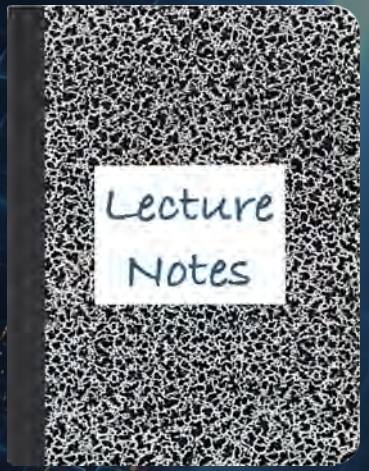
CS 417 – DISTRIBUTED SYSTEMS

Week 12: Security in Distributed Systems



Paul Krzyzanowski

Lecture Overview

- Security goals and the distributed system threat landscape
- Cryptographic building blocks:
  - Symmetric, asymmetric, hashes, TLS, PKI
- Identity and authentication: mTLS, broken authorization
- Tokens and delegation: OAuth, OIDC, JWTs
- Workload identity, Zero Trust, micro-segmentation
- Service meshes, API gateways, secret management

Security Fundamentals

Security Goals

- **Confidentiality:** keep data secret from unauthorized parties
- **Integrity:** detect unauthorized modification of data
- **Authentication:** establish who is on the other end of a connection
- **Authorization:** decide what an authenticated principal is allowed to do
- **Non-repudiation:** prove that a principal created or approved some data

Confidentiality vs. Integrity

These are separate properties: *one does not imply the other*

- **Confidentiality only**
  - Encrypted data can be modified without detection if there is no integrity check
  - Attackers who modify ciphertext can sometimes alter the plaintext in predictable ways
- **Integrity only**
  - Unmodified data can be fully visible to an attacker
- **Authenticated encryption** provides both at once

Common Authenticated Encryption functions

- **AES-GCM:**
  Advanced Encryption Standard (encryption) + Galois Counter Mode (authentication)
- **ChaCha20-Poly1305**:
  ChaCha20 (encryption) + Poly1305 (authentication)

Why Distributed Systems Are Different

Conventional system:

- The operating system enforces authentication and access

Distributed system:

- No single trust boundary: clients, gateways, services, and data stores are all separate
- Identity extends beyond users:
  services, *containers*, *databases*, *compute jobs*, and *agents* need identities

- Authorization is not a one-time check at the edge
- Secrets and keys are distributed across many machines
- A compromised component can be a stepping stone into the rest of the system

Threat Landscape

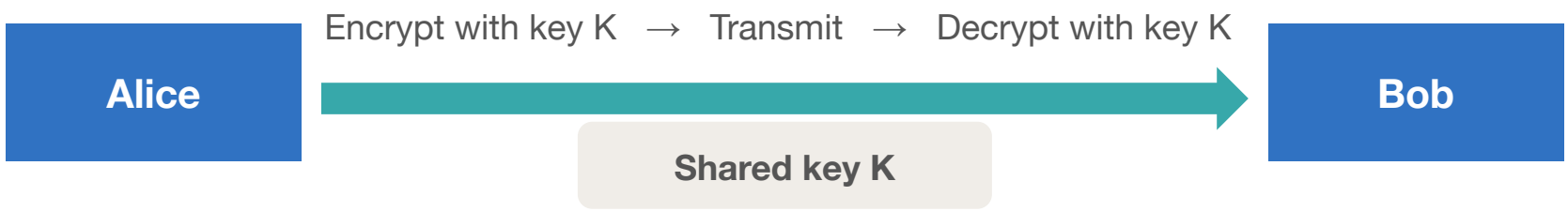
- **Eavesdropping:** attacker captures unencrypted traffic between services
- **Tampering:** messages modified in transit
- **Replay:** a valid captured message is retransmitted later
- **Stolen credentials:** authentication tokens, API keys, or certificates extracted from a compromised machine
- **Service impersonation:** attacker poses as a legitimate peer service
  - *This is why **mutual authentication** is important*

More Threats

- **Confused deputy**
  A service is tricked into using its permissions on behalf of an attacker
- **Lateral movement**
  A compromised component used as a foothold into the rest of the system
- **Over-privileged services**
  Excess permissions amplify the damage from a compromise
- **Broken authorization**
  A service authenticates correctly but fails object-level checks
- **Leaked secrets**
  Credentials end up in config files, container images, repositories, or logs

Cryptographic Foundations

Symmetric Encryption



Both parties must already hold this key

Problem: how do two parties share a key securely over an untrusted network?

