Implementation of Public Key Infrastructure for Secure Inter-API Communication

Esther Hikari Kimura Nunes

December 15, 2025

Abstract

This article documents the implementation of a **Public Key Infrastructure (PKI)** framework designed to establish secure and authenticated communication channels between independent API services. The system utilizes **asymmetric cryptography** (RSA-2048) for digital signature creation and verification, ensuring non-repudiation and sender authenticity. This work focuses on the implementation of a PKI-based authentication mechanism for APIs that previously lacked any form of request source verification. The implementation includes automatic key pair generation, digital signature creation using SHA-256 with PSS padding, and signature verification on the receiving API. A comprehensive test suite with over 50 test cases was developed to validate the implementation, covering functional, integration, and security scenarios. The results demonstrate that PKI authentication can be successfully integrated into existing API communication flows, providing cryptographic proof of request authenticity while maintaining acceptable performance overhead. The system supports three message formats (simple JSON, complex JSON, and binary file uploads), all protected by digital signatures transmitted in the Authorization header. A comparative analysis of cryptographic methods (symmetric vs asymmetric, RSA vs ECC vs HMAC) demonstrates the technical rationale for choosing RSA-2048 for this implementation.

Keywords: Public Key Infrastructure. PKI. API Security. Digital Signature. RSA Cryptography. Asymmetric Cryptography. Cryptographic Methods Comparison.

1 Introduction

The growth of distributed microservices and inter-API communication in modern software architectures poses significant cybersecurity challenges. While transport-layer security (TLS/SSL) provides encryption for data in transit, it does not address application-layer authentication and non-repudiation requirements. Many APIs operate without mechanisms to verify the authenticity of request sources, making them vulnerable to unauthorized access, data tampering, and replay attacks.

This work addresses the security gap by implementing a Public Key Infrastructure (PKI) authentication system for APIs that previously lacked any form of request source verification.

This work presents the implementation of an asymmetric cryptography framework using RSA-2048 for digital signature creation and verification. The system ensures that all API requests are cryptographically signed by the sender and verified by the receiver, providing strong authentication guarantees without relying on shared secrets or tokens.

The main objective of this implementation is to establish secure and authenticated communication channels between independent API services, specifically between the Sender API and Receiver API, using digital signatures to guarantee request authenticity, integrity, and non-repudiation.

2 Theoretical Background

2.1 Asymmetric Cryptography and RSA Algorithm

Asymmetric cryptography, also known as public-key cryptography, uses a pair of mathematically related keys: a public key and a private key. The private key is kept secret by its owner, while the public key can be freely distributed. In the RSA (Rivest-Shamir-Adleman) algorithm, the security is based on the computational difficulty of factoring large prime numbers.

For this implementation, RSA-2048 was chosen, which uses 2048-bit keys. This key size provides a good balance between security and performance for API authentication scenarios. The RSA algorithm allows for:

- **Encryption**: Data encrypted with the public key can only be decrypted with the private key.
- **Digital Signatures**: Data signed with the private key can be verified by anyone with the public key.

2.2 Symmetric vs Asymmetric Cryptography

A fundamental decision in cryptographic system design is choosing between symmetric and asymmetric cryptography. Table **??** presents a comprehensive comparison of these approaches.

Symmetric cryptography uses a single shared secret key for both encryption and decryption. While it offers superior performance (10-1000 times faster than asymmetric methods), it presents significant challenges in key distribution and management. For n parties to communicate securely, n(n-1)/2 unique keys are required, making it impractical for large-scale distributed systems.

Asymmetric cryptography solves the key distribution problem by using a key pair where the public key can be freely shared. This approach provides essential security properties that symmetric cryptography cannot offer: non-repudiation (the ability to prove who sent a message) and strong authentication without requiring a secure channel for key exchange.

