SDE: System Design and Engineering

Lecture – 7

Introduction to Security

From Zero to Google: Architecting the Invisible Infrastructure

by

Aatiz Ghimire

Sections

- Covers fundamentals: CIA Triad, threats, firewalls, VPNs, and Secure System design.
- Emphasizes secure practices: DevSecOps, permission systems, password hashing, and tokenization.
- Explains auth systems: JWT, OAuth 2.0 flows, Google Authenticator, and SSO.
- Highlights encryption types: symmetric, asymmetric, encoding vs. encryption vs. tokenization.
- Focuses on future security trends:zero-trust, and secure-by-design models.

Introduction to Cybersecurity

- **Cybersecurity** is the practice of **PROTECTING DATA, SYSTEMS, AND NETWORKS** from digital attacks.
- It prevents **UNAUTHORIZED ACCESS**, data theft, or service disruption.
- It is **ESSENTIAL** for **PRIVACY**, **BUSINESS CONTINUITY**, and **NATIONAL SECURITY**.

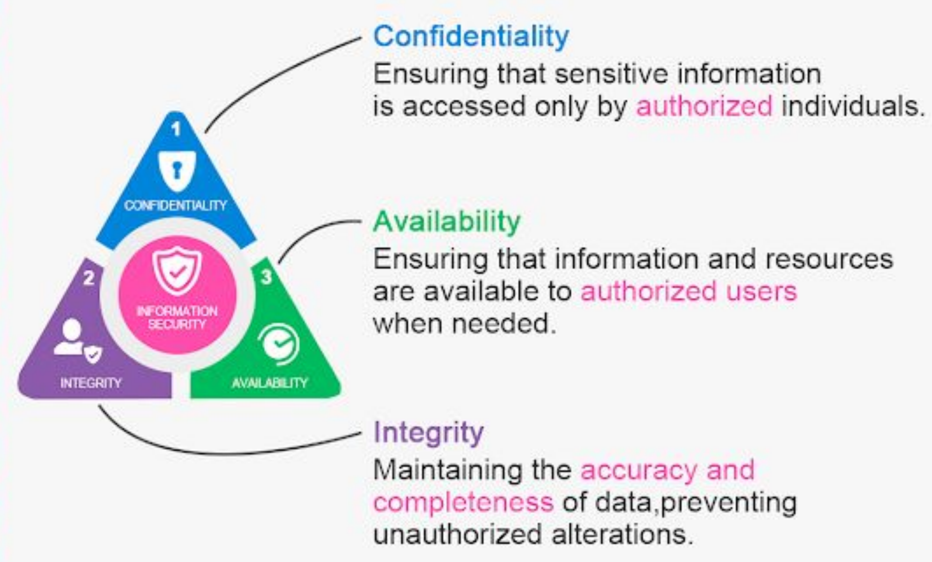
Why is Cybersecurity Important?

$4.35 MILLION: Average cost of a data breach in **2022**.

579–1287 password attacks occur **EVERY SECOND**.

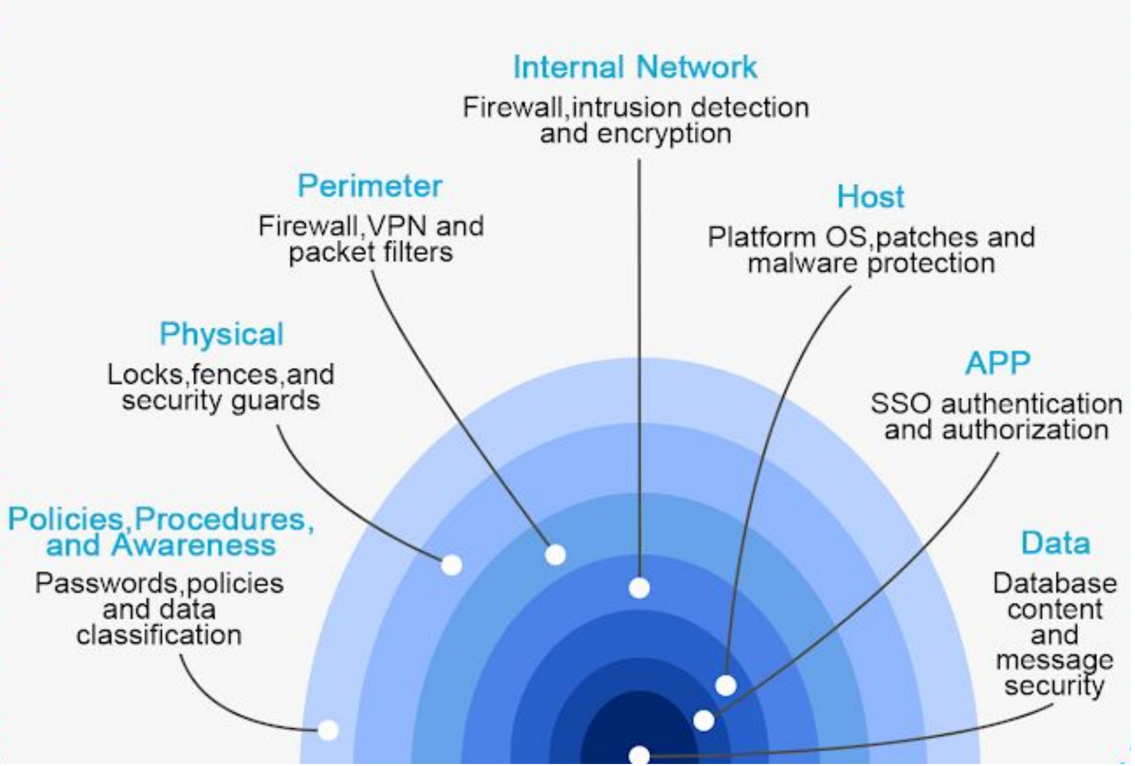
Over **65 TRILLION SIGNALS** are analyzed **DAILY** by Microsoft.

