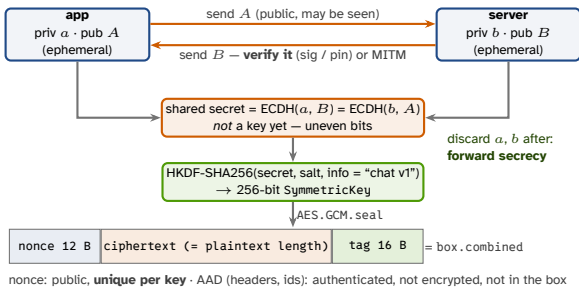
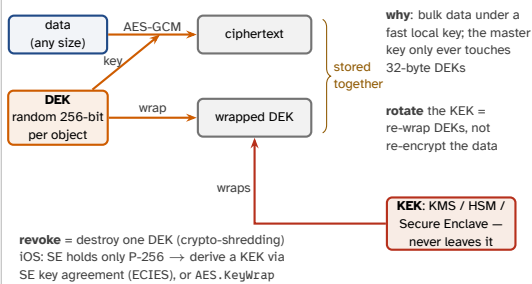


In one line: **Symmetric** AEAD (AES-GCM, ChaCha20-Poly1305) encrypts *and* authenticates bulk data with one shared key and a **unique nonce**; **asymmetric** keys (P-256, Curve25519) *agree* on that key (ECDH → HKDF) and *sign* (ECDSA, Ed25519). CryptoKit gives vetted primitives with safe defaults — your job is choosing them, managing keys and nonces, and **never inventing** a primitive, mode or protocol.

Agree → derive → seal (the shape of TLS 1.3)



Envelope encryption



How it works

- AEAD** = confidentiality + integrity + authenticity in one call; open throws on any flipped bit — never “decrypt then check”. **AES-GCM**: fastest with AES hardware (every Apple chip). **ChaCha20-Poly1305**: fast in pure software, constant-time without AES instructions. Both: 256-bit key, 96-bit nonce, 128-bit tag.
- Nonce reuse** under one key = catastrophe: same keystream, so $C_1 \oplus C_2 = P_1 \oplus P_2$, and in GCM it leaks the GHASH key → **forged tags**. Random 96-bit nonces (CryptoKit’s default) are safe up to $\sim 2^{32}$ messages per key; then rotate.
- ECDH**: combine your private with their public → the same secret on both sides. Run it through **HKDF** (extract-then-expand; salt + info bind it to a purpose) — one secret, many keys. **Ephemeral** keys give forward secrecy.
- Signatures**: sign with *your private*, anyone verifies with your public → authenticity, integrity, non-repudiation, *no* confidentiality. **ECDSA** (P-256) needs a unique per-signature nonce — reuse leaks the private key (PS3, 2010); **Ed25519** is deterministic. Encryption runs the other way: *their public* key.
- Key sizes**: P-256 / Curve25519 $\approx$ 128-bit security $\approx$ RSA-3072. CryptoKit has no RSA → SecKey.
- KDFs**: **HKDF** for high-entropy input (ECDH output, a master key) — fast, not for passwords. **PBKDF2** / Argon2id stretch a *password* slowly. CryptoKit has HKDF only; PBKDF2 = CCKeyDerivationPBKDF (CommonCrypto).

Example — the CryptoKit calls

```
import CryptoKit
let mine = Curve25519.KeyAgreement.PrivateKey() // ephemeral
let secret = try mine.sharedSecretFromKeyAgreement(with: theirPublic)
let key = secret.hkdfDerivedSymmetricKey(using: SHA256.self,
salt: salt, sharedInfo: Data("chat v1".utf8), outputByteCount: 32)
// AEAD: CryptoKit makes a random 12-byte nonce for you
let box = try AES.GCM.seal(msg, using: key, authenticating: header)
let wire = box.combined! // nonce | ciphertext | tag
let plain = try AES.GCM.open(AES.GCM.SealedBox(combined: wire),
using: key, authenticating: header)
// signatures; SecureEnclave.P256.Signing.PrivateKey() = same API
let signer = P256.Signing.PrivateKey()
let sig = try signer.signature(for: msg)
let ok = signer.publicKey.isValidSignature(sig, for: msg)
// MAC with its OWN key; the check is constant-time
let tag = HMAC<SHA256>.authenticationCode(for: msg, using: macKey)
let valid = HMAC<SHA256>.isValidAuthenticationCode(tag,
authenticating: msg, using: macKey)
```

What CryptoKit gives you (iOS 13+)

need	API
key	SymmetricKey(size: .bits256) — CSPRNG
AEAD	AES.GCM.seal/open, ChaChaPoly.seal/open
hash	SHA256/384/512.hash(data:); Insecure.MD5, Insecure.SHA1 — the namespace is the warning
MAC	HMAC<SHA256>
KDF	HKDF<SHA256>.deriveKey (iOS 14), SharedSecret.hkdfDerivedSymmetricKey
agreement	Curve25519.KeyAgreement, P256.KeyAgreement
signing	Curve25519.Signing (Ed25519), P256.Signing (ECDSA)
hardware	SecureEnclave.P256.Signing / .KeyAgreement; check SecureEnclave.isAvailable

Secure Enclave keys are P-256 only; the private key never leaves the chip — dataRepresentation is an *encrypted blob* only that device’s SE can use (store it in the Keychain). Operations can require biometry via SecAccessControl.

Symmetric vs asymmetric

	symmetric	asymmetric
keys	one shared secret	public + private pair
speed	GB/s (hardware AES)	$\sim 1000\times$ slower
for	bulk data, storage, sessions	key agreement, signatures, identity
problem	how to share the key	how to trust the public key (PKI, pinning)

Real systems are **hybrid**: asymmetric to agree or wrap a key, symmetric for the data (TLS, envelope encryption, ECIES).

Interview traps — the common mistakes

- ! **ECB** — equal blocks → equal ciphertext (the penguin). CBC without a MAC → malleable, padding oracles. Use AEAD.
- ! **Fixed or reset nonce** (a counter restarting after reinstall, two devices sharing a key + counter).
- ! **Hard-coded keys** — strings on the binary finds them. Keys are generated on device, kept in Keychain / Secure Enclave.
- ! **A password as a key** (SymmetricKey(data: pw.utf8)) — run a slow KDF first.
- ! **MAC compared with ==** — timing leaks the first bad byte.
- ! **One key, two jobs** (encrypt + MAC, two protocols) — derive separate keys with HKDF info.
- ! Unauthenticated ECDH = key agreement with the attacker (MITM).
- ! Key or nonce bytes from a seeded / custom PRNG (GameplayKit, a test generator) — use SymmetricKey(size:) / SecRandomCopyBytes.

Remember

Agree (ECDH) → derive (HKDF) → seal (AEAD, fresh nonce) → sign what must be proven. Don’t roll your own — not the primitive, the mode, nor the protocol.

Likely questions

- GCM nonce reused? — keystream reuse + tag forgery; rotate, random nonces.
- Why HKDF after ECDH? — raw secret is not uniform; bind keys to a purpose.
- Where does a Secure Enclave key live? — in the SE; the app holds a blob.
- Encrypt vs sign — which key? — their public / your private.
- Why envelope encryption? — cheap rotation, small master-key exposure.