**Quick notes on AWS Identity and Access Management (IAM)**



[AWS Identity and Access Management (IAM)](https://aws.amazon.com/iam/#:~:text=AWS%20Identity%20and%20Access%20Management%20(IAM)%20enables%20you%20to%20manage,offered%20at%20no%20additional%20charge.) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources.

If we breakdown the term Identity and Access Management,

**Identity**— stands for Authentication, and

**Access** — stands for Authorization.

In AWS, an API call is authenticated by signing the requests in [HMAC](https://en.wikipedia.org/wiki/HMAC) signature with the secret key.

When we talk about authorization in AWS, IAM policies comes into picture.

**Components of IAM**

1. Users
2. Groups
3. Roles
4. Policies

**Users —**Using IAM, we can create and manage AWS users and use permissions to allow and deny their access to AWS resources.

**Groups —** The users created can also be divided into groups and then the rules and policies that apply on the group will also apply on the user level as well.

**Roles —** An IAM role is an IAM entity that defines a set of permissions for making AWS service requests. Trusted entities such as IAM users, applications or AWS services like EC2, Lambda etc. assumes these roles to carry out the task on our behalf.

**Policies —** We create policies to assign permission to a user, group, role or resource. It is a document that explicitly lists the permissions.

IAM role has two main parts — **Permission Policy** and**Trust Policy**. These policies are **JSON objects**. **Permission Policy** describes the permission of the role and **Trust Policy** describes who can assume that role.

Once, the IAM Role is assumed by an allowed entity, **AWS STS**(Security Token service) provides temporary security credentials to the entity, which contain the following-

* Session Token
* Expiration
* Access key
* Secret access key

When an IAM user from a different AWS account assumes an IAM Role of another account, then his credentials of the trusted account (i.e. the permissions that he was having before assuming this new role) will get replaced by the temporary credentials from STS.

Any entity which makes API call to access AWS resources should have an identity. That’s why, IAM users are provided with **long-term security credentials**. Sometimes, AWs services like EC2, lambda etc. makes those API calls on our behalf. For that, they also need an identity. So, these services are provided with **short-term security credentials**.

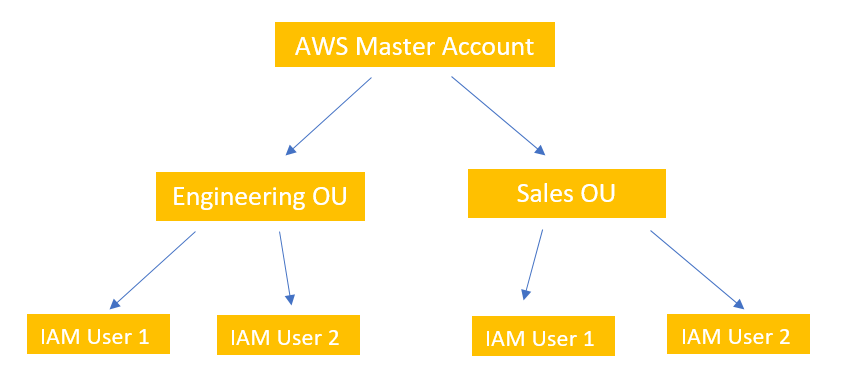
**Main policy types**

1. Service controlled policies (SCPs)
2. Identity based policies
3. Resource based policies

***Service control policies (SCPs)***

These are a type of organization policy that we can use to manage permissions in our organization. SCPs offer central control over the maximum available permissions for all accounts in our organization. SCPs help us to ensure our accounts stay within our organization’s access control guidelines. An SCP defines a **guardrail**, or sets limits, on the actions that the account’s administrator can delegate to the IAM users and roles in the targeted accounts. The administrator must still attach identity-based or resource-based policies to IAM users or roles, or to the resources in our accounts to actually grant permissions.

Let’s understand SCP with an example. Suppose there is a hypothetical organisation which has two departments — Engineering and Sales. So, there is a master AWS account for this organisation which has two Organisational Units (OUs) with the name of Engineering and Sales. Each OU has a root AWS account with two IAM users.



Now, if the admin of the master account attaches an SCP with the OU named Sales that denies EC2 access, then the root account of this OU won’t be able to access EC2.

***Identity-based policies***

These are attached to an IAM user, group, or role. These policies let us specify what that identity can do (its permissions). For example, we can attach the policy to the IAM user named Ram, stating that he is allowed to perform the Amazon DynamoDB getItem action.

A policy that is attached to an identity in IAM is known as an *identity-based policy*. Identity-based policies can include**AWS managed policies**, **customer managed policies**, and **inline policies**. AWS managed policies are created and managed by AWS. We can use them, but we can’t manage them. An inline policy is one that we create and embed directly to an IAM group, user, or role. Inline policies can’t be reused on other identities or managed outside of the identity where it exists.

As a best practice, we should use customer managed policies instead of inline policies. It is also best to use customer managed policies instead of AWS managed policies. AWS managed policies usually provide broad administrative or read-only permissions. **For greatest security, grant least privilege, which is granting only the permissions required to perform specific job tasks.**

Now, let’s look at a sample IAM policy structure.

{  
 "statement": [{  
 "Effect":"\_\_\_\_\_\_\_",  
 "Principal":"\_\_\_\_\_\_\_",  
 "Action":"\_\_\_\_\_\_\_",  
 "Resource":"\_\_\_\_\_\_\_",  
 "Condition":{  
 "condition":{  
 "key":"value"  
 }  
 }  
 }]  
}

In the above JSON object, in **Effect**, we either put **allow** or **deny** depending on whether we want to allow certain actions or not.

