TUTORIAL

C# Tutorial: Secure AES Encryption and Decryption in .NET

Build a practical AES encryption and decryption workflow in C# .NET. This tutorial covers prerequisites, key and IV handling, implementation, validation, and production checks so you can safely encrypt and decrypt data with confidence.

Programming - July 15, 2026

Why this matters operationally

Plaintext secrets, customer data, tokens, and internal configuration values often move through .NET services, worker processes, and API boundaries. If that data is stored or transmitted without encryption, a single log leak, database read, file exposure, or memory dump can become an incident. The practical goal of this tutorial is to show you how to implement secure AES encryption in C# using .NET primitives, then validate that the encrypted data can be safely decrypted only with the correct key material.

By the end, you will be able to decide whether AES is the right fit, build a working encrypt/decrypt workflow, verify it with repeatable checks, and identify the production controls you still need before treating it as safe for real data.

What you will build

You will build a small, production-shaped helper that:

- encrypts UTF-8 text with AES in CBC mode and PKCS7 padding
- generates a fresh random IV for every encryption operation
- derives a key from a password using PBKDF2, or accepts a securely managed raw key

- returns an encoded payload that contains everything needed for decryption
- validates correct decryption and fails safely when inputs are tampered with

The finished state should look like this: given a plaintext string, your code produces a Base64 payload containing the salt, IV, and ciphertext; given the same key material, it can recover the original plaintext; given a wrong key or altered payload, it fails rather than returning corrupted data.

Stop-here-if warnings before you start

Before implementing encryption, verify the following. If any item is unresolved, stop and fix it first.

- You do not have a safe key storage plan. Do not hardcode keys in source code or configuration files that are broadly readable.
- You need authenticity as well as confidentiality. AES encryption alone does not prove data was not modified; you will need an integrity check or an authenticated encryption mode.
- You are trying to encrypt passwords for storage. Passwords should be hashed with a password hashing algorithm, not encrypted.
- You do not know whether your data must remain decryptable across services, deployments, or rotations. Key lifecycle decisions affect the payload format.
- You are relying on a framework version without confirming which cryptographic APIs are available. Verify your target .NET version before you choose the implementation pattern.

If you are also securing telemetry or audit output, consider how encryption complements safe observability practices. For example, Secure C# Logging with Structured Serilog and Correlation IDs is useful when you need traceability without exposing sensitive fields.

Prerequisites and decision rules

Goal

Confirm that AES is appropriate for your use case and that you have the minimum inputs required for a safe implementation.

Action

You need:

- a .NET project targeting a current supported runtime
- access to System.Security.Cryptography
- a key management approach
- sample plaintext for local validation

Choose AES when you need symmetric encryption for data at rest or in transit within a controlled application boundary. Do not use AES as a replacement for transport security such as TLS, and do not use it for password storage.

For this tutorial, you will use:

- AES for confidentiality
- PBKDF2 to derive a key from a password when a raw key is not already available
- a random salt and IV per encrypted payload
- Base64 for transport or storage of binary data as text

Expected output

You should know:

- what is being encrypted
- where the key comes from
- how encrypted data will be stored or transported
- how the decryption path will retrieve the correct components

Validation

A good fit means you can answer yes to these questions:

- Is the data symmetric, small enough to encrypt per item or per field, and expected to be decrypted by the same application or trusted service?
- Can you protect the key separately from the ciphertext?
- Can you rotate the key without breaking all historical data at once?

Common failure

The most common design error is conflating secrecy with authenticity. AES can hide data, but by itself it does not stop an attacker from altering ciphertext in some modes. If tamper detection matters, add a separate authentication layer or use an authenticated encryption mode after confirming your .NET version supports the APIs you intend to use.

Implementation plan

Goal

Create a reusable helper with clear inputs and outputs so encryption and decryption behave predictably.

Action

Use a small utility with this contract:

- input: plaintext string and password or key material
- output: serialized encrypted payload
- decryption input: the same payload and matching key material
- error behavior: fail on wrong key, wrong salt, altered payload, or invalid encoding

A practical payload layout is:

- version byte or version marker
- salt
- IV
- ciphertext

That layout makes later rotation and compatibility work easier because you can recognize how the payload was produced.

Expected output

You will have an encrypted payload that can be safely stored in a database column, message body, file, or secret store field.

Validation

Confirm your design before coding:

- The IV is generated fresh for every encryption operation.
- The salt is generated fresh when using a password-based key derivation function.
- The key is never embedded in the payload.
- The payload format is documented in code comments or a small README for operators.

Common failure

A frequent mistake is reusing the same IV with the same key. For AES-CBC, IV reuse undermines security. Another common mistake is deriving a key without a unique salt, which makes repeated password-based keys predictable.

Build the encryption helper

Goal

Implement secure encryption with explicit key derivation, random IV generation, and a serialized result.

