AWS IAM

Thursday, December 12, 2024

11:37 AM

### Core and Important Concepts of AWS IAM

1. \*\*Users\*\*: An IAM user is an entity that you create in AWS to represent a person or service that interacts with AWS. Each user has a unique name within the account.

2. \*\*Groups\*\*: An IAM group is a collection of IAM users. You can use groups to specify permissions for multiple users, which can make it easier to manage the permissions.

3. \*\*Roles\*\*: An IAM role is an IAM identity that you can create in your account that has specific permissions. It is intended to be assumable by anyone who needs it, secure through policies, and it can be granted temporary permissions.

4. \*\*Policies\*\*: Policies are JSON documents that define permissions. These documents specify what actions are allowed or denied on which AWS resources. Policies can be attached to users, groups, or roles.

5. \*\*Authentication\*\*: The process of verifying the identity of someone or something. AWS provides several authentication mechanisms including passwords, access keys, MFA (Multi-Factor Authentication), and identity federation.

6. \*\*Authorization\*\*: The process of determining what actions can be performed by an authenticated principal (user, role, etc.). AWS IAM uses policies to manage this.

### Real-time Use Case: Managing IAM Roles for EC2 Instances to Access S3

\*\*Use Case Description:\*\*

You need to allow an application running on an EC2 instance to access objects in an S3 bucket. To achieve this, you will create an IAM role and attach it to the EC2 instance with the necessary permissions to access the S3 bucket.

### Steps and Important Points to Consider

#### Step 1: Create an IAM Role

Create an IAM role with permissions to access S3.

\*\*Java Code to Create an IAM Role\*\*:

\*\*Maven Dependencies (pom.xml)\*\*:

```xml

<dependencies>

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-iam</artifactId>

<version>1.11.1034</version>

</dependency>

</dependencies>

```

\*\*Create IAM Role\*\*:

```java

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;

import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;

import com.amazonaws.services.identitymanagement.model.\*;

public class IAMRoleCreation {

private static final String ROLE\_NAME = "EC2S3AccessRole";

private static final String POLICY\_NAME = "S3AccessPolicy";

public static void main(String[] args) {

final AmazonIdentityManagement iam = AmazonIdentityManagementClientBuilder.standard()

.withCredentials(new ProfileCredentialsProvider())

.withRegion("us-west-2")

.build();

// Create trust relationship policy for EC2 Service

String assumeRolePolicyDocument = "{\n" +

" \"Version\": \"2012-10-17\",\n" +

" \"Statement\": [\n" +

" {\n" +

" \"Effect\": \"Allow\",\n" +

" \"Principal\": {\n" +

" \"Service\": \"ec2.amazonaws.com\"\n" +

" },\n" +

" \"Action\": \"sts:AssumeRole\"\n" +

" }\n" +

" ]\n" +

"}";

CreateRoleRequest createRoleRequest = new CreateRoleRequest()

.withRoleName(ROLE\_NAME)

.withAssumeRolePolicyDocument(assumeRolePolicyDocument);

CreateRoleResult createRoleResult = iam.createRole(createRoleRequest);

String roleArn = createRoleResult.getRole().getArn();

System.out.println("Created Role Arn: " + roleArn);

// Create policy to allow S3 access

String policyDocument = "{\n" +

" \"Version\": \"2012-10-17\",\n" +

" \"Statement\": [\n" +

" {\n" +

" \"Effect\": \"Allow\",\n" +

" \"Action\": [\n" +

" \"s3:GetObject\",\n" +

" \"s3:PutObject\",\n" +

" \"s3:ListBucket\"\n" +

" ],\n" +

" \"Resource\": \"arn:aws:s3:::your-s3-bucket-name/\*\"\n" +

" }\n" +

" ]\n" +

"}";

CreatePolicyRequest createPolicyRequest = new CreatePolicyRequest()

.withPolicyName(POLICY\_NAME)

.withPolicyDocument(policyDocument);

CreatePolicyResult createPolicyResult = iam.createPolicy(createPolicyRequest);

String policyArn = createPolicyResult.getPolicy().getArn();

System.out.println("Created Policy Arn: " + policyArn);

// Attach policy to role

AttachRolePolicyRequest attachRolePolicyRequest = new AttachRolePolicyRequest()

.withRoleName(ROLE\_NAME)

.withPolicyArn(policyArn);

iam.attachRolePolicy(attachRolePolicyRequest);

System.out.println("Attached policy to role.");

}

}

```

#### Step 2: Attach the IAM Role to an EC2 Instance

To attach the IAM role to your EC2 instances, you need to create an instance profile and assign the role to it. Then you can associate the instance profile with your EC2 instance.