**Principal** is an identity in IAM. It is somebody who can make API calls to access AWS resources. It is the thing that we may not see very often because we mostly attach policies to principals (eg. IAM users or IAM roles). So, we don’t have to specify it.

**Action** is the type of access that is allowed or denied.

**Resource** is the Amazon resource on which specified Action will act.

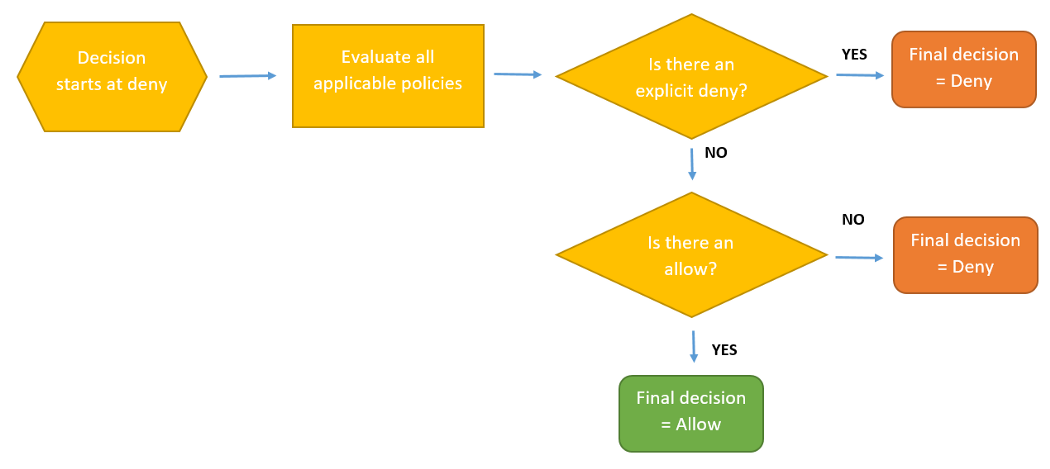
**Condition** defines the condition under the access defined is valid.

***Resource-based policies***

Withresource-based policies, we can specify who has access to the resource and what actions they can perform on it. These policies are attached to a resource and not to the principal.

**Policy Evaluation**

When AWS is evaluating whether an entity can access a resource, it follows the following logic.



At first, AWS assumes that all the resource access is denied by default. Then, it will evaluate all the applicable policies like Resource-based policies, IAM based policies or any other policies that is applicable to the resource in consideration. It will also evaluate the SCPs attached to the account. Then, it will check if there is any explicit deny. If so, then no matter how many ‘allow’ are there, the final decision will be ‘deny’.

Now, we’ll understand all the above mentioned concepts with some examples.

**Example 1**

Mathew is an IAM user with admin access permissions. He has created an EC2 instance and attached the following policy to it.

{  
 "Version":"2012-10-17",  
 "Statement":[  
 {  
 "Effect":"Allow",  
 "Action":"\*",  
 "Resource":"\*"  
 }  
 ]  
}

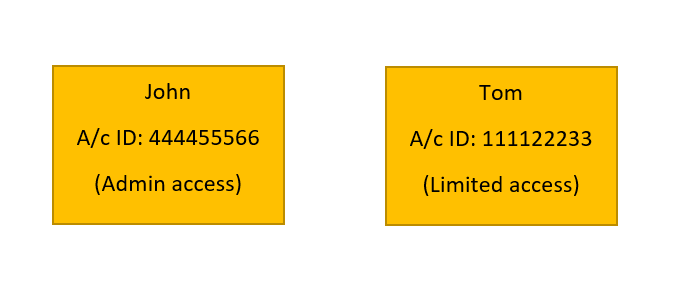
With the above policy, EC2 instance will get admin access permission.

Now, Mathew has to create a trust relationship policy document, which will define his trust over the EC2 instance. Here, EC2 instance will be the principal which will assume the role and perform the specified action on behalf of him.

{  
 "Version":"2012-10-17",  
 "Statement":[  
 {  
 "Effect":"Allow",  
 "Principal":{  
 "Service":"ec2.amazonaws.com"  
 }  
 "Action":"sts:AssumeRole"  
 }  
 ]  
}

**Example 2**

Suppose, there are two IAM users with name John and Tom, whose details are given below.



If John wants Tom to access his S3 bucket whose name is **myBucketJohn**, he can do it in two ways -

1. John will create a resource-based policy on **myBucketJohn**.

{  
 "Version":"2012-10-17",  
 "Statement":[  
 {  
 "Effect":"Allow",  
 "Principal":{  
 "AWS":"arn:aws:iam::111122233:root"  
 }  
 "Action":"S3:getObject"  
 "Resource":"arn:aws:s3:::myBucketJohn/\*"  
 }  
 ]  
}

This resource-based policy will allow Tom to read the objects stored in **myBucketJohn,** which is available in John’s account.

Note- All the other permissions given to Tom will remain as it is. With the above policy, he’ll get an extra permission to read objects stored in S3 bucket of John.

1. The second way is to create a cross account role for Tom by John. In this case, when the role is created for Tom, he’ll drop his existing permissions (called as Trusted account permission) and will assume new IAM cross account role that is created for him by John.

AWS Doc-links for IAM : <https://docs.aws.amazon.com/iam/index.html>

AWS CLI Command for IAM : <https://docs.aws.amazon.com/cli/latest/reference/iam/>