Action

The example below uses AES-CBC with PKCS7 padding and PBKDF2 via Rfc2898DeriveBytes. It is suitable for understanding the workflow and for scenarios where you need confidentiality and can separately manage integrity. If you need built-in authentication, verify whether your target runtime supports an authenticated mode you can adopt instead.

Expected output

You should be able to call Encrypt with a plaintext string and password, then call Decrypt with the returned Base64 string and the same password to recover the original value.

Validation

Use a quick console app or unit test to validate the round trip:

Expected result:

- the encrypted string should look like Base64 text
- the decrypted value should exactly match the original plaintext
- the equality check should return True

Common failure

If decryption fails with padding or cryptographic exceptions, check these first:

- the password is identical on both sides
- the payload was not truncated
- the Base64 string was not modified by line wrapping, storage, or transport encoding
- the IV and salt sizes match the values used during encryption

If your application also reconstructs objects from decrypted content, keep the deserialization boundary separate and safe. Insecure reconstruction can turn decrypted data into an execution path, which is why Secure C# Deserialization: Preventing RCE in .NET Applications is relevant when encrypted payloads are later parsed into objects.

Prefer authenticated encryption when integrity matters

Goal

Make a correct decision about tamper detection instead of assuming encryption alone is enough.

Action

AES-CBC provides confidentiality, but it does not inherently provide integrity. If an attacker can modify ciphertext, your decryption logic may fail noisily or produce altered data that downstream code interprets incorrectly. For operationally sensitive data, you should prefer an authenticated encryption approach when your platform and compatibility requirements allow it.

If you cannot use an authenticated mode, add a separate message authentication mechanism over the version, salt, IV, and ciphertext before storing or transmitting the payload. The exact mechanism depends on your application standard and threat model.

Expected output

You should be able to explain whether your implementation is confidentiality-only or confidentiality plus integrity.

Validation

Before production, verify:

- whether tamper detection is required for the data class
- whether the selected .NET runtime supports the authenticated mode you want to use
- whether your payload format can carry any extra authentication metadata

Common failure

The common failure here is treating a successful decrypt as proof that the data was never modified. In CBC mode, decryption success alone is not a complete security check.

Validate the implementation end to end

Goal

Prove that the helper works under normal use and fails safely under incorrect use.

Action

Run these checks:

- Encrypt and decrypt the same value with the same password.
- Encrypt the same plaintext twice and confirm the outputs differ because the IV and salt are random.

- Try decrypting with a wrong password.
- Truncate or alter one character of the Base64 payload.
- Confirm your application handles exceptions in a controlled way.

A simple test harness can look like this:

Expected output

- two encryptions of the same input should produce different payloads
- valid decrypt should return the original text
- wrong-password decrypt should fail
- malformed input should fail before any plaintext is returned

Validation

Your implementation is behaving correctly if:

- ciphertext changes every run for the same plaintext and password
- decryption round trips exactly
- corrupted payloads do not produce silent success
- exceptions are caught at the application boundary and converted to safe operational responses

Common failure

A dangerous failure mode is leaking decrypted values into logs, error pages, or telemetry. If you need traceability, log a correlation identifier and the failure category, not the secret itself.

Operational follow-up before production use

Goal

Move from a working demo to a deployable pattern with manageable risk.

Action

Before production, verify these controls:

- Keys are stored in a secure secret manager, hardware-backed store, or equivalent protected location.
- Rotation strategy is defined and tested with versioned payloads.
- Exceptions are handled without exposing plaintext, key material, or payload internals.
- Input size limits are enforced to reduce memory pressure and abuse potential.
- The team knows whether the encrypted data must remain readable by older code after a key change.
- Integrity protection is in place if attackers can tamper with the ciphertext.

If you need a stable operational pattern, version the payload format from the start. That way, when you introduce a new key or a stronger algorithm, your decryptor can choose the correct path instead of guessing.

Expected output

You should have a clear operational story for key storage, rotation, decryption compatibility, and failure handling.

Validation

Use this checklist before rollout:

- Can a new deployment decrypt existing payloads?
- Can old payloads be distinguished from new ones?
- Are secrets absent from logs and crash dumps?
- Are malformed payloads rejected quickly?
- Is there a defined recovery path if the key is rotated or lost?

Common failure

The most expensive mistake is waiting until after deployment to define how old ciphertext will be handled. Key rotation without versioning often creates avoidable outages.

Practical summary

AES in .NET is straightforward to implement, but secure use depends on the surrounding workflow more than the cipher itself. You need fresh randomness for IVs and salts, a real key management plan, safe exception handling, and a clear decision about whether confidentiality alone is enough.

If you can round-trip data, reject wrong keys, reject tampered payloads, and keep keys out of your code and logs, you have a workable base. From there, production readiness is mostly about operational controls: versioning, rotation, validation, and the right integrity guarantees for the data you are protecting.

Use this guidance together with adversarial attack detection to connect the workflow with related operational context already available on the site.

> Part of the Programming: C# Insights content cluster.