Certificates and Cryptographic Methods

CEN429 Secure Programming — Week 10

Asst. Prof. Dr. Uğur CORUH · 20.11.2026

Today's Plan (3 Hours)

Hour	Section	Topic
1	0–3	Basic concepts · algorithm/key selection · modes/padding · MAC/HMAC
2	4–7	RSA/ECC · OAEP/PSS · digital signature · Diffie–Hellman · PKI
3	8–13	X.509 · chain · CRL/OCSP · HSM/PKCS#11 · post-quantum · project

A Brief History — Keys, Certificates, PKI

- **1976–77** — Diffie–Hellman and **RSA**: talking securely with someone you don't know
- **1988** — the **X.509** certificate format; identity carries a **CA signature**
- **1995** — commercial CAs and **PKI**; then **CRL** and **OCSP** (revocation)
- **2014** Heartbleed · **2015** Let's Encrypt · **2018** TLS 1.3 · **2022–24 PQC** (Kyber/Dilithium)

> Today's rules (the right mode/padding, chain validation, revocation) came out of these painful lessons.

Where Does This Week Fit?

- **Week 3:** introduction to cryptography (confidentiality, integrity).
- **This week (10):** choosing the right **algorithm/mode/padding**, the key lifecycle, **PKI**.
- **Week 11:** if the key is protected in software → whitebox.

Learning Outcome

This week is about **LO.2 / LO.4**.

By the end, you will be able to:

- Choose the right algorithm, mode, padding, and key length
- Spot the **pitfalls** of signatures and key exchange
- Correctly validate a certificate **chain**

Main Idea

> In cryptography, mistakes are usually not in the algorithm but in the **usage**: the wrong mode, the wrong padding, an unverified signature,
> an unchecked chain.

Today we learn the correct **usage**.

0. Basic Concepts (From Scratch)

Symmetric vs Asymmetric (Reminder)

- **Symmetric:** one key, encrypts and decrypts (AES). Fast.
- **Asymmetric:** a public + private key pair (RSA, ECC). Slower, but easy key distribution.
- In practice, **together**: carry the key with asymmetric, encrypt the data with symmetric.

Block Cipher and Mode

- **Block cipher:** encrypts a fixed-size block (AES: 16 bytes).
- **Mode: how** we chain the blocks together (CBC, GCM…).
- The choice of mode is the **heart** of security.

Padding

- **Padding:** filling data out when it is not a multiple of the block size.
- Required by some modes (CBC), **absent** from others (GCM).
- Handling padding incorrectly → the **padding oracle** attack.

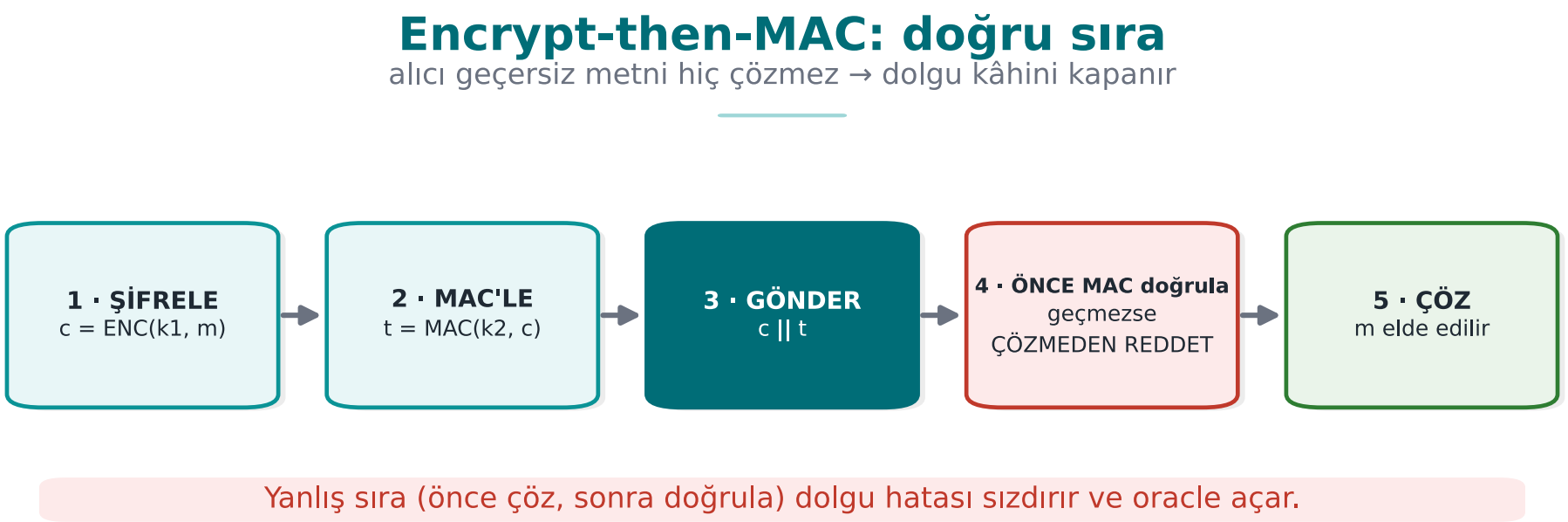
What Is AEAD?

- **AEAD (Authenticated Encryption with Associated Data):** encryption that gives confidentiality **and** integrity **together**.
- Example: AES-GCM, ChaCha20-Poly1305.
- Modern choice: don't bother with a separate MAC, use AEAD.

MAC and HMAC

- **MAC (Message Authentication Code):** a tag that proves a message **has not changed** and came from the right party (symmetric).
- **HMAC:** a common MAC built on a digest function.
- For integrity.

Encrypt-then-MAC — Diagram



Digest (Hash)

- **Digest:** a fixed-size fingerprint computed from data (SHA-256).
- One-way: you cannot get the data back from the digest.
- The building block of signatures and MACs.

Digital Signature

- **Signature:** asymmetric; sign with the **private** key, verify with the **public** key.
- Provides: integrity + **non-repudiation** (who signed it).
- Different from a MAC: asymmetric, anyone can verify.

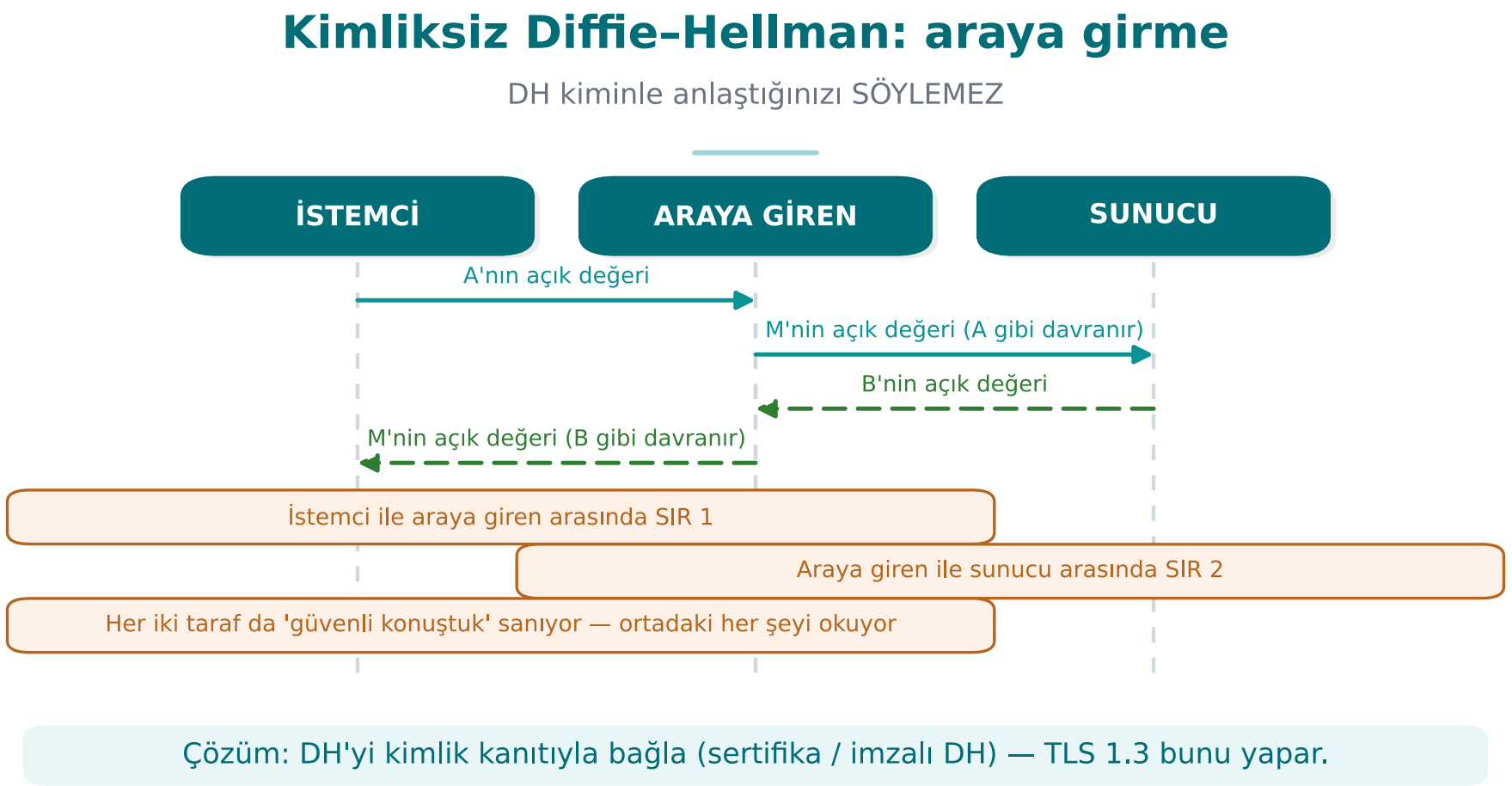
RSA and Elliptic Curve (ECC)

