MODULE 4 · STUDY NOTES

Securing Access to Cloud Resources

The shared responsibility model, and how IAM users, groups, roles, and policies authenticate and authorize access to AWS resources

Frank and Martha liked the café's proof-of-concept website, and now more team members are joining Sofía to develop the café's web application. Before development continues, Sofía decides to define the level of access that each user and system should have across the cloud resources the site uses — a developer role, a database administrator role, and so on. That decision is what this module is about. It begins with the **shared responsibility model**, which draws the line between what AWS secures and what the customer secures. It then introduces **AWS Identity and Access Management (IAM)**, the service that controls access to your AWS resources through users, groups, roles, and policies. From there the module separates the two halves of access control: **authentication** (establishing who is making a request) and **authorization** (deciding what that requester may do). Along the way it covers the credentials developers use, why IAM roles and temporary credentials are the safer choice for applications, how IAM policies are written and evaluated, and the principle of least privilege that should govern every grant.

LEARNING OBJECTIVES

- Describe the AWS shared responsibility model
- Explain how AWS Identity and Access Management (IAM) helps secure access to AWS resources
- Describe IAM user authentication
- Identify how to authorize access to AWS services by using IAM users, groups, and roles

4.1 The shared responsibility model

Ensuring security and compliance is a **shared responsibility** between AWS and the customer, so it is important to differentiate who is responsible for what. A useful summary: **AWS is responsible for security *of* the cloud, and the customer is responsible for security *in* the cloud.** The exact customer responsibilities depend on which AWS services a customer chooses to use, but customers are generally responsible for managing their data and for using IAM and other security services to configure the appropriate, implementation-specific security. The model has two objectives: it **reduces the customer's operational burden** by having AWS secure the physical infrastructure the customer might once have managed, and it **provides the flexibility and control** customers need to build custom applications and make practical use of cloud resources.

AWS – SECURITY OF THE CLOUD	CUSTOMER – SECURITY IN THE CLOUD
operates, manages, and controls the infrastructure	*responsible for what you implement and connect*
▪ Protects the global infrastructure that runs all AWS services — **Regions, Availability Zones, and edge locations** ▪ Secures the host operating system and the virtualization (hypervisor) layer ▪ Provides the physical security of the facilities where the services operate ▪ Covers software, compute, storage, databases, networking, and hardware	▪ Customer data; platform, applications, and identity and access management ▪ Guest OS, network, and firewall configuration — including updates and security patches ▪ Selecting and securing the OS and applications that run on EC2; security group configurations ▪ Client-side and server-side encryption, data integrity, traffic protection, and secure account management

In short, AWS secures the hardware, software, facilities, and networks that run all AWS products and services. You are responsible for what you implement by using those products and services and for the applications you connect to AWS. The security steps you must take depend on the services you use and the complexity of your system.

WORKED EXAMPLE – AN S3, EC2, AND RDS WORKLOAD

Suppose you store company data in **Amazon S3** and also run an **Amazon EC2** instance and an **Amazon RDS** instance; the RDS instance runs a MySQL database inside a VPC, and the EC2 instance hosts a web server whose application stores data in that database. In this scenario **AWS** protects the global infrastructure — the physical servers and storage hardware hosting your bucket, instances, and database; the physical networking that lets those components be reached; and the hypervisor layer that hosts the EC2 instances. **You (the customer)** manage the guest OS on the EC2 instances (including Windows or Linux updates and patches) and any application software you install; you configure the security groups that control network access to the EC2 instance and the RDS database; and you secure the S3 bucket and its objects — for example with bucket policies, data encryption, and S3 Block Public Access.

COMMON PITFALL

The tempting mistake is to assume that because AWS runs the infrastructure, AWS also secures what you put on it. AWS secures *of* the cloud; you secure *in* the cloud. Guest OS patching, application security, firewall and security group configuration, IAM, and the protection of your data are all the customer's responsibility.

4.2 Introducing AWS Identity and Access Management (IAM)

AWS Identity and Access Management (IAM) is the service you use to configure fine-grained access control to AWS resources. With IAM you follow security best practices by granting unique security credentials to users, groups, and roles — credentials that specify which AWS service APIs and resources those identities can access. A crucial default: **users have no access to AWS resources until permissions are explicitly granted.** IAM is integrated into most AWS services, so you can define access controls in one place, the AWS Management Console, and have them take effect throughout your environment.

