MODULE 5 · STUDY NOTES

Protecting Data in Your Application

Protecting data at rest and in transit — S3 access controls and protection features, client-side vs server-side encryption, AWS KMS envelope encryption, TLS and certificates, and the services that guard secrets and sensitive data

This module focuses on the portion of the shared responsibility model that the customer always owns: protecting the data itself. Data exists in two states, and each needs its own defenses. **Data at rest** sits in storage such as Amazon S3 or EBS volumes; **data in transit** moves across networks and often the public internet, where it is considered less secure. Encryption is the common thread — an algorithm and a key turn readable plaintext into ciphertext that is worthless without the key, which makes *managing the keys* the heart of the problem. The module works outward from S3's default-private posture and its built-in protection features, through the mechanics of client-side versus server-side encryption and the AWS KMS envelope-encryption model that powers SSE-KMS, into protecting connections with TLS and certificates, and finally to the services — Secrets Manager and Macie — that guard credentials and hunt down sensitive data. A hands-on lab reinforces it all by encrypting S3 objects and EBS volumes with a KMS key and watching CloudTrail record every use.

LEARNING OBJECTIVES

- Describe how to protect data at rest and in transit
- Identify Amazon Simple Storage Service (Amazon S3) protection features
- Encrypt data in Amazon S3
- Differentiate between client-side encryption (CSE) and server-side encryption (SSE)
- Identify AWS services that help protect your data

5.1 Protecting data at rest

Encrypting **data at rest** ensures the security of stored data even if an unauthorized party gains access to it — it makes compromise much harder even when an attacker has already breached an endpoint, and it is often required for business or compliance reasons. In the **shared responsibility model**, this is customer territory: the customer is responsible for client-side data encryption and data integrity (including authentication) and for server-side encryption of the file system and data. AWS secures the underlying infrastructure; the customer secures the data placed on it.

TABLE 5.1 Why protect data at rest – common scenarios and defenses

SCENARIO	HOW TO PROTECT AGAINST IT
Information disclosure	Limit who can access data, manage access with policies, and encrypt confidential data
Data integrity compromise	Limit who can modify data with resource permissions; use digital signatures and encryption; restore from backup or a previous S3 object version
Accidental or malicious deletion	Apply correct permissions and least privilege; restore from backup or a previous S3 object version
System, hardware, and software availability	In a failure or disaster, restore data from replicas

In Amazon S3 specifically, **all resources are private by default** — buckets, objects, and subresources such as lifecycle and website configuration. Only the resource owner, the AWS account that created the resource, can access it, and access requires AWS credentials. The owner grants more access by writing an access policy; the **S3 Block Public Access** feature acts as an additional layer that prevents accidental exposure. Beyond access control, AWS recommends you *consider encrypting data at rest* in S3.

IDENTITY-BASED POLICY	RESOURCE-BASED POLICY
attached to an IAM principal	*attached to an AWS resource*
▪ Indicates what the user (or role) is permitted to do ▪ Travels with the identity across the resources it touches ▪ Example: allow Bob to GET, PUT, and LIST on bucket X	▪ Indicates what a specified user may do with the resource ▪ Attach to S3 buckets, VPC endpoints, or KMS keys ▪ Can grant cross-account access by naming AWS account IDs ▪ Example: a bucket policy that DENIES PUT to Bob

WORKED EXAMPLE – IDENTITY POLICY VS BUCKET POLICY

Bob's **identity-based** policy allows GET, PUT, and LIST on bucket X. But bucket X's **resource-based** policy allows GET and LIST and **denies** PUT. Result: Bob cannot PUT objects into bucket X even though his identity policy allows it — the explicit deny wins. For bucket Y, Bob's identity policy allows only LIST and says nothing about GET or PUT; the bucket policy allows GET and LIST. Result: Bob **can** GET objects from bucket Y even though his identity policy never explicitly allowed it, because the resource policy grants it and nothing denies it. The lesson: evaluate identity and resource policies together, and remember that an explicit DENY overrides any allow.