Common symmetric encryption functions

- **AES**: block encryption, fast on newer CPUs
- **ChaCha20**: byte encryption, fast in older or embedded CPUs

Asymmetric (Public Key) Cryptography

Key pair = **public key** (shareable) + **private key** (never shared)

- **Encrypt data to a recipient:** ***encrypt using their public key***
  - Only the private key holder can decrypt
- **Sign your content (authentication/integrity):** ***encrypt using your private key***
  - Anyone with the corresponding public key can verify the signature
- Public keys can be **shared** over an untrusted channel
  - Used to establish symmetric session keys
  - Encrypt a symmetric key with the other side's public key
  - Only they will be able to decrypt
- Too slow for bulk data

Common asymmetric encryption functions

- **RSA**: the first and still widely used
- **ECC**: elliptic curve cryptography – faster & newer

Hashes, MACs, and Digital Signatures

- **Hash:** deterministic digest; detects accidental corruption; does not prove origin
- **MAC:** uses a shared secret key on a hash
  - Proves the message came from someone who holds the key
- **Digital signature:** uses a private key on a hash
  - Verifiable by anyone with the corresponding public key
  - Provides non-repudiation (only the user with the private key could have signed it)

Common functions

- **SHA-256**, **SHA-512**: common hash functions
- **MD5**, **SHA-1**: older hash functions (don't use)
- **HMAC**: standard MAC on top of any secure hash

Freshness and Replay Resistance

Integrity checks verify that a message was not modified, *not* that it is a fresh copy

Example:

- An attacker captures a valid signed "*transfer $1000*" request and retransmits it later
- The signature still passes: the original message was genuine

Freshness mechanisms:

- Nonces, timestamps, sequence numbers, short expiration times

A valid signed message replayed later still passes signature verification

DEFENSES

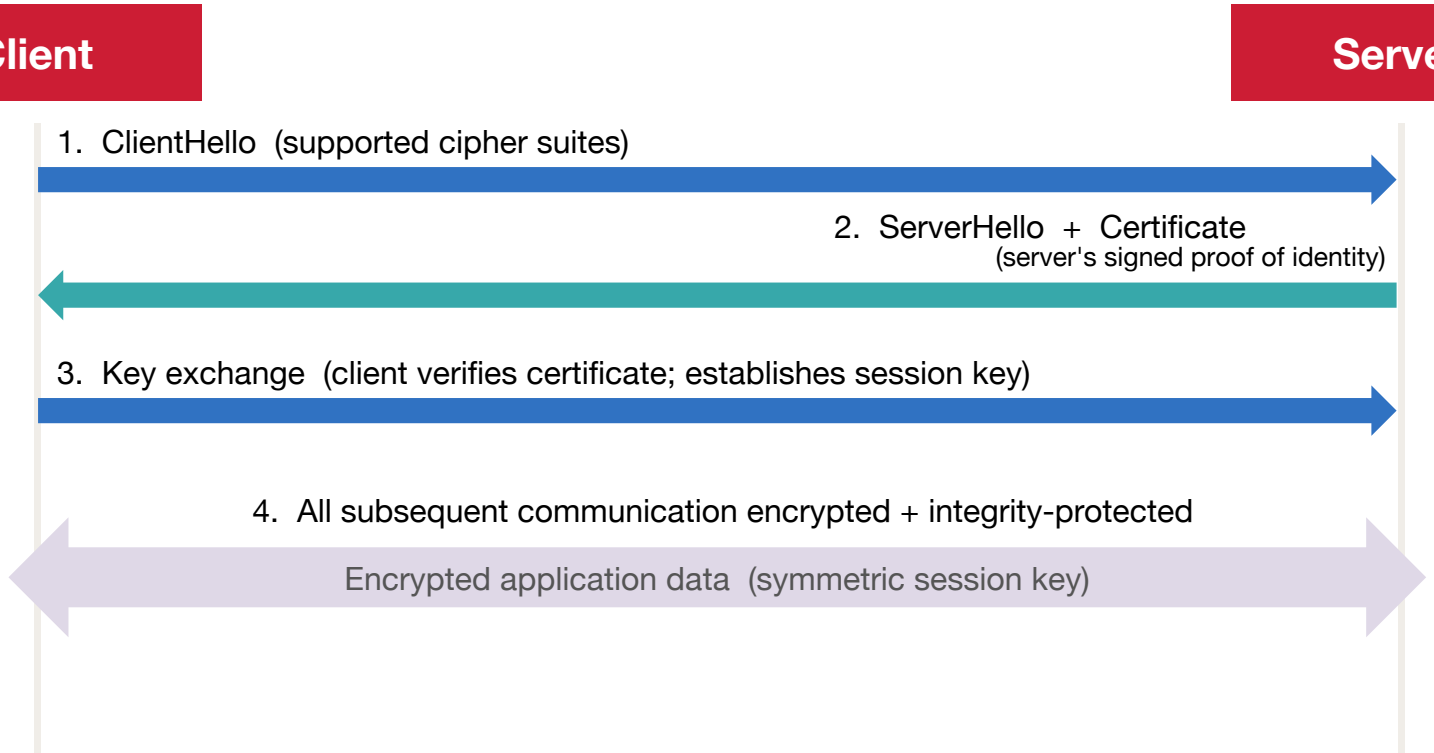
Nonce: a unique random value in each message.
The receiver tracks used nonces and rejects any repeat.