\*\*Maven Dependencies (pom.xml)\*\*:

```xml

<dependencies>

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-ec2</artifactId>

<version>1.11.1034</version>

</dependency>

</dependencies>

```

\*\*Create Instance Profile and Attach Role\*\*:

```java

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.ec2.AmazonEC2;

import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;

import com.amazonaws.services.ec2.model.AssociateIamInstanceProfileRequest;

import com.amazonaws.services.ec2.model.AssociateIamInstanceProfileResult;

import com.amazonaws.services.ec2.model.IamInstanceProfileSpecification;

public class EC2InstanceProfile {

private static final String INSTANCE\_ID = "i-0abcdef1234567890";

private static final String ROLE\_NAME = "EC2S3AccessRole";

public static void main(String[] args) {

final AmazonEC2 ec2 = AmazonEC2ClientBuilder.standard()

.withCredentials(new ProfileCredentialsProvider())

.withRegion("us-west-2")

.build();

IamInstanceProfileSpecification iamInstanceProfileSpecification = new IamInstanceProfileSpecification()

.withName(ROLE\_NAME);

AssociateIamInstanceProfileRequest associateIamInstanceProfileRequest = new AssociateIamInstanceProfileRequest()

.withInstanceId(INSTANCE\_ID)

.withIamInstanceProfile(iamInstanceProfileSpecification);

AssociateIamInstanceProfileResult result = ec2.associateIamInstanceProfile(associateIamInstanceProfileRequest);

System.out.println("Assigned IAM Role to EC2 Instance: " + result.getIamInstanceProfileAssociation().getAssociationId());

}

}

```

#### Step 3: Access S3 from EC2 Using IAM Role

From within your EC2 instance, utilize the AWS SDK to access S3 resources. The IAM role attached to the instance will provide the necessary permissions.

\*\*Java Code in EC2 to Access S3\*\*:

\*\*Maven Dependencies (pom.xml)\*\*:

```xml

<dependencies>

<dependency>

<groupId>com.amazonaws</groupId>

<artifactId>aws-java-sdk-s3</artifactId>

<version>1.11.1034</version>

</dependency>

</dependencies>

```

\*\*S3 Access in Java\*\*:

```java

import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;

import com.amazonaws.services.s3.AmazonS3;

import com.amazonaws.services.s3.AmazonS3ClientBuilder;

import com.amazonaws.services.s3.model.S3Object;

public class S3Access {

private static final String BUCKET\_NAME = "your-s3-bucket-name";

private static final String OBJECT\_KEY = "your-object-key";

public static void main(String[] args) {

final AmazonS3 s3 = AmazonS3ClientBuilder.standard()

.withCredentials(new DefaultAWSCredentialsProviderChain())

.withRegion("us-west-2")

.build();

// Get an object from S3

S3Object s3object = s3.getObject(BUCKET\_NAME, OBJECT\_KEY);

System.out.println("Content-Type: " + s3object.getObjectMetadata().getContentType());

}

}

```

### Important Points to Consider

1. \*\*Least Privilege Principle\*\*: Always grant the minimum permissions necessary for a role or user to perform their job functions. This reduces security risks.

2. \*\*Use IAM Roles for EC2\*\*: Using IAM roles for EC2 instances provides an easy and secure way to grant permissions without managing long-term access keys.

3. \*\*Rotate IAM Credentials\*\*: Regularly rotate IAM credentials to mitigate the impact of compromised credentials.

4. \*\*Monitor and Audit\*\*: Use AWS CloudTrail and CloudWatch to monitor and audit IAM changes and activities to ensure compliance and detect anomalies.

5. \*\*Multi-Factor Authentication (MFA)\*\*: Enable MFA for highly privileged users to add an extra layer of security.

6. \*\*Resource-Level Permissions\*\*: When possible, use resource-level permissions to limit access to specific resources and actions.