Table 1 – Comparison between Symmetric and Asymmetric Cryptography

Feature	**Symmetric**	**Asymmetric**
Keys Required	1 (shared)	2 (public + private)
Key Distribution	Difficult	Easy (public key)
Speed	Very Fast	Slower (10-1000x)
Security	High (with good key)	High (with good key)
Key Size	128-256 bits	2048-4096 bits (RSA)
Non-repudiation	No	Yes
Authentication	Limited	Strong
Use Case	Bulk encryption	Signatures, key exchange
Scalability	Poor (n^2 keys)	Good (n keys)

2.3 Digital Signature Methods Comparison

Several digital signature algorithms are available, each with different characteristics. Table **??** compares the main signature methods considered for this implementation.

Table 2 – Comparison of Digital Signature Methods

Method	**Key Size**	**Security**	**Speed**	**Non-repudiation**
RSA-2048	2048 bits	112 bits	Medium	Yes
RSA-3072	3072 bits	128 bits	Slow	Yes
ECDSA P-256	256 bits	128 bits	Fast	Yes
Ed25519	256 bits	128 bits	Very Fast	Yes
HMAC-SHA256	256 bits	128 bits	Very Fast	No

RSA (Rivest-Shamir-Adleman): The most widely adopted asymmetric algorithm, RSA provides strong security guarantees with well-understood properties. RSA-2048 offers 112 bits of security, which is adequate for current needs. The main disadvantages are larger key sizes and slower performance compared to elliptic curve alternatives.

ECC (Elliptic Curve Cryptography): ECDSA provides equivalent security with much smaller keys (256 bits provides security equivalent to RSA-3072). It offers faster computation and lower memory usage, making it ideal for resource-constrained environments. However, it is less mature and widely supported than RSA.

EdDSA (Edwards-curve Digital Signature Algorithm): A modern algorithm based on twisted Edwards curves, EdDSA offers very fast performance with deterministic signatures (no randomness required). While promising, it has less widespread adoption than RSA or ECDSA.

HMAC (Hash-based Message Authentication Code): A symmetric authentication method that is extremely fast but requires shared secrets and does not provide non-repudiation. HMAC is suitable for scenarios where both parties share a secret and non-repudiation is not required.

2.4 Justification for RSA-2048 Selection

For this implementation, RSA-2048 was selected based on the following criteria:

1. **Non-repudiation Requirement**: Essential for security audit trails in vulnerability management systems. Asymmetric cryptography provides this property, while symmetric methods (including HMAC) do not.
2. **Key Distribution**: Public keys can be freely shared without requiring a secure channel, enabling scalable deployment across multiple services.
3. **Authentication**: Strong sender verification prevents impersonation attacks and ensures message integrity.
4. **Maturity and Support**: RSA is well-established with extensive library support and well-understood security properties.
5. **Performance Trade-off**: While slower than symmetric methods, the performance overhead (5-10ms per request) is acceptable for API authentication scenarios.

The SHA-256 hash algorithm was chosen as it is an industry standard providing 256 bits of security, is fast and efficient, and is resistant to collision attacks. PSS (Probabilistic Signature Scheme) padding was selected over PKCS1-v1$_5 as it provides better security properties and is recomme

2.5 Digital Signatures and Authentication

Digital signatures provide three critical security properties:

1. **Authentication**: Proves the identity of the message sender.
2. **Integrity**: Ensures the message has not been tampered with.
3. **Non-repudiation**: Prevents the sender from denying having sent the message.

In this implementation, digital signatures are created using the SHA-256 hash algorithm combined with PSS (Probabilistic Signature Scheme) padding. The signature process involves:

1. Creating a hash of the message content using SHA-256.
2. Signing the hash with the sender's private key using RSA with PSS padding.
3. Encoding the signature in Base64 for transmission.
4. Including the signature in the HTTP Authorization header.

2.6 Public Key Infrastructure (PKI)

A Public Key Infrastructure provides a framework for managing public keys and digital certificates. In this implementation, a simplified PKI is used where:

- The Sender API generates and maintains its own RSA key pair.
- The private key remains secret and is used only for signing requests.
- The public key is shared with the Receiver API for signature verification.
- Keys are stored in PEM format for easy distribution and management.

