MODULE 3 · STUDY NOTES

# Securing Access

*The security principles behind an architecture, how IAM authenticates and authorizes access with users, groups, roles, and policies, and how AWS evaluates the permissions in an IAM policy*

Every AWS architecture rests on a question with two halves: who is allowed in, and what may they do once inside. This module is the architect's answer. It opens with the **security principles** that frame the decisions — the **shared responsibility model** (AWS secures the cloud; you secure what you put in it), the **security pillar** of the Well-Architected Framework, and the **principle of least privilege**. It then turns to **AWS Identity and Access Management (IAM)**, the service that both **authenticates** requesters (establishing who they are) and **authorizes** their requests (deciding what they may do) using **users, groups, roles, and policies**. From there it examines how AWS **evaluates** the permissions in those policies — a default deny that an explicit allow can override, and an explicit deny that always wins — and dissects the **parts of an IAM policy** itself. Throughout, take the architect's stance: work backward from the business need, apply security at every layer, and minimize the risk of unwanted access.

LEARNING OBJECTIVES

- Describe the security principles in the AWS Cloud
- Explain the purpose of AWS Identity and Access Management (IAM) users, groups, and roles
- Explain how IAM policies determine permissions in an AWS account

## 3.1 Security principles

As a cloud architect, you apply security best practices when you grant access to resources so that you minimize the risk of unwanted access at **each layer** of an architecture. This section reviews three principles that shape those decisions: consider the **shared responsibility model**, use the **security pillar** of the Well-Architected Framework to identify design principles, and always apply the **principle of least privilege**.

**Security and compliance are a shared responsibility between AWS and the customer.** AWS operates, manages, and controls security *of* the cloud — everything from the host operating system and virtualization layer down to the physical security of the facilities, plus the global infrastructure (Regions, Availability Zones, and edge locations) and foundation services (compute, storage, database, networking) that run AWS services. You assume responsibility *in* the cloud, and the steps you must take depend on the services you use and the complexity of your system — selecting and securing guest operating systems on EC2 instances, securing your applications, configuring security groups and firewalls, and managing your accounts, data, and encryption options.

| AWS – SECURITY OF THE CLOUD | CUSTOMER – SECURITY IN THE CLOUD |
|---|---|
| *the infrastructure that runs the services* | *what you put on that infrastructure* |
| ▪ Global infrastructure: Regions, Availability Zones, edge locations<br>▪ Foundation services: compute, storage, database, networking<br>▪ The host operating system and virtualization layer<br>▪ Physical security of the facilities | ▪ Your customer data<br>▪ Platform, applications, and identity and access management<br>▪ Guest operating system, network, and firewall configuration<br>▪ Client-side and server-side encryption, data integrity, and network traffic protection |

**Security is one of the six pillars of the Well-Architected Framework** (alongside operational excellence, reliability, performance efficiency, cost optimization, and sustainability). The framework provides best practices and design guidance across each pillar, and the **AWS Well-Architected Tool** gives you on-demand, self-service access to current best practices so you can review architectures against them. This module calls out the key aspects of two of the security pillar's design principles in particular: granting access, and protecting data at rest and in transit.

### THE SEVEN SECURITY-PILLAR DESIGN PRINCIPLES

- **Implement a strong identity foundation** — apply least privilege and separation of duties, centralize identity management, and reduce reliance on long-term static credentials.
- **Protect data in transit and at rest** — classify data by sensitivity, then use encryption, tokenization, and access control where appropriate.
- **Apply security at all layers** — defense in depth across the network edge, VPC, load balancing, every instance and compute service, the OS, the application, and code.
- **Keep people away from data** — use tools to reduce direct access and manual handling, lowering the risk of human error.
- **Maintain traceability** — monitor, alert, and audit changes in real time; integrate log and metric collection with automated response.
- **Prepare for security events** — keep incident-management processes, run response simulations, and automate detection and recovery.
- **Automate security best practices** — define and manage security controls as code in version-controlled templates.

**The principle of least privilege** is a key part of implementing a strong identity foundation. It means granting **only the permissions needed to perform a task**: start with a minimum set of permissions and grant additional permissions as necessary, rather than starting with permissions that are too permissive and trying to restrict them later. Research what access a task actually needs, then craft policies that allow exactly that — and revoke permissions that are no longer used. For example, developers might be allowed to create EC2 instances in production but not to stop or terminate them.

