Report on AWS VPC Peering and Flow Log Automation Script

This Python script utilizes the **Boto3 SDK** to manage various AWS resources, such as VPCs, route tables, VPC peering connections, and CloudWatch logs for flow logging. The script automates the process of setting up **VPC peering**, configuring **route tables**, and managing **flow logs** between a lab VPC and a shared VPC. The following report provides an overview of the key components and functions of the script.

1. AWS Credentials and Session Management

The script begins by retrieving AWS credentials and region settings from the environment variables:

```
python

aws_access_key_id = os.getenv('AWS_ACCESS_KEY_ID')
aws_secret_access_key = os.getenv('AWS_SECRET_ACCESS_KEY')
aws_default_region = os.getenv('AWS_DEFAULT_REGION')
```

The AWS SDK, Boto3, is used to initiate a session:

```
python

session = boto3.Session()
ec2_client = session.client('ec2')
```

2. VPC Retrieval and Identification

The script defines a function `get_vpc_id(cidr_block)` that retrieves VPCs and matches their CIDR blocks to return their corresponding VPC IDs:

```
python

def get_vpc_id(cidr_block):
    vpcs = ec2_client.describe_vpcs()['Vpcs']
    for vpc in vpcs:
        if vpc['CidrBlock'] == cidr_block:
            return vpc['VpcId']
    return None
```

This function is used to identify two VPCs: **Lab VPC** and **Shared VPC**, based on their CIDR blocks (`10.0.0.0/16` and `10.0.5.0/16`).

3. VPC Peering Setup

If the VPC IDs for both the Lab and Shared VPCs are found, the script initiates a **VPC Peering Connection**:

```
python

peering_connection = ec2_client.create_vpc_peering_connection(
    VpcId=lab_vpc_id,
    PeerVpcId=shared_vpc_id
)
```

After the peering connection is created, the script accepts the connection:

```
python

peering_connection_id =
peering_connection['VpcPeeringConnection']['VpcPeeringConnectionId']
ec2_client.accept_vpc_peering_connection(VpcPeeringConnectionId=peer
ing_connection_id)
```

The peering connection establishes a network link between the Lab and Shared VPCs.

4. Route Table Configuration

The script retrieves route tables for both VPCs using the `describe_route_tables` API, then adds routes to ensure traffic can flow between the VPCs via the peering connection:

```
python
```

```
ec2_client.create_route(
    RouteTableId=route_table_lab_id,
    DestinationCidrBlock='10.0.5.0/16',
    VpcPeeringConnectionId=peering_connection_id
)
```

Routes are added in both VPCs' route tables, allowing traffic from the Lab VPC to reach the Shared VPC and vice versa.

5. Flow Logs Configuration

To monitor traffic between the VPCs, the script creates **VPC Flow Logs**. A **CloudWatch Log Group** is created to store the flow log data:

```
python

logs_client.create_log_group(LogGroupName=log_group_name)
```

Flow logs for both VPCs are created using the `create_flow_logs` API:

```
python

flow_log_response = ec2_client.create_flow_logs(
    ResourceIds=[lab_vpc_id, shared_vpc_id],
    ResourceType='VPC',
    TrafficType='ALL',
    LogGroupName=log_group_name,
    DeliverLogsPermissionArn=flow_logs_role_arn
)
```

This step allows tracking of traffic within and between the VPCs, capturing information on accepted and rejected traffic.

6. Flow Log Query and Analysis

The script includes a function `get_flow_logs(log_group_name, start_time, end_time)` to retrieve flow logs from the created CloudWatch log group. It uses

CloudWatch Logs Insights to query the logs and retrieve network traffic data such as source/destination IPs and ports, protocols, and actions (e.g., accept/reject):

```
python

fields @timestamp, srcAddr, dstAddr, srcPort, dstPort, protocol,
action
| sort @timestamp desc
| limit 100
```

7. Additional Log Stream and Log Event Retrieval

The script also includes utility functions to list **log streams** and retrieve **log events** from the log group for further analysis:

- `get_log_streams(log_group_name)` lists the log streams in a log group.
- `get_log_events(log_group_name, log_stream_name)` retrieves log events for a specific stream.

These logs are crucial for investigating network traffic behavior and troubleshooting connectivity issues between the VPCs.

8. AWS Account Information and Debugging

For debugging and management purposes, the script prints out AWS account details, all available VPCs, and their route tables. It retrieves the AWS Account ID using the **STS** service:

```
python

account_id = sts_client.get_caller_identity()['Account']
```

Additionally, it lists all the VPCs and route tables in the account for reference.

Conclusion

This script automates key aspects of VPC management in AWS, specifically:

- **VPC Peering Setup**: Facilitates communication between two VPCs.
- **Route Table Updates**: Ensures proper routing between the peered VPCs.
- **Flow Logs Setup**: Provides monitoring and logging of VPC network traffic.
- **Automation of Network Monitoring**: Retrieves and analyzes flow logs to help identify and resolve potential network issues.

By using this script, an AWS architect can simplify and streamline the process of establishing a peering connection between VPCs, configuring routing, and setting up traffic monitoring. This automation is particularly useful for maintaining secure and efficient network communication in multi-VPC architectures.