The CIA Triad: Core Principles of Cybersecurity

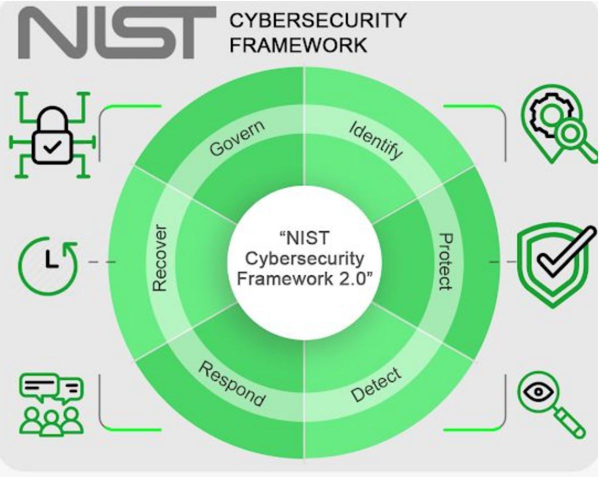


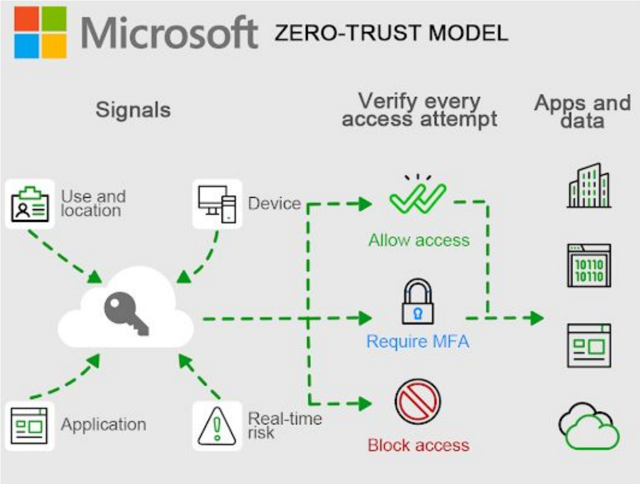
Top 15 Cybersecurity Threats

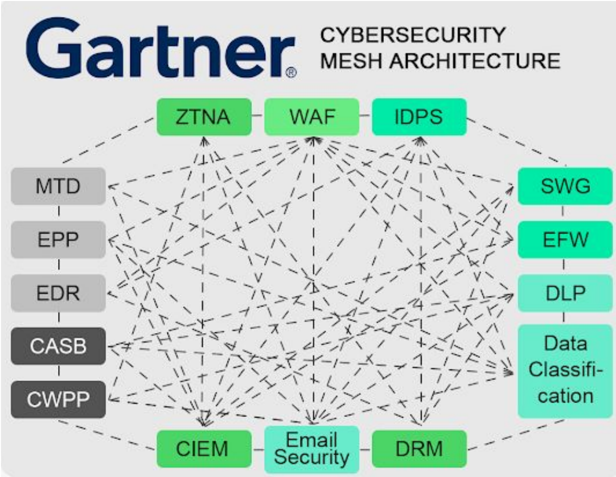
Basic Defense Mechanisms



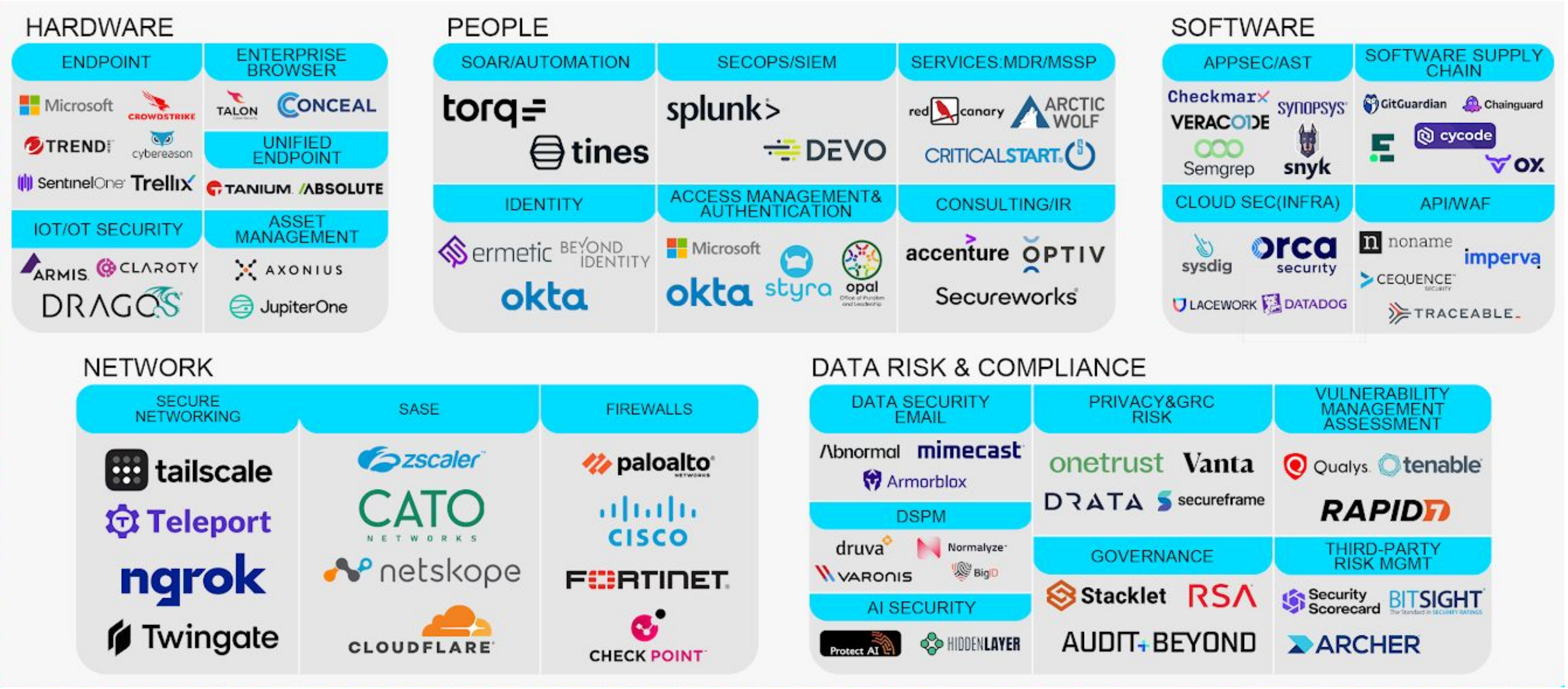
Cybersecurity Frameworks







Cybersecurity Ecosystem



What is DevSecOps?

- **DevSecOps** = **Development** + **Security** + **Operations**

- A **culture and methodology** that ensures **security is integrated at every stage** of the **software development lifecycle (SDLC)**.

- Emphasizes **automated security**, **shared responsibility**, and **collaboration** between developers, operations, and security teams.

Key Concepts in DevSecOps:

- 🔁 **CI/CD Automation**: Embed security checks into continuous integration and delivery pipelines.
- 🔒 **Automated Security Checks**: Linting, static code analysis, dependency scanning.\
- 📊 **Continuous Monitoring**: Real-time threat detection and alerting in runtime environments.

What is DevSecOps?

- ⚙️ **Infrastructure as Code (IaC)**: Secure, version-controlled infrastructure provisioning.
- 🐳 **Container Security**: Image scanning, runtime sandboxing, least privilege containers.
- 🔐 **Secret Management**: Secure storage of API keys, credentials, tokens.
- 📉 **Vulnerability Management**: Early detection and automated patching of security flaws.
- 🧠 **Threat Modeling**: Anticipate attack vectors early in design.
- 🧪 **QA Integration**: Functional + security tests in pre-production.
- 💬 **Collaboration and Communication**: Shared responsibility model across teams.

Principles of Secure System Design

- Secure system design involves **12 critical domains**.
- Emphasizes **defense-in-depth** and **zero trust** principles.
- Security must be built-in from the **architecture stage**.

1. **Authentication**

Scenarios to Protect:

- **User logins, employee access to internal systems**

Design Points:

- **Strong password policies**
- **Multi-factor authentication (MFA)**

Principles of Secure System Design

2. Authorization

Scenarios to Protect:

- Data access, user roles

Design Points:

- **Least privilege principle**
- **Role-based access control (RBAC)**
- **Regular permission reviews**

3. Encryption

Scenarios to Protect:

- Sensitive data, secure communication

Design Points:

- **TLS** for data transit
- **AES** or stronger encryption for data at rest
- **Robust key management systems**

Principles of Secure System Design

4. Vulnerability Management

Scenarios to Protect:

- Vulnerability exploits, zero-day attacks

Design Points:

- **Regular scanning**
- **Security patching**
- **Continuous monitoring**

5. Audit & Compliance

Scenarios to Protect:

- Regulatory requirements (e.g., GDPR, HIPAA)

Design Points:

- **Audit trails, data breach logs**
- **Compliance reporting**
- **Encrypted logging systems**

Principles of Secure System Design

6. Network Security

Scenarios to Protect:

- Internal and external network attacks

Design Points:

- **Firewalls, segregated VLANs**
- **Intrusion detection systems (IDS)**
- **Secure DNS configuration**

7. Terminal Security

Scenarios to Protect:

- Laptops, workstations, POS devices

Design Points:

- **Antivirus, full-disk encryption**
- **Device hardening, patch management**

Principles of Secure System Design

8. Emergency Responses

Scenarios to Protect:

- **DDoS**, ransomware, breach incidents

Design Points:

- **Incident response plans**
- **Runbooks & drills**
- **Security operations center (SOC)** procedures

9. Container Security

Scenarios to Protect:

- Microservices, containerized environments

Design Points:

- **Base image scanning**
- **Runtime sandboxing**
- **Role-limited containers**

Principles of Secure System Design

10. API Security

<u>Scenarios to Protect:</u>

- Public and internal APIs

<u>Design Points:</u>

- **OAuth2, API key management**
- **Rate limiting, input validation**

11.3rd-Party Management

<u>Scenarios to Protect:</u>

- Integrations with SaaS or vendor APIs

<u>Design Points:</u>

- **Vendor risk assessments**
- **Secure data-sharing agreements**
- **Monitoring third-party access logs**

Principles of Secure System Design

12. Disaster Recovery

Scenarios to Protect:

- Data center failures, ransomware events

Design Points:

- **Disaster recovery (DR) plans**
- **Redundant backup systems**
- **Geo-replication**

Designing a Permission System

Why It Matters:

- Permission systems determine **who can access what**.
- The **wrong model leads to security breaches or maintenance nightmares**.
- Choosing the right model depends on the **application domain**, **security requirements**, and **user complexity**.