Timestamp: include the send time and reject messages older than a short window (a few seconds or minutes).

Transport Layer Security (TLS)

- Standard protocol for securing communication over untrusted networks
- Protects HTTPS traffic, gRPC connections, and service-to-service calls
- Server presents a certificate; client validates it against a trusted CA
- Asymmetric operations establish a shared symmetric session key
- Result: confidentiality, integrity, and server authentication

TLS Handshake



Certificates and PKI

Certificate

- Binds a public key to an identity (the **subject**), signed by a **certificate authority** (**CA**)
- **Subject** may be a domain name, service name, or workload identifier

  Example subject: api.example.com

  Example subject:
  cn=CN=Paul Krzyzanowski, OU=Department of Computer Science, O=Rutgers University, L=Piscataway, ST=NJ, C=US

A CA signature means: *"I vouch that this public key belongs to this subject"*

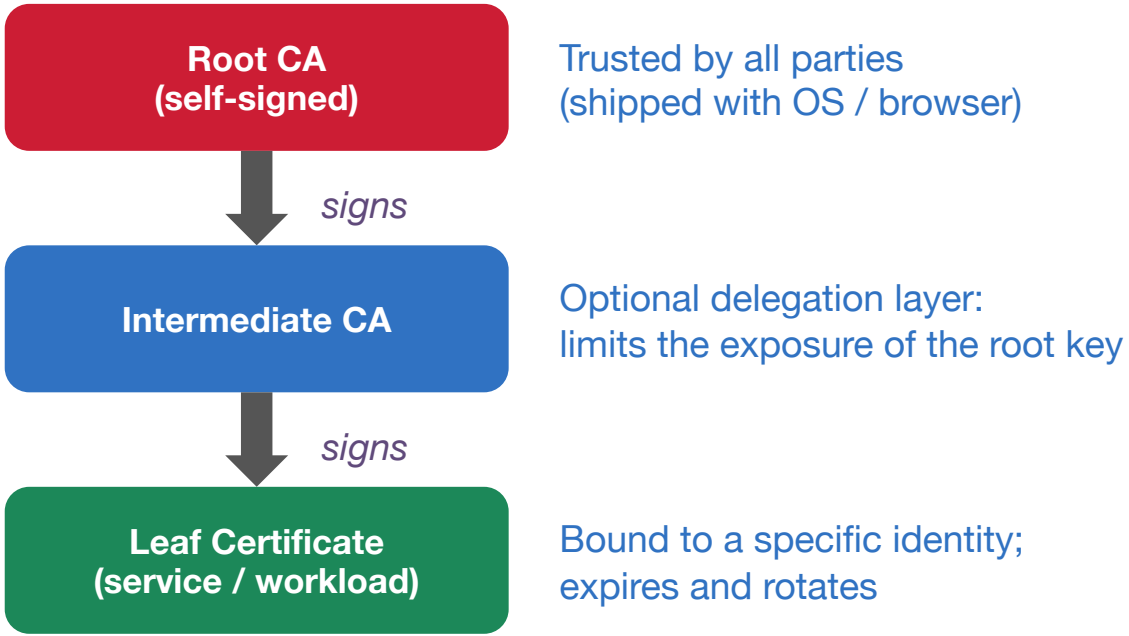
- Certificates
  - Have an expiration: systems rotate certificates (issue new ones periodically)
  - Can be revoked

PKI: Public Key Infrastructure

A security framework that manages digital certificates and public-key encryption

PKI Trust Chain

Trust chains:
Root CA signs intermediate CAs; intermediate CAs sign leaf certificates