Beyond limiting *who* has access, **encryption** protects the data itself, addressing the design principle of protecting data in transit and at rest. **Data in transit** is data actively moving from one location to another — across the internet or a private network — and it is often considered less secure while in motion; a cryptographic protocol such as **TLS** protects it as it travels. **Data at rest** is data stored anywhere in your solution, and it is protected with encryption applied either by the client before sending (client-side) or by the server as it stores the data (server-side).

**TABLE 3.1** Three ways encryption protects data

| APPROACH | HOW IT WORKS | EXAMPLE |
|---|---|---|
| **In transit** | Protect data as it moves between networks or locations using a cryptographic protocol such as **TLS** | Encrypting pictures as they are uploaded to a cloud service |
| **At rest — client-side** | The **client** encrypts data before sending it and decrypts it after retrieving it — end-to-end protection, in transit and at rest | Encrypting data on a mobile device that could be lost or stolen |
| **At rest — server-side** | The **server** encrypts data as it writes it to disk and decrypts it when requested — for example, Amazon S3 server-side encryption at the object level | Encrypting a Social Security number stored in a cloud database |

> **COMMON PITFALL**
>
> Least privilege is a *start-minimal-then-add* practice, not *start-broad-then-restrict*. Beginning with a minimum set of permissions and granting more as needs are proven is more secure than opening access wide and trying to claw it back later.

## 3.2 Authenticating and securing access

Controlling access to your AWS resources involves two distinct questions. **Authentication** asks *who* is requesting access to the account and its resources: it establishes the identity of the requester — a person or an application — through credentials. **Authorization** asks, once that requester is authenticated, *what* they should be allowed to do: it determines whether to allow or deny the request. A banking analogy captures it: before the bank lets you into your account it **authenticates** you with a username, password, and perhaps an MFA code; once you are in, you are **authorized** to pay bills from your own money but not from another customer's account.

| AUTHENTICATION (AUTHN) | AUTHORIZATION (AUTHZ) |
|---|---|
| *who is requesting access?*<br>▪ Establishes the identity of the requester through credentials<br>▪ The requester can be a person or an application<br>▪ Analogy: proving you are the account holder before you are let in | *what are they allowed to do?*<br>▪ Happens only after the requester is authenticated<br>▪ Determines whether to allow or deny the specific request<br>▪ Analogy: you may spend your own money, but not someone else's |

**AWS Identity and Access Management (IAM)** is the service that configures fine-grained access control to AWS resources. It controls individual and group access, integrates into most AWS services (so you define access from one place — the console — and it takes effect throughout your environment), provides **federated identity management** (federation from corporate systems such as Microsoft Active Directory and standards-based identity providers), supports **multi-factor authentication (MFA)**, and allows granular permissions. IAM is **secure by default**: users have no access to resources until permissions are explicitly granted. When MFA is enabled, a signing-in user is prompted for an authentication code delivered to a hardware or virtual device — for example, Google Authenticator running on a smartphone.

KEY TERMS

**IAM resource**
The user, group, role, policy, or identity-provider objects that are stored in IAM

**IAM entity**
The IAM objects AWS uses for authentication — users and roles

**IAM identity**
The IAM objects that can be authorized in policies to perform actions and access resources — users, groups, and roles

**Principal**
A person or application that signs in and makes requests to AWS; includes assumed roles and federated users (external identities with no AWS account)

You control access by creating **users, groups, and roles** and attaching **policies** to them. An IAM **user** represents one person or application and is given a permanent set of credentials that stays until a forced rotation. An IAM **group** is a collection of users used to grant the same permissions to many people at once. An IAM **role** is similar to a user — an identity you attach permissions policies to — but it has **no long-term credentials**; when a person or application assumes the role, they receive temporary security credentials for the role session. An IAM **policy** is a document that explicitly lists permissions and can be attached to a user, group, role, or any combination.