- **RSA:** the classic asymmetric algorithm; large keys (2048+ bits).
- **ECC:** elliptic curve; the same security with a **smaller** key (Ed25519, X25519).
- The modern choice is increasingly ECC.

Diffie–Hellman (DH)

- **DH:** two parties deriving a shared secret **without sharing** a private key.
- Key agreement over the network.
- Risk of **man-in-the-middle** (MITM) if identity is not authenticated.

Unauthenticated DH and MITM — Diagram



Çözüm: DH'yi kimlik kanıtıyla bağla (sertifika / imzalı DH) — TLS 1.3 bunu yapar.

PKI and CA

- **PKI (Public Key Infrastructure):** the trust system that answers "which public key belongs to whom?"
- **CA (Certificate Authority):** the trusted party that signs certificates.
- Root CA → intermediate CA → server certificate.

Certificate and X.509

- **Certificate:** a document, signed by a CA, that binds a public key to an identity (a domain name).
- **X.509:** the standard format for certificates.
- Contains: subject, public key, validity, signature, SAN.

CRL and OCSP

- **CRL (Certificate Revocation List):** a **list** of revoked certificates.
- **OCSP:** asking about a certificate's revocation status **on the spot**.
- The question "is this certificate still valid?"

HSM, PKCS#11, SoftHSM

- **HSM:** dedicated **hardware** that stores/operates keys; the key never leaves it.
- **PKCS#11:** the standard interface for talking to key modules.
- **SoftHSM:** a software simulation of an HSM (for testing).

Post-Quantum (PQC)

- **PQC (Post-Quantum Cryptography):** algorithms resistant to quantum computers.
- Today's RSA/ECC will be under threat in the future.
- Standardization is ongoing (e.g., ML-KEM).

Now We're Ready

Terms:

symmetric/asymmetric · block cipher/mode · padding · AEAD · MAC/HMAC · digest · signature · RSA/ECC · DH · PKI/CA · certificate/X.509 · CRL/OCSP · HSM/PKCS#11 · PQC

Now: choosing the right algorithm and key.

1. Algorithm and Key Selection

Current, Standard, Properly Used

Three rules:

- **Current:** an unbroken algorithm (AES, SHA-256, Ed25519).
- **Standard:** don't write your own crypto; use a proven library.
- **Properly used:** the right mode, padding, key management.

What to Avoid

- MD5, SHA-1 (for digests), DES/3DES, RC4.
- ECB mode.
- Your own "encryption" algorithm.
- A fixed/predictable IV or key.

Key Length

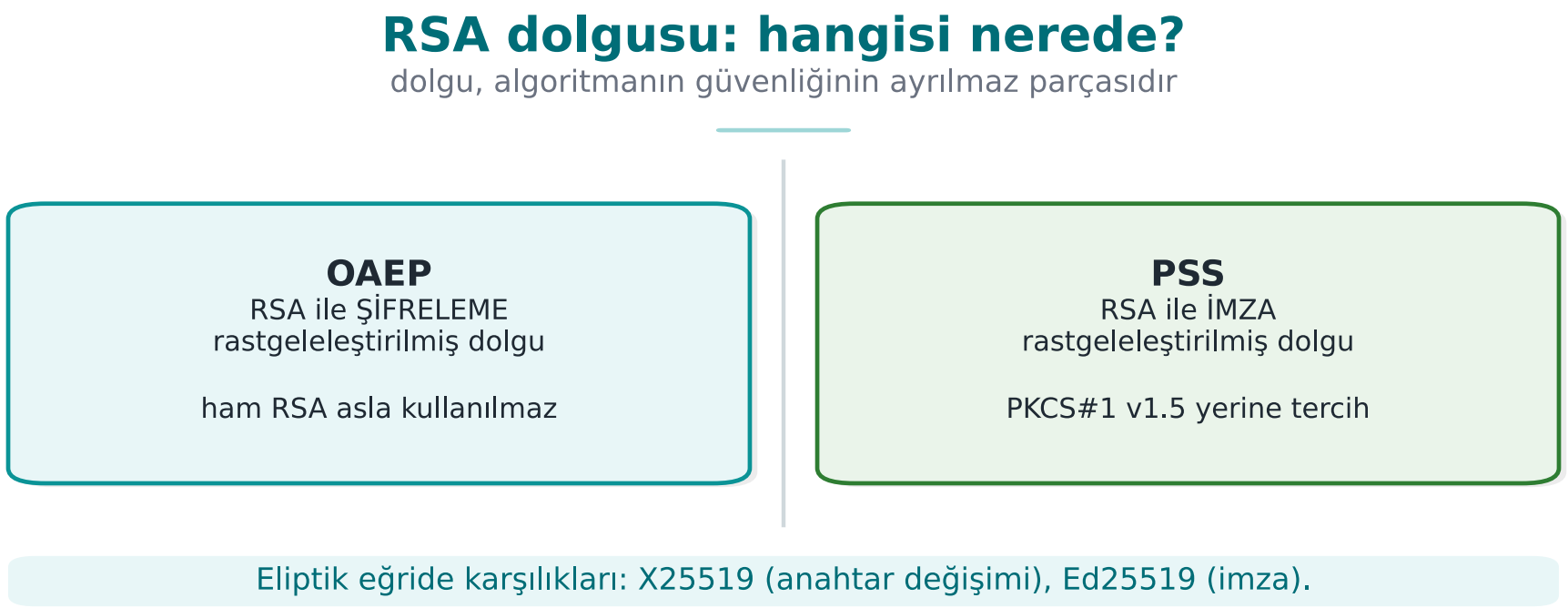
Purpose	Recommendation (approx.)
Symmetric	AES-128 (sufficient), AES-256
RSA	≥ 2048, prefer 3072
ECC	256-bit (≈ RSA-3072)
Digest	SHA-256+

The Weakest-Link Rule

- Choose components with **balanced** strength.
- Targeting 128 bits? Use RSA-3072 or ECC.

2. Block Cipher Modes and Padding

OAEP / PSS — Diagram



ECB · Never

- **ECB:** encrypts each block independently.
- Same block → same ciphertext block → **pattern leaks**.
- The famous "ECB penguin" example.

> Don't use ECB.

Why ECB Is Bad — Diagram

ECB neden asla kullanılmaz?

ünlü 'ECB pengueni' bu yüzden görünür kalır

ECB
her blok BAĞIMSIZ şifrelenir

aynı düz metin bloğu →
aynı şifreli blok
→ DESENLER görünür

CBC / CTR / GCM
her blok öncekine ya da
nonce'a bağlı

aynı metin → farklı şifreli
→ desen sızmaz

Kip seçimi şifrelemenin kendisi kadar önemlidir; varsayılan tercih AEAD (GCM).