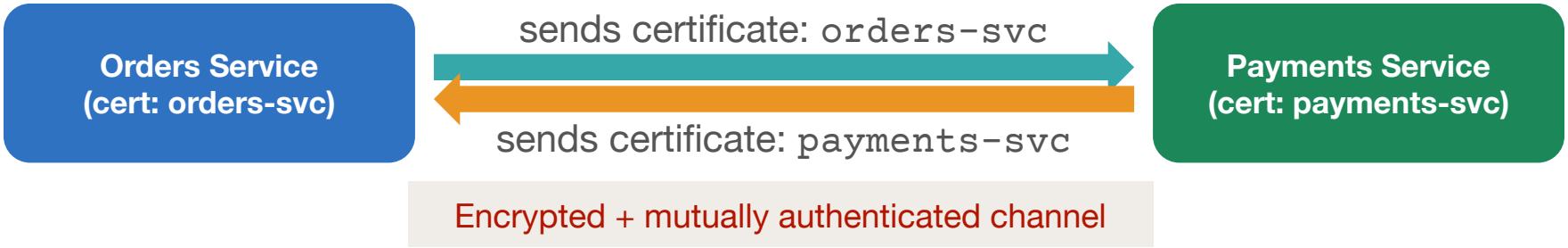


Identity and Authentication

Mutual TLS (mTLS)

- Standard TLS authenticates only the server
- Mutual TLS (mTLS) authenticates both sides
  - Each party presents a certificate to the other side
- Not used with end users (like the browser sessions) but critical for services
  - *Payments* service can verify: "*this connection is from the orders service*"
  - "*Inside the cluster*" becomes a verified cryptographic property, not a network assumption
- **Main challenge**: certificate issuance and rotation must be automated at scale

mTLS: Bidirectional Authentication



Without mTLS:
"connection from IP 10.0.1.12"

With mTLS:
"connection from orders-svc (cert verified)"

Authentication vs. Authorization

Authentication: who are you?

Authorization: what are you allowed to do?

A system can answer the first correctly while getting the second completely wrong

A valid authentication token at the entry point does not mean every resource inside is accessible

Each service must make its own authorization decision

– *It cannot delegate this fully*

Authorization at Multiple Layers

- **Edge:** is this client allowed to call this API at all?
- **Service:** is this caller allowed to invoke this specific operation?
- **Resource:** is this caller allowed to access this specific record or field?

None of these layers can fully delegate to the layer above

A valid entry token says nothing about access to a specific customer order

Broken Object-Level Authorization (BOLA)

Authentication and authorization are different checks: passing one does not substitute for the other

BOLA is the #1 API security risk (OWASP API Security Top 10)

Every endpoint must ask: does this caller own this specific resource?

Example:

- Suppose an API returns order details for an authenticated caller: `GET /orders/12345`
- An attacker has any valid token and changes the order ID in the request:

```
GET /orders/12346
GET /orders/12347
```

- **Authentication passed**: the token is genuine
- **Authorization missing**: no check that this caller owns this order

Appears consistently in API security audits of real production systems

Tokens and Delegation

What Is a Token?

Token: a credential issued by an authentication service after a successful login

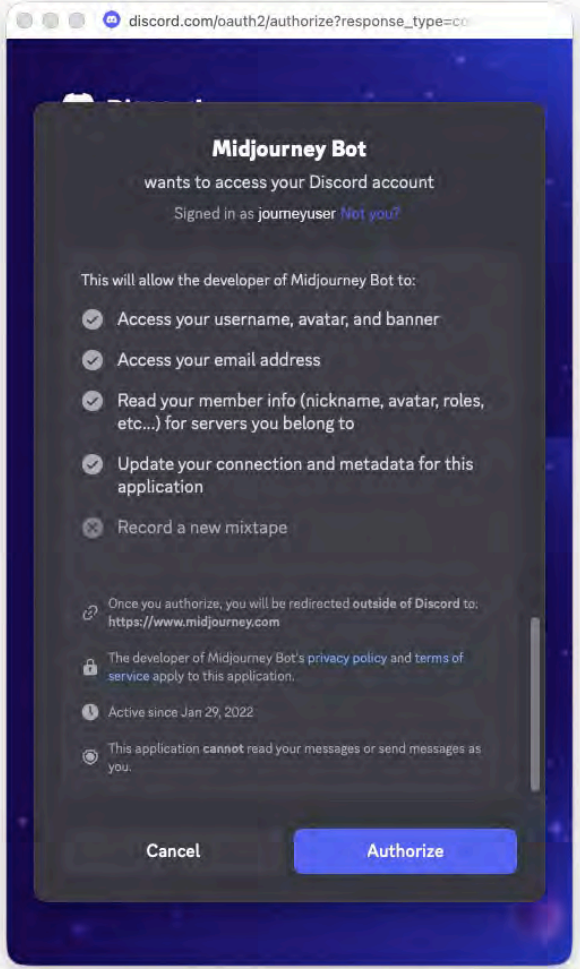
- Presented on subsequent requests to prove the caller was authenticated
- May contain information for the receiver to make an authorization decision
- **Has a limited lifetime**: once it expires, the caller must obtain a new one
  - Avoids sending a password on every request

OAuth 2.1

Authorization framework,
not an authentication protocol

> How do you allow an application to access ***specific*** resources from a service without giving away the user's password?