**TABLE 3.2** The four IAM building blocks

| BUILDING BLOCK | WHAT IT IS | CREDENTIALS / NOTE |
|---|---|---|
| **User** | An identity for one person or application that interacts with AWS | **Permanent** credentials until rotated |
| **Group** | A collection of IAM users granted identical permissions | No credentials; a management convenience |
| **Role** | An identity that grants a **temporary** set of permissions; assumed by a person, app, or service | **Temporary** session credentials; no long-term keys |
| **Policy** | A JSON document defining which resources can be accessed and the level of access | Attached to users, groups, roles, or resources |

The **type of credential** you use to authenticate depends on how you access AWS. You cannot use a username and password to authenticate through the CLI, SDKs, or direct API calls — those require an AWS access key, which is the combination of an access key ID and a secret access key.

**TABLE 3.3** IAM credentials for authentication

| ACCESS METHOD | CREDENTIALS NEEDED |
|---|---|
| **AWS Management Console** | Username and password |
| **AWS Command Line Interface (AWS CLI)** | An AWS access key (access key ID + secret access key) |
| **Programmatic calls (SDKs, direct API)** | An AWS access key (access key ID + secret access key) |

### BEST PRACTICES TO SECURE ACCESS

- **Follow least privilege** — give users, groups, and roles the minimum permissions their tasks require.
- **Enable MFA** — an extra sign-in factor (a code from a hardware or software token), especially for the root user.
- **Require temporary credentials for human users** — use IAM roles and **AWS IAM Identity Center** for federated access instead of long-term credentials.
- **Rotate access keys** — for use cases that still need long-term programmatic credentials, use last-used information to rotate and remove keys regularly.
- **Use strong, complex passwords** — for the root user and all IAM users, following AWS password-policy guidelines.
- **Secure local credentials** — store access keys, passwords, and MFA tokens in a secure password manager or hardware token.
- **Use AWS Organizations** — consolidate multiple accounts to manage billing, access control, and resources centrally.
- **Enable AWS CloudTrail** — keep a record of all account actions to identify risks and meet auditing requirements.
- **Protect the root user** — limit its use and monitor activity with CloudTrail logs, Trusted Advisor, and console sign-in events.

**Protecting the root user** deserves special attention. When you first create an account you get a single sign-in identity — the **root user** — that has complete access to all services and resources. Protect its credentials like a credit card number. For day-to-day work, create an **administrative user** rather than using the root user; if you create that admin user in **AWS IAM Identity Center** (successor to AWS Single Sign-On), it uses temporary rather than long-term credentials. Reserve the root user only for the few tasks that no other user can perform.

### WORKED EXAMPLE – SETTING UP AN ADMIN USER

AWS strongly recommends not using the root user for daily interactions. The setup steps: (1) log in as the **root user** and set up **MFA** on it; (2) create a new **admin user** (for example, John), add MFA, and download the programmatic keys — but do not use or store them on your system; (3) log out as the root user; (4) log in as the admin user; (5) as the admin user, create the individual user accounts (for example, John, Ji, and Nikki), each with a separate policy and permissions. After locking away the root credentials, the admin user handles the day-to-day administration.

For long-term access, a best practice is to **attach IAM policies to IAM groups and then assign IAM users to those groups.** A user who is a member of a group inherits the permissions attached to the group; you can also attach a policy directly to a user to further customize the access granted through the group. In a typical setup, a sales group and an IT group carry different policies, and each user placed in a group receives that group's permissions.

**IAM roles** provide temporary security credentials, are not uniquely associated with one person, can be assumed by a person, application, or service, and are often used to **delegate access**. When you use a role you avoid granting long-term credentials to every entity that needs access. When a user assumes a role, their prior permissions are temporarily forgotten and AWS returns temporary credentials for the session. Roles fit any case where you don't want to store or manage long-term credentials — an IAM user needing temporary cross-account access, federated users authenticated outside AWS, a mobile app that shouldn't ship with embedded credentials, or an application on an EC2 instance.