### Conclusion

This guide covered the core concepts of AWS IAM, provided a real-time use case of managing IAM roles for EC2 instances to access S3, and offered important best practices to consider. The Java code examples demonstrated how to create and manage IAM roles, attach them to EC2 instances, and use those roles to securely access S3 resources. Implementing these concepts helps ensure secure and efficient access management within AWS environments.

AWS IAM policies are essential for defining permissions in AWS. They are JSON documents that specify what actions are allowed or denied on which resources. Understanding the structure and fields of IAM policies is crucial for managing permissions effectively. Let's delve deeper into the components and elements of IAM policies.

### Structure of an IAM Policy

An IAM policy consists of one or more statements, each of which contains the following fields:

1. \*\*Version\*\*: Specifies the version of the policy language. The current version is `2012-10-17`, which you should use for new policies.

2. \*\*Id\*\* (optional): An identifier for the policy. It can be used to manage multiple versions of a policy.

3. \*\*Statement\*\*: A list of individual statements (`Statement` objects). Each statement specifies a single permission or set of permissions.

### Fields in a Policy Statement

Each `Statement` object contains the following fields:

1. \*\*Effect\*\*: Specifies whether the statement allows or denies access. Valid values are `Allow` and `Deny`. If a request is explicitly denied or not allowed by any statement in the policy, it will be denied.

2. \*\*Action\*\*: Specifies the actions that are allowed or denied. Actions usually correspond to API operations (e.g., `s3:PutObject`, `ec2:StartInstances`). You can specify multiple actions using a wildcard (e.g., `s3:\*`).

3. \*\*Resource\*\*: Specifies the resources to which the actions apply. This can be an ARN (Amazon Resource Name) or a wildcard (`\*`) to signify all resources.

4. \*\*Principal\*\*: Specifies the user, account, service, or other entity that is allowed or denied access to the resources. This field is primarily used in resource-based policies.

5. \*\*Condition\*\* (optional): Specifies conditions under which the statements will apply. These are key-value pairs that further refine when a policy is in effect (e.g., `aws:username`, `aws:SourceIp`).

### Example of an IAM Policy

Let’s look at an example of an IAM policy that allows read access to a specific S3 bucket.

```json

{

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

"Statement": [

{

"Effect": "Allow",

"Action": [

"s3:GetObject",

"s3:ListBucket"

],

"Resource": [

"arn:aws:s3:::example-bucket",

"arn:aws:s3:::example-bucket/\*"

]

}

]

}

```

### Breakdown of the Example IAM Policy

- \*\*Version\*\*: Specifies the policy language version (`2012-10-17`).

- \*\*Statement\*\*: Contains one statement that allows specific actions on specified resources.

- \*\*Effect\*\*: `Allow` indicates that the actions are permitted.

- \*\*Action\*\*: Specifies two actions:

- `s3:GetObject`: Allows reading objects from the bucket.

- `s3:ListBucket`: Allows listing the objects in the bucket.

- \*\*Resource\*\*: Specifies the resources to which the actions apply:

- `arn:aws:s3:::example-bucket`: The S3 bucket itself.

- `arn:aws:s3:::example-bucket/\*`: The objects within the S3 bucket.

### Detailed Example with Conditions

Here’s a more detailed example that includes conditions:

```json

{

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

"Statement": [

{

"Effect": "Allow",

"Action": "s3:PutObject",

"Resource": "arn:aws:s3:::example-bucket/\*",

"Condition": {

"StringEquals": {

"s3:x-amz-server-side-encryption": "AES256"

}

}

}

]

}

```

### Breakdown of the Example with Conditions

- \*\*Version\*\*: Specifies the policy language version (`2012-10-17`).

- \*\*Statement\*\*: Contains one statement that allows a specific action on specified resources, under certain conditions.

- \*\*Effect\*\*: `Allow` indicates that the action is permitted.

- \*\*Action\*\*: `s3:PutObject` allows writing objects to the bucket.

- \*\*Resource\*\*: Specifies the objects within the S3 bucket (`arn:aws:s3:::example-bucket/\*`).

- \*\*Condition\*\*: Specifies a condition that must be met for the statement to apply:

- \*\*StringEquals\*\*: Condition operator indicating an exact match.

- `"s3:x-amz-server-side-encryption": "AES256"`: The request must specify that server-side encryption using AWS-managed keys (SSE-S3) is used when uploading objects.

