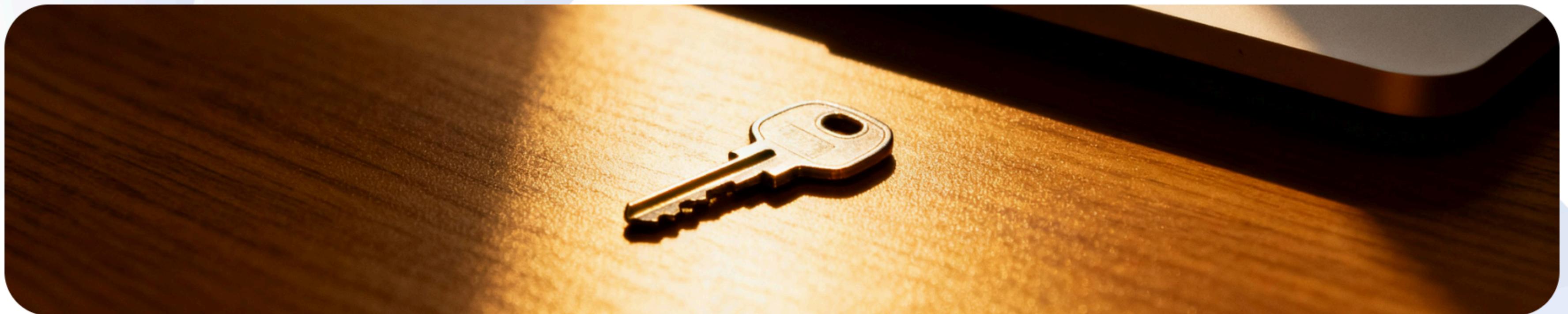


PROVING API IDENTITY WITH PKI AND RSA-2048

Authenticity, integrity, non-repudiation for APIs

Practical FastAPI integration with automated key management, JSON and binary payload support, and acceptable performance.

ESTHER HIKARI
Presenter



ASYMMETRIC CRYPTOGRAPHY AND RSA-2048

Public/private keys for signatures and encryption; security balanced with performance

CORE PRINCIPLES OF ASYMMETRIC CRYPTOGRAPHY



- Uses paired public and private keys for trust
- Enables encryption and digital signatures
- Private key secrecy is critical to security

WHY RSA-2048 IS CHOSEN



- Provides strong cryptographic security
- Common standard for signatures and key exchange
- Balanced security and performance for many uses

PRACTICAL TRADE-OFFS AND STRATEGY



- Computationally heavier than symmetric algorithms
- Use asymmetric for signing and key exchange
- Use symmetric crypto for bulk encryption

IMPLEMENTATION TIP: PROTECT PRIVATE KEYS



- Store keys in secure vaults or hardware modules
- Limit exposure in runtime environments
- Enforce access controls and rotated keys

INTRODUCTION: APPLICATION-LEVEL TRUST GAPS

Prove API request origin without shared secrets



PROBLEM

- MICROSERVICES GROWTH INCREASES CYBERSECURITY CHALLENGES
- TLS DOES NOT ADDRESS APPLICATION-LAYER AUTHENTICATION
- MANY APIs LACK REQUEST SOURCE VERIFICATION



OBJECTIVE

- ESTABLISH CRYPTOGRAPHIC REQUEST AUTHENTICITY BETWEEN SENDER API - RECEIVER API WITHOUT SHARED SECRETS



CHALLENGE

- ENSURE CONSISTENT MESSAGE CANONICALIZATION ACROSS LANGUAGES AND PLATFORMS FOR VERIFIABLE SIGNATURES

HYBRID CRYPTOGRAPHY: AUTHENTICATE WITH ASYMMETRIC, ENCRYPT WITH SYMMETRIC

Preserve performance while ensuring strong authentication

ASYMMETRIC FOR AUTHENTICATION

- Use asymmetric signatures solely for authentication and integrity
- Protects identity and non-repudiation without encrypting bulk payloads
- Common pattern: sign with RSA or ECDSA



