CHECKLIST

C# AES Encryption and Decryption in .NET ? Secure Implementation Checklist

A practical checklist for building, validating, and preparing AES encryption and decryption workflows in C# .NET, including key handling, IV/salt generation, payload format, testing, and production checks.

C# AES Encryption and Decryption in .NET ? Secure Implementation Checklist

Use this checklist to plan, build, test, and operationalize a secure AES encryption/decryption workflow in C# .NET.

---

1) Confirm AES is the right tool

- [ ] The data needs symmetric encryption, not password hashing.
- [ ] The use case is data at rest or within a trusted application boundary.
- [ ] Transport security such as TLS is still used where data moves across networks.
- [ ] You have confirmed the target .NET runtime and available cryptography APIs.
- [ ] You have decided whether you need confidentiality only, or confidentiality plus authenticity.

Stop here if any of these are true

- [ ] You do not have a secure key storage plan.
- [ ] You are trying to encrypt passwords for storage.
- [ ] You need tamper detection but have not planned for authentication.
- [ ] You do not know how keys will be rotated without breaking decryption.

---

2) Define the operational design

- [ ] Identify exactly what data will be encrypted.
- [ ] Identify where the ciphertext will be stored or transmitted.
- [ ] Decide whether the key comes from a password or from managed key material.
- [ ] Document who can decrypt the data and under what conditions.
- [ ] Confirm the decryption path can retrieve the matching key material.
- [ ] Decide how older payloads will be supported after key or format changes.

Recommended design rules

- [ ] Use a fresh random IV for every encryption operation.
- [ ] Use a fresh random salt for every password-derived key.
- [ ] Never embed the secret key directly in the payload.
- [ ] Include a version marker so payload formats can evolve safely.
- [ ] Store or transport binary output in Base64 only when text encoding is required.

---

3) Choose a safe payload format

A simple payload layout helps with compatibility and rotation.

- [ ] Version marker

- [ ] Salt
- [ ] IV
- [ ] Ciphertext

Payload checklist

- [ ] The version byte is documented in code comments.
- [ ] Salt size is fixed and known.
- [ ] IV size is fixed and known.
- [ ] Ciphertext begins only after the metadata fields.
- [ ] Invalid or truncated payloads fail safely.

---

4) Implement encryption correctly

Core cryptographic settings

- [ ] Use AES with a secure key size, commonly 256 bits.
- [ ] Use CBC mode only if you understand its limitations and are handling integrity separately.
- [ ] Use PKCS7 padding for block alignment.
- [ ] Use PBKDF2 when deriving a key from a password.
- [ ] Use a strong hash function for PBKDF2, such as SHA-256.
- [ ] Use an iteration count appropriate for current hardware and threat model.

Encryption implementation checklist

- [ ] Generate plaintext bytes using UTF-8 encoding.
- [ ] Generate a random salt.
- [ ] Generate a random IV.
- [ ] Derive the key using the salt and password, if applicable.
- [ ] Configure AES with the selected mode, padding, key, and IV.
- [ ] Write version, salt, and IV before the ciphertext in the payload.
- [ ] Return the final payload as Base64 when text output is required.

---

5) Implement decryption safely

Decryption checklist

- [ ] Decode Base64 input before processing.
- [ ] Reject payloads that are too short to be valid.
- [ ] Read the version marker first.
- [ ] Extract salt and IV using the exact sizes used during encryption.
- [ ] Re-derive the key using the same password and salt.
- [ ] Attempt decryption only after validating the payload structure.
- [ ] Fail closed on wrong keys, malformed payloads, or unsupported versions.

Safe failure behavior

- [ ] Do not return partially decrypted data.
- [ ] Do not silently substitute defaults.
- [ ] Do not reveal whether the salt, IV, or key was incorrect.
- [ ] Use consistent error handling for invalid ciphertext.

---

6) Add integrity protection

AES encryption alone does not guarantee the data was not modified.

