MODULE 3 · STUDY NOTES

# Securing Access to Cloud Resources

*How IAM authenticates and authorizes access to AWS resources — users, groups, roles, policies, federation, and multi-account control with AWS Organizations*

Securing access to cloud resources is squarely the customer's job: in the shared responsibility model, AWS secures the cloud itself, but the *platform, applications, and identity and access management* are yours to secure. The central tool is **AWS Identity and Access Management (IAM)**, a web service that controls who is **authenticated** (signed in) and who is **authorized** (has permissions). This module builds IAM from the ground up — the users, groups, roles, and policies you create; the credentials that authenticate them and the MFA that hardens them; the difference between identity-based and resource-based policies and the strict logic AWS uses to evaluate them; worked policy examples including cross-account access and permissions boundaries; the additional AWS services that handle federation and app sign-in (AWS SSO, AWS Directory Service, Amazon Cognito); and AWS Organizations, which extends control from a single account up to a whole fleet of accounts using service control policies. Throughout, one best practice recurs: grant the **least privilege** needed for the job, and add more only as required.

LEARNING OBJECTIVES

- Authorize access to AWS services by using IAM users, groups, and roles
- Differentiate between the types of security credentials in IAM
- Authorize access to AWS services by using identity-based and resource-based policies
- Identify other AWS services that provide authentication and access management services
- Centrally manage and enforce policies for multiple AWS accounts

## 3.1 IAM fundamentals

**AWS Identity and Access Management (IAM)** is a web service that helps you securely control access to AWS resources. You use it to control who is *authenticated* (signed in) and who is *authorized* (has permissions) to use those resources. IAM is integrated into most AWS services, so you can define access controls from one place — the AWS Management Console — and have them take effect throughout your environment. IAM supports **federated identity management** (trusting corporate systems such as Microsoft Active Directory and standards-based identity providers), **granular permissions**, **multi-factor authentication (MFA)**, and — through **AWS CloudTrail** — log records that tie each request back to the identity that made it, which supports information assurance. This module maps to the platform, applications, and IAM portion of the shared responsibility model that the customer is responsible to secure.

IAM does two distinct jobs. **Authentication** answers *who is requesting access?* — the requester, which IAM calls a **principal** (a person or application), must establish its identity through credentials. **Authorization** answers *what should they be allowed to do?* — once the requester is authenticated, IAM checks the policies relevant to the request to determine whether to allow or deny it. A banking analogy makes the split concrete: before a bank lets you into your account it *authenticates* that you are who you claim to be (user name, password, perhaps a code sent to your phone), and it *authorizes* you to spend the money in your own account but not in anyone else's. As the AWS account owner, you secure your account the same way.

KEY TERMS

**IAM user**
An entity you create to represent a person or application that interacts with AWS. A user has a *permanent* set of credentials until a forced rotation; on a new account, the root user is the first IAM user.

**IAM group**
A collection of IAM users. Use a group to grant the same set of permissions to multiple users at once.

**IAM role**
An AWS identity you attach permission policies to — like a user — but with *no* long-term credentials. A person, application, or service assumes it and receives temporary credentials for the session.

**IAM policy**
A document that explicitly lists permissions and defines which resources can be accessed and the level of access. It can attach to a user, group, role, or any combination.

You control access by creating users, groups, and roles and attaching policies. The best practice for long-term access is to **attach IAM policies to IAM groups and then assign users to those groups**: a user who is a member of a group inherits the group's permissions, and you can also attach a policy directly to a user to further customize the access granted through the group.

**TABLE 3.1** IAM terminology you will use throughout the course

| TERM | MEANING |
|---|---|
| **IAM entity** | The IAM objects AWS uses for authentication: users and roles. User entities can sign in to the console and make programmatic requests. |
| **IAM identity** | The IAM objects used to identify and group — users, groups, and roles. You can attach a policy to an IAM identity. |
| **IAM resource** | The user, group, role, policy, and identity provider objects stored in IAM. You can add, edit, and remove them. |
| **Principal** | A person or application that uses the root user, an IAM user, or an IAM role to sign in and make requests — includes assumed roles and federated users (external identities with no AWS account). |

A **request** is made any time a principal uses the console, API, or AWS CLI. AWS gathers all of the request's information into a **request context**, which it then uses to evaluate and authorize (or deny) the request.

WHAT A REQUEST CONTAINS

- **Actions or operations** — what the principal wants to perform.
- **Resources** — the object or objects the actions are performed on.
- **Principal** — the person or application (using a user or role) sending the request.
- **Environment data** — IP address, user agent, SSL-enabled status, or time of day.
- **Resource data** — data related to the resource being requested.