SYMMETRIC FOR BULK ENCRYPTION

- Use symmetric algorithms such as AES for confidentiality
- Optimized for high throughput and low latency
- Combine with ephemeral keys for forward secrecy when needed

DIGITAL SIGNATURES AND PUBLIC KEY INFRASTRUCTURE

Authentication, integrity, non-repudiation with SHA-256 and PSS



AUTHENTICATION: SIGNER IDENTITY TIED TO PUBLIC KEY

Verifies who created the message



INTEGRITY: SHA-256 ENSURES MESSAGE HAS NOT CHANGED

Detects any modification to the signed data



NON-REPUDIATION: SIGNER CANNOT DENY THE SIGNATURE

Provides legal and forensic evidence



SIGNATURE METHOD: SHA-256 WITH PSS PADDING

Probabilistic padding used for signatures



SIGNING STEPS: CREATE DIGEST, APPLY PSS, BASE64 ENCODE

Include resulting string in Authorization header



WHY PSS: PROBABILISTIC PADDING REDUCES DETERMINISTIC RISKS

Adds randomness to prevent signature reuse attacks



SIGN HEADERS AND METADATA: INCLUDE CANONICALIZED VALUES

Prevents header tampering and inconsistent parsing

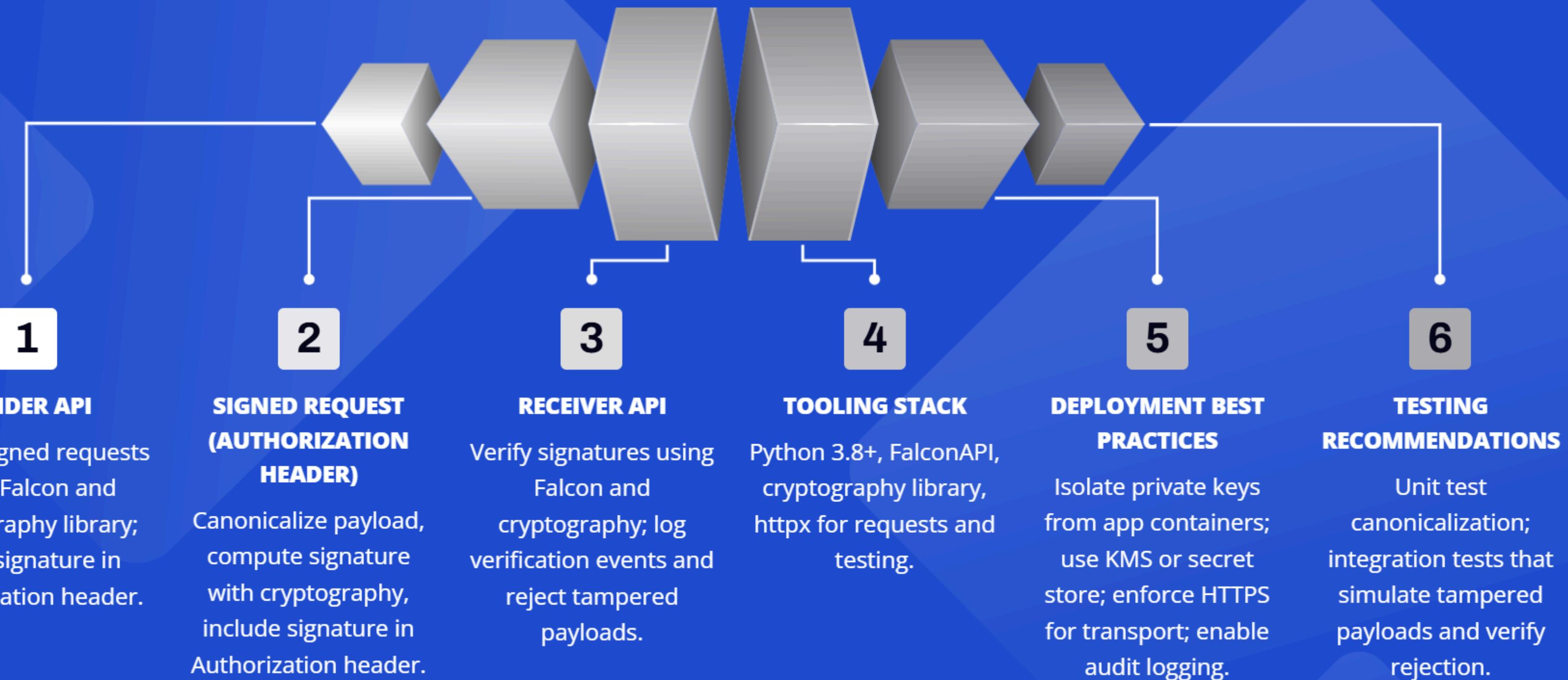


AUTHORIZATION HEADER PLACEMENT: CENTRALIZES VERIFICATION

Requires consistent header handling across servers

SECURE REQUEST SIGNING ARCHITECTURE

Sender API signing to Receiver API verification with deployment and testing guidance



SECURE KEY GENERATION, STORAGE AND MANAGEMENT

Practical workflow and recommended protections for private and public keys



1

GET_OR_CREATE_KEYS

Generate or load an existing RSA/ECC key pair using generate_key_pair()

2

SAVE PRIVATE AND PUBLIC

Persist private_key.pem and public_key.pem using save/load functions

3

SHARE PUBLIC KEY

Distribute public_key.pem to receivers or public key store for discovery

SIGNATURE CREATION FOR SENDER API

Deterministic JSON signing with PKI Authorization

1 SERIALIZE JSON WITH SORTED KEYS

Ensure consistent key ordering across languages

