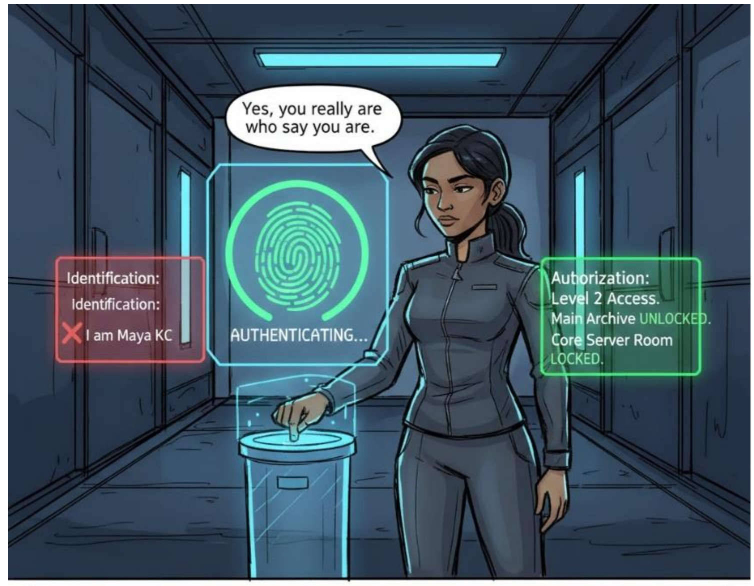
Unit 5
Authentication
3 Hrs

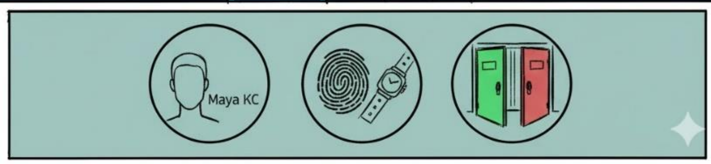
Sanjeev Thapa, Er. DevOps, SRE, CKA, RHCSA, RHCE, RHCSA-Openstack, MTCNA, MTCTCE, UBSRS, HEv6, Research Evangelist

Authentication

Authentication answers one question:

“Who are you?”





It is different from:

- **Identification**: “I claim I am Maya KC.”
- **Authentication**: “Prove it.”
- **Authorization**: “What are you allowed to do?”

Core concepts

- **Subject**: user/device trying to access
- **Verifier**: system checking proof (server)
- **Credential**: what you use to prove identity

- **Factors of authentication**
    1. **Something you know** (password/PIN)
    2. **Something you have** (token, phone, smart card)
    3. **Something you are** (biometrics)
    4. **Somewhere you are** (location)
    5. **Something you do** (behavior/keystrokes)

Authentication System

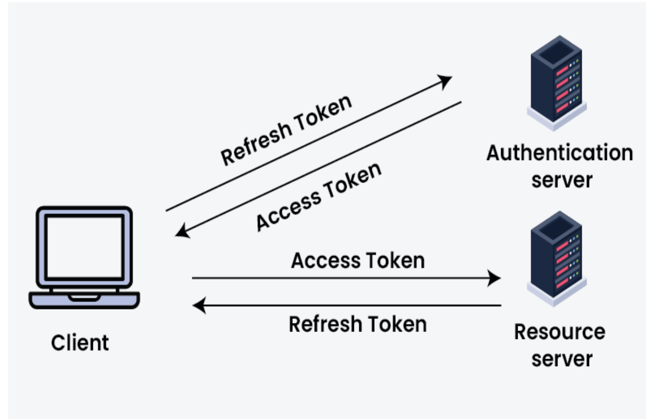
An **authentication system** is the full mechanism that:

1. collects credentials,
2. verifies them securely, and

3. establishes a trusted session.

Components

- **User/Client**: enters password/uses biometrics/token
- **Authenticator**: mechanism generating proof (password, OTP app, fingerprint sensor)
- **Verification server**: checks proof
- **Credential database**: stores *verifiers* (not raw secrets)
- **Session manager**: creates a session (cookie/token) after login
- **Audit/logging**: detects abuse, supports forensics



Basic flow (typical login)

1. User enters credential
2. Server verifies credential
3. If valid → server creates a **session token**
4. User uses token for future requests

Design goals

- **Correctness**: real users get in, attackers don’t
- **Confidentiality**: credentials not leaked
- **Integrity**: no tampering with login process
- **Availability**: login works reliably
- **Accountability**: logs for tracing

Common threats

- Password guessing, phishing, replay, MITM, credential stuffing, database leaks, session hijacking.

