Experiment-11

Aim: To understand the working of Identity and Access management (IAM) in cloud computing and to demonstrate the case study based on IAM on AWS/Azure cloud platform.

Prerequisites: AWS subscription

Theory :-

AWS identity and access management (IAM) is a web service that helps you securely control access to AWS resources. You can use IAM to control who is authenticated and authorized to use resources.

IAM Terms:

1. **IAM resources**: The user, group, role, policy and identity provider objects that are stored in IAM. As with other AWS services, you can add, edit and remove resources from IAM .
2. **IAM identities**: The IAM resource objects that are used to identify and group. You can attach a policy to an IAM identity. These include users, groups and roles.
3. **IAM entities**: The IAM resource objects that AWS uses for authentication. These include IAM users and roles.
4. **Principals**: A person or application that uses the AWS account root user, an IAM user, or an IAM role to sign in and make requests to AWS. Principals include federated users and assumed roles.

IAM Features:

1. **Shared access to your AWS account**:
   You can grant other people permission to administer and use resources in your AWS account without having to share your password or access key.

2. **Granularpermissions**:
   You can grant different permissions to different people for different resources. For example, you might allow some users complete access to Amazon Elastic Compute Cloud (Amazon EC2), Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, Amazon RedShift, and other AWS services. For other users, you can allow read-only access to just some S3 buckets, or permission to administer some EC2 instances, or to access your billing information but nothing else.
3. **Secure access to AWS resources for applications that run on Amazon EC2**:
   You can use IAM features to securely provide credentials for applications that run on EC2 instances. These credentials provide permissions for your application to access other AWS resources. Examples include S3 buckets and DynamoDB tables.
4. **Multi-factor authentication (MFA)**:
   You can add two-factor authentication to your account and to individual users for extra security. With MFA you or your users must provide not only a password or access key to work with your account, but also a code from a specially configured device.
5. **Identityfederation**:
   You can allow users who already have passwords elsewhere—for example, in your corporate network or with an internet identity provider—to get temporary access to your AWS account.

Principal

A principal is a person or application that can make a request for an action or operation on an AWS resource. The principal is authenticated as the AWS account root user or an IAM entity to make requests to AWS. As a best practice, do not use your root user credentials for your daily work. Instead, create IAM entities (users and roles). You can also support federated users or programmatic access to allow an application to access your AWS account.

Request

When a principal tries to use the AWS Management Console, the AWS API, or the AWS CLI, that principal sends a request to AWS. AWS gathers the request information into a request context, which is used to evaluate and authorize the request. The request includes the following information:

1. Actions or operations
2. Resources
3. Principal
4. Environment data
5. Resource data

Authentication

A principal must be authenticated (signed in to AWS) using their credentials to send a request to AWS. Some services, such as Amazon S3 and AWS STS, allow a few requests from anonymous users. However, they are the exception to the rule.

Authorization

You must also be authorized (allowed) to complete your request. During authorization, AWS uses values from the request context to check for policies that apply to the request. It then uses the policies to determine whether to allow or deny the request. Most policies are stored in AWS as JSON documents and specify the permissions for principal entities. There are several types of policies that can affect whether a request is authorized. To provide your users with permissions to access the AWS resources in their own account, you need only identity-based policies. Resource-based policies are popular for granting cross-account access. The other policy types are advanced features and should be used carefully. Resources

After AWS approves the operations in your request, they can be performed on the related resources within your account. A resource is an object that exists within a service. Examples include an Amazon EC2 instance, an IAM user, and an Amazon S3 bucket. The service defines a set of actions that can be performed on each resource. If you create a request to perform an unrelated action on a resource, that request is denied.

Using IAM with AWS SDK

The following code sample illustrates how to:

- Create a user who has no permissions.
- Create a role that grants permission to list Amazon S3 buckets for the account.
- Add a policy to let the user assume the role.
- Assume the role and list Amazon S3 buckets using temporary credentials.
- Delete the policy, role, and user.

```
# Create an IAM user and a role that grants permission to list buckets.
# The user has rights only to assume the role. After assuming the role, use temporary credentials to list buckets for the account.
import json
import sys
import time
from uuid import uuid4
```

```
import boto3
from botocore.exceptions import ClientError

def progress_bar(seconds):
    for _ in range(seconds):
        time.sleep(1)
        print('.', end='')
        sys.stdout.flush()
    print()
def setup(iam_resource):
    try:
        user = iam_resource.create_user(UserName=f'demo-user-{uuid4()}')
        print(f"Created user {user.name}.")
    except ClientError as error:
        print(f"Couldn't create a user for the demo. Here's why: "
              f"{error.response['Error']['Message']}")
        raise
    try:
        user_key = user.create_access_key_pair()
        print(f"Created access key pair for user.")
    except ClientError as error:
        print(f"Couldn't create access keys for user {user.name}. Here's why: "
              f"{error.response['Error']['Message']}")
        raise
    print(f"Wait for user to be ready.", end='')
    progress_bar(10)
    try:
        role = iam_resource.create_role(
            RoleName=f'demo-role-{uuid4()}',

            AssumeRolePolicyDocument=json.dumps({

                'Version': '2012-10-17',

                'Statement': [{

                    'Effect': 'Allow',
                    'Principal': {'AWS': user.arn},
                    'Action': 'sts:AssumeRole'}]}))
        print(f"Created role {role.name}.")
    except ClientError as error:
        print(f"Couldn't create a role for the demo. Here's why: "
              f"{error.response['Error']['Message']}")
        raise
    try:
```

```
        policy = iam_resource.create_policy(
            PolicyName=f'demo-policy-{uuid4()}',
            PolicyDocument=json.dumps({
                'Version': '2012-10-17',
                'Statement': [{
                    'Effect': 'Allow',
                    'Action': 's3:ListAllMyBuckets',
                    'Resource': 'arn:aws:s3:::*'}]}))
        role.attach_policy(PolicyArn=policy.arn)
        print(f"Created policy {policy.policy_name} and attached it to the role.")
    except ClientError as error:
        print(f"Couldn't create a policy and attach it to role {role.name}. Here's why: f"{error.response['Error']['Message']}")
        raise
    try:
        user.create_policy(
            PolicyName=f'demo-user-policy-{uuid4()}',
            PolicyDocument=json.dumps({
                'Version': '2012-10-17',
                'Statement': [{
                    'Effect': 'Allow',
                    'Action': 'sts:AssumeRole',
                    'Resource': role.arn}]}))
        print(f"Created an inline policy for {user.name} that lets the user assume "
              f"the role.")
    except ClientError as error:
        print(f"Couldn't create an inline policy for user {user.name}. Here's why: "
              f"{error.response['Error']['Message']}")
        raise
    print("Give AWS time to propagate these new resources and connections.", end='')
    progress_bar(10)
    return user, user_key, role
def show_access_denied_without_role(user_key):
    print(f"Try to list buckets without first assuming the role.")
    s3_denied_resource = boto3.resource(
        's3', aws_access_key_id=user_key.id, aws_secret_access_key=user_key.secret)
    try:
        for bucket in s3_denied_resource.buckets.all():
            print(bucket.name)
        raise RuntimeError("Expected to get AccessDenied error when listing buckets!")
    except ClientError as error:
        if error.response['Error']['Code'] == 'AccessDenied':
            print("Attempt to list buckets with no permissions: AccessDenied.")
        else:
```

```
            raise
def list_buckets_from_assumed_role(user_key, assume_role_arn, session_name):
    sts_client = boto3.client(
        'sts', aws_access_key_id=user_key.id, aws_secret_access_key=user_key.secret)
    try:
        response = sts_client.assume_role(
            RoleArn=assume_role_arn, RoleSessionName=session_name)
        temp_credentials = response['Credentials']
        print(f"Assumed role {assume_role_arn} and got temporary credentials.")
    except ClientError as error:
        print(f"Couldn't assume role {assume_role_arn}. Here's why: "
              f"{error.response['Error']['Message']}")
        raise

    # Create an S3 resource that can access the account with the temporary credentials.
    s3_resource = boto3.resource(
        's3',
        aws_access_key_id=temp_credentials['AccessKeyId'],
        aws_secret_access_key=temp_credentials['SecretAccessKey'],
        aws_session_token=temp_credentials['SessionToken'])
    print(f"Listing buckets for the assumed role's account:")
    try:
        for bucket in s3_resource.buckets.all():
            print(bucket.name)
    except ClientError as error:
        print(f"Couldn't list buckets for the account. Here's why: "
              f"{error.response['Error']['Message']}")
        raise

def teardown(user, role):
    try:
        for attached in role.attached_policies.all():
            policy_name = attached.policy_name
            role.detach_policy(PolicyArn=attached.arn)
            attached.delete()
            print(f"Detached and deleted {policy_name}.")
        role.delete()
        print(f"Deleted {role.name}.")
    except ClientError as error:
        print("Couldn't detach policy, delete policy, or delete role. Here's why: "
              f"{error.response['Error']['Message']}")
        raise
    try:
```

```
        for user_pol in user.policies.all():
            user_pol.delete()
            print("Deleted inline user policy.")
        for key in user.access_keys.all():
            key.delete()
            print("Deleted user's access key.")
        user.delete()
        print(f"Deleted {user.name}.")
    except ClientError as error:
        print("Couldn't delete user policy or delete user. Here's why: "
              f"{error.response['Error']['Message']}")
def usage_demo():
    print('-'*88)
    print(f"Welcome to the IAM create user and assume role demo.")
    print('-'*88)
    iam_resource = boto3.resource('iam')
    user = None
    role = None
    try:
        user, user_key, role = setup(iam_resource)
        print(f"Created {user.name} and {role.name}.")
        show_access_denied_without_role(user_key)
        list_buckets_from_assumed_role(user_key, role.arn, 'AssumeRoleDemoSession')
    except Exception:
        print("Something went wrong!")
    finally:
        if user is not None and role is not None:
            teardown(user, role)
        print("Thanks for watching!")
if __name__ == '__main__':
    usage_demo()
```

Conclusion-

Sign and Remark:

R1 (3 Marks)	R2 (2 Marks)	R3 (5 Marks)	R4 (5 Marks)	Total (15 Marks)	Signature