### Important Points

1. \*\*Least Privilege Principle\*\*: Always grant the minimum necessary permissions required for the user or role to perform their tasks. Avoid using wildcards like `\*` which can grant overly broad access.

2. \*\*Explicit Deny\*\*: Deny statements take precedence over allow statements. If a request is explicitly denied, it cannot be allowed by another statement or policy.

3. \*\*Policy Limits\*\*: There are limits on the number of policies you can attach to an IAM identity or resource, and there are also size limits for policy documents.

4. \*\*Testing Policies\*\*: Test policies using AWS IAM Policy Simulator to verify their effects before applying them in a production environment.

5. \*\*Versioning\*\*: Use the `Version` field to specify the policy language version for backward compatibility and preventing issues with syntax changes in future versions.

6. \*\*Condition Keys\*\*: Leverage condition keys to add additional constraints to policies, such as requiring requests to come from specific IP ranges or using certain encryption keys.

### Real-time Use Case: Managing Access to a Restricted S3 Bucket

\*\*Use Case\*\*:

You need to create an IAM policy that restricts access to an S3 bucket to only authorized personnel. Additionally, you want to ensure that data uploaded to the bucket is encrypted using AWS Key Management Service (KMS).

### Steps and Important Points for Implementing the Use Case

#### Step 1: Define the IAM Policy

\*\*Policy JSON\*\*:

```json

{

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

"Statement": [

{

"Effect": "Allow",

"Action": [

"s3:ListBucket",

"s3:GetObject",

"s3:PutObject"

],

"Resource": [

"arn:aws:s3:::restricted-bucket",

"arn:aws:s3:::restricted-bucket/\*"

],

"Condition": {

"StringEquals": {

"aws:username": [

"john.doe",

"jane.doe"

]

},

"StringEquals": {

"s3:x-amz-server-side-encryption": "aws:kms"

}

}

}

]

}

```

\*\*Breakdown\*\*:

- \*\*Effect\*\*: `Allow` specifies that the actions are permitted.

- \*\*Action\*\*: Allows `s3:ListBucket`, `s3:GetObject`, and `s3:PutObject` actions.

- \*\*Resource\*\*: Applies to the bucket and its contents.

- \*\*Condition\*\*: Conditions applied to the policy:

- \*\*`aws:username`\*\*: The action is allowed only for specified users (`john.doe` and `jane.doe`).

- \*\*`s3:x-amz-server-side-encryption`\*\*: Requires server-side encryption with KMS for the `PutObject` action.

#### Step 2: Attach the Policy to Users

\*\*Java Code to Attach the Policy\*\*:

```java

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;

import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;

import com.amazonaws.services.identitymanagement.model.AttachUserPolicyRequest;

public class AttachPolicyToUser {

private static final String POLICY\_ARN = "arn:aws:iam::aws:policy/YourPolicyArn";

private static final String[] USERS = { "john.doe", "jane.doe" };

public static void main(String[] args) {

final AmazonIdentityManagement iam = AmazonIdentityManagementClientBuilder.standard()

.withCredentials(new ProfileCredentialsProvider())

.withRegion("us-west-2")

.build();

for (String user : USERS) {

AttachUserPolicyRequest attachRequest = new AttachUserPolicyRequest()

.withPolicyArn(POLICY\_ARN)

.withUserName(user);

iam.attachUserPolicy(attachRequest);

System.out.println("Attached policy to user: " + user);

}

}

}

```

### Conclusion

Understanding the structure and fields of IAM policies is fundamental to managing AWS permissions effectively. This guide provided a detailed explanation of policy components and fields, along with a real-time use case for managing access to an S3 bucket. You learned how to create a policy with specific conditions and attach it to users using Java code. Applying these concepts ensures secure and granular access control within your AWS environment.

AWS Identity and Access Management (IAM) is a vast and critical area for securing and managing AWS resources. Below are some additional concepts, best practices, and advanced features you may find useful to deepen your understanding of IAM.

### Additional Concepts

1. \*\*Service-Control Policies (SCPs):\*\*

- Used within AWS Organizations to manage permissions across accounts. SCPs enable central control over the maximum available permissions for accounts in your organization.

- SCPs do not grant permissions but serve as a filter, limiting what permissions can be used by IAM principals (users/roles) within the accounts.