Password-based authentication

User proves identity by demonstrating knowledge of a **secret password**.

How systems should store passwords (important)

Never store plaintext passwords. Store a **salted password hash**:

- salt = random value stored with the hash
- hash = H(salt || password) with a slow hashing function

Recommended password hashing (conceptually)

- Use slow, adaptive algorithms: **bcrypt / scrypt / Argon2**
- Why slow? It makes offline guessing expensive.

Strength factors

- **Length** is usually more important than complexity.
- **Unique per site** prevents reuse damage.

A system stores passwords like:

$$\boldsymbol{Stored = H(\text{salt} + \text{password})}$$

Given

- Salt = 12
- Password = 34
- Assume a simple hash function for learning:

$$H(x) = (x \bmod 100)$$

- Combine salt + password as a number: 1234

Question: What is stored?

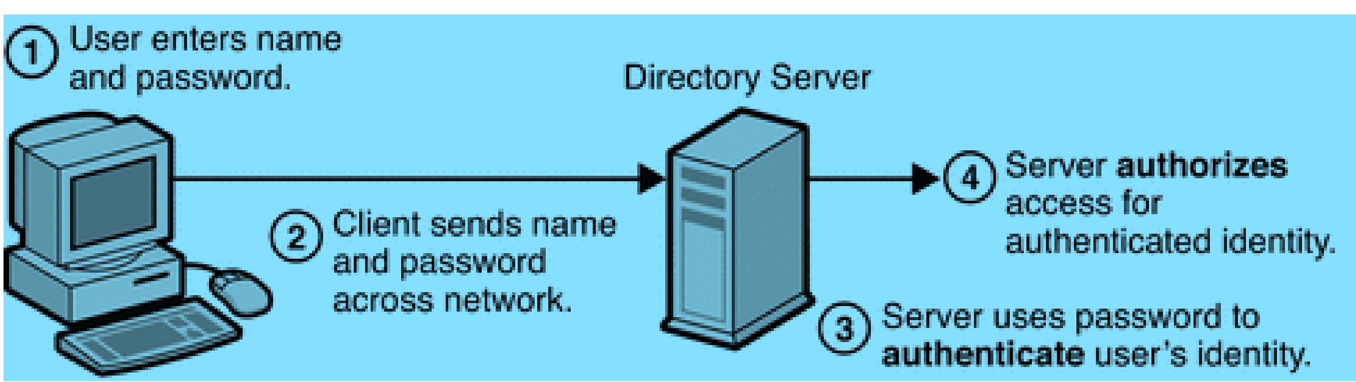
Solution

$$H(1234) = 1234 \bmod 100 = 34$$

✅ Stored value = **34** (and salt = 12 stored too)

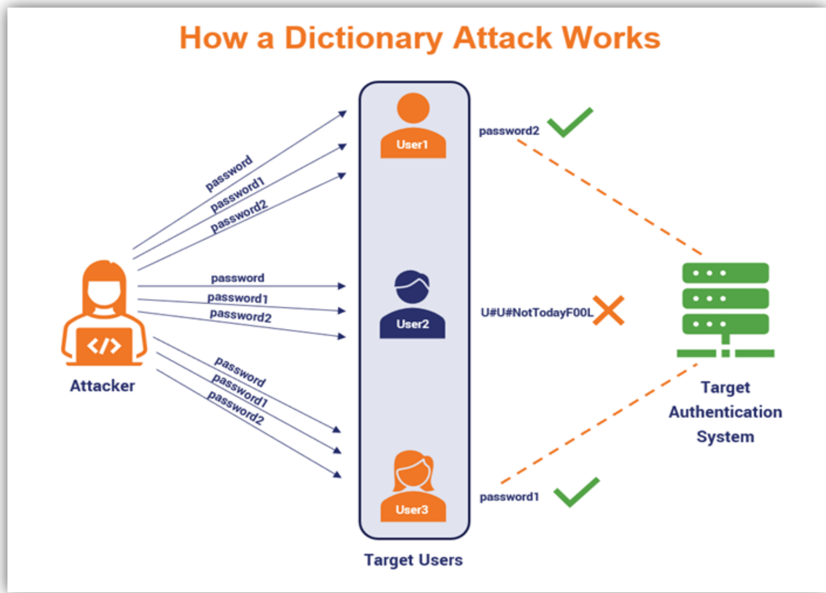
Meaning: Even if attacker sees “34”, they still need password. In real systems hash is very strong, not mod.