ABOUT ACCESS CONTROL LISTS (ACLS)

- In addition to IAM and bucket policies, S3 supports **ACLs** — an independent mechanism that can be combined with policies.
- Most modern S3 use cases no longer require ACLs; AWS recommends **disabling** them except when you must control access per individual object.
- With ACLs disabled, the **bucket policy becomes the sole audit surface**, which simplifies review.

5.2 Amazon S3 protection features

S3 provides several features to build into your security posture. Three stand out: **Block Public Access** overrides risky permissions, **Versioning** protects against deletion and overwrite, and **Object Lock** makes objects immutable. They complement, rather than replace, the encryption and access controls already covered.

TABLE 5.2 Amazon S3 Block Public Access settings

SETTING	WHAT IT BLOCKS
BlockPublicAcls	Blocks public access granted by *new* ACLs; existing policies and ACLs are not changed
IgnorePublicAcls	Ignores *all* public ACLs on the bucket and its objects
BlockPublicPolicy	Rejects PUT of a bucket policy that would allow public access (existing policies unaffected)
RestrictPublicBuckets	Restricts a publicly-policied bucket to AWS services and authorized users in the owner's account

The four settings are independent and combinable, and you apply them to an **access point, a bucket, or an entire account** — not per object. If settings differ across those levels, S3 applies the **most restrictive** combination. When a request arrives, S3 checks whether a Block Public Access setting prohibits it and rejects the request if so. The console also highlights publicly accessible buckets and warns you before a change would make a bucket public.

Versioning keeps multiple variants of an object in the same bucket so you can preserve, retrieve, and restore *every version of every object*. It is disabled by default and must be explicitly enabled. Delete an object and S3 inserts a **delete marker** as the current version rather than removing the object — you can restore the prior version; overwrite an object and the old version is still recoverable. By default the most recent version is retrieved; specify a version ID to get an older one. Pair versioning with **S3 Lifecycle** to control storage costs. Critically, once enabled versioning **cannot be turned off — only suspended**, which stops accruing new versions without changing existing ones. Versioning also unlocks **MFA delete**, which requires two forms of authentication to delete a version or change the bucket's versioning state.

Object Lock stores objects using the **write-once-read-many (WORM)** model, protecting data that must not change or be deleted, for a fixed time or indefinitely — useful for regulatory WORM requirements. It works **only in versioned buckets**. It offers two ways to manage retention: a **retention period** locks a version for a fixed span, and a **legal hold** applies the same protection with no expiration until you explicitly remove it.

TABLE 5.3 Object Lock retention modes

MODE	WHO CAN OVERWRITE OR DELETE A PROTECTED VERSION
Governance	Only users with special permissions can overwrite/delete a version or change its lock settings
Compliance	No user can overwrite or delete a protected version — including the AWS account root user

> **COMMON PITFALL**
>
> Two easy points on the exam: S3 Versioning **cannot be disabled once enabled** (only suspended), and Object Lock **requires a versioned bucket**. Choose compliance mode only when you truly need immutability that even the root user cannot bypass.

5.3 Protection through encryption

Encryption works by using an algorithm with a key to convert plaintext into unreadable **ciphertext** that becomes readable again only with the right key — the phrase *Hello World!* might become a string like `1c28df2b595b4e30b7b07500963dc7c`. A strong algorithm relies on mathematical properties so the ciphertext cannot be decrypted with any practical amount of computing power without the key. That is why **protecting and managing the keys** is the critical part of any encryption solution.

CLIENT-SIDE ENCRYPTION (CSE)	SERVER-SIDE ENCRYPTION (SSE)
encrypt before submitting to AWS ▪ Encryption happens before data reaches AWS; decryption after retrieval ▪ S3 receives your encrypted data and plays no role in encrypting or decrypting it ▪ Use a key stored in AWS KMS or one you store within your application ▪ Libraries: AWS Encryption SDK, DynamoDB Encryption Client, S3 encryption clients	*encrypt at the destination* ▪ The receiving service encrypts data as it writes to disk and decrypts on access ▪ Source data comes from your data center or an EC2 instance, uploaded over HTTPS ▪ The endpoint handles encryption and key management for you ▪ In S3, encryption is an optional step you specify at upload

The two approaches differ in *when, where, and who* encrypts and decrypts — not necessarily in *how* — and they are **not exclusive**: you can apply both to the same data for a stronger profile. Most AWS services that store customer data offer SSE or perform it by default, and most SSE services integrate with **AWS KMS** to protect the keys.