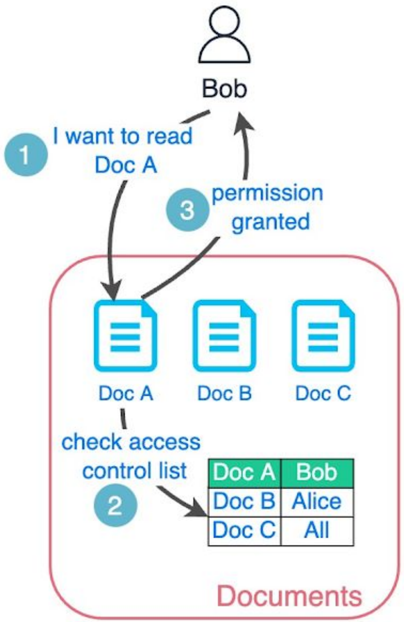
Designing a Permission System

1. ACL – Access Control List

- A **list of rules** specifying which users are **granted or denied** access to a resource.

- **Pros**:
    - Simple, intuitive
    - Easy to attach to individual resources

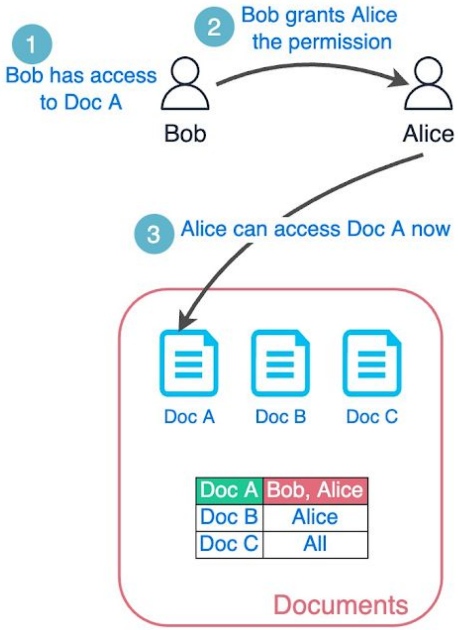
- **Cons**:
    - Hard to maintain at scale
    - **Error-prone** and repetitive



Designing a Permission System

2. DAC – Discretionary Access Control

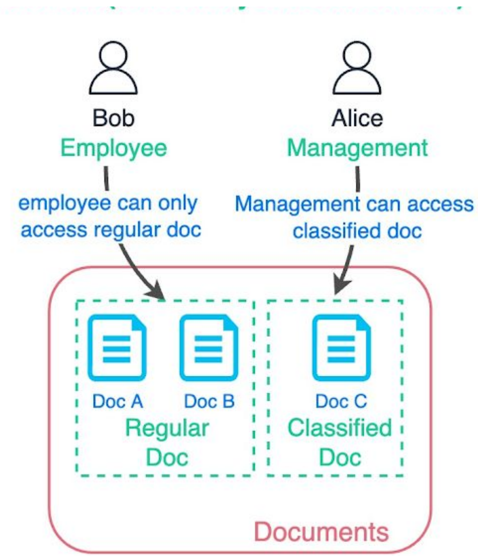
- Access is controlled by the **resource owner**.
- Based on **ACL**, but allows owners to grant access at their discretion.
- **Pros**:
  - **Flexible** and commonly used (e.g., **Linux file systems**)
- **Cons**:
  - Decentralized control → potential misconfigurations
  - Too much **authority in the owner's hands**



Designing a Permission System

3. MAC – Mandatory Access Control

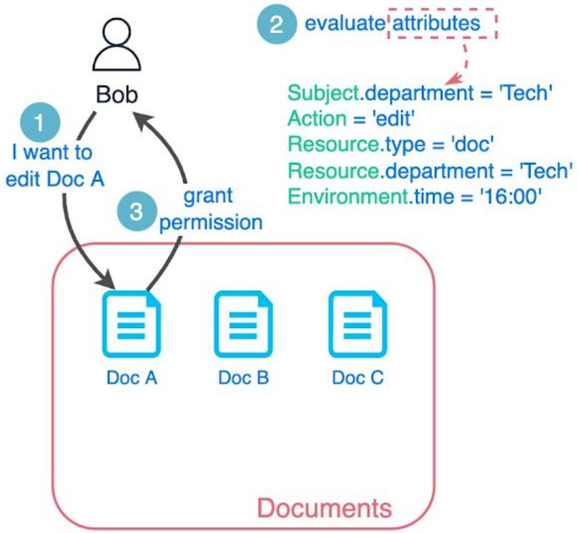
- System-enforced access rules based on **classification labels**.
- Used in **military and government systems**.
- **Pros**:
    - **Strict and secure**, prevents privilege escalation
- **Cons**:
    - **Rigid and difficult to configure**
    - Not suitable for dynamic user environments



Designing a Permission System

4. ABAC – Attribute-Based Access Control

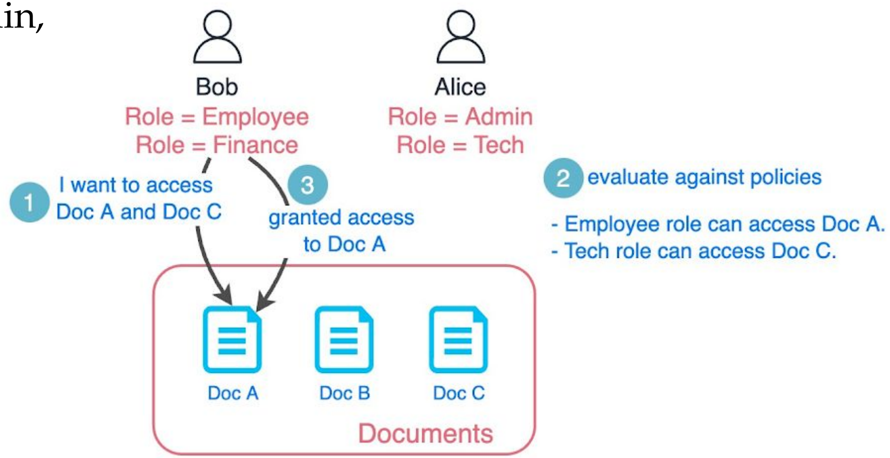
- Permissions depend on **attributes**: user role, resource type, action, and environment.
- **Pros**:
  - **Highly flexible**, supports context-aware decisions
- **Cons**:
  - **Complex rules**, harder to audit and implement
  - Rarely used in mainstream applications



Designing a Permission System

5. RBAC – Role-Based Access Control

- Access decisions based on **user roles** (e.g., admin, viewer).
- Users inherit permissions via roles.
- **Pros**:
    - **Scalable**, easy to group users
    - Widely adopted in enterprises and SaaS platforms
- **Cons**:
    - Static roles may not capture **contextual needs**



Storing Passwords Safely

Common Mistakes to Avoid

- **NEVER** store passwords in **plain text**.
- Avoid storing raw **password hashes** without **additional safeguards**.
- Raw hashes are **vulnerable to rainbow table attacks** and brute-force attempts.