- **Rate limiting + lockouts** reduce online guessing.



Dictionary Attacks

A **dictionary attack** tries passwords from a list of common passwords/words.



Two types

1. **Online dictionary attack**
   - Attacker tries logins against the real system.
   - Limited by rate limiting, CAPTCHA, lockout, monitoring.
2. **Offline dictionary attack**
   - Attacker steals password hashes (database leak) and guesses locally.
   - Much more dangerous: attacker can try billions of guesses.

Given

- Attacker has a dictionary of **10,000** passwords
- System allows **5 login attempts per minute** (rate limit)

Question: Worst-case time to try all?

Solution

- Attempts per minute = 5
- Total attempts needed = 10,000

$$Time = \frac{10000}{5} = 2000 \text{ minutes}$$

Convert to hours:

$$2000/60 = 33.33 \text{ hours}$$

✅ Worst-case ≈ **33 hours 20 minutes**

Lesson: Rate-limiting slows online attacks a lot.

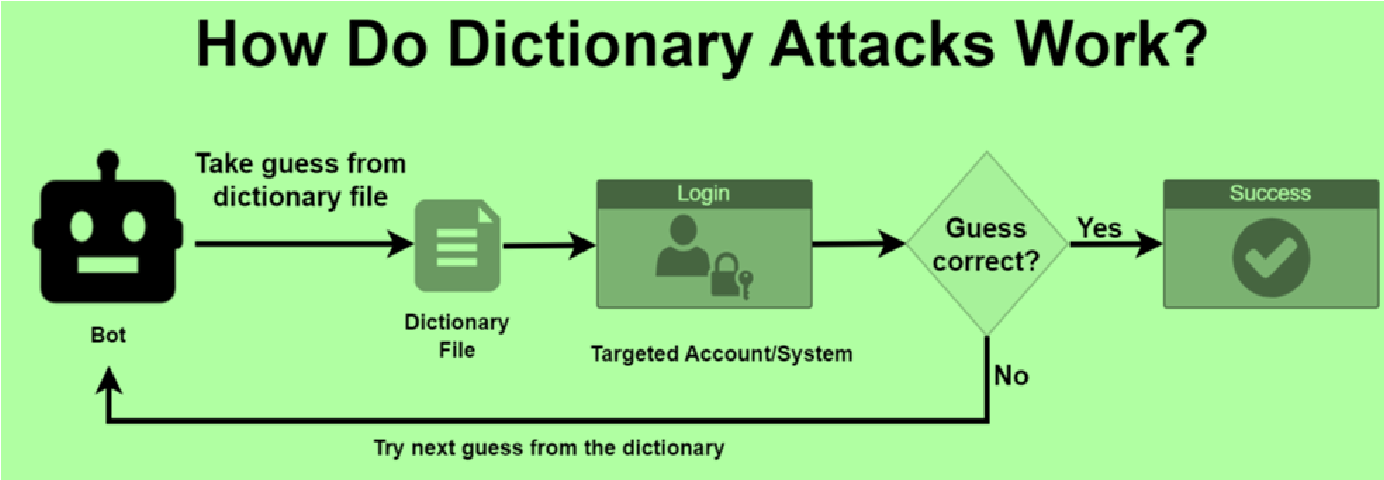
Why salts matter

Without salt: attacker can use precomputed tables (rainbow tables).
With salt: each user’s hash is unique even for same password.

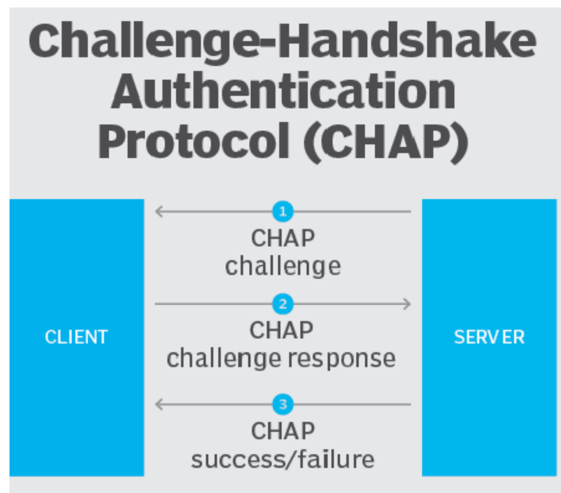
Defenses

- **Strong password policy** (length, avoid common passwords)
- **Password blacklist** (block “123456”, “password”, etc.)
- **Rate limiting / progressive delays**
- **Account lockout** (careful: can be DoS)
- **2FA/MFA** (even if password guessed, attacker blocked)