2. \*\*IAM Roles for Cross-Account Access:\*\*

- IAM roles can be used to grant permissions to entities in a different AWS account. This is useful for scenarios where organizations need to provide access to resources or services across multiple AWS accounts.

- It typically involves setting up a trust relationship between the accounts.

3. \*\*Temporary Security Credentials:\*\*

- Temporary (short-term) security credentials provide permissions for a limited time and reduce the risk associated with long-lived credentials.

- These credentials are generally obtained using AWS Security Token Service (STS) and are commonly used for IAM roles in Amazon EC2 and federated users.

4. \*\*Permissions Boundaries:\*\*

- Permissions boundaries are policies that define the maximum permissions an IAM entity (user or role) can have. They are particularly useful for delegation tasks, ensuring that delegated accounts remain within specified permissions limits.

5. \*\*AWS Identity Center (formerly AWS Single Sign-On):\*\*

- AWS Identity Center simplifies the management of user identities at scale across multiple AWS accounts and applications.

- Provides Single Sign-On (SSO) capabilities, enabling users to authenticate once and gain access to multiple accounts and environments.

### Best Practices

1. \*\*Use Roles for EC2 Instances:\*\*

- For applications running on EC2 instances, use IAM roles to grant permissions instead of embedding long-term credentials. This approach is more secure and manageable.

2. \*\*Enable MFA Where Possible:\*\*

- Enable Multi-Factor Authentication (MFA) for privileged IAM users and roles to provide an additional layer of security.

3. \*\*Rotate Access Keys Regularly:\*\*

- Regularly rotate access keys for IAM users to reduce the risk of compromised credentials.

4. \*\*AWS Managed Policies:\*\*

- Prefer using AWS managed policies for standard permissions. These policies are maintained and updated by AWS and provide a good starting point for many use cases.

5. \*\*Grant Least Privilege:\*\*

- Follow the principle of least privilege by granting only the permissions necessary for users and roles to perform their tasks. Avoid broad permissions.

6. \*\*Review Permissions Regularly:\*\*

- Regularly review and audit IAM policies and permissions to ensure compliance and security.

### Advanced Features

1. \*\*Access Analyzer:\*\*

- AWS IAM Access Analyzer helps to identify resources in your account that are shared with external entities, providing insights to reduce security risks due to unintended exposure.

2. \*\*Policy Simulator:\*\*

- Use the IAM Policy Simulator to test and validate the effects of IAM policies before deploying them in your environment. This simulator can help identify policy issues and understand the permissions granted by policies.

3. \*\*Resource-Level Permissions:\*\*

- Define permissions at the resource level to grant or restrict access to specific resources. This granularity helps in creating more precise and secure policies.

- Example: Allowing `s3:GetObject` only for objects in a specific S3 bucket.

4. \*\*Tag-Based Access Control:\*\*

- Implement tag-based access control by attaching metadata tags to AWS resources and defining policies based on these tags. This method simplifies permissions management in large environments.

5. \*\*AWS Organizations:\*\*

- AWS Organizations allows you to manage multiple AWS accounts centrally. Use service control policies (SCPs) for establishing permission guardrails and managing compliance across your organization.

### Example Use Case: Cross-Account Access with IAM Roles

#### Scenario:

Enable an application in Account A to access resources (e.g., S3 bucket) in Account B using IAM roles.

\*\*Step 1: Configure Trust Policy in Account B\*\*

1. Define a role in Account B that allows access to the necessary resources. Create a trust policy that allows an entity in Account A to assume the role.

\*\*Trust Policy Example\*\*:

```json

{

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

"Statement": [

{

"Effect": "Allow",

"Principal": {

"AWS": "arn:aws:iam::AccountA-ID:role/RoleNameInAccountA"

},

"Action": "sts:AssumeRole"

}

]

}