Real-World Breach Example

- In several major breaches (e.g., Yahoo, RockYou), millions of passwords were leaked due to **lack of proper hashing/salting**.

Key Point:

- *"If your database is compromised, your password scheme must still protect users."*

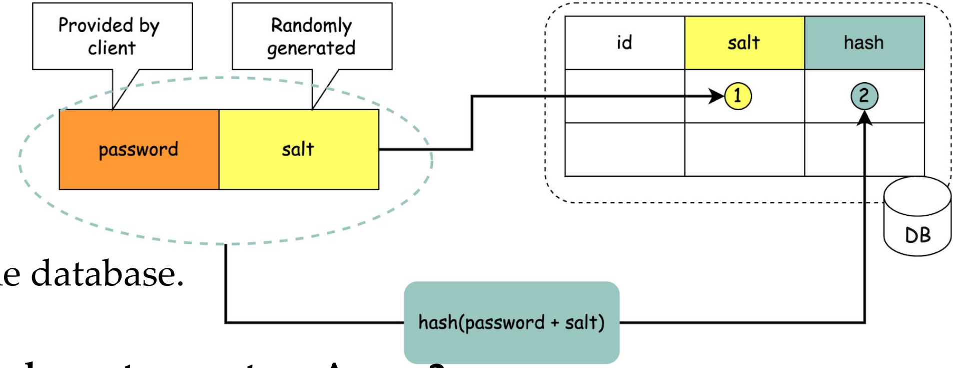
Best Practices in Password Storage

What is Salt?

- A **salt** is a **random string** added to a password before hashing.
- It **ensures uniqueness** in the hash output, even for identical passwords.
- Defined by **OWASP** as:
  "A unique, randomly generated string that is added to each password as part of the hashing process."

How to Store Passwords Securely

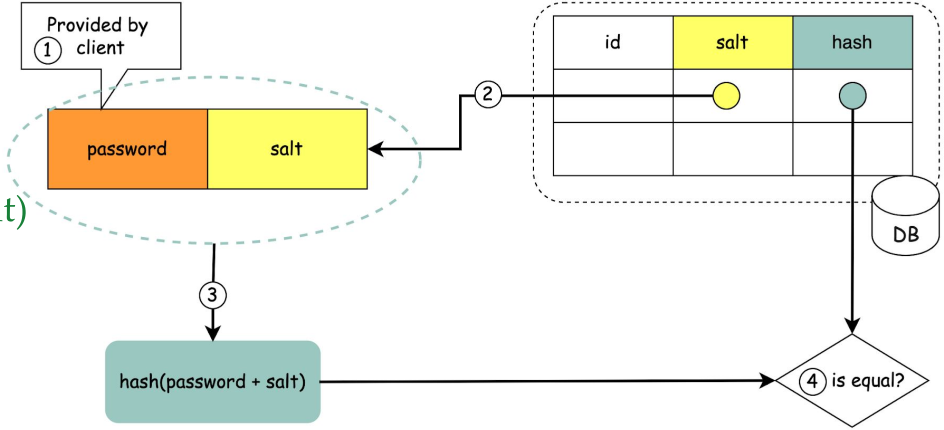
1. **Generate a random salt** per user.
2. Compute:
   hash(password + salt)
3. **Store both salt and hashed password** in the database.
   - Salt is stored in **plain text**.
   - Hash must use **strong algorithms** like **bcrypt**, **scrypt**, or **Argon2**.



Best Practices in Password Storage

How to Validate a Password

1. User submits password.
2. Fetch salt from database.
3. Compute:
   H1 = hash(submitted_password + stored_salt)
4. Compare H1 with stored hash (H2).
   - If H1 == H2, the password is **valid**.



Best Practices in Password Storage

Recommended Hashing Algorithms

- **bcrypt** – built-in salting, adaptive
- **scrypt** – resistant to GPU attacks
- **Argon2** – OWASP’s preferred algorithm (memory-hard)

Password Managers

What Is a Password Manager?

- A tool that **generates, stores, and autofills** passwords.
- Can be accessed via **browser extension**, **app**, or **command line**.
- Stores credentials in an **encrypted vault**.
- Only **one master password** needs to be remembered by the user.

Use Cases:

- Individuals: Web logins, app credentials.
- Teams: Shared vaults, role-based access.
- Enterprises: Audit logging, compliance, SSO integration.

Key Benefit:

"You never store or transmit plain passwords—the system is built on strong encryption and **zero-knowledge principles**."

How Password Managers Work (Internals)

Step-by-Step Process:

Step 1: Account Creation

- User provides:
  - **Email address**
  - **Account password**
- Password manager generates a **Secret Key**.
- These 3 values are used to compute:
  - **MUK (Master Unlock Key)**
  - **SRP-X (Secure Remote Password)** via 2SKD algorithm.
- **Secret key is stored locally, never sent to the server.**

How Password Managers Work (Internals)

Step 2: Key Derivation

- MUK is used to generate an encrypted **Master Password Key (MP key)**.

Step 3–5: Key Hierarchy & Vault

- MP Key → derives:
    - A **Private Key**
    - A **Vault Key**
- Vault key encrypts the vault items (passwords, notes, keys).
- All data is **stored encrypted on the cloud**, but **decrypted only on your device**.

How Password Managers Work (Internals)

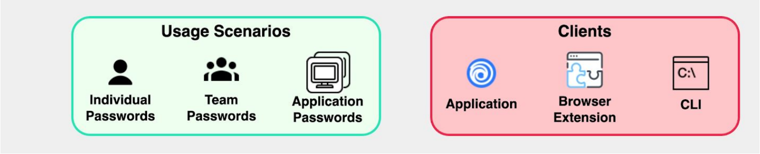
Step 6: Zero-Knowledge Encryption

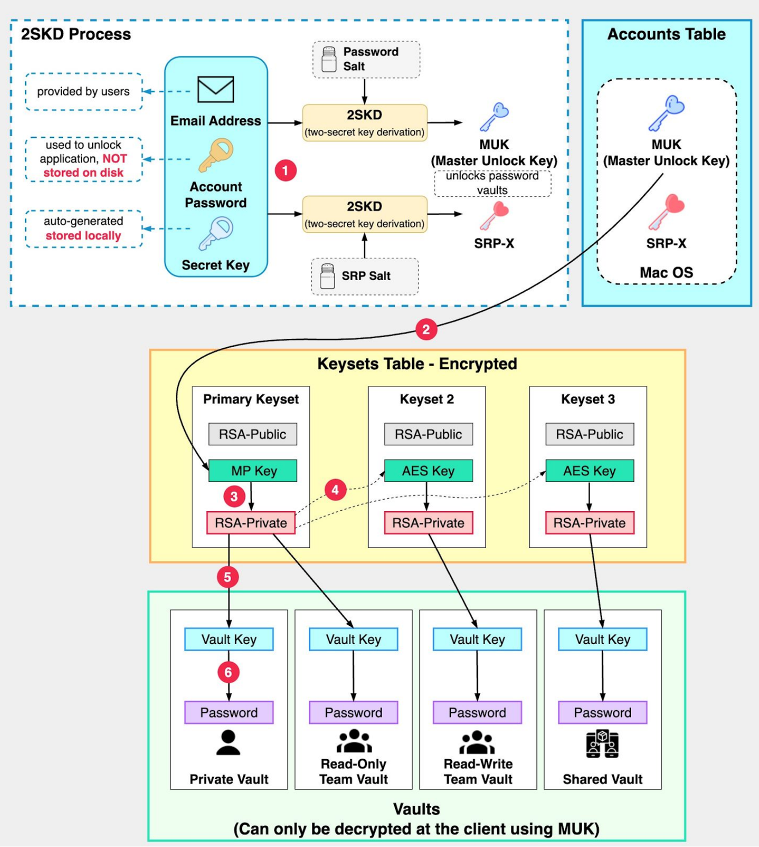
- All encryption happens **locally**.
- Even the **password manager service cannot read** your data.

