Secure Messaging Client: Technical Documentation

xAI

October 20, 2025

Contents

1 Introduction

The Secure Messaging Client is a Python-based application designed to demonstrate secure communication using asymmetric and symmetric cryptography. It employs RSA (2048-bit) for key exchange and AES-256-GCM for message encryption, ensuring confidentiality, integrity, and authentication. This system is intended for educational purposes, such as a Phase 2 project in Code Theory and Cryptography, and simulates end-to-end encrypted messaging between users.

Key features include:

- User key pair generation and management.
- Session key establishment via RSA.
- Message encryption and decryption.
- Export/import capabilities for keys and messages.
- Interactive command-line interface for demonstration.

The system relies on the `cryptography` library for cryptographic primitives and does not handle network communication, focusing instead on local simulation of cryptographic operations.

2 System Overview

The application consists of two primary classes: `User` and `SecureMessagingClient`. The `User` class manages individual user identities, RSA key pairs, and session keys. The `SecureMessagingClient` class orchestrates multiple users, handles message exchanges, and provides utility functions for viewing, verifying, and sharing cryptographic data.

2.1 Architecture

- **Asymmetric Encryption (RSA)**: Used for secure exchange of symmetric session keys.
- **Symmetric Encryption (AES-GCM)**: Used for efficient encryption of messages once a session is established.
- **State Management**: Sessions are stored per user-pair, allowing key reuse for multiple messages.
- **Data Formats**: Keys and messages are handled in PEM (for RSA) and Base64/Hex (for AES outputs) for readability and portability.
- **Persistence**: Supports JSON-based export/import of keys and message packages.

The system operates in a local environment, simulating peer-to-peer interactions without actual network transmission. All operations include detailed logging for transparency in cryptographic steps.

3 Key Components

3.1 User Class

Represents an individual user with cryptographic capabilities.

- **Initialization**:
  - Generates a 2048-bit RSA key pair using `rsa.generate_private_key`.
  - Stores public and private keys, along with extracted components (e.g., modulus n, primes p and q) for display.
  - Maintains a dictionary of session keys per peer.

- **Key Export/Import**:
  - `get_public_key_pem()`: Exports public key in PEM format.
  - `save_keys_to_file()`: Saves full key pair (public and private) to JSON.
  - `load_keys_from_file()`: Loads key pair from JSON, recreating the User instance.
- **Session Management**:
  - `establish_session()`: Generates a 256-bit AES session key, encrypts it with the peer's RSA public key using OAEP padding (SHA-256), and stores it locally.
  - `decrypt_session_key()`: Decrypts a received encrypted session key using the user's private RSA key and stores it.
- **Message Encryption/Decryption**:
  - `encrypt_message()`: Uses AES-256-GCM with a random 96-bit IV. Returns IV, ciphertext, and authentication tag in Base64 and Hex formats.
  - `decrypt_message()`: Decrypts using the stored session key, IV, ciphertext, and tag. Verifies integrity via GCM's built-in authentication.

3.2 SecureMessagingClient Class

Manages multiple users and coordinates messaging.

- **User Management**:
  - `create_user()`: Creates a new User and displays detailed RSA key info (e.g., digit lengths of large numbers for security insight).
  - `show_user_info()`: Displays user's key details and active sessions.
- **Messaging**:
  - `send_message()`: Establishes a session if needed, encrypts the message, and logs it. Checks for recipient's private key availability.
  - `view_messages()`: Displays all messages with cryptographic details (e.g., plaintext, IV, ciphertext, tag, lengths).
  - `verify_message()`: Decrypts a specific message and checks if it matches the original plaintext.
- **Export/Import Features**:
  - `export_message_package()`: Creates a JSON package with encrypted message, sender's public key, and encrypted session key (for v1.1 format).
  - `import_message_package()`: Loads a package, decrypts the session key (if included), decrypts the message, and adds to local history. Supports backward compatibility with v1.0 packages.
  - `export_public_key()`: Exports only the public key in JSON for sharing.
  - `import_public_key()`: Imports a public key, creating a limited User instance (no private key, can only send messages).
- **Interactive Interface**:
  - The `main()` function provides a menu-driven CLI for all operations, including a demo mode that creates users "Alice" and "Bob" and exchanges sample messages.