- Works for user-facing flows (acting on behalf of a user) and machine-to-machine flows
- Output is an **access token**
  - The service validates the token and decides what access to grant



OpenID Connect (OIDC)

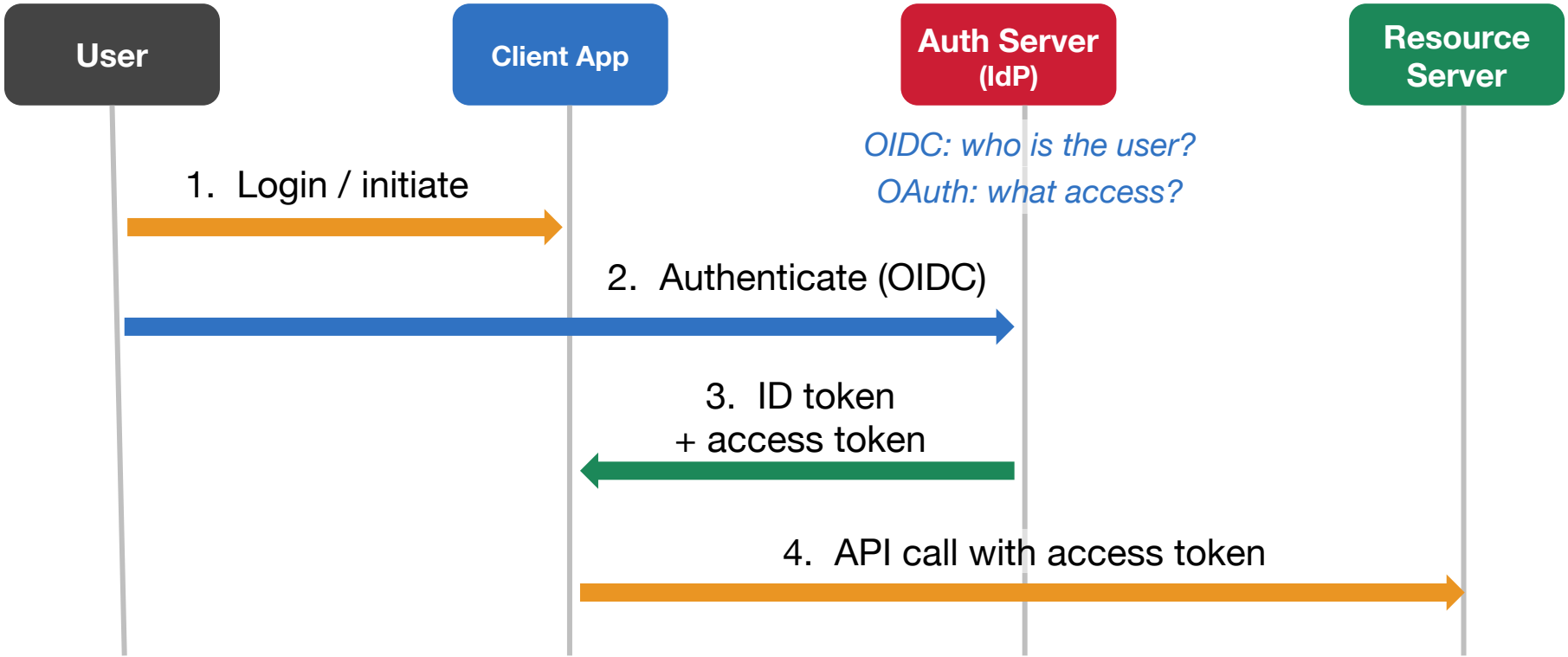
Identity layer built on top of OAuth 2.0

- Lets a client verify user identity via an **identity provider** (IdP) service:
  - Google, Microsoft Entra ID, Okta, Facebook, Apple
- OAuth was designed for *authorization*, not *authentication*
  - OAuth produces an access token
  - OIDC produces an ID token (who is the user?)