TABLE 5.4 Types of Amazon S3 server-side encryption

OPTION	KEYS	HOW IT WORKS
SSE-C	You provide	You manage the keys and supply one with each request; S3 handles encrypt/decrypt but stores only a salted **HMAC** of the key — lose the key and you lose the object
SSE-S3	AWS managed	Each object is encrypted with a unique key, which is itself encrypted by a regularly rotated key; uses 256-bit **AES-256**
SSE-KMS	AWS KMS	Similar to SSE-S3 but adds separate key-use permissions, a **CloudTrail** audit trail, customer/AWS managed KMS keys, and **envelope encryption**

AWS KMS is a managed service to create and control the keys used to encrypt your data. You can give keys aliases and descriptions, schedule automatic rotation, disable or delete keys, and even import your own. It uses **FIPS 140-2 validated hardware security modules (HSMs)** to protect keys, and it is integrated with many AWS services. The keys **never leave AWS KMS** — you submit data to be encrypted or decrypted under keys you control — which reduces the risk of a data key being compromised. You set **usage policies** to determine which users can use each key, and **all** KMS operations are logged in **AWS CloudTrail**, so you know who used which key and when.

WORKED EXAMPLE – SSE-KMS ENCRYPT THEN DECRYPT

Encrypting an upload: you request to store a file as an encrypted object; S3 requests a data key from KMS; KMS generates a plaintext data key and a copy of it encrypted under your customer managed key, and sends both to S3; S3 encrypts the object with the plaintext key, stores the object, deletes the plaintext key, and keeps the encrypted data key in the object metadata. **Decrypting on access:** you request the object; S3 sees it is encrypted and sends the encrypted data key to KMS; KMS decrypts it under the customer managed key (which never leaves KMS) and returns the plaintext data key; S3 decrypts the object, serves it to you, and deletes the plaintext key. That data-key-under-a-root-key pattern is **envelope encryption**.

The same model powers **Amazon EBS encryption**, a straightforward way to encrypt EBS volumes and snapshots without building your own key infrastructure. Each volume is encrypted with **AES-256-XTS**, which uses two 256-bit keys (think of it as one 512-bit key); the data key is encrypted under a KMS key in your account. You grant EBS access to a customer managed key so it can create and use data keys: EBS stores the encrypted data key with the data, a host retrieves it, a call over **TLS** asks KMS to decrypt it inside an HSM, and the returned plaintext key stays in memory to encrypt and decrypt all traffic to the attached volume.

COMMON PITFALL

Under SSE-KMS the **plaintext data key is deleted** after use — only the *encrypted* data key persists in object metadata, and it is useless without KMS. And with **SSE-C**, S3 never stores your key: if you lose it, the object cannot be recovered.

5.4 Protecting data in transit

Data in transit is data actively moving from one location to another — across the internet or through a private network. Cloud applications often communicate over public links, so protecting traffic between clients and servers, and between servers, is critical; data is generally considered less secure while in motion. A **man-in-the-middle (MITM)** attack exploits weak web protocols to insert an attacker into a communication channel, enabling packet sniffing and modification.

RISKS TO DATA IN TRANSIT

- Accidental information disclosures
- Data integrity compromises
- Identity compromises — including **man-in-the-middle (MITM)** attacks and **identity spoofing**

The primary defenses are **SSL endpoints over TLS (HTTPS)** and **client-side encryption**: you can upload to and download from S3 over SSL endpoints using HTTPS, and CSE protects data in transit because it is already encrypted before you send it. You can also use **Amazon VPC endpoints** — with S3 bucket policies — to limit bucket access to specific endpoints. AWS applies a **multi-level** approach: all traffic between AWS data centers is transparently encrypted at the physical layer; traffic within a VPC and between peered VPCs across Regions is transparently encrypted at the network layer on supported EC2 instance types; and at the application layer, customers choose whether and how to use TLS. All AWS service endpoints support TLS for secure HTTPS API calls.