To connect to an AWS service you use its **endpoint** — the URL of the service's entry point. The SDKs and CLI use the default endpoint for each service in a Region; for example, `ec2.us-east-1.amazonaws.com` is the Amazon EC2 endpoint in the US East (N. Virginia) Region. You can specify alternate endpoints when configuration requires. You can

attach an **endpoint policy** to an endpoint to control access from that endpoint to the specified service, but it does *not* override or replace IAM user policies or service-specific policies. Unlike an IAM policy, an endpoint policy must contain a principal element; you can attach only one per endpoint (though you can modify it at any time), and it cannot exceed 20,480 characters.

## 3.2 Authenticating with IAM

An **IAM role** is an identity that defines a specific set of permissions to access the resources a user or service needs. As with a user, you attach permission policies to a role — but you do not attach the permissions to a user or group; instead the user or the service **assumes** the role as needed. When a user assumes a role, the user's prior permissions are *temporarily forgotten*, and AWS returns temporary security credentials — issued by the **AWS Security Token Service (AWS STS)** — that the user or application uses to make requests for the duration of the session. Roles mean you don't have to grant long-term credentials to every entity that needs access. The principal that assumes a role may even be a user, group, or role from another AWS account, including accounts you don't own.

### ROLE CHARACTERISTICS AND COMMON USE CASES

- Provides **temporary** security credentials, issued by AWS STS.
- Is **not** uniquely associated with one person.
- Can be assumed by a person, an application, or an AWS service.
- Is often used to **delegate** access.
- Common use cases: applications running on Amazon EC2, cross-account access for an IAM user, and mobile applications.

### WORKED EXAMPLE – A ROLE FOR AN EC2 INSTANCE

An administrator creates the `Get-pics` role in IAM, defines a policy that grants access to an Amazon S3 bucket named `photos`, and attaches the policy to the role. An EC2 instance then assumes the `Get-pics` role and is granted *temporary* permissions to access the `photos` bucket — no long-term credentials are ever stored on the instance.

When you interact with AWS, you must provide credentials, and the two primary types depend on *how* you access AWS. To authenticate from the **console**, you sign in with a user name and password. To authenticate **programmatically** through the AWS CLI, SDKs, and APIs, you provide an AWS **access key** — the combination of an access key ID and a secret access key. In most cases a human user signs in with the CLI while an automated program uses the SDK or API. You may also be required to provide additional security information, such as MFA.

**TABLE 3.2** Two types of credentials for authentication

| ACCESS METHOD | CREDENTIAL |
|---|---|
| **Console (interactive)** | User name and password |
| **Programmatic (CLI, SDKs, APIs)** | AWS access key = access key ID + secret access key |

**Multi-factor authentication (MFA)** is a best practice that adds an extra layer of protection on top of your user name and password. MFA requires two or more factors: something you *know* (a password), something you *have* (a cryptographic token), or something you *are* (a fingerprint). By default MFA is **not** activated. When it is enabled and a user signs in, they enter their user name and password (the first factor — what they know) and are then prompted for an authentication code from their AWS MFA device (the second factor — what they have).

**MFA AT A GLANCE**

- Activate MFA for **AWS Management Console** users and for **AWS API** users (API use requires temporary security credentials).
- Devices can be **hardware based** or **virtual**.
- Examples: security keys such as YubiKey and Gemalto; applications such as Google Authenticator and Authy.

**WORKED EXAMPLE – CROSS-ACCOUNT AUTHENTICATION WITH A ROLE**

An organization isolates its **production** and **development** accounts (each account is a separate container for resources). After testing an update in development, members of the Developers group need temporary access to update the `productionapp` S3 bucket. (1) An administrator in the production account creates a role and defines a **trust policy** that specifies the development account as a principal — authorizing development users to assume the `UpdateApp` role. (2) An administrator in the development account grants the Developers group permission to assume `UpdateApp` (and to call AWS STS). (3) A user requests to assume the role through the console, API, or CLI. (4) AWS STS verifies the request and returns temporary credentials. (5) Using those credentials, the user updates the `productionapp` bucket in the production account.

## 3.3 Authorizing with IAM

When you grant permissions, follow the **principle of least privilege**: grant only the permissions needed to perform a task, and add more as needed. Starting with a minimal set and expanding is more secure than starting too permissive and trying to restrict later. When you write policies, determine exactly what your users need to do and allow only those tasks; for individual resources, identify precisely who is allowed access and grant only the minimum. For example, developers might be allowed to *create* EC2 instances in production but not to *stop* or *terminate* them.