CBC · Use with Care

- **CBC:** each block is chained with the previous one; an **IV** is required.
- The IV must be **random** and never repeated.
- Does **not provide integrity** on its own → a separate MAC is required.

CBC + Padding = Risk

- CBC requires padding.
- Handling padding incorrectly → **padding oracle** (coming up next).
- This is why the modern choice is **AEAD**.

PKCS#7 Padding · Example

Let's encrypt the 13-byte text `"MERHABA DUNYA"` with AES-128-CBC (K and IV are fixed here only for the demo):

```
printf 'MERHABA DUNYA' | openssl enc -aes-128-cbc -K "$K" -iv "$IV" -out cikti.bin
xxd -p cikti.bin
```

The output is **16 bytes** (32 hex characters) — the input was 13 bytes, so **3 bytes of padding** were added.

PKCS#7 Padding · Output

Let's look at the padding content with `-nopad` (for teaching purposes only):

```
4d45 5248 4142 4120 4455 4e59 4103 0303  MERHABA DUNYA...
```

The last three bytes are **`03 03 03`**: the number of missing bytes (16 − 13 = 3) worth of bytes, each with that value, were added — exactly the PKCS#7 rule. A normal `openssl enc -d` reads and strips these bytes automatically.

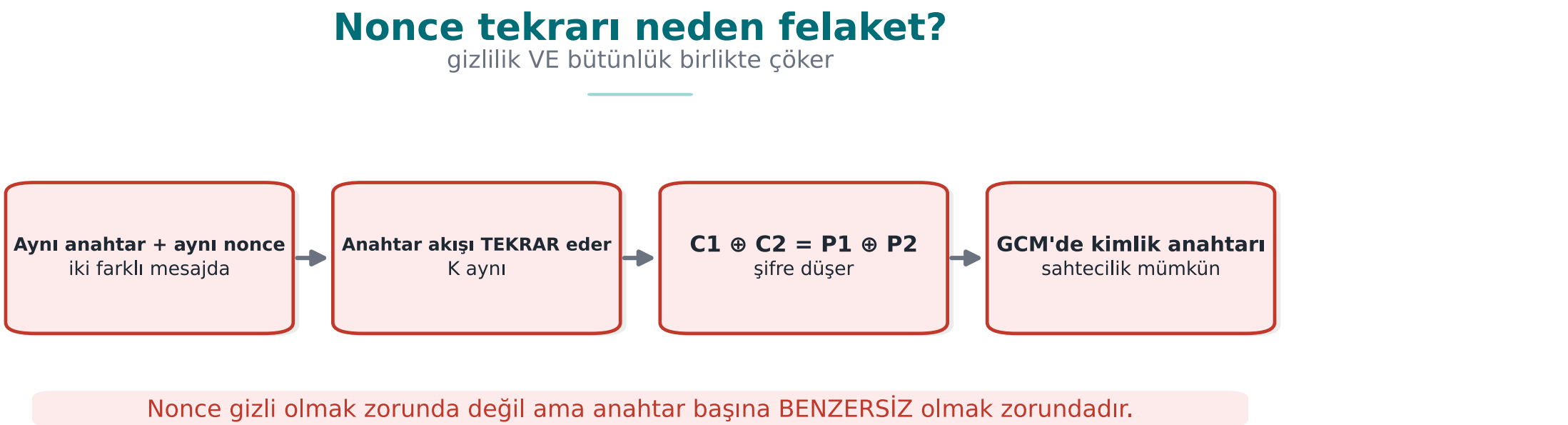
GCM · the Modern Choice (AEAD)

- **AES-GCM:** confidentiality **and** integrity together.
- **No** padding; a **nonce** (a number used once) is required.
- The nonce must **never** repeat (under the same key).

Nonce/IV Rule

- The IV/nonce must be **unique**.
- Nonce reuse in GCM is **catastrophic** (the key/data can leak).
- Use a counter or a random value (of sufficient length).

Nonce Reuse — Diagram



ChaCha20-Poly1305

- An AEAD alternative to AES-GCM.
- Fast when there is no hardware AES support.
- The same nonce rule applies.

Padding Oracle

The Problem · Step by Step

1. While decrypting CBC, the server gives a **different** response/timing for **invalid padding** versus **invalid MAC**.
2. The attacker modifies the ciphertext and watches the responses.
3. From the response difference, they decrypt the **plaintext** byte by byte.

Why Does It Happen?

- The error message/timing **leaks** internal state.
- The attacker uses this like an **oracle**.

The Fix

- **Use AEAD** (GCM): no padding, a single verification.
- If CBC is unavoidable: **encrypt-then-MAC** + **one and the same** error.
- A timing difference is a leak too → make it constant-time.

Lesson

> Encryption alone is not enough; **integrity** and a **consistent error** are required.

The padding oracle is the classic example of a "usage mistake."

Section 1–2 — Quick Check

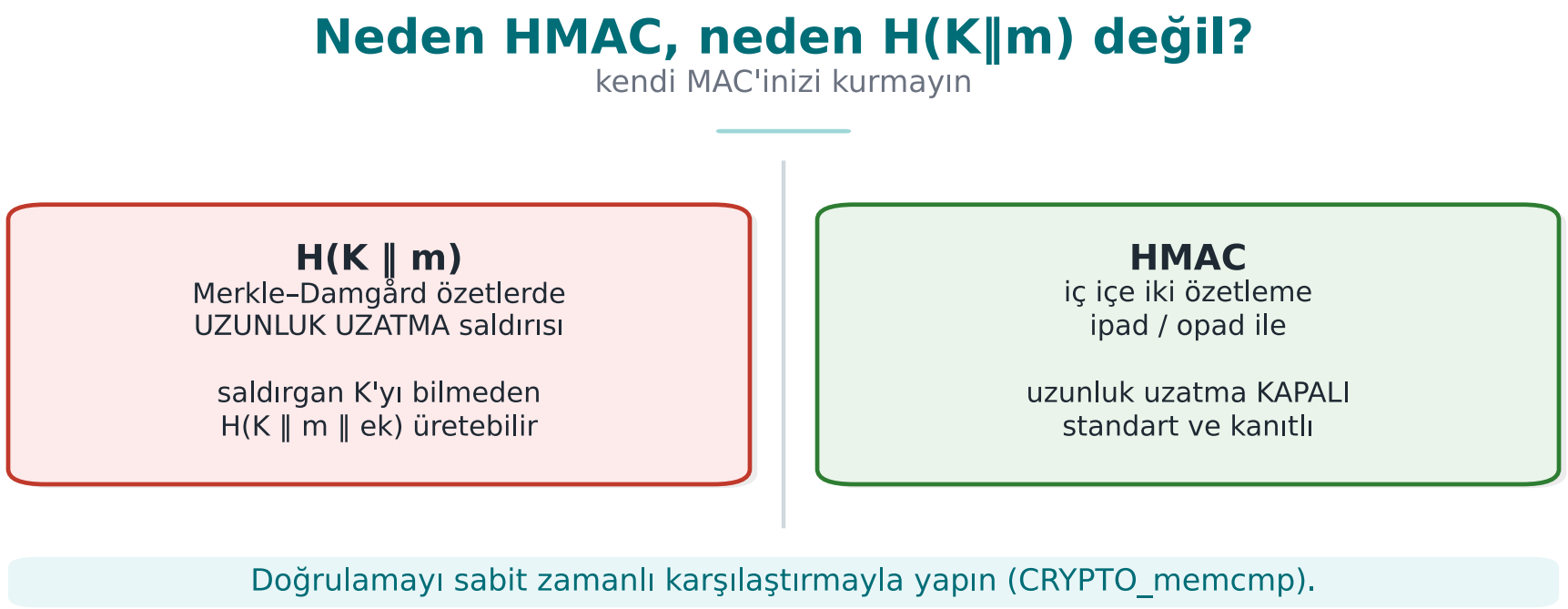
1. Explain the weakest-link rule with an example.
2. Why is ECB never used?
3. Why is nonce reuse catastrophic in GCM?
4. How does AEAD close off the padding oracle?

Section 1–2 — Answers

1. A system is only as secure as its **weakest component**. E.g., using AES-256 but storing the key in a plain file → the key is the weakest link.
2. **ECB** turns the same plaintext block into the same ciphertext block → the pattern leaks, no semantic security.
3. The same key+nonce repeats the **keystream** (confidentiality collapses), and the GHASH **authentication key** can be recovered → catastrophe.
4. **AEAD** verifies the tag first and **refuses to decrypt** text with an invalid tag → no oracle is left for a padding error.