RDP – WINDOWS SERVERS	SSH – LINUX SERVERS
Remote Desktop Protocol	*Secure Shell*
▪ Establishes an underlying SSL/TLS connection ▪ Issue a trusted **X.509 certificate** for better security ▪ Don't rely on the default **self-signed** certificates	▪ Provides a secure communications channel ▪ Use **tunneling** to protect the application session in transit ▪ Don't allow the **root user** to log in over SSH ▪ Require an SSH **key pair** and deactivate password auth

AWS Certificate Manager (ACM) makes it easy to enable SSL/TLS and meet encryption-in-transit compliance requirements. It handles the complexity of creating and managing **public** SSL/TLS certificates for your AWS-based sites and apps, and can issue **private** X.509 certificates to identify internal users, computers, applications, and devices. One interface manages both; deploy certificates to load balancers, CloudFront distributions, and API Gateway endpoints; and ACM **automatically renews** certificates to minimize downtime and outages from expiration.

ACM Private Certificate Authority (Private CA) lets you build a **public key infrastructure (PKI)** inside AWS — creating root and subordinate CA hierarchies without the cost of running an on-premises CA. Private CAs issue end-entity X.509 certificates for creating TLS channels, authenticating users/computers/API endpoints/IoT devices, signing code, and checking revocation via **OCSP**. Certificates a private CA issues are trusted **only within your organization**, not on the internet. The service is secured with HSMs at **FIPS 140-2 Level 3**, controlled with **IAM** policies, and it automatically publishes certificate revocation lists (CRLs) and audit reports to S3 buckets.

> **COMMON PITFALL**
>
> Match the defense to the data's state: **SSL/TLS (HTTPS), VPC endpoints, and ACM certificates** protect data *in transit*; SSE and CSE protect data *at rest*. On remote-access questions, remember RDP/SSH should use trusted certificates and key pairs — never default self-signed certs or root SSH login.

5.5 Best practices for protecting data in Amazon S3

All S3 objects are **private by default**, and only the owner can access them — but the owner can share an object without making it public by generating a **presigned URL** with their own credentials, granting time-limited permission to upload or download that one object. A presigned URL uses three parameters to limit access: the **Bucket** (where the object is or will be), the **Key** (the object's name), and **Expires** (how long the URL is valid). Presigned URLs even let someone *without* an AWS account upload an object you can then receive.

> **SECURITY BEST PRACTICES FOR AMAZON S3**
>
> - Consider encryption of data **at rest**, and enforce encryption of data **in transit**.
> - Ensure buckets use correct policies and are **not publicly accessible**; combine bucket policies with IAM policies.
> - Apply the **principle of least privilege** — e.g. an upload-only role should have no read permission.
> - Enable **MFA delete** for versioning-enabled buckets.
> - **Enforce SSE on each PUT request** — deny requests that don't specify SSE.
> - Set **default encryption** on a bucket so all stored objects are encrypted.
> - Use **presigned URLs** for applications that refer to S3 objects with anonymous access.
> - Use the appropriate **Object Lock** retention mode, enable **Block Public Access**, and enable **Versioning**.
> - Upload over SFTP using **AWS Transfer for SFTP**, a fully managed service for file transfer directly in and out of S3.

WORKED EXAMPLE – ENFORCE ENCRYPTION ON PUT

To guarantee everything written to a bucket is encrypted, combine two controls. First, set **default encryption** on the bucket so any object stored is encrypted even if the uploader forgets. Second, add a bucket policy that **enforces SSE for each PUT request** — the policy denies any PUT that does not request server-side encryption, so an unencrypted upload is rejected outright. Layer least-privilege IAM roles on top (upload-only roles get no read permission), and the bucket resists both accidental exposure and misconfigured clients.