By default, an authenticated IAM user, group, or role **can't access anything** in your account until you grant permissions. You grant permissions by creating a **policy** — an object that defines the permissions for the identity or resource it is associated with. Most policies are stored as **JSON** documents that define the **effect**, **actions**, **resources**, and optional **conditions** under which an entity can invoke API operations. Anything not explicitly allowed is denied. When a principal makes a request, AWS evaluates it against the applicable policies to determine allow or deny.

**TABLE 3.3** The six types of policies AWS supports

| POLICY TYPE | WHAT IT DOES |
|---|---|
| **Identity-based** | Attached to IAM identities (user, group, role); grants permissions to the identity. |
| **Resource-based** | Attached to a resource; grants permissions to the principal specified in the policy. |
| **Permissions boundary** | Sets the maximum permissions an identity-based policy can grant to an entity; does not grant permissions. |
| **SCP (Organizations)** | Sets the maximum permissions for member accounts of an organization or OU; can limit but cannot grant. |
| **ACL** | Controls which principals in *other* accounts can access a resource; the only policy type not written in JSON. |
| **Session policy** | Used with the CLI/API to create a temporary session for a role or federated user; limits, cannot grant. |

Two policy types grant permissions, and the difference is *where* they are applied. **Identity-based policies** are attached to an IAM user, group, or role and indicate what that identity can do (for example, granting a user access to a DynamoDB table). **Resource-based policies** are attached to a resource and indicate what a specified user or group may do with it (for example, granting access to an S3 bucket or granting cross-account access between two trusted accounts). One asks *what does this identity have access to?*; the other asks *who has access to this resource?*

| MANAGED POLICIES | INLINE POLICIES |
|---|---|
| *standalone and reusable — AWS recommends these* | *embedded and one-to-one* |
| ▪ Standalone, identity-based policies you attach to multiple users, groups, and roles<br>▪ **AWS managed** (created by AWS) or **customer managed** (created by you)<br>▪ Features: reusability, central change management, versioning and rollback, delegated permissions management<br>▪ Also provide permissions boundaries | ▪ Embedded in a single principal entity (user, group, or role)<br>▪ Each entity keeps its *own copy* — impossible to centrally manage<br>▪ Useful for a strict **1:1** relationship between policy and entity<br>▪ Can't be inadvertently attached to the wrong entity |

**EVALUATION LOGIC FOR IAM POLICIES**

- By default, **all requests are denied**.
- An explicit **Allow** overrides the default deny.
- An explicit **Deny** overrides any explicit Allow.
- **Order doesn't matter** — all applicable policies are evaluated; the result is always allow or deny.
- On a conflict (one policy allows, another denies), the **most restrictive** policy (the deny) is applied.

**COMMON PITFALL**

Don't confuse *where* a policy lives with *what* it can do. Identity-based and resource-based policies are the only ones that grant permissions; permissions boundaries, SCPs, and session policies only cap them. And never assume evaluation order matters — an explicit deny anywhere in the applicable policies wins, every time.

## 3.4 Examples of authorizing with IAM

The same policy grammar — effect, action, resource — expresses everything from a user managing their own credentials to one account trusting another. These worked examples show how the logic behaves in practice.

**IDENTITY-BASED POLICY – SELF-SERVICE CREDENTIALS**

Applied to a user, group, or role, this policy allows the entity to perform specific actions. In the example, the policy lets a user manage only their *own* sign-in password (IAM `LoginProfile` actions), their *own* access keys (`AccessKey` actions), and their *own* SSH public keys (`SSHPublicKey` actions). The actions use wildcards (`*`) to include a set of related actions conveniently, and the `Resource` key gets the user name dynamically with `${aws:username}`, so the policy applies to whoever it is attached to.

#### CROSS-ACCOUNT, RESOURCE-BASED POLICY

Account A creates a resource-based policy on its S3 bucket `DOC-EXAMPLE-BUCKET`. The policy grants Account B (account number `111122223333`) permission to perform any S3 API operation (`s3:*`) on that bucket. Notice the policy does **not** specify any IAM users, groups, or roles — it specifies Account B as the principal. To actually use the access, Account B must then create an IAM (identity-based) policy allowing one of its users to reach Account A's bucket. Both sides must permit the action.