3. MAC, HMAC, and Integrity

Why HMAC — Diagram



Why MAC?

- Encryption gives confidentiality, **not integrity**.
- An attacker can modify the ciphertext.
- MAC: the message **has not changed** and came from the right party.

HMAC

- A MAC built on a digest function (`HMAC-SHA256`).
- A symmetric key; both parties know the same key.
- Verified with a **constant-time** comparison.

HMAC-SHA-256 · Real Output

Let's compute the HMAC of a payment instruction (a 32-byte key, fixed here only for the demo):

```
printf 'tutar=100;alici=TR00' | \
  openssl dgst -sha256 -mac HMAC -macopt hexkey:$ANAHTAR
```

```
SHA2-256(stdin)= a08fb115...c5f9a1
```

HMAC-SHA-256 · Avalanche Effect

Using the same key, let's change only the amount (`100` → `900`):

```
SHA2-256(stdin)= 8a786eac...91a46be
```

A single-character change (`1` → `9`) makes the tag come out **completely different** — this is called the **avalanche effect**. Without knowing the key, an attacker cannot produce a valid new tag; the receiver catches the mismatch.

Constant-Time Comparison

```c
/* YANLIS: ilk farkta durur, sure farki sizdirir */
if (memcmp(hesaplanan, gelen, 32) == 0) { /* kabul */ }

/* DOGRU: her zaman 32 baytin tamamini gezer */
if (CRYPTO_memcmp(hesaplanan, gelen, 32) == 0) { /* kabul */ }
```

Rule: always compare secret values such as a MAC/signature/password digest with a constant-time comparison; `memcmp` / `==` leaves an open door to a timing attack.

This Section's Rule · MAC/HMAC

- Provide integrity with **HMAC**, not plain `H(K‖m)` (risk of length extension).
- When combining encryption + MAC, the order is **encrypt-then-MAC**.
- Tag/signature comparison must be **constant-time**; don't use a function with an early exit.

The Correct Order · Encrypt-then-MAC

```
1. şifrele:  c = ENC(k1, m)
2. MAC'le:   t = MAC(k2, c)
3. gönder:   c || t
```

- Encrypt first, **then** MAC the ciphertext.
- If the MAC does not pass, **don't even attempt** to decrypt.

Wrong Orders

- **MAC-then-encrypt:** open to the padding oracle.
- **Encrypt-and-MAC:** the MAC can leak the plaintext.
- The correct way: **encrypt-then-MAC** (or AEAD, which already does this).

Replay

- The attacker resends a valid message **again**.
- The MAC is valid (the message hasn't changed), but the operation is **repeated**.
- Fix: **nonce**, timestamp, counter.

AEAD Does It All

- AES-GCM: encryption + integrity **together**.
- "Associated data" (AAD) also authenticates headers/context.
- Modern choice: AEAD instead of a separate MAC.

4. Asymmetric Cryptography: RSA and ECC

Symmetric ↔ Asymmetric — Diagram

Simetrik ve asimetrik: hangisi ne için?

pratikte HİBRİT: asimetrikle anahtar kur, simetrikle veri şifrele

SİMETRİK (AES)
tek anahtar
ÇOK HIZLI

sorun: anahtarı nasıl paylaşırsın?

ASİMETRİK (RSA/EC)
açık + gizli anahtar çifti
YAVAŞ

anahtar dağıtımı ve İMZA çözülür

TLS tam olarak bunu yapar: ECDHE ile anahtar, AES-GCM ile veri.

RSA · Two Jobs

- **Encryption:** encrypt with the public key, decrypt with the private key.
- **Signing:** sign with the private key, verify with the public key.
- Large keys (2048+).

RSA · Padding Is Required

- Raw RSA is **insecure**.
- **For encryption:** OAEP padding.
- **For signing:** PSS padding.
- Old PKCS#1 v1.5: avoid where possible.

OAEP vs PSS

- **OAEP:** RSA **encryption** padding.
- **PSS:** RSA **signature** padding.
- Common mix-up: OAEP is not for signing, PSS is not for encryption.

OAEP's Randomness · Command

Let's encrypt the same message twice with OAEP:

```
openssl pkeyutl -encrypt -pubin -inkey rsa_acik.pem -in kisa.txt -out c1.bin \
    -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha256
openssl pkeyutl -encrypt -pubin -inkey rsa_acik.pem -in kisa.txt -out c2.bin \
    -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha256
cmp c1.bin c2.bin && echo AYNI || echo FARKLI
```

OAEP's Randomness · Result

```
c1.bin c2.bin differ: char 1, line 1
FARKLI
```

Same key, same plaintext, but **different** ciphertext — OAEP mixes in a fresh random value on every encryption. **Raw RSA** (without padding) is deterministic (`c = m^e mod n`); that is why it is never used directly for encryption.

ECC · Why?

- The same security with a **smaller** key (256-bit ≈ RSA-3072).
- Faster, less space.
- Ideal for mobile/embedded.

Ed25519 and X25519

- **Ed25519:** the modern **signature** algorithm.
- **X25519:** modern **key agreement** (DH).
- Common mix-up: Ed25519 signs, X25519 exchanges keys.

RSA-3072 vs Ed25519 · Measurement

Both at roughly 128 bits of security (see the table in Section 1):

```
wc -c rsa_acik.pem ed_acik.pem
wc -c belge.rsa.sig belge.ed.sig
```

RSA-3072 vs Ed25519 · Result

File	RSA-3072	Ed25519
Public key (PEM)	636 bytes	116 bytes
Signature	384 bytes	64 bytes

The public key is **~5.5 times** smaller, the signature **6 times** smaller. In IoT/mobile handshakes, in embedded flash, or on a blockchain, this difference adds up.

Symmetric + Asymmetric Together

```
1. X25519 ile ortak sır türet
2. Ondan bir AES anahtarı çıkar (KDF)
3. AES-GCM ile veriyi şifrele
```

- Asymmetric: carry the key. Symmetric: encrypt the data.
- TLS does exactly this.

Section 3–4 — Quick Check

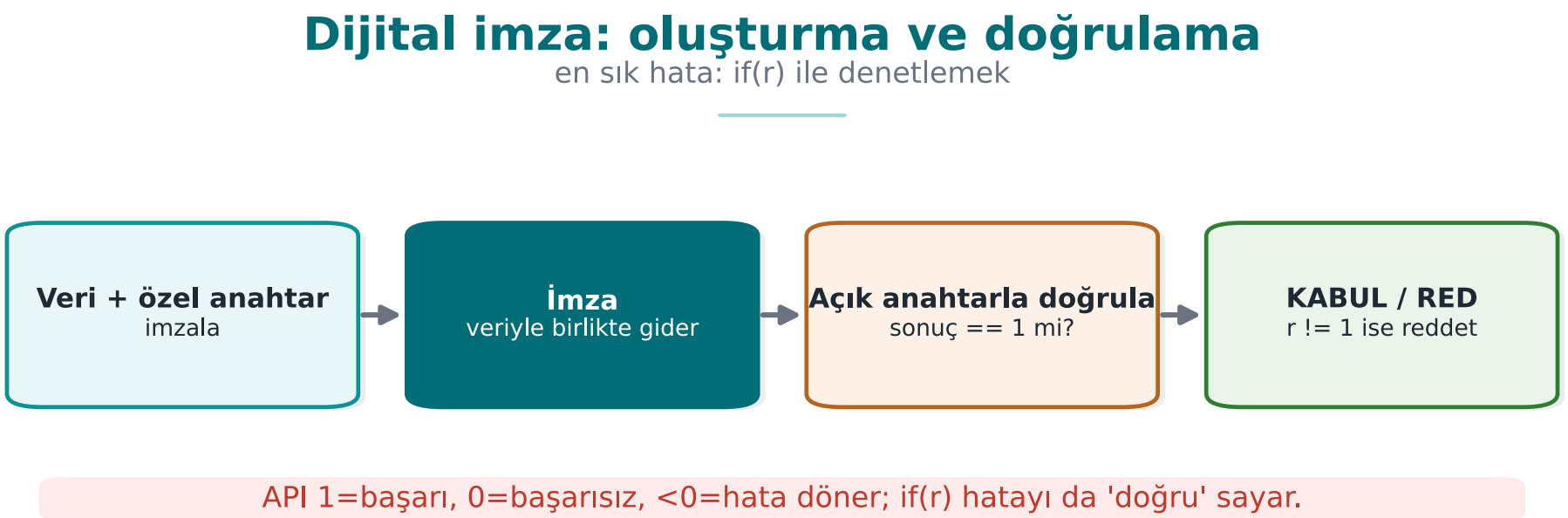
1. Why is encrypt-then-MAC the correct order?
2. How is replay prevented?
3. Which is OAEP and which is PSS for?
4. What's the difference between Ed25519 and X25519?