Crypto Principles Used:

- **AES-256 encryption** for vault data
- **2SKD protocol** for password-based key derivation
- **SRP (Secure Remote Password)** for safe authentication
- **Zero-Knowledge Architecture** for privacy

How Password Managers Work (Internals)





Encoding, Encryption, and Tokenization

1. Encoding

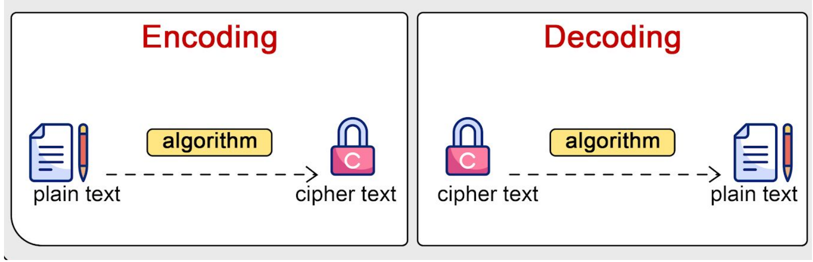
Transforming data into a **different, reversible format** for **data transmission or storage**.

Example:

- **Base64 encoding**: Encodes binary data into ASCII for network transmission.

Key Points:

- **Purpose**: Data **representation, not security**.
- **No key is required** — anyone can decode it.
- Use in: **Web development**, **data serialization**, **file transfer (e.g., email attachments)**.

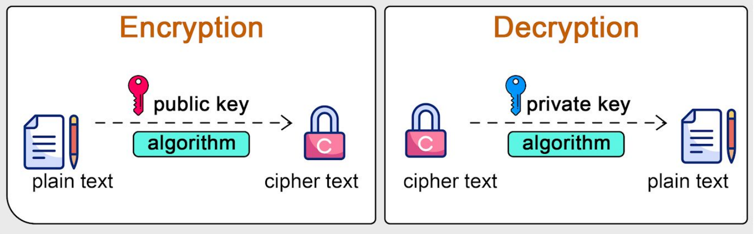


Encoding, Encryption, and Tokenization

2. Encryption
Transforms **plaintext** into **ciphertext** using an **algorithm + key** to ensure **confidentiality**.

Types:

- **Symmetric** (AES): Same key for encryption and decryption.
- **Asymmetric** (RSA): Public key for encryption, private key for decryption.



Key Points:

- **Purpose**: **Confidentiality and access control**.
- Requires a **key** to decrypt.
- Use in: **TLS/SSL, email encryption, data-at-rest security**.

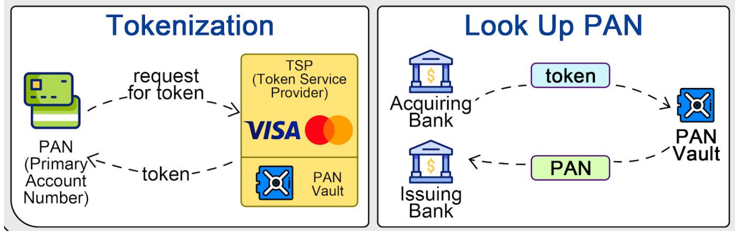
Encoding, Encryption, and Tokenization

3. Tokenization

Replaces **sensitive data** with **non-sensitive tokens**, stored separately in a **secure token vault**.

Example:

- Credit card **PAN** → **Token** in payment processing.



Key Points:

- Tokens have **no mathematical link** to original data.
- Cannot be reverse-engineered.
- Use in: **PCI-DSS compliance, payment gateways, PII protection**.

Symmetric vs. Asymmetric Encryption

Symmetric Encryption

- **One key** is used for both **encryption and decryption**.
- **Fast and efficient** — ideal for **bulk data encryption**.

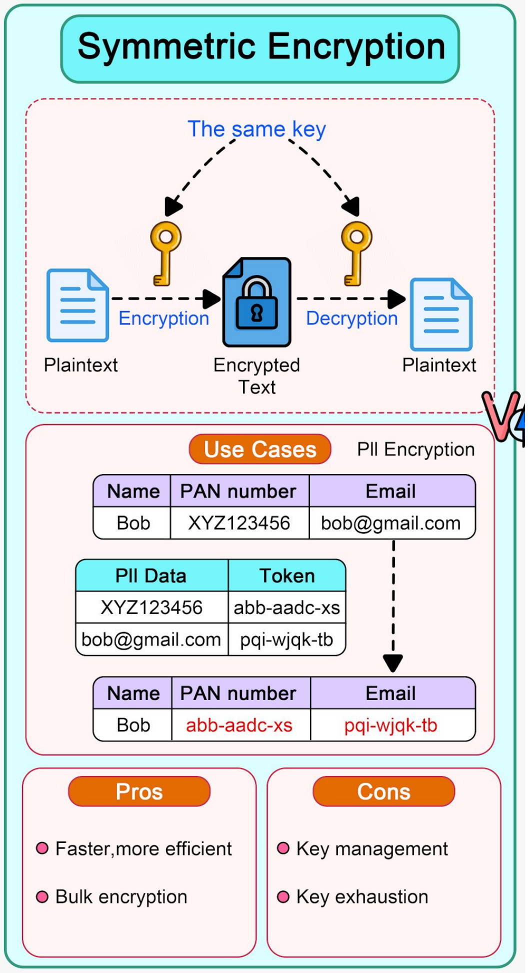
- Used in:
  - **AES (Advanced Encryption Standard)**
  - **Encrypting PII, backups, and files**

Pros:

- High speed
- Minimal computational overhead

Cons:

- **Key distribution problem**
- Risk if key is intercepted



Symmetric vs. Asymmetric Encryption

Asymmetric Encryption

- Uses a **key pair**:
  - **Public key** to encrypt & **Private key** to decrypt
- Only the **private key holder** can decrypt the message.

Used in:

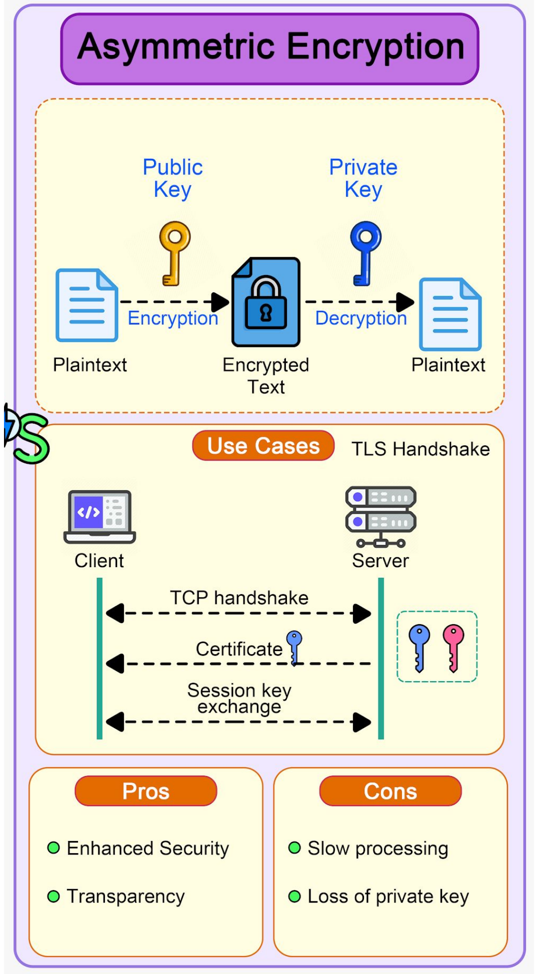
- **TLS Handshakes, email encryption (PGP), digital signatures**

Pros:

- **Secure communication** with no shared secret
- Enables **digital identity verification**

Cons:

- **Slower performance**
- **Complex key management and generation**



Understanding JSON Web Tokens (JWT)

What is JWT?