WHAT IAM PROVIDES

- Securely controls individual and group access to your AWS resources
- Integrates with most other AWS services
- Supports **federated identity management** — federation from corporate systems such as Microsoft Active Directory and standards-based identity providers
- Supports **granular permissions** — fine-grained control over who can do what
- Supports **multi-factor authentication (MFA)** — an authentication code from a hardware or virtual device on top of the password

IAM cleanly separates the two fundamentals of access control. **Authentication** asks *who is requesting access?* — it establishes the identity of the requester through credentials. The requester may be a person or an application, and IAM calls either one a **principal**. **Authorization** assumes the requester has been authenticated and asks *what should they be allowed to do?* — IAM checks the policies relevant to the request to determine whether to allow or deny it. A banking analogy captures the split: before the bank lets you into your account it authenticates you with a user name and password (and perhaps a code sent to your phone); once you are in, authorization is what stops you from paying your bills with another customer's money.

You control access to your resources by creating **users, groups, and roles** and by attaching **policies** to them. The four building blocks work together, and understanding how they differ is the heart of this module.

TABLE 4.1 The four IAM building blocks

ENTITY	WHAT IT IS	CREDENTIALS	TYPICAL USE
IAM user	An entity that represents a person or application interacting with the account	A permanent set of credentials, kept until a forced rotation	A person's console login; an application's programmatic access
IAM group	A collection of IAM users	None of its own	Grant the same set of permissions to multiple users at once
IAM role	An AWS identity you attach permission policies to; similar to a user but not tied to one person	No long-term credentials; temporary security credentials issued when assumed	Delegate access — EC2 apps, cross-account, federation, mobile apps
IAM policy	A JSON document that explicitly lists permissions	Not applicable	Attach to a user, group, role, or resource to allow or deny access

BEST PRACTICE FOR LONG-TERM ACCESS

- Attach IAM policies to **IAM groups**, then assign IAM users to those groups.
- A user who is a member of a group **inherits** the permissions attached to that group.
- You can also attach a policy **directly to a user** to further customize the access granted through the group.
- A role is different — rather than carrying long-term credentials, it is **assumed**, and the principal receives temporary credentials for the session.

4.3 Authenticating with IAM

When you develop applications, authentication is required at several different moments for different people and systems. Consider a mobile photo app whose users take pictures and upload them to AWS. As the developer, you use your AWS credentials to authenticate to the account so you can build the app (**AWS account authentication**). An app user signs in to the photo app, authenticating with your app (**application authentication**). When that user uploads a picture, the app must authenticate to the account before it can reach a service such as Amazon S3 (**resource authentication**). Finally, a photo landing in the S3 bucket can trigger a backend job that updates an Amazon RDS database, which requires **database authentication**. Each moment calls for the right credential in the right place.

TABLE 4.2 IAM credentials for authentication

ACCESS METHOD	CREDENTIALS	USED WITH
Console access	User name and password	AWS Management Console
Programmatic access	Access key ID and secret access key (an AWS access key)	AWS CLI, AWS SDKs, and API access

You must authenticate to supported services from the console, the AWS CLI, and the SDKs by providing AWS credentials, and which type you use depends on how you access AWS. To sign in to the console, you provide your user name and password. To authenticate programmatically, you provide an AWS access key — the combination of an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). You may also be required to provide additional security information; AWS recommends MFA to increase account security.

> **MULTI-FACTOR AUTHENTICATION (MFA)**
>
> - Adds an extra layer of protection on top of the user name and password by requiring **two or more factors**.
> - Factors are something you **know** (a password), something you **have** (a token or device), or something you **are** (a fingerprint).
> - After the password, the user is prompted for an authentication code from a **hardware or virtual** MFA device.
> - Enable MFA for **AWS Management Console users** and for **AWS API users** — API access requires temporary security credentials, and you pass optional MFA parameters in AWS STS requests to call MFA-protected APIs and resources.

An **IAM role** lets you define the permissions a user or service needs without attaching them to any one user or group; instead the permissions attach to the role, and the user or service **assumes** the role. When a principal assumes a role, its prior permissions are temporarily forgotten and AWS returns **temporary security credentials** it can use to make programmatic requests. Because a role has no long-term credentials, you do not have to grant standing keys to every entity that needs access. A role is *not* uniquely associated with one person; it can be assumed by a person, an application, or a service, and it is often used to **delegate access**.