Section 3–4 — Answers

1. The receiver verifies the **MAC first**; if it fails, it does not decrypt → a modified ciphertext is never processed (closes off the padding oracle).
2. **Freshness:** a nonce/counter, a timestamp + window, a one-time challenge.
3. **OAEP** = RSA **encryption** padding; **PSS** = RSA **signature** padding.
4. **Ed25519** = signing (EdDSA); **X25519** = key exchange (ECDH). Same curve family, different job.

5. Digital Signatures and Pitfalls

Digital Signature — Diagram



Signature · What Does It Provide?

- **Integrity:** the message has not changed.
- **Identity/non-repudiation:** the private key holder signed it.
- Verification: with the public key, anyone can do it.

Signature · Where Is It Used?

- Signing software/updates.
- Certificates (CA signature).
- Signing documents/transactions.
- Server identity in TLS (CertificateVerify).

Pitfall 1 · Not Checking the Return Value

```
int r = EVP_DigestVerify(ctx, imza, n, veri, m);
if (r) guncellemeyi_kur();   /* HATALI */
```

A negative error value is also treated as "true." The correct check is `r == 1`.

Pitfall 2 · Not Including the Version in the Signature

- If the signature covers only the content, an attacker can install an **old** but validly signed version (downgrade).
- The signed content must also cover the **version number**; an old version must be rejected.

Pitfall 3 · Verifying with the Wrong Key

- Is the verification key trusted? Where did it come from?
- If you verify with the attacker's key, their signature is "valid."
- The key must be **pinned**, or come from a trusted chain.

Signature · the Digest Rule

- A signature is really the signing of a **digest**.
- A weak digest (MD5/SHA-1) → a weak signature.
- Use SHA-256+.

6. Diffie–Hellman and MITM

DH · What Does It Do?

- Two parties derive a shared secret **without sharing** a private key.
- Key agreement over the network.
- The modern form: X25519.

DH · Step by Step (Concept)

```
Ali:  a gizli, A = g^a açık gönderir
Veli: b gizli, B = g^b açık gönderir
Ortak sır: A^b = B^a = g^(ab)
```

An eavesdropper sees `g^a` and `g^b` but cannot compute `g^(ab)`.

X25519 · Both Sides Reach the Same Secret

```
openssl genpkey -algorithm X25519 -out alice.key
openssl genpkey -algorithm X25519 -out bob.key
openssl pkey -in alice.key -pubout -out alice.pub
openssl pkey -in bob.key -pubout -out bob.pub

openssl pkeyutl -derive -inkey alice.key -peerkey bob.pub -out alice_sir.bin
openssl pkeyutl -derive -inkey bob.key -peerkey alice.pub -out bob_sir.bin
```

X25519 · Result

```
8c53744a000c1a6f...bbc1376
8c53744a000c1a6f...bbc1376
AYNI
```

Alice and Bob, without ever knowing each other's private key, reached the **same** 32-byte secret by exchanging only public keys over the network. This raw secret is not used directly as an AES key; it is first passed through a **KDF** (HKDF) to derive session keys.

Unauthenticated DH · MITM Risk

- If Alice and Bob don't **verify** each other's identity...
- An attacker **sits in the middle**: doing DH separately with Alice and separately with Bob.
- Listening to/modifying both.

MITM · Diagram

```
Ali ⇔ [Saldırgan] ⇔ Veli
     iki ayrı DH; saldırgan ortada
```

Both sides think "I've established a secure channel."

The Fix · Authenticated DH

- The DH values are **signed** with a key whose identity is known.
- `CertificateVerify` in TLS 1.3.
- The other side **verifies** this identity (certificate chain).

Forward Secrecy

- A **new** DH key for every session.
- Even if the long-term key leaks, **old** sessions cannot be decrypted.
- Modern TLS provides this.

Section 5–6 — Quick Check

1. Why is `if (r)` wrong in signature verification?
2. Why must the version be part of the signature?
3. How does MITM happen against unauthenticated DH? The fix?
4. What does forward secrecy provide?

Section 5–6 — Answers

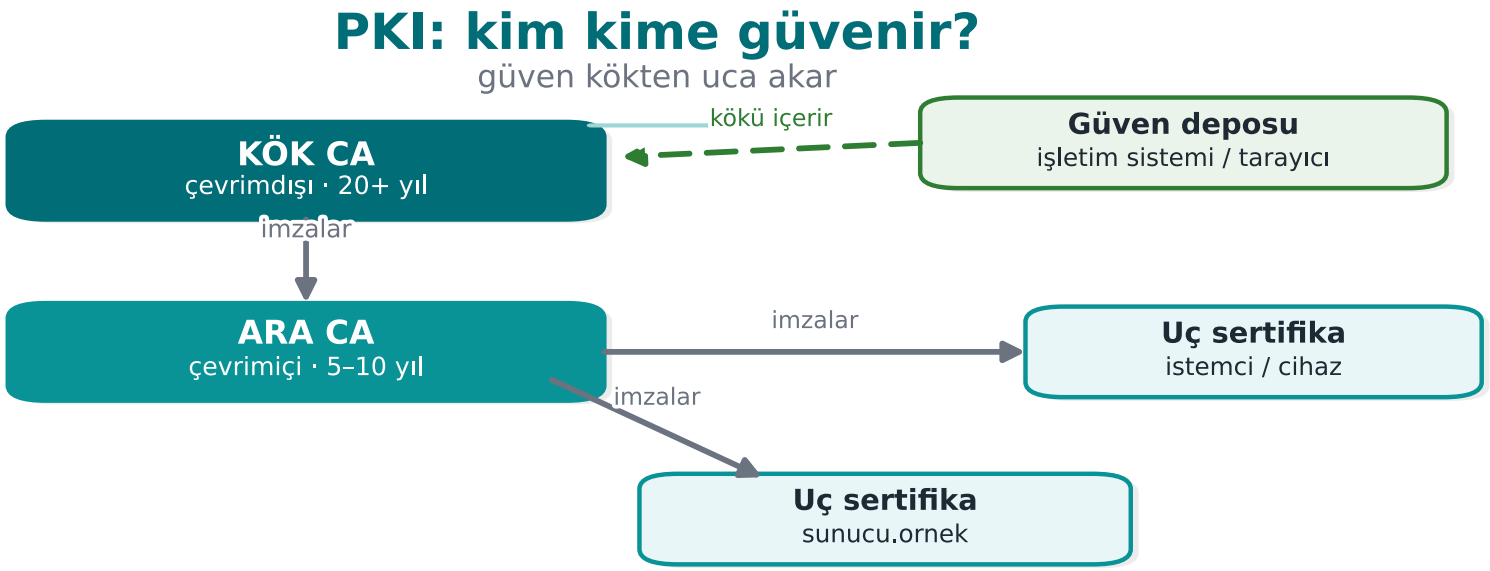
1. Verification returns 1=success, 0=failure, <0=error; `if (r)` also treats the error (−1) as "true." Check for `r == 1`.
2. Otherwise an attacker can present an **old/insecure version** as "valid" with the same signature (downgrade). The version must be part of the signed data.
3. The attacker establishes a separate key with each side. Fix: bind DH to **authentication** (signed DH / a certificate).
4. Each session uses an **ephemeral** key; even if the long-term key leaks, **past sessions** cannot be decrypted.

7. Public Key Infrastructure (PKI)

The Problem · Who Do We Trust?

- You've seen a public key. Does it really belong to that person/site?
- A malicious actor can present their own key as "the bank."
- PKI solves this trust problem.

Chain of Trust



Each level **signs** the one below it.

Chain Validation — Diagram

Eksik ara sertifika zinciri kurdurmaz; fail-open ile 'geçti' saymak en sık hatadır.