Typical usage:

- OIDC for user login
- OAuth for API access tokens

OAuth / OIDC Flow



JSON Web Tokens (JWR)

- A compact, **self-contained** token format:

  header . payload . signature

- **Header:** signing algorithm and token type
- **Payload:** a list of claims (attributes)
  - **aud** (audience), **sub** (subject), **iss** (issuer), **exp** (expiry), **iat** (issued at)
- **Signature:** cryptographic proof that the issuer created this specific payload
- OAuth access tokens and OIDC ID tokens are frequently encoded as JWTs

JWT Structure

Header	.	Payload (claims)	.	Signature
algorithm: HS256 type: JWT		(sub, iss, aud, exp, iat) Inluding custom fields		HMAC-SHA256 (or RSA / ECDSA)

base64url(Header) . base64url(Payload) . base64url(Signature)

Receiver verifies signature locally: no roundtrip to the auth server required

OAuth access tokens and OIDC ID tokens are frequently encoded as JWTs, but JWTs can carry any claims

JWT Trade-offs

- Local signature verification
  - Fast, no dependency on the auth server at request time
  - Verifying a JWT signature proves authenticity, not that the token is still valid
- **Cost**: ***revocation requires coordination***
  - A revoked token stays valid until it expires
- Same tradeoff as distributed state
  - Local reads are fast, but global consistency costs coordination

Short token lifetime is the most common mitigation for the revocation problem

Use short-lived access tokens (minutes) with longer-lived refresh tokens

For high-value operations, use token introspection to validate against the auth server on each request

Workload Identity

The Workload Identity Problem

- Users are not the only principals in a distributed system
  - Every service, container, VM, batch job, and serverless function needs an identity
- "Inside the cluster" is a network location, not a verified identity
  - *We can't rely on IP addresses as proof of identity*
- When *orders* makes an API call to *payments*,
  payments must know which specific workload is calling

Authorization policy depends on **caller identity**

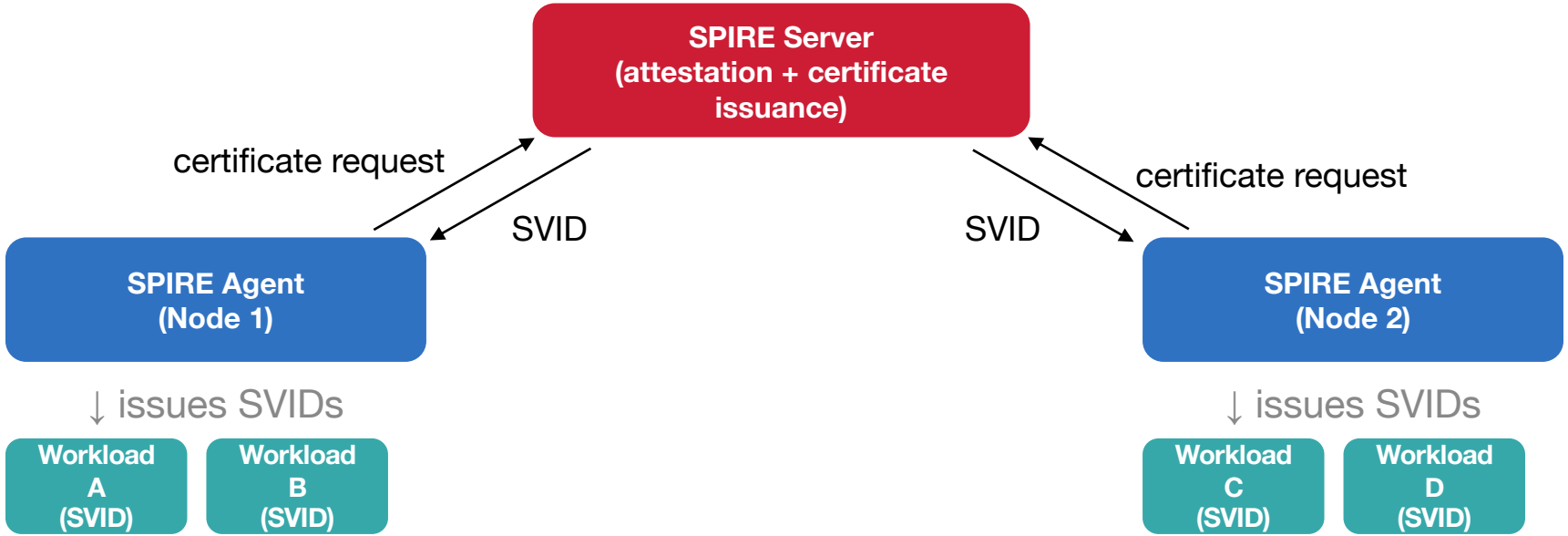
Workload Identity: giving processes cryptographically verifiable credentials

SPIFFE and SPIRE

- **SPIFFE**: open standard for workload identity in distributed systems
  - Each workload gets an SVID: a short-lived certificate encoding its identity
  - SVID example: `spiffe://cluster.local/ns/default/sa/orders-service`
- The certificate is used directly in mTLS connections
- **SPIRE**: the reference implementation
  - Issues and rotates SVIDs automatically