IAM ROLE CHARACTERISTICS	WHEN TO USE TEMPORARY CREDENTIALS
temporary by design ▪ Provides temporary security credentials ▪ Is not uniquely associated with one person ▪ Is assumable by a person, application, or service ▪ Is often used to delegate access	*common use cases* ▪ An application that runs on Amazon EC2 needs to reach AWS resources ▪ An IAM user in one account needs temporary access to another account (**cross-account**) ▪ Identities authenticated outside AWS need access (**identity federation** — enterprise or web identity providers) ▪ A **mobile app** needs access without embedding credentials that are hard to rotate

Roles also change how you grant access to outsiders. By creating a role for external account access you avoid managing user names and passwords for third parties; if you no longer want someone or some system to have access, you modify or delete the role. For applications that run on EC2 instances and other AWS compute services, applications or AWS services can **programmatically assume a role at runtime**, so you never store long-term credentials locally.

Two developer-workflow details round out authentication. First, **IDE access**: AWS provides toolkits for many popular IDEs to integrate access to AWS resources into your development environment, and if you run your IDE remotely on an EC2 instance you can use **AWS managed temporary credentials**, which manage credentials in the EC2 environment on your behalf and follow security best practices — avoiding the need to distribute or embed long-term credentials. Second, the **AWS credentials file**: after installing the AWS CLI you run `aws configure`, and the CLI stores your credentials in a local file at `~/.aws/credentials` on Linux, macOS, or Unix (`C:\Users\USERNAME\.aws\credentials` on Windows).

BENEFITS OF THE CREDENTIALS FILE AND PROFILES

- Credentials live **outside your projects**, so you are unlikely to commit them into version control by accident.
- You can define and name **multiple sets of credentials** (profiles) in one place — for example a `default` profile and a `prod` profile.
- You can **reuse** the same credentials across projects and reference them from supported AWS SDKs and tools.
- When you use the CLI you can **specify which profile** to use and switch between profiles and access keys.

SECTION KEY TAKEAWAYS

- IAM roles with temporary credentials are **preferred** for applications that run on Amazon EC2 and for mobile apps.
- **Avoid storing AWS credentials in code or in publicly accessible places** — use credential files instead.

COMMON PITFALL

Do not treat a role as "a user with extra steps." A role has *no* long-term credentials of its own — no password, no standing access keys. Its value is precisely that a principal assumes it to receive short-lived credentials that expire, so there is nothing durable for an attacker to steal and nothing for you to rotate.

4.4 Authorizing with IAM

Once a principal is authenticated, IAM **policies** decide what it can do. By default an authenticated user, group, or role **cannot access anything** in your account until you grant permissions by creating a policy. An IAM policy is a **JSON document** that defines the `Effect` (allow or deny), the `Action` list, the `Resource` it applies to, and optional `Condition` elements under which an entity can invoke API operations. Anything not explicitly allowed is denied by default. A single policy can be attached to an IAM user, group, or role, or to a resource — or to any combination.

TABLE 4.3 Two types of policies, by attachment point

POLICY TYPE	ATTACHED TO	ANSWERS	EXAMPLE
Identity-based	An IAM user, group, or role	What does this identity have access to?	Grant a user the ability to access a DynamoDB table
Resource-based	A resource	Who is permitted to do what with this resource?	Grant access to an S3 bucket, or cross-account access between two trusted accounts

WORKED EXAMPLE – A CROSS-ACCOUNT BUCKET POLICY

Account A creates a **resource-based policy** on its S3 bucket (named DOC-EXAMPLE-BUCKET). The policy grants **Account B** (AWS account number 111122223333) permission to perform any Amazon S3 API operation on that bucket. Notice what it does *not* do: the bucket policy names no IAM users, groups, or roles — it specifies Account B as a whole. To finish the grant, Account B must create an IAM user policy that allows a specific user in Account B to access Account A's bucket. Access requires both sides: Account A permits the account, and Account B permits its user.

TABLE 4.4 Managed policies vs. inline policies

ASPECT	MANAGED POLICIES	INLINE POLICIES
What they are	Standalone, identity-based policies (AWS-managed, created by AWS; or customer-managed, created by you)	Policies embedded in a principal entity — an inherent part of one user, group, or role
Relationship	Can be attached to multiple users, groups, and roles	Strict 1:1 relationship between the policy and the entity
Benefits	Reusability, central change management, versioning and rollback, delegable permissions management	Useful when you want a strict one-to-one link; each entity keeps its own copy, so they cannot be centrally managed

When it evaluates policies, AWS follows a fixed logic and evaluates **all** applicable policies — the order has no effect on the outcome. **By default, all requests are denied.** An **explicit allow** overrides that default deny. An **explicit deny** overrides any explicit allow. So if one policy allows an action and another denies it, the conflict resolves to the most restrictive result: the request is denied. Every request ends as either allowed or denied.