Root and Intermediate CA

- **Root CA:** offline, self-signed, heavily protected.
- **Intermediate CA:** issues day-to-day certificates.
- If the root leaks, it's a **catastrophe**; this is why an intermediate CA is used.

Trust Store

- The OS/browser carries the trusted **root CAs**.
- If a chain reaches one of these roots, it is trusted.
- If it can't, it is **untrusted**.

8. The X.509 Certificate

X.509 Fields — Diagram

What's Inside?

- **Subject:** who it belongs to (domain name).
- **Public key.**
- **Validity:** start/end.
- **Issuer:** which CA signed it.
- **Signature.**

SAN · Name Checking

- **SAN (Subject Alternative Name):** the domain names the certificate is valid for.
- Name checking is done against the **SAN, not the CN**.
- A certificate for `example.com` must not be valid for `baska.com`.

Real Certificate · Top-Level Fields

```
Version: 3 (0x2)
Serial Number: 35:e2:f6:8d:...:76:e7
Signature Algorithm: ecdsa-with-SHA256
Issuer: CN=CEN429 Lab Ara CA
Validity: Sep 23 2026 – Dec 22 2026
Subject: CN=localhost
Subject Public Key Info: 256 bit, NIST CURVE: P-256
```

Real `openssl x509 -text` output; each field carries the answer to one of the "four questions" (next section).

Real Certificate · Extensions

```
X509v3 Basic Constraints: CA:FALSE
X509v3 Key Usage: critical, Digital Signature
X509v3 Extended Key Usage: TLS Web Server Authentication
X509v3 Subject Alternative Name:
    DNS:localhost, IP Address:127.0.0.1
X509v3 Authority Key Identifier: F5:BA:39:86:...
```

The `Authority Key Identifier` is the **fingerprint** of the CA that signed the certificate; when building the chain, it prevents confusion with a "fake intermediate CA with the same name but a different key."

9. Chain Validation

Four Questions

When validating a certificate chain:

1. Is the **signature** valid? (at every level)
2. Does the **chain** reach a trusted root?
3. Has the **validity period** not expired?
4. Does the **name** match (SAN)?

All four must be **yes**.

Question 1 · Chain (Issuer/Subject)

```
openssl x509 -in sunucu.crt -noout -issuer -subject
openssl x509 -in ara.crt    -noout -issuer -subject
openssl x509 -in kok.crt    -noout -issuer -subject
```

```
issuer=CN=...Ara CA   subject=CN=localhost
issuer=CN=...Kok CA   subject=CN=...Ara CA
issuer=CN=...Kok CA   subject=CN=...Kok CA
```

The chain reads: `sunucu`'s issuer = `ara`'s subject; the root signs **itself**.

Question 2 · Validity

```
openssl x509 -in sunucu.crt -noout -dates
openssl x509 -in sunucu.crt -noout -checkend 0
```

```
notBefore=Sep 23 2026 GMT
notAfter=Dec 22 2026 GMT
Certificate will not expire
```

`-checkend 0`: "has it expired as of right now?" TLS clients ask this automatically on every connection.

Question 3 · Usage (CA:TRUE/FALSE)

```
openssl x509 -in sunucu.crt -noout -ext basicConstraints,keyUsage
openssl x509 -in ara.crt    -noout -ext basicConstraints,keyUsage
```

```
sunucu.crt:  CA:FALSE            (baska sertifika imzalayamaz)
ara.crt:     CA:TRUE, pathlen:0  (yalniz uc sertifika imzalayabilir)
```

Question 4 · Name (SAN)

```
X509v3 Subject Alternative Name:
    DNS:localhost, IP Address:127.0.0.1
```

The client compares the address it connected to against this list — **not** the `CN` in `Subject`.

A client library asks these four questions **on your behalf**; turning off `verify` or swallowing errors means none of the four questions are ever asked.

With OpenSSL · Example

```
openssl verify -CAfile kok.crt \
    -untrusted ara.crt sunucu.crt
```

- `-CAfile` : the trusted root.
- `-untrusted` : the intermediate certificate(s).

Common Mistake · Missing Intermediate Certificate

- If the server doesn't send the **intermediate** certificate, the chain can't reach the root.
- `verify` fails; it passes with `-untrusted ara.crt`.
- A very common configuration mistake in the field.

SPKI Pinning

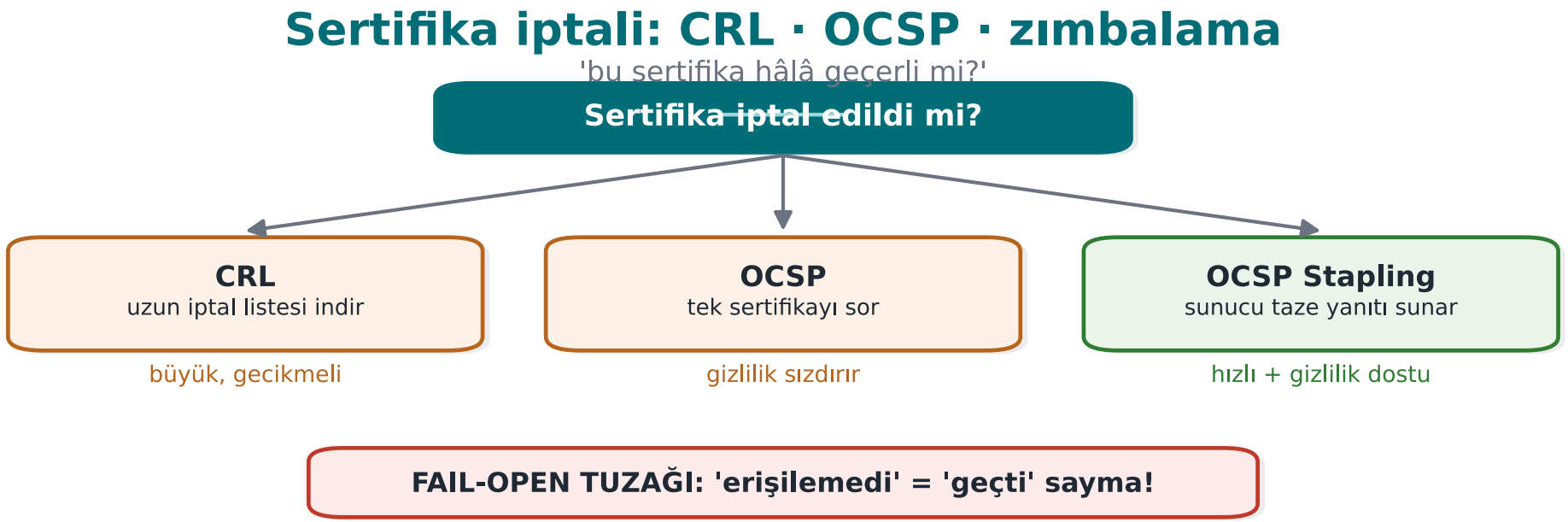
- The application embeds the **digest** of the expected server key.
- Even a fake but "valid" certificate is not accepted.
- A **backup pin** is required (so you're not locked out when the key changes).

This Section's Rule · Chain Validation

- Don't skip any of the chain validation's **four questions** (signature, validity, usage, name).
- `verify` does not check the name — **SAN** checking must be done separately.
- Even a cryptographically correct chain can't connect if the server doesn't send the **intermediate certificate**.

10. Certificate Revocation

Certificate Revocation — Diagram



Why Revoke?

- If a certificate's private key leaks, it must be **revoked** before it expires.
- A revoked certificate must not be accepted.

CRL

- **CRL:** a **periodic list** of revoked certificates.
- Can be large, and can be out of date.

OCSP

- **OCSP:** asking about a certificate's status **on the spot**.
- More current; but has privacy/speed issues.

OCSP Stapling

- The server fetches the OCSP response **itself** and presents it along with the certificate.
- Gains privacy + speed.

CRL Workflow · Revoke and Generate

```
openssl ca -config ara.cnf -revoke sunucu.crt
openssl ca -config ara.cnf -gencrl -out ara.crl
```

```
Revoking Certificate 35E2F68D...76E7.
Database updated
```

The certificate file **does not change**; only a revocation record is added to the CA's ledger (`index.txt`). `-gencrl` turns this record into a list **signed** with the CA's own key.