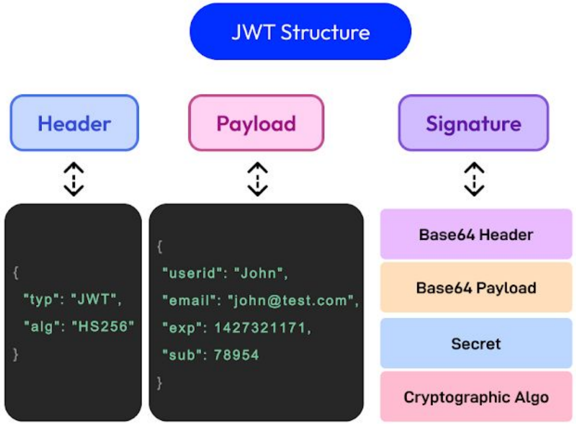
- **JWT (JSON Web Token)** is an **open standard (RFC 7519)** used for **securely transmitting information** between parties.

- Commonly used in **authentication** and **authorization** mechanisms in modern web applications.

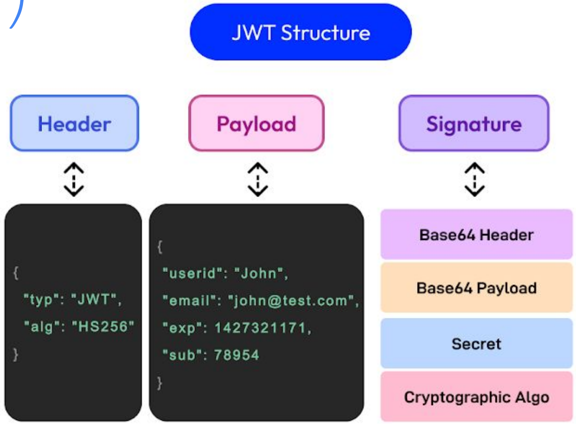
Structure of a JWT

A JWT consists of **three parts**, separated by dots (.):

1. **Header**
   - Contains algorithm (alg) and token type (typ)
   - e.g. { "alg": "HS256", "typ": "JWT" }



Understanding JSON Web Tokens (JWT)



2. Payload

- Contains **claims** and **user data**
- Claim types:
    - Registered claims (e.g., iss, exp)
    - Public claims
    - Private claims

3. Signature

- Ensures **integrity** and **authenticity**
- Created using:
  HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

Example:
eyJhbGciOi...header
eyJzdWIiOi...payload
SflKxwRJSMe...signature

Understanding JSON Web Tokens (JWT)

Why Use JWT?

- Stateless (no need to store sessions)
- Portable (can be passed in headers, cookies)
- Self-contained (holds authentication data)

Signing Methods in JWT

Symmetric Signature

- Uses a **shared secret key** to **sign and verify** the JWT.
- Algorithm: **HMAC (e.g., HS256)**.
- Simpler but both parties must securely manage the **same key**.

Use Cases:

- Internal services where **both signing and verification** happen on the **same server or trusted cluster**.

Signing Methods in JWT

Asymmetric Signature

- Uses a **private key** to sign and a **public key** to verify.
- Algorithms: **RS256, ES256** (RSA/ECC based).
- Ensures verification can happen **without exposing the private key**.

Use Cases

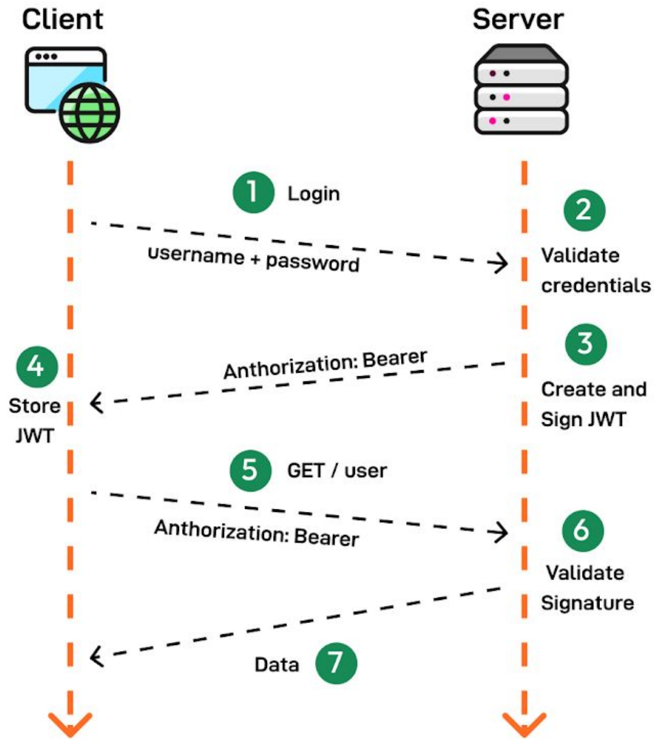
- Microservices, third-party APIs, or public-facing apps where clients or external systems must verify tokens.

JWT Flow in Authentication

1. User logs in → server generates JWT and sends it back.

2. Client includes JWT in subsequent requests (usually in **Authorization header**).

3. Server verifies JWT signature before granting access.



OAuth 2.0

- A **secure authorization framework** that enables applications to **access resources on behalf of a user, without sharing credentials**.
- Standardized in **RFC 6749** and widely used by **Google, Facebook, GitHub**, etc.

👥 OAuth 2.0 Entities

1. **User**: The resource owner
2. **Client (App)**: Requests access to the user's data
3. **Authorization Server**: Issues access tokens (typically the IDP)
4. **Resource Server**: Hosts user data and verifies access via token

OAuth 2.0

Use Case Example

- Logging into **Spotify with your Google account**
- Google is the **Identity Provider**, Spotify is the **Client**, and you are the **User**

Grant Types

- **Authorization Code** (most secure, used by web apps)
- **Implicit** (legacy; discouraged)
- **Client Credentials** (machine-to-machine)
- **Resource Owner Password Credentials** (rare; legacy)

OAuth Token

- A **short-lived, secure token** issued by the Authorization Server
- Represents **identity + permissions** of the user
- Sent with requests to access protected APIs or user data

Capabilities of OAuth Tokens

1. Single Sign-On (SSO)

- Use one login to access **multiple apps or services**
- Enhances **usability** and **reduces password fatigue**

2. Cross-System Authorization

- Allows authorized access to **external APIs or services**
- Example: Slack posting to your Google Calendar on your behalf

OAuth Token

3. Scoped Data Access

- Tokens carry **scopes** (permissions)
- Only allows access to **specific resources** (e.g., email address, not full inbox)

Security Considerations

- Tokens are **not passwords**—but they can still be **stolen or intercepted**
- Always use **HTTPS**, set **expiration time**, and **revoke** when compromised

OAuth 2.0 Flows

- A **workflow pattern** used by OAuth 2.0 to **issue access tokens**.
- Each flow suits a **specific use case** depending on:
  - Client type (web, mobile, server)
  - Security posture
  - Deployment model

OAuth 2.0 Flows

1. Authorization Code Flow

Best For: Web applications with a backend
How It Works:

- User authenticates via **Identity Provider (IDP)**
- App receives a **temporary authorization code**
- App exchanges code for an **access token** and optionally a **refresh token**

Pros:

- Most secure (tokens issued on backend)
- Supports refresh tokens

Example: Login to Slack using Google account (with server validation)

OAuth 2.0 Flows

2. Client Credentials Flow

Best For: Machine-to-machine communication

How It Works:

- No user involved
- Client uses its **client ID + secret** to request an access token from the Authorization Server

Pros:

- Lightweight
- Ideal for **microservices and background jobs**

Example: Internal service calling another service API within the same system

OAuth 2.0 Flows

3. Implicit Flow *(Deprecated / Legacy)*

Best For: Single Page Applications (SPAs) *(Not Recommended)*

How It Works:

- User logs in, and the **access token is returned directly in the URL fragment**
- No authorization code stage

Cons:

- No refresh token support
- Vulnerable to token leakage in browser history

Status: 🔴 Deprecated by the OAuth Working Group — use **Authorization Code with PKCE** instead.