- [ ] Confirm whether tamper detection is required.
- [ ] If yes, add an authentication layer or use an authenticated encryption mode supported by your runtime.
- [ ] Validate that altered ciphertext fails decryption or verification.
- [ ] Treat ciphertext integrity failures as security events.

Integrity checklist

- [ ] The design includes a MAC, signature, or authenticated encryption strategy.
- [ ] Authentication is checked before accepting decrypted output.
- [ ] Integrity failures are logged without exposing sensitive content.

---

7) Validate with repeatable tests

Functional tests

- [ ] Encrypt known plaintext and verify output is not readable as plaintext.
- [ ] Decrypt with the correct key and verify the original text is recovered.
- [ ] Decrypt with the wrong key and confirm failure.
- [ ] Tamper with the payload and confirm failure.
- [ ] Test empty strings, short strings, and multiline text.

- [ ] Test non-ASCII characters and verify UTF-8 round-tripping.

Negative tests

- [ ] Truncate the payload and confirm rejection.
- [ ] Corrupt the Base64 string and confirm failure.
- [ ] Change the version byte and confirm rejection.
- [ ] Reuse a payload with the wrong key context and confirm it does not decrypt.

Regression checks

- [ ] Save a known-good ciphertext sample for future test runs.
- [ ] Verify the payload layout after code changes.
- [ ] Confirm iteration count and cryptographic settings have not been weakened.

---

8) Prepare for production

Key management

- [ ] Store keys outside source control.
- [ ] Restrict key access to the smallest possible set of services and operators.
- [ ] Define key rotation procedures before deployment.
- [ ] Document how to decrypt historical payloads after rotation.
- [ ] Separate secrets from application binaries and non-secret configuration.

Operational safety

- [ ] Avoid logging plaintext, keys, or full ciphertext payloads.
- [ ] Redact sensitive fields in traces and exception messages.
- [ ] Review memory handling and object lifetime for sensitive material.
- [ ] Confirm backups and replicas are protected to the same standard as primary data.
- [ ] Verify access controls for databases, files, message queues, and secret stores.

Deployment checklist

- [ ] Run encryption and decryption smoke tests in the target environment.
- [ ] Confirm the same runtime version is used in development, staging, and production where required.
- [ ] Confirm monitoring alerts exist for decryption failures and malformed payloads.
- [ ] Confirm rollback will not invalidate unreadable encrypted data.

---

9) Review common mistakes

- [ ] Reusing the same IV with the same key.
- [ ] Using a fixed salt for password-based key derivation.
- [ ] Hardcoding secrets in code or configuration.
- [ ] Encrypting passwords instead of hashing them.
- [ ] Treating encryption as a substitute for TLS.
- [ ] Ignoring integrity protection when tamper resistance matters.
- [ ] Failing to document payload format and versioning.

---

10) Final readiness check

Before shipping, confirm all of the following:

- [ ] You can explain what is encrypted and why.
- [ ] You know exactly where the key comes from and how it is protected.
- [ ] Every encryption call uses a fresh IV.
- [ ] Every password-derived key uses a fresh salt.
- [ ] The payload format includes versioning and can support future changes.
- [ ] Wrong keys and altered payloads fail safely.
- [ ] Integrity protection is present if authenticity matters.
- [ ] Tests cover success and failure cases.
- [ ] Logging and monitoring do not expose sensitive data.
- [ ] Key rotation and recovery procedures are documented.

Quick go-live summary

If you can check every item below, your AES workflow is in good shape for a controlled application boundary:

- [ ] You chose AES for the right use case.
- [ ] You generated a fresh IV per encryption.
- [ ] You used salt correctly when deriving keys from passwords.
- [ ] You serialized payloads in a documented, versioned format.
- [ ] You verified decryption with the correct key material.
- [ ] You verified failure on tampering or wrong keys.
- [ ] You planned for integrity, key storage, rotation, and observability.

Notes

- AES provides confidentiality, not automatic authenticity.
- Passwords should be hashed, not encrypted.
- Treat key management as part of the design, not an afterthought.
- If your runtime offers an authenticated encryption option you can support, prefer it when integrity is required.