WORKED EXAMPLE – AN ALLOW-THEN-DENY POLICY

A single policy can grant narrow access and then wall off everything else. Its first statement uses `Effect` = `Allow` to grant DynamoDB and S3 actions on exactly three resources — the DynamoDB table `pollynotes`, the S3 bucket `polly-notes-web`, and the S3 bucket `polly-notes-mp3` and all objects it contains. Its second statement uses `Effect` = `Deny` with a `NotResource` element listing those same three resources, which denies the DynamoDB and S3 actions on *every other* table and bucket. The Action lists use a wildcard (an asterisk) to mean all DynamoDB and all S3 actions. Because an explicit deny takes precedence over an allow, the net effect is exact: the entity can act only on the specified resources and on nothing else in the account, even if another policy would have granted it.

Every grant should follow the **principle of least privilege**: grant only the permissions needed to perform a task. It is more secure to start with a minimal set of permissions and add more as needed than to start too permissive and try to restrict later — which takes research to determine exactly what access a task requires. For example, developers might be allowed to create EC2 instances in a production environment but *not* to stop or terminate them.

SECTION KEY TAKEAWAYS

- Permissions for accessing AWS account services and resources are defined in **IAM policy documents**.
- Attach IAM policies to IAM **users, groups, or roles**.
- Follow the **principle of least privilege** when you grant account access.
- When IAM determines permissions, an explicit **deny** will always override any allow statement.

COMMON PITFALL

On authorization questions, watch two things. First, deny wins: an explicit deny overrides any allow, and evaluation order is irrelevant. Second, match the policy type to the attachment point — identity-based attaches to a user, group, or role, while resource-based attaches to a resource and is what makes cross-account access possible.

4.5 An IAM scenario, end to end

Sofía created the café's AWS account, so she has the **account root user** credentials and signs in with the account's email address. The root user has full access to everything in the account — billing information, personal profile data, and every resource in every service — and you **cannot restrict** the privileges of the root user. Because of that power, AWS recommends **not** using root credentials for day-to-day work. Instead, Sofía uses the root user once to create an IAM user for herself, gives that user full access to all services, secures the root credentials, and from then on logs in as her IAM user.

Now authenticated as her IAM user, Sofía builds a permission structure with groups and least privilege. She creates an **Admins** group, attaches administrative policies to it, and adds her own user, which then inherits those permissions. She hires a consultant, **Mateo**, and adds his user to Admins — but she wants to keep group-policy management for herself, so she also attaches a **custom policy directly to Mateo that explicitly denies** creating IAM groups and attaching policies to them. Because an explicit deny overrides the allow he inherits from the group, Mateo has full admin access *except* those specific group-management actions.

Applying least privilege, Sofía then creates a **DBAdmins** group whose permissions are more restrictive than Admins' — full access to DynamoDB, for instance, but not full access to Amazon EC2 — because database administrators do not need administrator permissions across all services. Mateo creates the DBAdmin users and places them in the group, where they inherit its permissions; although he cannot create groups or manage their policies, the Admins policy still

lets him **add and remove** users from DBAdmins. Next they create a **Developers** group the same way; when a group policy is too permissive for one individual, they attach a policy **directly to that user** to deny specific actions.

A few days later, temporary needs call for **roles** rather than group membership. Sofía wants to give two DBAdmins members, **Wei and Ana**, extra Amazon RDS access for a project without granting it to the whole group (whose members only need DynamoDB). She creates a **TempDBRole**, attaches a policy granting full RDS access, and temporarily attaches the role to Wei and Ana; when the project ends she removes them. She also creates a **TempDevRole** with a more restrictive policy for a temporary contractor — she defines a **TempDev** user but does *not* add it to the Developers group, attaching the role instead so the contractor does not inherit the permanent staff's broad access. The same role also lets her grant Ana some temporary developer access to help on the project.

WORKED EXAMPLE – THE SAMPLE EXAM QUESTION

A company stores an access key (access key ID and secret access key) in a text file on a custom Amazon Machine Image (AMI) and uses it to access DynamoDB tables from instances created from the AMI. The security team wants a more secure solution. Isolate the key phrases: **storing an access key**, **on a custom AMI**, **to access DynamoDB tables**, **more secure**. The answer is to **create an IAM role with permissions to access the table and launch all instances with the new role.** IAM roles for EC2 instances let applications on the instance reach AWS resources **without creating or storing any access keys**; every alternative that keeps an access key — in an S3 bucket, in instance user data, or on a key server — still introduces the complexity and risk of managing that secret.