5.6 Additional data protection services

Two services extend data protection beyond storage and transit. **AWS Secrets Manager** manages access to secrets — database credentials, passwords, third-party API keys, even arbitrary text — stored and controlled centrally through the console, CLI, or API. Its core value is removing **hard-coded credentials** from source code and config files: instead of embedding a password, the application makes an API call to retrieve the secret at runtime, so the secret simply isn't in the code to be found. Secrets Manager can also **automatically rotate** secrets on a schedule, replacing long-term secrets with short-term ones to reduce the risk of compromise, and it helps meet regulatory and compliance requirements while auditing the lifecycle of secrets.

WORKED EXAMPLE – SECURING DATABASE CREDENTIALS

A common pattern secures database credentials without hardcoding them: clients call a RESTful API on **Amazon API Gateway** → API Gateway runs an **AWS Lambda** function → the Lambda function retrieves the database secret via the **Secrets Manager** API, which decrypts the protected text and returns it over a secured channel → the function connects to the **Amazon RDS** database using those credentials and returns the query results. The secret is never in the code or an environment variable, protecting the backend database.

For **rotation**, Secrets Manager can automatically rotate passwords for Aurora, MySQL, MariaDB, PostgreSQL, and Oracle databases hosted on AWS (administrator permissions required). A **Lambda** rotation function does the work — the built-in function already exists for RDS DB engine credentials, while other credentials (such as a SAML login) may need a custom function. Rotation uses **staging labels** across four steps: the function contacts the DB for new credentials, Secrets Manager stores them with the **AWSPENDING** label, the new credentials are tested, and then they are promoted to the default with the **AWSCURRENT** label.

Amazon Macie is a security service that uses **machine learning and pattern matching** to automatically discover, classify, and protect sensitive data in AWS. It recognizes **personally identifiable information (PII)** such as passport, medical ID, and tax ID numbers, along with financial information, encryption keys, and credentials — and you can add **custom data identifiers** using regular expressions for proprietary data. Macie currently protects data stored in **Amazon S3 only**, analyzing S3 resource-based policies and ACLs and continuously monitoring buckets. It provides full API coverage and integrates with **AWS Organizations** to manage many accounts centrally.

WHAT MACIE FLAGS

- Macie scans supported objects at the **bucket level** and evaluates them for sensitive data that meets the job criteria.
- It generates near-real-time **findings** for buckets that are **public**, **not encrypted**, or **shared or replicated** with accounts outside your organization.
- Jobs run one-time, daily, weekly, or monthly; after the first run Macie tracks changes and evaluates only **new or modified** objects over time.

WORKED EXAMPLE – THE MODULE'S SAMPLE EXAM QUESTION

A company requires that data stored in AWS be encrypted at rest. Which approach would meet this requirement? The keywords are **data stored in AWS** and **encryption at rest**. The correct answer is **server-side encryption in Amazon S3**, which protects data at rest. Using EBS-optimized EC2 instances alone does not guarantee protection at rest; a transit-focused control such as TLS does not encrypt S3 objects at rest; and you don't store this data in an instance store. Anchor on the state of the data (at rest) and the storage service (S3), and SSE is the match.

