Secure Messaging Client: Technical Documentation

MASIKO NICHOLAS 2024/U/MMU/BCS/00163
AKANKUNDA PATIENCE 2024/U/MMU/BCS/01383
WASSWA ATIBU 2024/U/MMU/BCS/00061
BAGUMA IVAN 2024/U/MMU/BCS/00217

October 20, 2025

Prepared for: Phase 2 Project - Code Theory and Cryptography

Contents

1 Introduction **2**

2 System Overview **2**
2.1 Architecture 2

3 Key Components **2**
3.1 User Class 2
3.2 SecureMessagingClient Class 3

4 Cryptographic Mechanisms **4**
4.1 Key Generation 4
4.2 Session Establishment 4
4.3 Message Encryption 4
4.4 Message Decryption 4
4.5 Padding and Hashing 4

5 Features and Usage **4**
5.1 User Management 4
5.2 Message Handling 4
5.3 Sharing and Portability 5
5.4 Demo Mode 5
5.5 Limitations 5

6 Security Analysis **5**
6.1 Strengths 5
6.2 Potential Weaknesses 5
6.3 Best Practices 5

7 Usage Guide **6**
7.1 Installation 6
7.2 Interactive Commands 6
7.3 Example Workflow 6

8 Conclusion **6**

A Source Code **6**

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.