SPIFFE = Secure Production Identity Framework for Everyone

SPIFFE/SPIRE Architecture



SVID = SPIFFE Verifiable Identity Document: a short-lived X.509 certificate for each workload

Cloud IAM and Workload Identity Federation

IAM =Identity and Access Management

- Cloud IAM controls which identities can access which cloud resources:
  - Storage, queues, databases, secrets, …

Over-privileged services amplify the damage from a compromise

- **Traditional approach**:
  - A long-lived key file proves the service's identity to the cloud provider
- **Workload identity federation**:
  - The service presents its SPIFFE certificate
  - The cloud provider issues a short-lived token in return
- **No long-lived key files needed**

Zero Trust

Zero Trust Architecture

Traditional model

- Trust the internal network & concentrate security at the boundary
- Network firewalls restrict traffic between the internet & internal network
- This model breaks for cloud, microservices, remote workers, and shared infrastructure
  - It also breaks when people move devices into and out of the internal network

Zero Trust principle: network location does not imply trust

Zero trust model

- All requests, even from inside the cluster, need authentication and authorization
- Trust is earned by verified credentials and policy, not by network position

Deploying Zero Trust

- **Authenticate** every service and workload
  - Not just users at the edge or remote systems
- **Encrypt** all traffic: internal as well as external
- Push **authorization decisions** as close to the resource as possible

The **principle of least privilege** states that every component should have access only to the resources it actually needs and nothing more

- Prefer **short-lived credentials:** a compromised token will have less value
- Apply the **principle of least privilege** at every layer
  - IAM, service mesh policy, data access

Assume a partial compromise will happen

- *Design for blast radius containment*

Micro-segmentation

- **Old model**
  - The entire internal network is one trusted zone
  - Lateral movement is easy
- **Micro-segmentation**
  - Divide the system into fine-grained zones
  - Explicitly control which services can communicate
- Each service is allowed to communicate only with the services it legitimately needs
- Reduces blast radius: a compromise of one service does not spread automatically

Service Mesh and Gateway

API Gateways: Handling External Traffic

Handles incoming external traffic from clients and third parties

Functions:

TLS termination | Token validation | Rate limiting | Routing

- Centralizes common edge functions so individual services do not reimplement them
- Limitations
  - Does not protect internal service-to-service traffic
  - A compromised internal service bypasses the gateway entirely

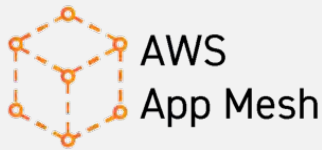
Service Meshes: Handling Internal Traffic

Goal: secure all service-to-service traffic
inside the cluster without modifying application code

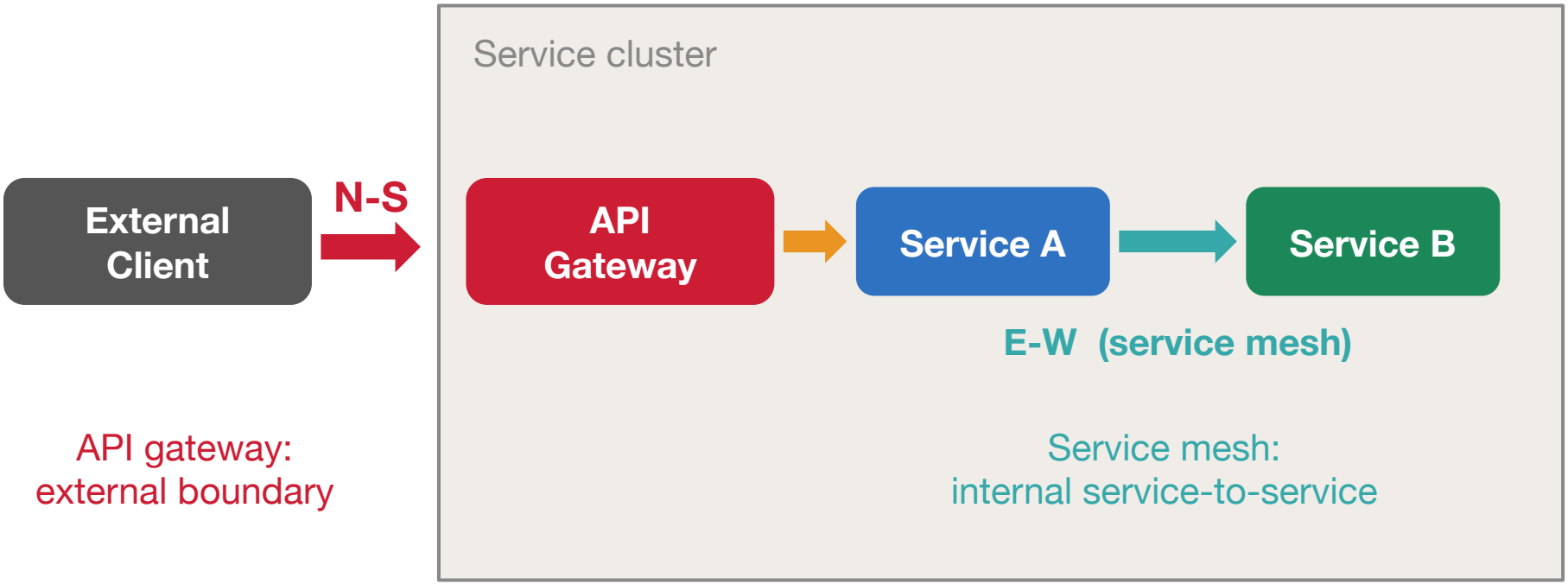
Inserts a proxy alongside each service to handle all communications