COMMON PITFALL

The scenario's recurring lesson: reach for a **role** whenever access is temporary, delegated, or belongs to an application, and reach for **least privilege** whenever you size a permission set. Group membership grants standing permissions; a role grants short-lived ones you can attach and detach as projects begin and end.

Module summary

- Security is a shared responsibility: AWS secures *of* the cloud (hardware, facilities, and the global infrastructure of Regions, Availability Zones, and edge locations), and the customer secures *in* the cloud (data, guest OS and patches, applications, firewall and security group configuration, and IAM). The model reduces the customer's operational burden while giving them flexibility and control.
- IAM controls access to AWS resources through users, groups, roles, and policies; it integrates with most AWS services, supports federation and MFA, and grants a principal no access until permissions are explicitly assigned.
- Access control has two halves — authentication establishes who the principal is (a person or application), and authorization determines what that principal may do by evaluating the relevant policies.
- Credentials come in two forms: a user name and password for console access, and an access key ID with a secret access key for programmatic access via the CLI, SDKs, and APIs; MFA adds another factor and is recommended for console and API users.
- An IAM role has no long-term credentials; a principal assumes it and receives temporary security credentials from AWS STS. Roles are preferred for EC2 applications, mobile apps, cross-account access, and identity federation, and let you delegate access without managing standing keys.
- Store credentials in the AWS credentials file with named profiles (for example `default` and `prod`) rather than in code or public places; on EC2, use roles or AWS managed temporary credentials so nothing long-term is embedded.

- IAM policies are JSON documents defining effect, actions, resources, and optional conditions; identity-based policies attach to a user, group, or role, while resource-based policies attach to a resource and enable cross-account access. Managed policies are reusable and versioned; inline policies are embedded 1:1.
- Policy evaluation is default deny, an explicit allow overrides the default, and an explicit deny overrides any allow; order does not matter and the most restrictive rule wins. Grant access by least privilege — start minimal and add only what a task requires.

Review questions

1. State the dividing line of the shared responsibility model, and give one AWS example and one customer example.

Answer: AWS is responsible for security of the cloud; the customer is responsible for security in the cloud. AWS example: protecting the global infrastructure (Regions, Availability Zones, edge locations), the hypervisor layer, and the physical facilities. Customer example: patching the guest OS on EC2 instances, configuring security groups, securing applications, and protecting data (for instance with S3 bucket policies and encryption).

2. Distinguish authentication from authorization in IAM, and define principal.

Answer: Authentication establishes the identity of the requester through credentials — who is making the request. Authorization determines what an authenticated requester may do by checking the relevant policies. A principal is the person or application whose identity is being authenticated.

3. Compare an IAM user, an IAM group, and an IAM role in terms of credentials.

Answer: An IAM user has a permanent set of credentials kept until a forced rotation. An IAM group has no credentials of its own; it is a collection of users that share the permissions attached to the group. An IAM role has no long-term credentials at all — a principal assumes it and receives temporary security credentials for the session.

4. Why are IAM roles with temporary credentials preferred for applications on EC2 and for mobile apps?

Answer: Because you never have to store or rotate long-term access keys with the application. The application or service assumes the role at runtime and gets temporary credentials that expire, so there are no durable keys to embed in code, on an AMI, or in a mobile app where they could be extracted.

5. How does IAM evaluate policies when several apply, including a conflict between an allow and a deny?

Answer: AWS evaluates all applicable policies and the order does not matter. By default every request is denied; an explicit allow overrides the default deny; and an explicit deny overrides any allow. If one policy allows an action and another denies it, the most restrictive result applies — the request is denied.

6. What is the difference between an identity-based policy and a resource-based policy, and which enables cross-account access?

Answer: An identity-based policy is attached to an IAM user, group, or role and says what that identity can access. A resource-based policy is attached to a resource and says who can access that resource. Resource-based policies enable cross-account access — for example, granting another AWS account access to an S3 bucket.

7. State the principle of least privilege and give the module's developer example.

Answer: Grant only the permissions needed to perform a task, starting with a minimal set and adding more as needed rather than starting broad and restricting later. Example: developers might be allowed to create EC2 instances in a production environment but not to stop or terminate them.