**TABLE 3.4** Three ways to use an IAM role

| USE CASE | HOW THE ROLE IS USED |
|---|---|
| **IAM user needs an EC2 app** | Create a policy and attach it to a role; the IAM user assumes the role to access the application running on the EC2 instance |
| **EC2 app needs an S3 bucket** | Create a role, add it to an **instance profile**, and attach the profile to the instance; the app assumes the role at runtime to reach the bucket |
| **Cross-account access** | Create a role in account A that defines account B as a **trusted entity**; an IAM user in account B switches to that role to access the resource in account A |

## 3.3 Authorizing users

A **policy** is an object that defines permissions for the identity or resource it is associated with; you use policies to fine-tune the permissions granted to **principals** (IAM users, IAM roles, and AWS services). Without policies, a principal would have no access to resources. There are **two types of policies**, and the biggest difference is *where* they are applied. **Identity-based policies** are attached to an IAM user, group, or role and indicate what that identity is permitted to do. **Resource-based policies** are attached to a resource and indicate what a specified user or group of users may do with that resource. Most policies are stored as JSON documents, and a best practice is to follow least privilege.

| IDENTITY-BASED POLICY | RESOURCE-BASED POLICY |
|---|---|
| *attached to an IAM user, group, or role* | *attached to an AWS resource* |
| ▪ Answers: what does this identity have access to?<br>▪ The **Principal** is implied — the identity the policy is attached to<br>▪ Example: grant a user the ability to access a DynamoDB table | ▪ Answers: who has access to this particular resource?<br>▪ Must name the **Principal** (account, user, role, or federated user)<br>▪ Example: grant cross-account access to an S3 bucket |

**Permissions are determined at the time of the request.** When IAM decides whether an action is allowed, it evaluates all applicable policies and applies a fixed logic. The order in which policies are evaluated has no effect on the outcome — every applicable policy is evaluated, and the result is always that the request is either allowed or denied. If a conflict occurs (one policy allows an action and another denies it), the most restrictive policy — the deny — is applied. The **IAM policy simulator** is a helpful tool for testing and troubleshooting policies before you rely on them.

THE EVALUATION RULES

- By default, **all requests are denied** (an implicit deny).
- An **explicit allow** overrides the default deny.
- An **explicit deny** overrides any explicit allow.
- All applicable policies are evaluated; the **order has no effect** on the result.
- On a conflict, the **most restrictive** policy wins.
- A request succeeds only if it is **explicitly allowed and not explicitly denied**.

**WORKED EXAMPLE – BOB AND BUCKET X (DENY WINS)**

Bob has an **identity-based** policy that allows the GET, PUT, and LIST actions on S3 bucket X. However, the **resource-based** policy on bucket X allows GET and LIST but **denies** PUT. Can Bob PUT objects into bucket X? **No.** The explicit deny in the resource-based policy overrides the allow in his identity-based policy — Bob can read and list, but cannot write, even though his own policy says he can.

**WORKED EXAMPLE – BOB AND BUCKET Y (ALLOW FROM EITHER SIDE)**

For bucket Y, Bob's **identity-based** policy allows only the LIST action and says nothing about GET or PUT. The **resource-based** policy on bucket Y allows GET and LIST and does not specify PUT. Can Bob GET (read) objects from bucket Y? **Yes.** The resource-based policy explicitly allows GET and nothing denies it, so Bob can read the objects — even though his identity-based policy does not explicitly allow GET. An allow from either policy is enough when there is no deny.

**COMMON PITFALL**

Watch the two flavors of deny. An **implicit deny** is the default that applies whenever no policy allows the action; an **explicit deny** is a deny statement that overrides even an explicit allow. Both end in the request being refused, but only the explicit deny beats an allow.

## 3.4 Parts of an IAM policy

IAM policies are stored in AWS as **JSON documents**, and each statement carries information about a single permission. If a policy includes multiple statements, AWS applies a **logical OR** across them when evaluating. A statement is built from a small set of elements — the ones that most often decide the outcome are the **Effect**, **Action**, and **Resource**.

**TABLE 3.5** Common elements of an IAM policy document

| ELEMENT | WHAT IT SPECIFIES |
|---|---|
| **Version** | The version of the policy language you want to use — for example, `2012-10-17` |
| **Statement** | The main container for the elements below; a policy can hold more than one statement |
| **Effect** | `Allow` or `Deny` — whether the statement allows or denies access |
| **Principal** | For a resource-based policy, the account, user, role, or federated user to allow or deny; for an identity-based policy the Principal is implied and omitted |
| **Action** | The list of actions the policy allows or denies — for example, `s3:GetObject` |
| **Resource** | The resource(s) the actions apply to, given by ARN (AWS Resource Name) |
| **Condition** | Optional — the circumstances under which the statement grants its permissions |

**A resource-based example.** A policy can combine an explicit allow and an explicit deny to fence access in. The module's example **allows** all DynamoDB and Amazon S3 actions on one named DynamoDB table and two named S3 buckets, then adds a **Deny** statement that uses the `NotResource` element to block those same services on every *other* resource — even if another policy granted access. Because an explicit deny always takes precedence, the net effect is access to exactly those named resources and nothing else.