OAuth 2.0 Flows

4. Resource Owner Password Credentials (ROPC) Flow

Best For: Legacy or **trusted first-party applications**

How It Works:

- User provides **username + password** directly to the client
- Client exchanges them for an **access token**

Cons:

- Breaks the separation of concerns
- User credentials are handled directly by the app

Status: 🟠 Discouraged in modern systems due to security risks

Google Authenticator

- A **software-based authenticator app** used for **2FA**
- Implements **TOTP (Time-based One-Time Password)** algorithm (RFC 6238)
- Adds a **second layer of authentication** beyond username & password

Authentication Workflow: Two Main Stages

Stage 1: Setup

1. **User enables 2FA** in the application
2. **Authentication server** generates a **unique secret key** for the user
3. Server sends a **URI (otpauth://)** with issuer, username, and secret
4. This URI is **converted into a QR code**
5. **User scans QR code** using Google Authenticator → stores secret key

Google Authenticator

Stage 2: Login

1. User logs in using **username + password**
2. Google Authenticator generates a **6-digit TOTP code** every 30 seconds
3. User enters the code from their app
4. Server retrieves the same user's **secret key from the DB**
5. Server generates TOTP and compares with user's input
6. If the codes **match**, authentication is successful

Google Authenticator

Time-Based One-Time Passwords

- TOTP = HMAC-SHA1(secret + current timestamp)
- Each 6-digit code is valid for **30 seconds**
- Based on **clock sync**, not internet access

Is TOTP 2FA Secure?

Yes, But with Important Conditions:

Protection of Secret Key

- Secret is **generated server-side** and shared **once**
- Must be transmitted via **HTTPS**
- Must be **encrypted** in:
  - **User's mobile device**
  - **Authentication server database**

Is TOTP 2FA Secure?

Why 6-Digit TOTP is Hard to Break

- Each code has **1,000,000 combinations** (000000 to 999999)
- Code changes every **30 seconds**
- To guess correctly within the window, an attacker would need to try:
  - **~30,000 guesses/sec**
  - Which is **impractical** without massive brute-force capability and bypassing rate-limits

Security of Google Authenticator

Best Practices

- Enable **rate-limiting** and **account lockout** on repeated failure
- Use **time drift tolerance** of ±30s to avoid legitimate failure
- Regularly **rotate and expire** old secrets if needed
- Prefer **hardware-backed secure storage** (e.g., Android Keystore, iOS Secure Enclave)

⚠️ **Risks If Implemented Poorly**

- Secret key leakage via:
  - Non-HTTPS transmission
  - Insecure frontend QR display
- Device compromise = Authenticator compromise

Firewall

- A **firewall** is a **network security device or software** that monitors, filters, and controls **incoming and outgoing network traffic**.

- It establishes a **barrier between trusted internal networks and untrusted external networks**, such as the internet.

Firewall Categories

- **Software-Based:** Installed on devices (e.g., desktops, servers) to monitor local traffic
- **Hardware-Based:** Standalone appliances securing entire network segments

Why Use Firewalls?

- Prevent **unauthorized access**
- Block **malicious traffic**
- Enforce **security policies**
- **Inspect traffic** before it reaches endpoints

Types of Firewalls

1. Packet Filtering Firewall

- Filters packets based on **source IP**, **destination IP**, **port**, and **protocol**
- Stateless — makes decisions **per packet**
- Lightweight but limited in context awareness

2. Circuit-Level Gateway

- Monitors **TCP handshakes** to validate the **legitimacy of sessions**
- Operates at **Session Layer (Layer 5)** of the OSI model
- Efficient, but does **not inspect payload**

Types of Firewalls

3. Application-Level Gateway (Proxy Firewall)

- Acts as an **intermediary between user and service**
- Inspects traffic at **Application Layer (Layer 7)**
- Can detect **malware**, **malicious scripts**, and **content-based attacks**

4. Stateful Inspection Firewall

- Maintains **state tables** for active connections
- Analyzes packet context (e.g., is this a response to a legitimate request?)
- **Combines speed and security**: tracks connection state + inspects traffic

Types of Firewalls

5. Next-Generation Firewall (NGFW)

- Integrates:
  - **Deep Packet Inspection (DPI)**
  - **Intrusion Prevention System (IPS)**
  - **Application-layer awareness**
  - **Identity-based access control**

- Ideal for **modern threat landscapes** (zero-day exploits, APTs, etc.)

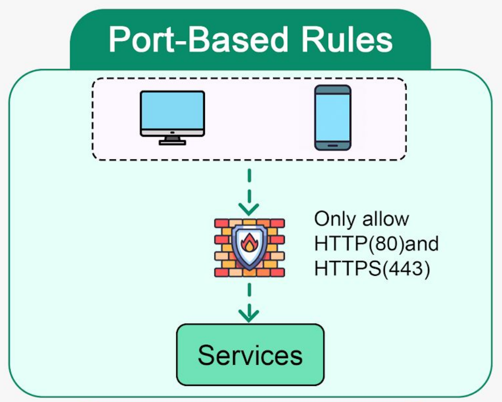
Top 6 Firewall Use Cases

1. Port-Based Rules

- Controls traffic based on **TCP/UDP port numbers**
- Example:
  - Allow: Port **80** (HTTP), Port **443** (HTTPS)
  - Block: Port **23** (Telnet), Port **445** (SMB)

Use Case:

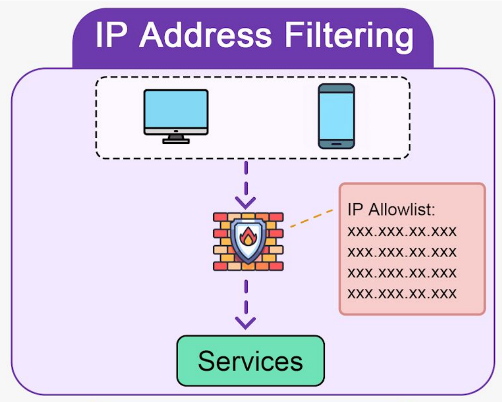
Permit only web traffic; block outdated or insecure protocols.



Top 6 Firewall Use Cases

2. IP Address Filtering

- Permit or deny traffic based on **source/destination IP addresses**
- Tactics:
  - **Whitelist** trusted IPs
  - **Blacklist** known malicious IPs (e.g., threat intel feeds)



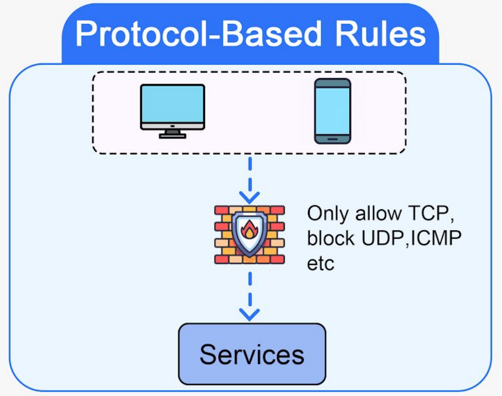
Use Case:

Only allow access to internal services from corporate VPN IP ranges.