4 Cryptographic Mechanisms

4.1 Key Generation

- **RSA Key Pair**: 2048-bit keys with public exponent 65537. Primes p and q are generated securely via the `cryptography` library.
- **Session Key**: 256-bit random key generated via `os.urandom(32)`.
- **IV for AES**: 96-bit random nonce per message via `os.urandom(12)`.

4.2 Session Establishment

- Sender generates AES key and encrypts it with recipient's RSA public key (OAEP padding with SHA-256).
- Recipient decrypts using their private key.
- Session keys are reused for efficiency, reducing RSA operations.

4.3 Message Encryption

- **Algorithm**: AES-256 in GCM mode for authenticated encryption.
- **Process**:
  1. Generate IV.
  2. Create Cipher object with session key and IV.
  3. Encrypt plaintext (UTF-8 encoded).
  4. Produce ciphertext and 128-bit authentication tag.
- **Output**: Base64-encoded for portability, with Hex for debugging.

4.4 Message Decryption

- Reconstruct Cipher with session key, IV, and tag.
- Decrypt and verify integrity; GCM raises an exception on tampering.

4.5 Padding and Hashing

- RSA uses OAEP with MGF1 (SHA-256) for secure padding.
- Hashes use SHA-256 for consistency.

5 Features and Usage

5.1 User Management

- Create users with unique usernames.
- Import public keys to enable sending to external parties (without decryption capability).

5.2 Message Handling

- Send messages with automatic session setup.
- View history with cryptographic details.
- Verify decryption for integrity checks.

5.3 Sharing and Portability

- Export full keys for backup.
- Export public keys for distribution.
- Export message packages for offline sharing (includes encrypted session key in v1.1).

5.4 Demo Mode

- Automatically creates “Alice” and “Bob”.
- Sends three messages demonstrating session reuse.
- Displays full history.

5.5 Limitations

- Local-only; no network integration.
- No forward secrecy (session keys persist).
- Basic error handling; assumes trusted environment.

6 Security Analysis

6.1 Strengths

- **Key Sizes**: RSA-2048 and AES-256 provide strong resistance to brute-force attacks.
- **Authenticated Encryption**: GCM ensures messages cannot be tampered with undetected.
- **Secure Key Exchange**: RSA-OAEP prevents chosen-ciphertext attacks.
- **Randomness**: Uses OS-level entropy for keys and IVs.

6.2 Potential Weaknesses

- **Session Key Reuse**: Vulnerable to compromise if a key is exposed; no perfect forward secrecy.
- **No Key Revocation**: Once shared, public keys cannot be revoked.
- **Side-Channel Risks**: Logging of key hex values could leak info in production (intended for demo).
- **Dependency on Library**: Relies on `cryptography` for secure implementations; vulnerabilities there affect the system.
- **No Authentication Beyond Encryption**: Assumes users are authenticated via usernames; no signatures on messages.

6.3 Best Practices

- Use in isolated environments.
- Avoid logging sensitive data in production.
- Extend with digital signatures (e.g., RSA-PSS) for non-repudiation.

7 Usage Guide

7.1 Installation

- Requires Python 3.x and `cryptography` library (`pip install cryptography`).
- Run the script: `python rsa.py`.

7.2 Interactive Commands

- Select options from the menu (1–13).
- For demo: Choose 6 to run a full example.

7.3 Example Workflow

1. Create users “Alice” and “Bob”.
2. Export Alice’s public key and import it elsewhere if needed.
3. Send message from Alice to Bob.
4. Export the message package for sharing.
5. Import and decrypt on recipient’s side.

8 Conclusion

This Secure Messaging Client effectively demonstrates hybrid cryptography, combining RSA for key exchange with AES for efficient messaging. It serves as an educational tool to explore cryptographic concepts, with extensible features for further development, such as network integration or advanced security mechanisms. Future enhancements could include perfect forward secrecy using Diffie-Hellman or multi-factor authentication.