#### ALLOW PLUS EXPLICIT DENY – PRECEDENCE IN ACTION

A single policy can hold two statements. The first (`Effect: Allow`) gives access to only specific resources — a DynamoDB table named `coursenotes`, an S3 bucket `course-notes-web`, and bucket `course-notes-mp3` and its objects. The second (`Effect: Deny`) uses `NotResource` to deny all DynamoDB and S3 actions on everything **except** those same resources. Even if another policy granted broader access, this explicit deny overrides it. In total, the policy allows those specific resources and explicitly denies every other DynamoDB table and S3 bucket in the account.

#### PERMISSIONS BOUNDARY – ALLOWED BY BOTH, OR NOT AT ALL

A permissions boundary is an advanced feature that uses a managed policy to set the maximum permissions an identity-based policy can grant. An entity can perform only the actions allowed by **both** its identity-based policies **and** its boundary. Suppose policy 1 (the boundary) allows managing only Amazon S3, CloudWatch, and EC2. If policy 2 is then attached to the same user and grants `iam:CreateUser`, the operation still **fails** — the boundary doesn't permit IAM. Only by adding IAM to the boundary would `iam:CreateUser` succeed. The effective permission is the **intersection** of the two.

## 3.5 Additional authentication and access management services

**Identity federation** is a system of trust between two parties to authenticate users and convey the information needed to authorize resource access. **Identity providers (IdPs)** are responsible for user authentication; **service providers (SPs)** — services or applications — are responsible for controlling access to resources. Through administrative agreement and configuration, the SP trusts the IdP to authenticate users and then grants them access to the requested resources. Two AWS services provide federation to AWS accounts and applications: **AWS Single Sign-On (AWS SSO)** and **IAM**. Use AWS SSO if you have a single centralized directory; use IAM if you have multiple directories or want attribute-based permissions.

| IDENTITY PROVIDER (IDP) | SERVICE PROVIDER (SP) |
|---|---|
| *proves who the user is*<br>▪ Responsible for **user authentication**<br>▪ Vouches for the user's identity to the service provider<br>▪ Examples: corporate Active Directory, social or standards-based IdPs | *controls what the user reaches*<br>▪ Responsible for **controlling access to resources**<br>▪ **Trusts** the IdP to authenticate users<br>▪ Grants authenticated users access to requested resources |

**TABLE 3.4** Additional identity and access management services

| SERVICE | WHAT IT PROVIDES |
|---|---|
| **AWS Single Sign-On (AWS SSO)** | Create or connect identities once and manage access centrally across accounts; a unified admin experience for fine-grained access and a user portal for all assigned accounts or cloud apps; can run parallel to or replace IAM access management; supports apps such as Microsoft 365 and Salesforce. |
| **AWS Directory Service** | AWS Managed Microsoft AD lets directory-aware workloads and AWS resources use managed Active Directory in the cloud; extend on-premises AD to AWS using existing user credentials, with AD single sign-on to AWS applications. |
| **Amazon Cognito** | Adds sign-up, sign-in, and access control to web and mobile apps; a secure identity store scaling to millions of users; supports enterprise and social IdPs (Apple, Google, Facebook, Amazon). |

Amazon Cognito relies on two components. **User pools** are directories that provide sign-up and sign-in for your app users, integrate with social IdPs, and support security features such as MFA and phone verification. **Identity pools** grant your users access to AWS services through temporary, limited-privilege AWS credentials. In the bank scenario, Cognito is the kind of service that could support a planned mobile app for members while IAM keeps internal access tightly controlled.

> COMMON PITFALL
>
> AWS SSO and IAM both provide federation, but the choice runs opposite to intuition: a **single centralized directory** points to AWS SSO, while **multiple directories** or a need for **attribute-based permissions** points to IAM. Cognito is for app sign-in (web/mobile end users), not for administering access to your AWS accounts.

## 3.6 Using AWS Organizations

**AWS Organizations** is an account management service you use to consolidate multiple AWS accounts into a single, centrally managed organization. It provides centralized account creation and management plus consolidated billing, so you can handle security, compliance, and budget needs more efficiently. Organizations lets you group accounts hierarchically into **organizational units (OUs)** and attach different policies to each; you can nest OUs up to five levels deep.

> WHAT AWS ORGANIZATIONS PROVIDES
>
> - Consolidate and centrally manage multiple AWS accounts.
> - Account creation and management plus **consolidated billing**.
> - **Hierarchical grouping** of accounts into OUs (nested up to five levels).
> - Centralized policy control over services and API actions using **service control policies (SCPs)**.
> - Integration with IAM and other services.

A key feature is the **service control policy (SCP)**, which specifies the *maximum* permissions for member accounts. SCPs help ensure accounts stay within your access-control guidelines, but **they cannot grant any permissions** — they can only restrict access to a service, resource, or API action for a member account, user, or role. Any restriction an SCP puts in place remains even if the action is implicitly allowed elsewhere. Organizations builds on IAM by expanding control to

the account level: a user can access an operation only if **both** the Organizations policies **and** the IAM policies allow it — if either service blocks the operation, it is denied.