Validating with a CRL · Result

```
openssl verify -crl_check -CAfile kok.crt \
    -untrusted ara.crt -CRLfile ara.crl sunucu.crt
```

```
error 23 at 0 depth lookup: certificate revoked
error sunucu.crt: verification failed
```

Even though the certificate **has not yet expired**, it is rejected because it is listed in the CRL — revocation checking works **independently** of validity checking.

⚠️ The Fail-Open Trap

- **Accepting** a certificate when OCSP is unreachable = fail-open.
- An attacker can block OCSP and slip a revoked certificate through.
- Mitigation: **Must-Staple**, short-lived certificates.

Section 7–10 — Quick Check

1. What are the four questions of chain validation?
2. What is name checking done against (CN or SAN)?
3. Why does a missing intermediate certificate break `verify`?
4. What is fail-open, and how is it mitigated?

Section 7–10 — Answers

1. (1) is the chain **valid** to the root, (2) is it within its **validity** period, (3) does it have the right **purpose/constraint** (CA:TRUE, KeyUsage/EKU), (4) has it been **revoked** (CRL/OCSP).
2. **SAN** (Subject Alternative Name); CN is no longer used.
3. The chain **cannot be built** to the root; the trust path is never completed → failure. The server must also send the **intermediate** certificate.
4. **Fail-open:** counting an error/unreachability as "passed." Mitigation: **fail-closed**, OCSP stapling / must-staple, reject on error.

11. Storing Keys in Hardware

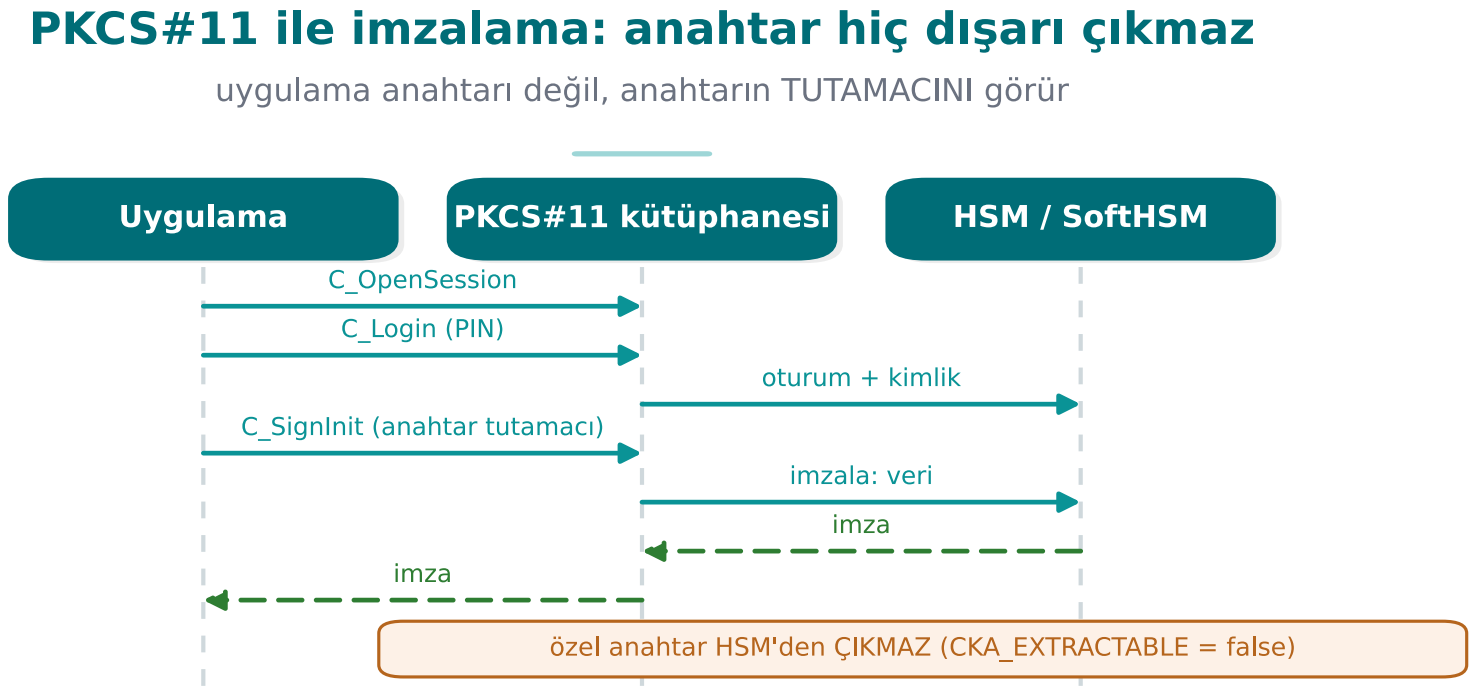
What Is an HSM?

- **HSM:** dedicated hardware that stores/operates keys.
- The key never leaves the HSM; you say "sign this," and the result comes back.
- Tamper-resistant.

PKCS#11

- The **standard interface** for talking to key modules.
- The application never **sees** the value of the key.
- HSMs and software simulations expose this same interface.

Signing with PKCS#11 — Diagram



Anahtar çalınamaz; ama HSM'e erişimi olan imzalatabilir — erişim denetimi yine şarttır.

SoftHSM

- A **software simulation** of an HSM; the same PKCS#11 interface.
- For development/testing; **no real hardware protection**.
- A real HSM in production.

Why Hardware?

- In software, the key is exposed to a whitebox attacker (Week 11).
- In hardware, the key is isolated → far stronger.
- If possible, never put the key in software at all.

Key Lifecycle

Stage	What's Done
Generation	Strong randomness (CSPRNG)
Storage	HSM/TEE or protected
Use	Least privilege, constant-time
Crypto-period	A lifespan limit
Renewal	Regular
Destruction	Secure erasure from memory

This Section's Rule · Key Storage

- When generating a valuable key, `CKA_EXTRACTABLE=false` must be set **explicitly**; don't rely on the default.
- On an HSM the key never leaves; the application only says "sign" through a **handle**.
- SoftHSM is for testing; production requires a real HSM/hardware protection.

12. Post-Quantum Cryptography

Post-Quantum — Diagram

The Threat

- A sufficiently powerful quantum computer could break today's **RSA/ECC**.
- The "harvest now, decrypt later" attack: store encrypted data today, decrypt it in the future.

PQC · Approach

- Quantum-resistant algorithms (e.g., ML-KEM key encapsulation).
- Standardization is ongoing.
- **Crypto agility:** a design that lets you swap the algorithm easily.

PQC Sizes · Command

OpenSSL 3.5 can generate the standardized PQC algorithms **today**:

```
openssl genpkey -algorithm ML-KEM-768 -out mlkem.pem
openssl genpkey -algorithm ML-DSA-65  -out mldsa.pem
openssl pkeyutl -sign -inkey mldsa.pem -rawin -in belge.txt -out belge.mldsa.sig
wc -c mlkem_pub.pem mldsa_pub.pem belge.mldsa.sig
```

PQC Sizes · Result

Algorithm	Public Key	Signature
Ed25519 (classical)	116 bytes	64 bytes
RSA-3072 (classical)	636 bytes	384 bytes
ML-KEM-768 (PQC)	1,714 bytes	—
ML-DSA-65 (PQC)	2,770 bytes	3,309 bytes

ML-DSA-65: the public key is **~24 times** larger than Ed25519's, the signature **~52 times** larger. TLS's **hybrid** (classical + PQC) approach exists to manage this transition cost.

What Should You Do Today?

- Be crypto-agile (don't hardcode the algorithm).
- Watch PQC for long-lived secrets.
- No panic; but prepare.

13. Project: This Week (S8, S11)

Project · S8/S11

- [ ] **Algorithm inventory:** purpose, algorithm, mode, key length, library.
- [ ] A **key lifecycle** table.
- [ ] TLS/certificate validation; pinning + backup pin if applicable.
- [ ] Signature verification (on the update file).

Solved Self-Check

Question 1

What is the security level of an RSA-2048 + AES-256 system?

Answer: ~112 bits (RSA-2048 is the weakest link). For 128 bits, use RSA-3072 or ECC.

Question 2

"Invalid padding" and "invalid MAC" as separate messages: what's the risk?

Answer: A padding oracle; the attacker can decrypt the message by watching the responses. Use one and the same error; prefer AEAD.