Module summary

- Data protection is the customer's job in the shared responsibility model, for data at rest (in storage) and data in transit (moving across networks); encrypting at rest adds a defense layer even if an endpoint is compromised.
- S3 resources are private by default and require AWS credentials; access is granted by identity-based policies (on IAM principals) and resource-based policies (on the resource, e.g. bucket policies), and an explicit DENY always overrides an allow. AWS recommends disabling ACLs.
- S3 protection features: Block Public Access (four override settings, applied at account/bucket/access-point level, most restrictive wins), Versioning (recover from delete/overwrite; can only be suspended; enables MFA delete), and Object Lock (WORM on versioned buckets; governance vs compliance retention plus legal holds).
- Encryption turns plaintext into ciphertext with an algorithm and a key, so key management is the critical part. CSE = you encrypt before sending (keys known only to you, S3 never sees plaintext); SSE = AWS encrypts on arrival, transparently.
- S3 offers three mutually exclusive SSE options: SSE-C (you provide keys, S3 stores only a salted HMAC), SSE-S3 (AWS-managed AES-256), and SSE-KMS (KMS keys with separate permissions, a CloudTrail audit trail, and envelope encryption).
- AWS KMS creates and controls keys in FIPS 140-2 HSMs, keys never leave the service, and every use is logged in CloudTrail; envelope encryption (encrypt data with a data key, then encrypt the data key with a root key) powers SSE-KMS and EBS encryption.
- Protect data in transit with SSL/TLS (HTTPS) endpoints, client-side encryption, and VPC endpoints; secure remote admin with RDP (trusted X.509 certs, not self-signed) and SSH (key pairs, no root login). ACM issues, deploys, and auto-renews public/private certificates; ACM Private CA builds an internal PKI trusted only within your org.
- S3 best practices: default bucket encryption, enforce SSE on every PUT, least privilege, MFA delete, Block Public Access, Versioning, Object Lock retention, presigned URLs (Bucket/Key/Expires) for time-limited object access, and AWS Transfer for SFTP.
- Secrets Manager keeps credentials out of code and auto-rotates them via Lambda using AWSPENDING/AWSCURRENT staging labels; Amazon Macie uses ML to discover and classify sensitive data in S3 and flags buckets that are public, unencrypted, or externally shared.

Review questions

1. **A user's identity-based policy allows PUT on a bucket, but the bucket's resource-based policy explicitly denies PUT. Can the user PUT objects, and why?**

   Answer: No. An explicit DENY overrides any allow, so the resource-based deny wins over the identity-based allow — the deck's Bob-and-bucket-X example. Policies from both sources are evaluated together.

2. **Differentiate client-side and server-side encryption.**

   Answer: CSE: your application encrypts data before sending it to AWS; the keys and algorithms are known only to you, and S3 stores ciphertext and never encrypts or decrypts it. SSE: AWS encrypts data at its destination after receiving it, transparently to the user, with the receiving service handling the keys and crypto.

3. **Name the three S3 server-side encryption options and who manages the keys in each.**

   Answer: SSE-C — you provide the keys, and S3 stores only a salted HMAC of them. SSE-S3 — AWS manages the keys, using AES-256. SSE-KMS — AWS KMS keys, adding separate key-use permissions, a CloudTrail audit trail, and envelope encryption.

4. **Explain envelope encryption as SSE-KMS uses it when storing an object.**

   Answer: S3 asks KMS for a data key; KMS returns a plaintext copy and a copy encrypted under the customer managed key; S3 encrypts the object with the plaintext key, stores the object, deletes the plaintext key, and keeps only the encrypted data key in object metadata. To decrypt, S3 sends the encrypted data key to KMS, which decrypts it under the CMK (which never leaves KMS) and returns the plaintext key.

5. **Why does S3 Object Lock require versioning, and how do governance and compliance modes differ?**

   Answer: Object Lock applies WORM protection to object versions, so the bucket must be versioned. In governance mode, only users with special permissions can overwrite/delete a protected version; in compliance mode, no user can — not even the AWS account root user — until the retention period expires.

6. **What three risks does the deck highlight for data in transit, and what are the primary defenses?**

   Answer: Risks: accidental information disclosure, data integrity compromise, and identity compromise (MITM attacks and identity spoofing). Defenses: SSL/TLS (HTTPS) endpoints and client-side encryption, plus VPC endpoints to limit access to S3 buckets.

7. **How does AWS Secrets Manager reduce the risk of leaked credentials?**

   Answer: It removes hard-coded credentials from source code and config files — the application retrieves the secret via an API call at runtime, so the secret isn't in the code — and it can automatically rotate secrets on a schedule via a Lambda function (using AWSPENDING and AWSCURRENT staging labels), replacing long-term secrets with short-term ones.

8. **What does Amazon Macie do, and which bucket conditions does it flag as findings?**

   Answer: Macie uses machine learning and pattern matching to discover, classify, and protect sensitive data (PII, financial information, encryption keys, credentials) in S3, and supports custom data identifiers. It generates near-real-time findings for buckets that are public, not encrypted, or shared/replicated outside your organization.