- Applications need no code changes
  - The proxy handles mTLS, identity, and policy
- Provides:
  - Automatic mTLS • Workload identity • Authorization policy enforcement • Telemetry

Popular service meshes:



North-South vs. East-West Traffic



Implementing Micro-segmentation

- **Network controls:** firewall rules restrict which services can establish connections at all
- **Service mesh:** authorization policies verify caller identity on every connection
- **Application:** per-operation authorization checks on the specific resource
- Callers must present a verified workload identity, not just arrive from an allowed IP address

Defense in depth: an attacker must bypass multiple independent controls

Secret Management

Secret Management

Systems need API keys, database passwords, signing keys, and TLS private keys

- **Challenge**:
  - Distribute keys safely, rotate regularly, revoke when compromised, audit all access
- **Dedicated systems:**
  - HashiCorp Vault, AWS Secrets Manager, GCP Secret Manager, Azure Key Vault
- **Goal:**
  - Short-lived credentials issued at runtime to authenticated workloads

Dynamic secrets: Vault generates a unique credential per request, valid for a defined duration

Common Secret Failures

- **Version control**
  - Secrets in source code or config files that get committed to version control
- **Container images**
  - Credentials baked into container images at build time
- **Environment variables**
  - Secrets in environment variables that appear in process listings or log output
- Long-lived credentials that have never been rotated
- *base64 encoding is not encryption*

SC Media A CRA Resource | CyberRisk ALLIANCE

Data Security, API security, Identity, Cloud Security

Thousands of API credentials exposed on public websites

April 2, 2026

By SC Staff

(Adobe Stock)

As reported by The Register, a recent analysis of 10 million websites has uncovered nearly 2,000 API credentials found scattered across 10,000 webpages. Researchers highlight that this exposure method, often overlooked compared to code repositories, poses a direct threat to sensitive infrastructure.

Rotation and Revocation

- **Any credential can leak:**
  - Certificates, API keys, database passwords
- **Short lifetimes** limit the damage window from a stolen credential
- **Automation is good**
  - If rotation requires manual steps across many machines, the design is brittle
  - Rotation must be automated, tested regularly, and treated as routine
  - If rotation requires manual downtime, it will be skipped or delayed after a compromise

Common Design Mistakes

Common Design Mistakes

- Assuming the internal network is trusted
- Authenticating users at the edge but not authenticating services to each other
- Using long-lived shared secrets instead of short-lived per-workload credentials
- Treating JWTs as a security architecture rather than a compact token format
- Relying on the entry gateway for all security and skipping authorization inside the system

More Design Mistakes

- Checking authentication but not per-object authorization
- Over-privileged service accounts
- Storing secrets in repositories, container images, or environment variable dumps
- Failing to plan for revocation and rotation
- No replay resistance: signed messages without freshness mechanisms

Putting It Together

Example: Online Store

1. User authenticates via OpenID Connect (OIDC)
   - IdP issues ID token and OAuth access token
2. API gateway validates access token, applies rate limits, routes to frontend
3. Frontend calls *orders* service over mTLS
   - *orders* service verifies the caller's identity
4. Orders calls *payments* over mTLS
   - *payments* makes its own authorization decision

- Database credentials retrieved at runtime from a secret manager

The Full Security Stack

OIDC: authenticates the user • **OAuth**: access • **JWTs**: the token format

TLS: encrypts communication & verifies server's identity
mTLS: encrypts & mutually authenticates both sides

SPIFFE: provides workload identity certificates used in mTLS

API gateway: controls incoming external traffic

Service mesh: controls internal service-to-service calls

Secret management: no static credentials, automated rotation

The End