- **Secure hashing** with salt + slow KDF (Argon2/bcrypt/scrypt)



Challenge–Response System



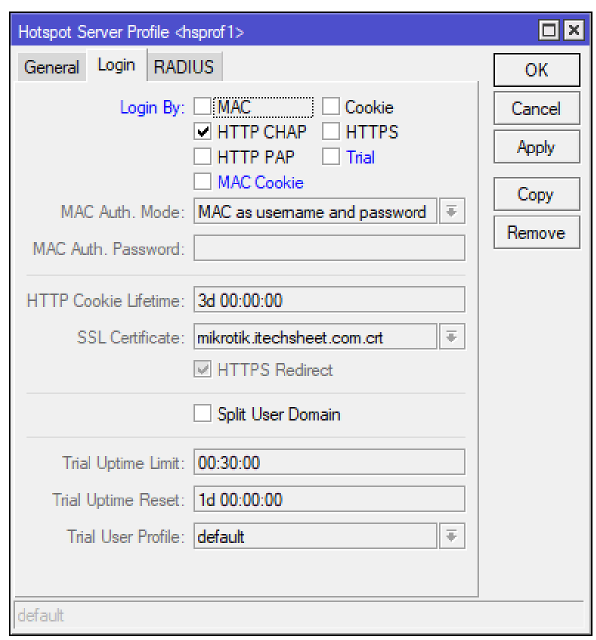
A **challenge–response** system avoids sending the secret directly.

Idea

1. Server sends a random **challenge** (nonce).
2. Client computes a **response** using secret + challenge.
3. Server verifies response.

Why it's useful

- Prevents **replay attacks** (because nonce changes each time)
- Secret is not transmitted as plaintext



Simple example (high-level)

- Server → Client: nonce
- Client → Server: H(secret || nonce)
- Server checks if response matches expected

Requirements for security

- Challenge must be **random and fresh** (nonce)
- Response must be computed with a **secure function**
- Prefer using **HMAC** (keyed hash), not plain hash
- Use **TLS** as well, because MITM can still interfere in some designs

Where used

- One-time password systems
- Smart cards and token-based auth
- Some network authentication schemes

Let's use a simple "response formula" for understanding:

$$Response = (Secret + Nonce)\, mod\, 100$$

Given

- Shared Secret = **47**
- Server sends Nonce = **18**

Question: Client response?

Solution

$$(47 + 18)\, mod\, 100 = 65$$

✅ Response = **65**

Now server uses a different nonce:

- New Nonce = **90**

$$(47 + 90)\, mod\, 100 = 37$$

✅ New Response = **37**

Lesson: Even if attacker records old response **65**, it won't work later because nonce changes.

Biometric System

Biometrics authenticate using **physical or behavioral traits**.

Types

Physiological

- Fingerprint, face, iris, retina

Behavioral

- Voice, signature dynamics, keystroke patterns, gait

Biometric authentication pipeline

1. **Enrollment**: capture sample → create a **template**
2. **Storage**: store template securely (ideally encrypted / hardware-backed)
3. **Matching**: compare live sample to stored template → similarity score
4. **Decision**: accept/reject based on threshold

Key terms

- **FAR (False Acceptance Rate)**: attacker incorrectly accepted
- **FRR (False Rejection Rate)**: real user incorrectly rejected
- **CER/EER**: point where FAR = FRR (overall accuracy indicator)

Pros

- Convenient, fast, hard to "guess"
- Good as a second factor

Biometrics (FAR / FRR) – Numerical

Given
In a day:

- Genuine user attempts = **200**
- Impostor attempts = **100**
  Results:
- Genuine rejected wrongly = **10**
- Impostor accepted wrongly = **3**

Find FRR and FAR

FRR (False Rejection Rate)

$$FRR = \frac{\text{False rejections}}{\text{Genuine attempts}} = \frac{10}{200} = 0.05 = 5\%$$

✅ FRR = **5%**

FAR (False Acceptance Rate)

$$FAR = \frac{\text{False acceptances}}{\text{Impostor attempts}} = \frac{3}{100} = 0.03 = 3\%$$

✅ FAR = **3%**

Lesson: Lower FAR = more secure; lower FRR = more usable

Cons / Risks

- **Not secret** (your face/fingerprint is exposed everywhere)
- **Cannot be changed** easily if compromised
- Spoofing (fake fingerprints, face masks, deepfake voice)
- Privacy concerns (tracking, misuse)
- Sensor and environment issues (wet fingers, poor lighting)

Best practices

- Use biometrics as **unlocking a local secret** (e.g., device key), not as the only server-side secret.
- Add **liveness detection** (blink, depth, pulse, challenge gestures).