**An identity-based example.** A policy attached to a user can grant **self-service** permissions — for instance, letting the user create, delete, get, and update their **own** login profile (password), access keys, and SSH public keys. The action

names use **wildcards** (the asterisk stands in for a set of related actions, a convenient way to include several at once), and the `Resource` element uses the dynamic variable `${aws:username}` so the policy automatically applies to whichever user it is attached to.

**A cross-account, resource-based example.** Sharing a resource across accounts takes a policy on each side. In the module's example, **account A** attaches a bucket policy that names **account B** (account number `111122223333`) as the `Principal` and allows all Amazon S3 actions on a bucket named `DOC-EXAMPLE-BUCKET`. That bucket policy alone is not enough — **account B must also create an identity-based policy** allowing its own user to use the bucket. Cross-account access is a resource-based policy on the owning side plus an identity-based policy on the accessing side.

> **WORKED EXAMPLE – READING THREE IAM POLICIES (MODULE ACTIVITY)**
>
> **(1) Read-only IAM:** a policy that allows `iam:Get*` and `iam:List*` on all resources grants read access to the IAM service only — the user cannot create a user, group, policy, or role, because those are neither Get nor List actions. **(2) Conditional terminate:** a policy that allows `ec2:TerminateInstances` but adds a Deny unless the caller's `aws:SourceIp` is in `192.0.2.0/24` or `203.0.113.0/24` lets the user terminate instances only from those IP ranges — an address of `192.0.2.243` falls inside `192.0.2.0/24`, so it is allowed. **(3) Instance-type guardrail:** a policy that denies `ec2:RunInstances` and `ec2:StartInstances` when the instance type is not `t2.micro` or `t2.small` allows nothing on its own (its only effect is a deny). Add a second statement allowing all EC2 actions, and the user gains full EC2 access but can still launch or start only `t2.micro` or `t2.small` instances — yet could terminate an `m3.xlarge`, because the deny restricts only running and starting.

> **COMMON PITFALL**
>
> The `Principal` element is required in a resource-based policy but must be omitted from an identity-based one. If you are attaching a policy to a user or role, the principal is already known (it is that identity); naming a principal there is an error.

## 3.5 Putting it into practice

The point of all of this is a concrete design habit: as the cloud architect, work backward from the business need and apply security best practices at **each layer** of the architecture, knowing the decisions evolve as needs are better understood. The module's guided lab, **Exploring AWS Identity and Access Management (IAM)**, puts the concepts to work — you experiment with IAM policies, add users to IAM groups, and use the IAM sign-in URL; you reach the service in the console by choosing **Services → Security, Identity, & Compliance → IAM**. The same thinking applies to any scenario, such as the course café: ask what access the staff actually need, group people by role with a least-privilege policy per group, use roles to give applications temporary credentials instead of embedded keys, require MFA, and keep the root user locked away.

> **MANAGED POLICIES (FROM THE LAB)**
>
> - A **managed policy** is a pre-built policy — created either by AWS or by your administrators — that can be attached to IAM users and groups.
> - When a managed policy is **updated**, the changes apply **immediately** to all users and groups it is attached to.
> - The lab examines a group with the AWS-managed **`AmazonEC2ReadOnlyAccess`** policy attached.

**WORKED EXAMPLE – THE SAMPLE EXAM QUESTION**

*A team of developers needs access to several services and resources in a VPC for 9 months. Team members are likely to change during the project. Which option follows key security principles?* The key phrases are **team of developers likely to change**, **access for 9 months**, and **follows key security principles**. The answer is to **create an IAM user for each developer, put them all in one IAM group, and attach the required policies to the group.** Attaching the policy to a group applies the same access to every member and automatically grants it to users added later and revokes it from users removed — ideal for changing membership. A single shared user (whether or not it is placed in a group) is a bad practice: the password is shared and you lose individual accountability. Giving each developer their own user with policies attached directly works, but is less efficient to manage as the team changes.