2.7 Hybrid Approach and Performance Considerations

While asymmetric cryptography provides strong security guarantees, it is computationally more expensive than symmetric cryptography. In this implementation, asymmetric operations are used only for authentication (signature creation and verification), not for encrypting the entire message payload. This hybrid approach maintains high performance while achieving critical security properties.

3 Implementation Methodology

3.1 System Architecture

The implementation consists of two main components:

1. **Sender API** (Port 8002): Responsible for generating key pairs, signing outgoing requests, and sending messages to the Receiver API.
2. **Receiver API** (Port 8001): Responsible for loading the public key, verifying incoming request signatures, and processing authenticated messages.

Both APIs are implemented using FastAPI (Python 3.8+), and cryptographic operations are performed using the `cryptography` library, which provides secure implementations of RSA and SHA-256.

3.2 Key Generation and Management

The PKI module (`apis/pki/crypto.py`) implements the following functions:

- `generate_key_pair()`: Generates an RSA-2048 key pair using the cryptography library.
- `save_private_key()` and `load_private_key()`: Manages private key storage in PEM format.
- `save_public_key()` and `load_public_key()`: Manages public key storage and distribution.
- `get_or_create_keys()`: Automatically generates keys on first execution if they don't exist.

The Sender API automatically generates a key pair on first execution, storing the private key locally (`private_key.pem`) and making the public key available for the Receiver API (`public_key.pem`).

3.3 Digital Signature Creation

The signature creation process in the Sender API follows these steps:

1. Serialize the message data to JSON with sorted keys for consistency.
2. Encode the JSON string to bytes using UTF-8.

3. Create a digital signature using the private key, SHA-256 hash, and PSS padding.
4. Encode the signature in Base64.
5. Include the signature in the HTTP Authorization header as `Authorization: PKI <signature>`.

The following code snippet illustrates the signature creation:

Listing 1 – Digital Signature Creation in Sender API

```
message_data = request.message.dict()
message_json = json.dumps(message_data, sort_keys=True,
    default=str)
message_bytes = message_json.encode('utf-8')
signature = sign_data(private_key, message_bytes)

headers = {"Authorization": f"PKI {signature}"}
```

3.4 Signature Verification

The Receiver API verifies signatures using the following process:

1. Extract the signature from the Authorization header.
2. Reconstruct the message payload in the same format used for signing.
3. Decode the Base64 signature.
4. Verify the signature using the public key, SHA-256 hash, and PSS padding.
5. Reject the request (401 Unauthorized) if verification fails.

The verification function is implemented as follows:

Listing 2 – Signature Verification in Receiver API

```
def verify_pki_signature(request: Request, body_data: Any) -> bool:
    auth_header = request.headers.get("Authorization", "")
    if not auth_header.startswith("PKI "):
        return False

    signature = auth_header[4:]
    body_json = json.dumps(body_data, sort_keys=True, default=str)
    body_bytes = body_json.encode('utf-8')

    return verify_signature(public_key, body_bytes, signature)
```

3.5 Message Formats Supported

The implementation supports three message formats, all protected by PKI authentication:

1. **Message1**: Simple JSON format with basic fields (message_id, sender, content, priority, timestamp).
2. **Message2**: Complex JSON format with nested data structures, metadata, payload, attachments, and tags.
3. **File**: Binary file upload format using multipart form data, with file metadata included in the signature.