Question 3

Why is `if (EVP_DigestVerify(...))` wrong?

Answer: A negative error value also counts as true; the check should be `== 1`. Also, the signed content must cover the version.

Question 4

Why does `verify` fail without the intermediate certificate?

Answer: The chain can't reach the root; the intermediate certificate is missing. In the field, this usually means the server isn't sending the intermediate certificate.

Question 5

Accepting when OCSP is unreachable: what pattern is this, and how is it mitigated?

Answer: Fail-open (a soft failure). Must-Staple or short-lived certificates.

Question 6

How is MITM against unauthenticated DH prevented?

Answer: The DH values are signed with a key whose identity is known (TLS 1.3 CertificateVerify), and the other side verifies it.

Question 7

What are the four questions of chain validation?

Answer: Is the signature valid, does it reach a trusted root, has the validity period not expired, does the name (SAN) match.

Question 8

Why shouldn't the key be placed in software? Alternative?

Answer: A whitebox attacker can extract a key embedded in software. Alternative: HSM/TEE (PKCS#11); otherwise whitebox + a layer (Week 11).

Glossary

Term	Meaning
AEAD	Encryption + integrity together
Encrypt-then-MAC	The correct order
OAEP/PSS	RSA encryption/signature padding
Ed25519/X25519	Signature / key agreement
SAN	Certificate name checking
CRL/OCSP	Revocation list / on-the-spot query

Summary: This Week in One Sentence

> In cryptography, security lies less in the right algorithm than in the right **usage**: AEAD, the right padding, a verified signature, a checked
> chain, and a well-managed key.

Next Week

Week 11 — Whitebox Cryptography

What happens when the key is protected in software? The whitebox attacker, table-based WBC, and its limits.

Appendix A · Building a Chain with OpenSSL (Step by Step)

Goal

Let's build a small chain:

Root CA → Intermediate CA → Server certificate.

Then let's validate it with `verify` and see the typical mistake.

Step 1 · Root CA (Self-Signed)

```
openssl req -x509 -newkey ed25519 \
  -keyout kok.key -out kok.crt \
  -subj "/CN=Ders Kok CA" -days 3650 -nodes
```

The root signs itself; it is kept offline.

Step 2 · Intermediate CA (Signed by the Root)

```
openssl req -newkey ed25519 -keyout ara.key \
  -out ara.csr -subj "/CN=Ders Ara CA" -nodes
openssl x509 -req -in ara.csr -CA kok.crt -CAkey kok.key \
  -CAcreateserial -out ara.crt -days 1825
```

Step 3 · Server Certificate (Signed by the Intermediate)

```
openssl req -newkey ed25519 -keyout sunucu.key \
  -out sunucu.csr -subj "/CN=ornek.test" -nodes
openssl x509 -req -in sunucu.csr -CA ara.crt -CAkey ara.key \
  -CAcreateserial -out sunucu.crt -days 365
```

Step 4 · Validate (Missing Intermediate)

```
openssl verify -CAfile kok.crt sunucu.crt
# HATA: unable to get local issuer certificate
```

The chain can't reach the root: the **intermediate certificate is missing**.

Step 5 · Validate (With the Intermediate)

```
openssl verify -CAfile kok.crt \
  -untrusted ara.crt sunucu.crt
# OK
```

Given the intermediate certificate, the chain is complete.

Lesson

- The server must also send the **intermediate** certificate.
- If it's missing, the client says "issuer not found."
- The most common TLS configuration mistake in the field.

The Chain's Four Questions · in Practice

- **Signature:** each level signed by the one above ✓
- **Root:** `-CAfile kok.crt` trust store ✓
- **Validity:** within `-days` ✓
- **Name:** SAN checking in the application (verify does not check the name!)

Appendix B · A Crypto Design Case Study

Scenario

A mobile app:

- Encrypts sensitive data locally.
- Talks to a server securely.
- Verifies updates.

Let's design the crypto.

Local Data Encryption

- **AES-256-GCM** (AEAD): confidentiality + integrity.
- A counter or random nonce, **never repeated**.
- The key in TEE/HSM; otherwise whitebox (Week 11).

Server Communication

- **TLS 1.3**: X25519 key agreement + AEAD.
- Certificate chain validation + (if applicable) SPKI pinning + a backup pin.
- Not fail-open.

Update Verification

- An **Ed25519** signature; `verify == 1`.
- The signed content covers the **version**; downgrade rejected.
- The verification key is pinned/embedded (protected by integrity).

Key Inventory (S8)

Key	Purpose	Alg/Length	Storage
Data key	Local encryption	AES-256	TEE/whitebox
Session key	TLS	X25519-derived	Memory, short-lived
Signature verification	Update	Ed25519 public	Embedded

Appendix C · Decision Tables

Which Mode?

Need	Choice
Confidentiality + integrity	AES-GCM / ChaCha20-Poly1305
Speed only (no hardware AES)	ChaCha20-Poly1305
Never	ECB
Legacy CBC system	+ encrypt-then-MAC, a single error

Which Asymmetric Algorithm?

Need	Choice
Signature (modern)	Ed25519
Key agreement	X25519
RSA encryption	RSA-OAEP (≥3072)
RSA signature	RSA-PSS

Decision Rule

- Choose current, standard, properly used.
- AEAD first; ECC first.
- Write the decision and its rationale into **S8**.

Appendix D · Ten Common Crypto Mistakes

Mistake 1 · Writing Your Own Crypto

- **Wrong:** your own "encryption" algorithm.
- **Right:** a proven library, a standard algorithm.

Mistake 2 · ECB Mode

- **Wrong:** ECB (pattern leaks).
- **Right:** AEAD (GCM / ChaCha20-Poly1305).

Mistake 3 · IV/Nonce Reuse

- **Wrong:** a fixed or repeating nonce.
- **Right:** a unique, non-repeating nonce.

Mistake 4 · Encryption Without Integrity

- **Wrong:** CBC alone, no MAC.
- **Right:** AEAD or encrypt-then-MAC.

Mistake 5 · Raw RSA / Wrong Padding

- **Wrong:** raw RSA, PKCS#1 v1.5.
- **Right:** OAEP (encryption), PSS (signing).

Mistake 6 · Weak Digest

- **Wrong:** MD5, SHA-1.
- **Right:** SHA-256+.

Mistake 7 · Not Checking the Signature Return Value

- **Wrong:** `if (verify(...))`.
- **Right:** `== 1`, and the version is covered.

Mistake 8 · Not Checking the Chain/Name

- **Wrong:** looking only at the signature.
- **Right:** the four questions + SAN checking.

Mistake 9 · Fail-Open Revocation

- **Wrong:** accepting when OCSP is unreachable.
- **Right:** Must-Staple, short lifetimes.

Mistake 10 · Embedding the Key in Plaintext

- **Wrong:** `static uint8_t k[16]`.
- **Right:** HSM/TEE; otherwise whitebox + a layer (Week 11).

Appendix E · Setting Up TLS in Code (a Critical Read)

Turning Off Certificate Validation

```
/* ASLA: */
SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);
```

This opens the door to MITM. A line left in "for testing" is a catastrophe in the field.

Skipping Name Checking

- If the chain is valid but the **name** is not verified, the attacker's valid certificate is accepted.
- The hostname (SAN) must always be checked.

Pinning · Use with Care

- SPKI pinning is strong, but without a **backup pin** you get locked out when the key changes.
- At least one backup pin.

Critical Reading Rule

Ask this when reading TLS setup code:

- Is validation turned on?
- Is the name checked?
- Is it **fail-closed** on error?

Appendix F · Quick Reference

Glossary (Continued)

Term	Meaning
Nonce	A number used once
KDF	Key derivation function
Forward secrecy	Old sessions can't later be decrypted
Trust store	Trusted root CAs
Crypto-period	A key's lifespan
Crypto agility	Easily swapping the algorithm

Crypto Checklist

- [ ] AEAD (GCM), nonce never repeated
- [ ] Integrity is present (AEAD/EtM)
- [ ] RSA-OAEP/PSS or Ed25519/X25519
- [ ] Signature `==1` + version covered
- [ ] Chain + SAN validated, fail-closed
- [ ] Key lifecycle + storage documented

Final Word (Week 10)

> Judge cryptography not by "on/off," but by whether it is **used correctly**.

The right mode, the right padding, a verified signature, a checked chain, a well-managed key.