sequence\_ownership\_usage\_privs\_list**: This function processes sequences, ownership, and usage privilege results and stores them in **sequence\_ownership\_usage\_privs\_list**.

**get\_roles\_specific\_privileges\_sequences\_list**: Deals with specific privilege sequences and appends them to **roles\_specific\_privileges\_sequences\_list**.

**get\_roles\_privs\_fdw\_list**, **get\_roles\_login\_fdw\_list**, and **get\_roles\_privs\_language\_list**: These functions handle foreign data wrapper (FDW) permissions, login permissions for FDWs, and language privileges, respectively.

**get\_functions\_ownership\_roles\_list**: This function processes functions, ownership, and roles related to them.

**get\_audit\_roles\_list**: Parses audit roles data and appends it to **audit\_roles\_list**.

**get\_public\_roles\_list**: Processes public roles data and appends it to **public\_roles\_list**. This function appears to handle different cases based on the number of keys in the dictionary, suggesting varying result formats.

Python function, generate\_csv\_and\_pdf\_reports\_for\_the\_drift\_tables, performs several tasks related to generating and storing CSV and PDF reports based on data from various database tables. This function connects to a reporting database, retrieves data from specified tables for a given scan, generates CSV and PDF reports, and uploads these reports to an S3 bucket for storage. It processes multiple tables, handling different structures and default parameters as needed for each table.

**Function Signature and Parameters**:

**secrets\_client**: A client object used to connect to a database with secrets.

**reporting\_db\_secret\_arn**: ARN (Amazon Resource Name) of the secret used to access the reporting database.

**table\_names\_list**: A list of names of the tables for which reports need to be generated.

**scan\_id**: An identifier representing the current scan. This value will typically be incremented with each invocation

**S3 Bucket Creation**:

The function first checks if an S3 bucket exists for storing CSV and PDF reports. The bucket name is constructed based on the environment using **os.getenv('environment')**. If the bucket doesn't exist, it is created in the **us-east-2** AWS region.

**Font and Default Parameters**:

The function sets up a default font and a dictionary of default parameters for different table names. These parameters include the columns that should be included in the report.

**Table Loop**:

The function iterates through each table name in the **table\_names\_list**.

**SQL Query Execution**:

For each table, it constructs an SQL query that selects data from that table for the given **scan\_id**. It then uses **pgs.run\_query\_using\_secrets** to execute the query using the provided secrets client and reporting database secret ARN. If an exception occurs during query execution, an empty result list is assigned.

**CSV and PDF Report Generation**:

The function prepares to generate CSV and PDF reports based on the retrieved data. It calculates the header list for the CSV file and the column sizes for formatting the PDF. It constructs file names for both the CSV and PDF reports based on the table name and scan ID.

If the CSV file does not exist, it is created and populated with data. If the CSV file already exists, data is appended to it. The function calculates the total width and maximum column height needed for PDF generation based on the column sizes. It converts the CSV file to a PDF file using a custom **convert** function. The PDF is then saved locally.

**File Upload to S3**:

The function uploads both the CSV and PDF files to the S3 bucket under appropriate folders (**csv\_files/** and **pdf\_files/**) using the S3 client.

1. CIS statements used & purpose.
2. How to call the function.
3. Sample
4. Sample