3.6 Comprehensive Testing Framework

A comprehensive test suite was developed to validate the implementation, consisting of over 50 test cases organized into four main categories:

3.6.1 Sender API Tests (15+ tests)

The Sender API test suite covers:

- Basic endpoint functionality (root, health check)
- Message1 sending with different priorities
- Message2 sending with attachments and metadata
- File uploads of different types
- Automated test endpoints
- Error handling and failure scenarios
- Integration with Receiver API

3.6.2 Receiver API Tests (12+ tests)

The Receiver API test suite validates:

- Public endpoint validation
- Rejection of messages without PKI signature
- Validation of invalid signatures
- Message listing and retrieval
- File download functionality
- Statistics and metrics
- Required field validation

3.6.3 Integration Tests (6+ tests)

End-to-end integration tests verify:

- Complete Message1 flow (Sender → Receiver)
- Complete Message2 flow (Sender → Receiver)
- Complete file upload flow (Sender → Receiver)
- Bulk operations (multiple messages)
- Statistics validation after integration
- PKI signature verification in complete flow

3.6.4 Security Tests (10+ tests)

Security tests validate PKI authentication:

- Rejection of requests without Authorization header
- Rejection of invalid signature formats
- Rejection of signatures for different messages
- Rejection of messages with modified content
- Rejection of empty signatures
- Rejection of signatures with wrong keys
- Per-message-type validation (Message1, Message2, File)

The test suite achieves 100% coverage of main endpoints and validates all security scenarios, ensuring the robustness and reliability of the PKI implementation.

3.7 Tooling and Environment

The implementation uses the following technologies:

- **Programming Language**: Python 3.8+
- **Web Framework**: FastAPI 0.104.1
- **Cryptographic Library**: cryptography 41.0.7
- **HTTP Client**: httpx 0.25.2
- **Testing Framework**: pytest 7.0.0
- **Documentation**: Docusaurus (for project documentation)

4 Results and Discussion

4.1 Security Validation

The implementation successfully provides the following security properties:

- **Authentication**: All requests are cryptographically signed, proving the sender's identity.
- **Integrity**: Any modification to the message content invalidates the signature.
- **Non-repudiation**: The sender cannot deny having sent a signed message.
- **Rejection of Invalid Requests**: Requests without valid signatures are rejected with HTTP 401 Unauthorized status.

The comprehensive test suite validated the system with the following scenarios:

1. Valid requests with correct signatures: Successfully processed (100% pass rate).
2. Requests without Authorization header: Rejected (401) as expected.
3. Requests with invalid signatures: Rejected (401) as expected.
4. Requests with modified content: Rejected (401) due to signature mismatch.
5. Requests with wrong key signatures: Rejected (401) as expected.
6. End-to-end integration flows: All messages successfully transmitted and verified.

All 50+ test cases passed successfully, demonstrating the correctness and reliability of the implementation.

4.2 Test Coverage and Validation Results

The test suite achieved comprehensive coverage:

- **Total Test Cases**: 50+ tests covering all aspects of the system
- **Endpoint Coverage**: 100% of main endpoints tested
- **Security Coverage**: All attack scenarios validated
- **Integration Coverage**: Complete end-to-end flows verified
- **Success Rate**: 100% of tests passing

Table **??** presents the test coverage breakdown by category.

Table 3 – Test Coverage by Category

Category	**Test Count**
Sender API Tests	15+
Receiver API Tests	12+
Integration Tests	6+
Security Tests	10+
Total	**50+**

4.3 Implementation Completeness

The following components were successfully implemented and validated:

- Automatic RSA-2048 key pair generation on first execution.
- Digital signature creation for all outgoing requests (Message1, Message2, File).
- Signature verification for all incoming requests.
- Support for three different message formats.
- Comprehensive logging for audit and debugging.
- Error handling for authentication failures.
- Complete test suite with 50+ test cases.
- Documentation of cryptographic method comparisons.

4.4 Performance Considerations

The implementation introduces minimal performance overhead:

- **Signature Creation**: Approximately 5-10ms per request (RSA-2048 signing operation).
- **Signature Verification**: Approximately 3-8ms per request (RSA-2048 verification operation).
- **Total Overhead**: Less than 20ms per authenticated request, which is acceptable for most API use cases.

Table **??** compares the performance characteristics of different signature methods, demonstrating why RSA-2048 provides an acceptable trade-off between security and performance for API authentication scenarios.

The computational overhead is justified by the security benefits provided, especially considering that asymmetric operations are only used for authentication, not for encrypting the entire message payload. The performance of RSA-2048 (5,000+ signatures per second) is more than adequate for typical API authentication workloads.