2 ENCODE UTF-8 THEN SIGN WITH PRIVATE KEY USING SHA-256 AND PSS

Use SHA-256 with PSS padding for robust signatures

3 BASE64-ENCODE THE SIGNATURE

Prepare binary signature for header transport

4 INCLUDE IN AUTHORIZATION HEADER AS "PKI "

Header value format required by Sender API

5 ADD TIMESTAMP AND NONCE TO SIGNED PAYLOAD (FUTURE WORK)

Protect against replay attacks with time and nonce

6 VALIDATE MAXIMUM HEADER SIZES FOR LARGE SIGNATURES OR BINARY UPLOADS

Avoid server rejects due to oversized headers

7 TESTING TIP: NEGATIVE TESTS THAT ALTER FIELDS OR ORDERING

Confirm altered payloads are rejected by verifier

8 ENSURE DETERMINISTIC SERIALIZATION ACROSS LANGUAGES

Normalize whitespace, numeric formats, and key ordering

9 CODE SNIPPET: SERIALIZE -> UTF8 -> SIGN(SHA-256+PSS) -> BASE64 -> AUTHORIZATION

Concise callout of the code flow

SIGNATURE VERIFICATION AND MESSAGE FORMATS

Receiver API steps, canonicalization, multipart handling, and defensive logging

1

EXTRACT AUTHORIZATION HEADER

Read Authorization header and check prefix equals "PKI " before decoding signature.

2

DECODE SIGNATURE

Base64-decode the signature value after the PKI prefix for verification.

3

RECONSTRUCT CANONICAL BODY

Recreate body JSON with keys sorted; apply identical canonicalization rules used by sender.

4

HANDLE CONTENT TYPES

For application/json use canonical JSON; for multipart/form-data canonicalize non-file fields and treat files consistently.

5

VERIFY SIGNATURE

Verify using the sender's public key with SHA-256 and PSS padding.

6

REJECT ON FAILURE

Return 401 Unauthorized when verification fails.

7

LOG FAILURES SAFELY

Record verification failures but avoid logging sensitive payloads; include metadata only.

8

RATE LIMIT FAILED ATTEMPTS

Throttle repeated failed authentications to mitigate abuse.