Top 6 Firewall Use Cases

3. Protocol-Based Rules

- Filter based on network protocols such as **TCP**, **UDP**, **ICMP**, etc.
- Example:
  - Allow only **TCP traffic** on Port 22 (SSH)
  - Block all **ICMP ping requests**



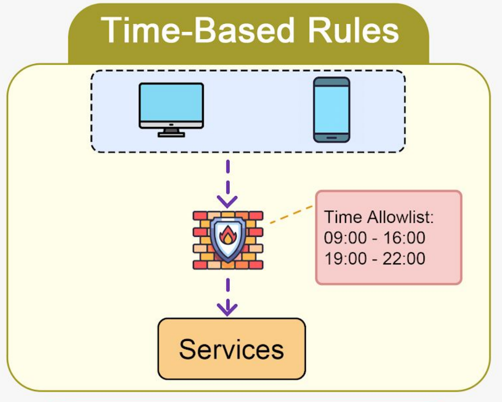
Use Case:

Allow SSH for administrators, disable ICMP to mitigate reconnaissance.

Top 6 Firewall Use Cases

4. Time-Based Rules

- Enable or disable rules **based on time schedules**
- Example:
  - Block database access after **business hours**
  - Allow web traffic only **9AM–5PM**



Use Case:

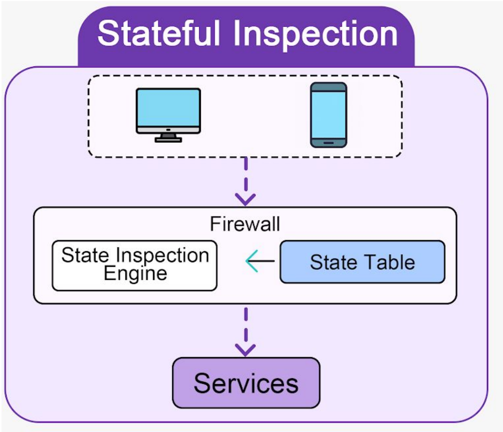
Reduce the attack surface by limiting access windows.

Top 6 Firewall Use Cases

5. Stateful Inspection

- Monitor **connection state** to allow traffic only from **established sessions**

- Evaluates context of packets: sequence, timing, legitimacy



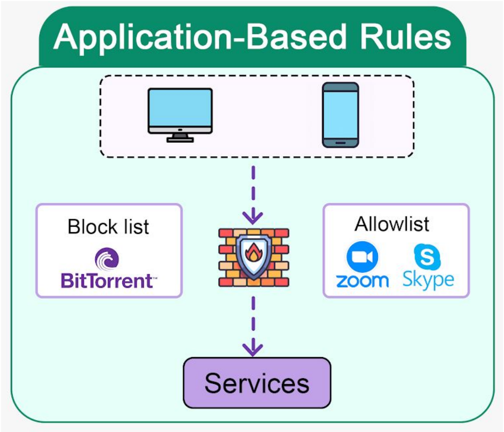
Use Case:

Prevent **unauthorized inbound traffic** while allowing valid response traffic.

Top 6 Firewall Use Cases

6. Application-Based Rules

- Filter traffic based on **application signatures or behaviors**
- Example:
    - Block: **BitTorrent**, **TOR**, **gaming apps**
    - Allow: **Microsoft Teams**, **Zoom**, **Slack**



Use Case:

Enforce company-wide **acceptable use policies** and limit bandwidth abuse.

VPN

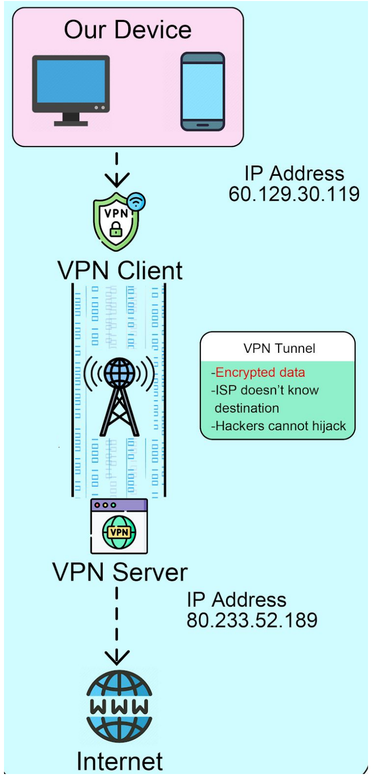
What is a VPN?

- A **Virtual Private Network (VPN)** creates a **secure, encrypted tunnel** between your device and a **remote server** over the public internet.
- It allows users to **send and receive data as if they were directly connected to a private network**.

How a VPN Works – 4-Step Flow

Step 1: Tunnel Initialization

- VPN client establishes a **secure tunnel** to the VPN server using protocols like:
  - **OpenVPN**
  - **IPSec**
  - **WireGuard**



VPN

Step 2: Encryption

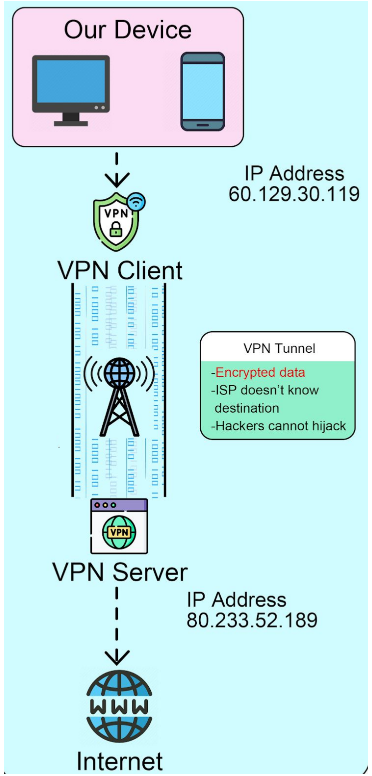
- All outgoing traffic from your device is **encrypted** before transmission.
- Prevents eavesdropping on public networks (e.g., Wi-Fi in cafes).

Step 3: IP Masking

- Your **real IP address is hidden**.
- Replaced with the **VPN server's IP address**, providing **anonymity**.

Step 4: Traffic Routing

- All internet traffic is routed through the **VPN server**, making it appear as if your activity originates from that server's location.



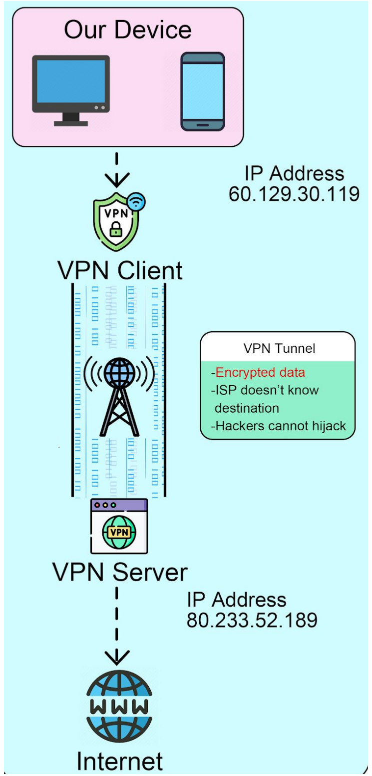
VPN

Advantage:

- ISP and others can't see your activity
- Your IP is masked
- Prevents MITM and packet sniffing attacks
- Protects sensitive data over public/untrusted network
- Geo-Spoofing : Access content as if you're in a different country

Disadvantage:

- Some websites (e.g., streaming services) block VPN traffic
- Due to encryption overhead and routing delays
- You must trust the VPN provider not to log or misuse data



Thank you!
Any Questions?