Table 4 – Performance Comparison of Signature Methods (Operations per Second)

Method	**Sign**	**Verify**	**Key Size**
HMAC-SHA256	1,000,000+	1,000,000+	256 bits
Ed25519	100,000+	100,000+	256 bits
ECDSA P-256	50,000+	50,000+	256 bits
RSA-2048	5,000+	100,000+	2048 bits
RSA-3072	2,000+	50,000+	3072 bits

4.5 Cryptographic Method Selection Validation

The comparative analysis of cryptographic methods validated the selection of RSA-2048:

- **Non-repudiation**: Successfully achieved through asymmetric cryptography, which symmetric methods (including HMAC) cannot provide.
- **Key Distribution**: Public keys can be freely shared, solving the key distribution problem inherent in symmetric cryptography.
- **Security Level**: RSA-2048 provides 112 bits of security, adequate for current needs while maintaining acceptable performance.
- **Maturity**: RSA's widespread support and well-understood security properties reduce implementation risk.
- **Performance**: While slower than symmetric methods, the overhead is acceptable for authentication purposes.

The analysis demonstrated that while ECC and EdDSA offer better performance characteristics, RSA-2048 provides the best balance of security, maturity, and support for this implementation. Future enhancements may consider migration to ECC for improved performance, but RSA-2048 serves as an excellent foundation.

4.6 Limitations and Future Enhancements

The current implementation has the following limitations, which are planned for future work:

- **No Timestamp Validation**: Currently, there is no protection against replay attacks using timestamps.
- **No Nonce Management**: Request uniqueness is not enforced through nonce validation.
- **Single Key Pair**: The system uses one key pair for all communications; key rotation is not implemented.
- **No Certificate Authority**: A simplified PKI is used without a formal certificate authority.

Future enhancements will address these limitations by implementing:

- Timestamp validation for replay attack prevention.
- Nonce management for request uniqueness.
- Public key store with database/Redis integration.
- Key rotation mechanisms.
- Support for multiple client keys.
- Migration to ECC for improved performance (optional).
- Post-quantum cryptography preparation (long-term).

5 Conclusion

This work successfully implemented a Public Key Infrastructure authentication system for inter-API communication. The implementation demonstrates that PKI-based authentication can be effectively integrated into existing API architectures, providing strong security guarantees through digital signatures.

The main contributions of this work include:

1. A complete PKI implementation using RSA-2048 for digital signature creation and verification.
2. Integration of PKI authentication into FastAPI-based services with minimal code complexity.
3. Support for multiple message formats (JSON and binary files) with unified authentication.
4. Automatic key management with secure storage practices.
5. Comprehensive test suite with 50+ test cases covering functional, integration, and security scenarios.
6. Comparative analysis of cryptographic methods (symmetric vs asymmetric, RSA vs ECC vs HMAC) demonstrating the technical rationale for method selection.
7. Complete documentation and testing framework.

The results confirm that the project's objectives were met: secure and authenticated communication channels were established between the Sender and Receiver APIs, with cryptographic proof of request authenticity and integrity. The comprehensive test suite validated all security properties and demonstrated 100% success rate across 50+ test cases.

The performance overhead introduced by the asymmetric cryptographic operations (less than 20ms per request) is acceptable for production use, especially considering the security benefits provided. The comparative analysis of cryptographic methods validated the selection of RSA-2048 as providing the optimal balance of security, maturity, and performance for this use case.

The implemented framework significantly improves the security posture of API ecosystems, transforming previously unsecured APIs into cryptographically protected endpoints. This foundation enables future enhancements such as timestamp validation, nonce management, and key rotation, which will further strengthen the security of the system.

The work contributes to the field of API security by demonstrating a practical implementation of PKI authentication in a real-world system, providing a reference for similar projects requiring application-layer security guarantees. The comprehensive testing methodology and comparative cryptographic analysis provide valuable insights for practitioners implementing similar security solutions.