## Module summary

- Security is a **shared responsibility**: AWS secures the cloud (global infrastructure, foundation services, host/hypervisor, and facilities); the customer secures what runs in it (data, IAM configuration, guest OS and firewall/network settings, and encryption). What you must do depends on the services you use.
- The **security pillar** of the Well-Architected Framework supplies seven design principles; this module emphasizes a strong identity foundation (least privilege) and protecting data in transit (a cryptographic protocol such as TLS) and at rest (client-side and server-side encryption).
- **Least privilege**: grant only the permissions a task requires — start minimal, add as needed, and revoke what is unused.
- **IAM authenticates and authorizes.** Authentication establishes who is requesting (credentials for a person or app); authorization decides what they may do. IAM is secure by default — no access until it is explicitly granted.
- **Users, groups, roles, policies:** a user is an identity with long-term credentials; a group shares one policy set across many users; a role hands out temporary credentials and is assumed by people, apps, or services; a policy is the JSON document that defines the permissions. Best practice: attach policies to groups.
- **Credentials:** the console uses a username and password; the CLI, SDKs, and API calls require an AWS access key (access key ID + secret access key).
- **Protect the root user:** it has complete access — enable MFA, use it only for tasks nothing else can do, and create an admin user (in AWS IAM Identity Center) for daily work.
- **Two policy types, one evaluation logic, one anatomy:** identity-based policies attach to an identity, resource-based policies attach to a resource and must name a Principal. Requests are denied by default, an explicit allow overrides that default, and an explicit deny overrides any allow. Policies are JSON documents built from Version, Statement, Effect, Principal, Action, Resource, and Condition — with Effect, Action, and Resource doing most of the work.

## Review questions

**1. In the shared responsibility model, name one security task that belongs to AWS and one that belongs to the customer.**

*Answer:* AWS: security OF the cloud — the global infrastructure (Regions, Availability Zones, edge locations), foundation services, the host OS and virtualization layer, and physical facilities. Customer: security IN the cloud — customer data, IAM configuration, the guest OS and firewall/network settings, and encryption.

**2. Distinguish authentication from authorization, and say which of them IAM performs.**

*Answer:* Authentication establishes who is making a request (via credentials); authorization determines what that authenticated requester is allowed to do (allow or deny). IAM performs both.

**3. How do an IAM user and an IAM role differ in their credentials?**

*Answer:* A user has long-term credentials (a password and/or access keys) that stay until rotated; a role has no long-term credentials — assuming it returns temporary session credentials, and it can be assumed by a person, application, or service.

**4. Why is attaching a policy to a group preferred over attaching it to each individual user?**

*Answer:* Permissions are managed in one place and follow group membership — adding a user to the group grants the access and removing them revokes it, which is efficient when membership changes.

**5. State the rules AWS uses to decide whether a request is permitted.**

*Answer:* By default all requests are denied (implicit deny); an explicit allow overrides the default deny; an explicit deny overrides any explicit allow. The order of evaluation does not matter, and the most restrictive result wins — so a request succeeds only if it is explicitly allowed and never explicitly denied.

**6. What is the difference between an identity-based and a resource-based policy, and which one must specify a Principal?**

*Answer:* An identity-based policy attaches to an IAM user, group, or role and says what that identity can do; a resource-based policy attaches to a resource and says who may do what to it. The resource-based policy must name the Principal; in an identity-based policy the Principal is implied.

**7. List four of the seven elements of an IAM policy statement and what each does.**

*Answer:* Any four of: Version (policy language version), Statement (container for permissions), Effect (Allow or Deny), Principal (who — required for resource-based, implied for identity-based), Action (the API actions), Resource (the ARNs the actions apply to), Condition (optional circumstances that must be met).

**8. A team of developers whose members will change needs the same access for nine months. What is the best-practice design, and why not a shared user?**

*Answer:* Create an IAM user for each developer, put them all in one IAM group, and attach the policies to the group — new members inherit the access and removed members lose it automatically. A single shared user is a bad practice: the password is shared and you lose individual accountability.