```

\*\*Step 2: Define IAM Role with Permissions in Account B\*\*

```java

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.identitymanagement.AmazonIdentityManagement;

import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClientBuilder;

import com.amazonaws.services.identitymanagement.model.\*;

public class CrossAccountRole {

private static final String ROLE\_NAME = "CrossAccountS3AccessRole";

private static final String POLICY\_NAME = "S3ReadOnlyAccessPolicy";

public static void main(String[] args) {

final AmazonIdentityManagement iam = AmazonIdentityManagementClientBuilder.standard()

.withCredentials(new ProfileCredentialsProvider())

.withRegion("us-west-2")

.build();

String trustPolicy = "{ \"Version\": \"2012-10-17\", \"Statement\": [ { \"Effect\": \"Allow\", " +

"\"Principal\": { \"AWS\": \"arn:aws:iam::AccountA-ID:role/RoleNameInAccountA\" }, \"Action\": \"sts:AssumeRole\" } ] }";

CreateRoleRequest createRoleRequest = new CreateRoleRequest()

.withRoleName(ROLE\_NAME)

.withAssumeRolePolicyDocument(trustPolicy);

CreateRoleResult createRoleResult = iam.createRole(createRoleRequest);

String roleArn = createRoleResult.getRole().getArn();

System.out.println("Created Role Arn: " + roleArn);

String policyDocument = "{ \"Version\": \"2012-10-17\", \"Statement\": [ { \"Effect\": \"Allow\", " +

"\"Action\": \"s3:ListBucket\", \"Resource\": \"arn:aws:s3:::example-bucket\" } ] }";

CreatePolicyRequest createPolicyRequest = new CreatePolicyRequest()

.withPolicyName(POLICY\_NAME)

.withPolicyDocument(policyDocument);

CreatePolicyResult createPolicyResult = iam.createPolicy(createPolicyRequest);

String policyArn = createPolicyResult.getPolicy().getArn();

System.out.println("Created Policy Arn: " + policyArn);

AttachRolePolicyRequest attachRolePolicyRequest = new AttachRolePolicyRequest()

.withRoleName(ROLE\_NAME)

.withPolicyArn(policyArn);

iam.attachRolePolicy(attachRolePolicyRequest);

System.out.println("Attached policy to role.");

}

}

```

\*\*Step 3: Assume Role in Account A and Access Resources in Account B\*\*

\*\*Java Code to Assume Role and Access S3 in Another Account\*\*:

```java

import com.amazonaws.auth.AWSStaticCredentialsProvider;

import com.amazonaws.auth.AWSCredentialsProvider;

import com.amazonaws.auth.profile.ProfileCredentialsProvider;

import com.amazonaws.services.securitytoken.AWSSecurityTokenService;

import com.amazonaws.services.securitytoken.AWSSecurityTokenServiceClientBuilder;

import com.amazonaws.services.securitytoken.model.AssumeRoleRequest;

import com.amazonaws.services.securitytoken.model.AssumeRoleResult;

import com.amazonaws.services.s3.AmazonS3;

import com.amazonaws.services.s3.AmazonS3ClientBuilder;

public class CrossAccountAccess {

public static void main(String[] args) {

AWSCredentialsProvider credentialsProvider = new ProfileCredentialsProvider();

AWSSecurityTokenService stsClient = AWSSecurityTokenServiceClientBuilder.standard()

.withCredentials(new AWSStaticCredentialsProvider(credentialsProvider.getCredentials()))

.withRegion("us-west-2")

.build();

AssumeRoleRequest roleRequest = new AssumeRoleRequest()

.withRoleArn("arn:aws:iam::AccountB-ID:role/CrossAccountS3AccessRole")

.withRoleSessionName("cross-account-session");

AssumeRoleResult roleResult = stsClient.assumeRole(roleRequest);

AWSCredentialsProvider assumeRoleCredentials = new AWSStaticCredentialsProvider(

new com.amazonaws.auth.BasicSessionCredentials(

roleResult.getCredentials().getAccessKeyId(),

roleResult.getCredentials().getSecretAccessKey(),

roleResult.getCredentials().getSessionToken()

)

);

AmazonS3 s3Client = AmazonS3ClientBuilder.standard()

.withCredentials(assumeRoleCredentials)

.withRegion("us-west-2")

.build();

// Access resources in Account B

s3Client.listObjects("example-bucket").getObjectSummaries().forEach(s -> System.out.println(s.getKey()));

}

}

```

### Conclusion

AWS IAM is an essential service for managing access and permissions in AWS. Understanding its core concepts, best practices, and advanced features is crucial for maintaining a secure and compliant AWS environment. By leveraging IAM policies, roles, permissions boundaries, and other features effectively, you can achieve fine-grained access control and secure your AWS resources efficiently. This comprehensive overview and examples should provide a solid foundation for using IAM in real-world scenarios.