**SCP – PREVENT ACCOUNTS FROM LEAVING THE ORGANIZATION**

This SCP has one statement with `Effect: Deny`, `Action: organizations:LeaveOrganization`, and `Resource: *`. Its effect is to explicitly deny the leave-organization action, so member accounts cannot remove themselves from the organization — a restriction that holds regardless of what any member account's own IAM policies permit.

**COMMON PITFALL**

An SCP is a ceiling, not a key. It never grants access; it only caps what member accounts can do. Even if an IAM policy in a member account allows an action, an SCP that denies it wins — and an SCP that allows a service still leaves the member account needing its own IAM policy to actually use it.

## Module summary

- In the shared responsibility model the customer secures the platform, applications, and IAM; IAM controls both authentication (who is signed in) and authorization (what they may do), and calls any requesting person or application a principal.
- IAM's building blocks: users (permanent credentials, one person/app), groups (identical permissions for many users), roles (temporary permissions that are assumed), and policies (JSON documents defining access). Best practice: attach policies to groups, then add users.
- Roles have no long-term credentials — a user, application, or service assumes a role and gets temporary credentials from AWS STS; assuming a role sets aside the assumer's own permissions. Roles delegate access for EC2 apps, cross-account access, and mobile apps.
- Two credential types authenticate you: user name and password for the console, and an access key (access key ID + secret access key) for programmatic access. MFA — off by default — adds a second factor (something you have) and is a best practice.
- Grant least privilege. By default everything is denied until a policy grants access; policies define effect, actions, resources, and optional conditions in JSON (except ACLs).
- AWS supports six policy types; only identity-based and resource-based policies grant permissions. Permissions boundaries, SCPs, and session policies only set a maximum. Managed policies are reusable and recommended; inline policies keep a strict 1:1 relationship.
- Policy evaluation: default deny → an explicit allow overrides it → an explicit deny overrides any allow. Order never matters, and the most restrictive result stands.
- Federation is a trust between an identity provider (authenticates) and a service provider (controls access). AWS SSO suits a single centralized directory; IAM suits multiple directories or attribute-based permissions; AWS Directory Service extends on-premises Active Directory; Amazon Cognito adds sign-in to web/mobile apps via user pools and identity pools.
- AWS Organizations consolidates and centrally manages many accounts with OUs, consolidated billing, and SCPs; an action is allowed only when both Organizations and IAM permit it, and SCPs can restrict but never grant.

## Review questions

**1. What is the difference between authentication and authorization in IAM?**

*Answer:* Authentication establishes who the requester (a principal) is, through credentials; authorization determines what an authenticated principal is allowed to do, by evaluating the applicable policies to allow or deny the request.

**2. How does an IAM role differ from an IAM user, and what issues the credentials when a role is assumed?**

*Answer:* A user has permanent credentials and represents one person or application; a role has no long-term credentials and is assumed by a user, application, or service, which then receives temporary security credentials issued by AWS STS. Assuming a role temporarily sets aside the assumer's own permissions.

**3. A resource-based policy on an S3 bucket in Account A names Account B as principal. Can a user in Account B use the bucket immediately?**

*Answer:* Not yet — the resource-based policy grants access to Account B, but Account B must also create an identity-based (IAM) policy allowing its user to access Account A's bucket. Both sides must permit the action.

**4. Walk through how AWS decides whether to allow a request, and explain why policy order doesn't matter.**

*Answer:* All requests are denied by default; an explicit allow overrides that default deny; an explicit deny overrides any allow. AWS evaluates all applicable policies and the result is always allow or deny, so the order of evaluation has no effect — on a conflict, the most restrictive (deny) wins. Only identity-based and resource-based policies grant; permissions boundaries, SCPs, and session policies only cap.

**5. You have one centralized corporate directory and want single sign-on across your AWS accounts. Which federation service fits, and when would you choose IAM instead?**

*Answer:* Use AWS Single Sign-On (AWS SSO) for a single centralized directory. Choose IAM for federation when you have multiple directories or want attribute-based permissions.

**6. How do AWS Organizations SCPs and IAM policies interact when deciding whether an action is allowed?**

*Answer:* Organizations extends control to the account level: an action is allowed only if both the SCPs and the IAM policies permit it. SCPs set a maximum and can only restrict — never grant — so if either the SCP or IAM blocks the action, it is denied.