- Combine with **PIN/password** for higher security (multi-factor).

Needham–Schroeder Scheme (Symmetric-Key)

A classic protocol for authentication using a **trusted key server** (KDC-like).

Goal

Allow two parties (A and B) to establish a **session key** securely, using help from a trusted server **S**.

High-level steps (symmetric version)

1. A → S: "A wants to talk to B"
2. S → A: session key **KAB** + a **ticket for B** encrypted for B
3. A → B: sends ticket (B can open it and get KAB)
4. A and B use KAB to prove freshness using nonces/challenges

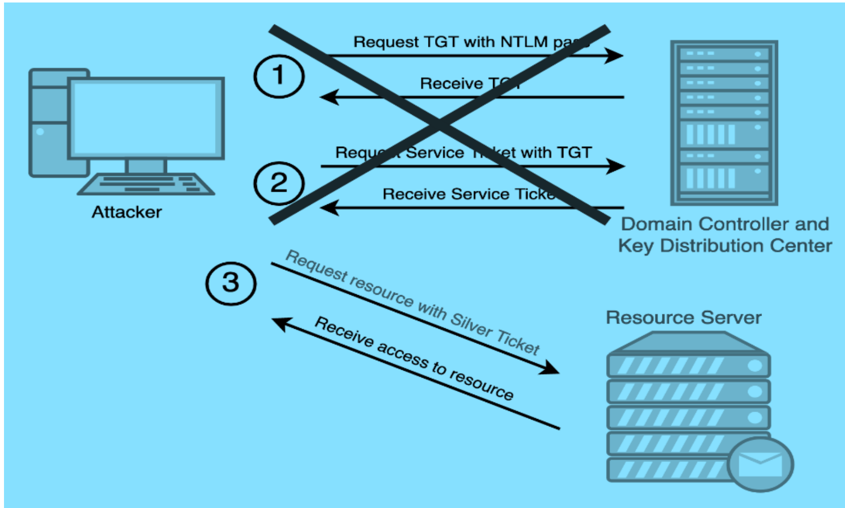
Strength

- Provides a session key without A and B sharing a long-term key directly.

Known issue (Denning–Sacco replay problem)

If an attacker later steals an **old session key KAB**, they may replay old tickets to impersonate.
Fix: include **timestamps** (this is exactly why Kerberos uses them).

Kerberos is a real-world authentication system based on trusted third party + tickets.



Kerberos key idea

Instead of sending passwords over the network, Kerberos uses:

- **Tickets**
- **Session keys**
- **Time-based freshness (timestamps)**

Kerberos main components

- **Client (C)**
- **Service server (V)** (e.g., file server)
- **KDC** (Key Distribution Center), which includes:
    - **AS (Authentication Server)**
    - **TGS (Ticket Granting Server)**

Kerberos flow (easy-to-remember)

Step 1: Login → get TGT

- Client proves identity to **AS** (often using password-derived key).

Kerberos (Ticket lifetime) – Numerical

Kerberos uses **timestamps** to stop replay.

Given

- Ticket issued at **10:00**
- Ticket validity = **2 hours**

Question: Until what time is ticket valid?

Solution

$$10:00 + 2 \text{ hours} = 12:00$$

✅ Ticket valid until **12:00**

Replay example

- Attacker steals ticket at **13:00** → system rejects because ticket expired at **12:00**.

- AS returns **TGT (Ticket Granting Ticket)** + a client-TGS session key.

Step 2: Request service ticket

- Client presents TGT to **TGS** and asks for access to service V.
- TGS returns **Service Ticket** + a client-service session key.

Step 3: Access the service

- Client presents Service Ticket to **V**.
- V verifies ticket and optionally performs mutual authentication.

Why Kerberos is strong

- Password isn't sent across the network.
- Tickets are time-limited.
- Supports **Single Sign-On (SSO)**: login once, access multiple services.

Kerberos requirements / limitations

- **Clock synchronization** is important (timestamps).
- If **KDC is down**, authentication may fail (availability concern).
- Ticket theft is a risk if attackers compromise a machine (protect cache).