A Source Code

The complete source code for the Secure Messaging Client (`rsa.py`) is provided below.

```
"""
Secure Messaging Client
Phase 2 Project - Code Theory and Cryptography
Uses RSA for key exchange and AES for symmetric encryption
"""

from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os
import json
import base64
from datetime import datetime
import pickle


class User:
    """Represents a user with RSA key pair and active sessions"""

    def __init__(self, username):
        self.username = username
        print(f"\n Generating RSA-2048 key pair for {username}...")
```

```
24
25          # Generate RSA key pair (2048-bit)
26          self.private_key = rsa.generate_private_key(
27              public_exponent=65537,
28              key_size=2048,
29              backend=default_backend()
30          )
31          self.public_key = self.private_key.public_key()
32          self.sessions = {}  # {peer_username: session_key}
33
34          # Extract key components for display
35          private_numbers = self.private_key.private_numbers()
36          public_numbers = self.public_key.public_numbers()
37
38          self.key_info = {
39              'public_exponent': public_numbers.e,
40              'modulus': public_numbers.n,
41              'private_exponent': private_numbers.d,
42              'prime1': private_numbers.p,
43              'prime2': private_numbers.q
44          }
45
46          print(f" Key generation complete for {username}")
47
48      def get_public_key_pem(self):
49          """Export public key in PEM format"""
50          return self.public_key.public_bytes(
51              encoding=serialization.Encoding.PEM,
52              format=serialization.PublicFormat.SubjectPublicKeyInfo
53          ).decode('utf-8')
54
55      def save_keys_to_file(self, filepath):
56          """Save user's keys to a file"""
57          key_data = {
58              'username': self.username,
59              'public_key_pem': self.get_public_key_pem(),
60              'private_key_pem': self.private_key.private_bytes(
61                  encoding=serialization.Encoding.PEM,
62                  format=serialization.PrivateFormat.PKCS8,
63                  encryption_algorithm=serialization.NoEncryption()
64              ).decode('utf-8')
65          }
66
67          with open(filepath, 'w') as f:
68              json.dump(key_data, f, indent=2)
69          print(f"Keys saved to {filepath}")
70
71      @classmethod
72      def load_keys_from_file(cls, filepath):
73          """Load user from saved keys file"""
74          with open(filepath, 'r') as f:
75              key_data = json.load(f)
76
77          # Create user instance
78          user = cls.__new__(cls)
79          user.username = key_data['username']
80          user.sessions = {}
81
82          # Load private key
83          user.private_key = serialization.load_pem_private_key(
84              key_data['private_key_pem'].encode('utf-8'),
85              password=None,
86              backend=default_backend()
```

```
87          )
88
89          # Load public key
90          user.public_key = serialization.load_pem_public_key(
91              key_data['public_key_pem'].encode('utf-8'),
92              backend=default_backend()
93          )
94
95          # Extract key components for display
96          private_numbers = user.private_key.private_numbers()
97          public_numbers = user.public_key.public_numbers()
98
99          user.key_info = {
100             'public_exponent': public_numbers.e,
101             'modulus': public_numbers.n,
102             'private_exponent': private_numbers.d,
103             'prime1': private_numbers.p,
104             'prime2': private_numbers.q
105         }
106
107         print(f"Keys loaded for user {user.username}")
108         return user
109
110     def get_key_info(self):
111         """Get formatted key information for display"""
112         return self.key_info
113
114     def establish_session(self, peer_username, peer_public_key):
115         """Establish a new session with a peer"""
116         print(f"\n Establishing session with {peer_username}...")
117
118         # Generate AES-256 session key
119         session_key = os.urandom(32)  # 256 bits
120         print(f" Generated AES-256 session key: {session_key.hex()[:32]}...")
121
122         # Encrypt session key with peer's RSA public key
123         print(f" Encrypting session key with {peer_username}'s RSA public key...")
124         encrypted_session_key = peer_public_key.encrypt(
125             session_key,
126             padding.OAEP(
127                 mgf=padding.MGF1(algorithm=hashes.SHA256()),
128                 algorithm=hashes.SHA256(),
129                 label=None
130             )
131         )
132
133         print(f" Encrypted session key (hex):
                {encrypted_session_key.hex()[:64]}...")
134
135         # Store session key
136         self.sessions[peer_username] = session_key
137
138         return encrypted_session_key, session_key
139
140     def decrypt_session_key(self, encrypted_session_key, peer_username):
141         """Decrypt a session key received from a peer"""
142         print(f"\n Decrypting session key from {peer_username}...")
143         print(f" Received encrypted key (hex):
                {encrypted_session_key.hex()[:64]}...")
144
145         # Check if this user has a private key
146         if self.private_key is None:
147             print(f"[!] Cannot decrypt session key - user '{self.username}' has no
```

```
                private key!")
148             print("This user was imported with public key only and cannot receive
                    messages.")
149             raise ValueError(f"User {self.username} cannot decrypt - no private
                    key available")
150
151         session_key = self.private_key.decrypt(
152             encrypted_session_key,
153             padding.OAEP(
154                 mgf=padding.MGF1(algorithm=hashes.SHA256()),
155                 algorithm=hashes.SHA256(),
156                 label=None
157             )
158         )
159
160         print(f" Decrypted session key: {session_key.hex()[:32]}...")
161
162         # Store session key
163         self.sessions[peer_username] = session_key
164         return session_key
165
166     def encrypt_message(self, message, peer_username):
167         """Encrypt a message using AES-GCM with the session key"""
168         if peer_username not in self.sessions:
169             raise ValueError(f"No session established with {peer_username}")
170
171         session_key = self.sessions[peer_username]
172
173         print(f"\n Encrypting message with AES-256-GCM...")
174         print(f" Using session key: {session_key.hex()[:32]}...")
175
176         # Generate random IV (96 bits for GCM)
177         iv = os.urandom(12)
178         print(f" Generated IV (hex): {iv.hex()}")
179
180         # Create AES-GCM cipher
181         cipher = Cipher(
182             algorithms.AES(session_key),
183             modes.GCM(iv),
184             backend=default_backend()
185         )
186         encryptor = cipher.encryptor()
187
188         # Encrypt the message
189         plaintext_bytes = message.encode('utf-8')
190         print(f" Plaintext bytes (hex): {plaintext_bytes.hex()[:64]}...")
191
192         ciphertext = encryptor.update(plaintext_bytes) + encryptor.finalize()
193         tag = encryptor.tag
194
195         print(f" Ciphertext (hex): {ciphertext.hex()[:64]}...")
196         print(f" Authentication tag (hex): {tag.hex()}")
197
198         # Return IV, ciphertext, and authentication tag
199         return {
200             'iv': base64.b64encode(iv).decode('utf-8'),
201             'ciphertext': base64.b64encode(ciphertext).decode('utf-8'),
202             'tag': base64.b64encode(tag).decode('utf-8'),
203             'iv_hex': iv.hex(),
204             'ciphertext_hex': ciphertext.hex(),
205             'tag_hex': tag.hex()
206         }
207
```

```
208     def decrypt_message(self, encrypted_data, peer_username):
209         """Decrypt a message using AES-GCM with the session key"""
210         if peer_username not in self.sessions:
211             raise ValueError(f"No session established with {peer_username}")
212 
213         session_key = self.sessions[peer_username]
214 
215         print(f"\n Decrypting message from {peer_username}...")
216         print(f" Using session key: {session_key.hex()[:32]}...")
217         print(f" IV (hex): {encrypted_data.get('iv_hex', 'N/A')}")
218         print(f" Ciphertext (hex): {encrypted_data.get('ciphertext_hex',
                'N/A')[:64]}...")
219         print(f" Auth tag (hex): {encrypted_data.get('tag_hex', 'N/A')}")
220 
221         # Decode the encrypted data
222         iv = base64.b64decode(encrypted_data['iv'])
223         ciphertext = base64.b64decode(encrypted_data['ciphertext'])
224         tag = base64.b64decode(encrypted_data['tag'])
225 
226         # Create AES-GCM cipher
227         cipher = Cipher(
228             algorithms.AES(session_key),
229             modes.GCM(iv, tag),
230             backend=default_backend()
231         )
232         decryptor = cipher.decryptor()
233 
234         # Decrypt the message
235         plaintext_bytes = decryptor.update(ciphertext) + decryptor.finalize()
236         print(f" Decrypted bytes (hex): {plaintext_bytes.hex()[:64]}...")
237 
238         return plaintext_bytes.decode('utf-8')
239 
240 
241 class SecureMessagingClient:
242     """Main messaging client managing users and messages"""
243 
244     def __init__(self):
245         self.users = {}  # {username: User object}
246         self.messages = []  # List of all messages
247 
248     def create_user(self, username):
249         """Create a new user with RSA key pair"""
250         if username in self.users:
251             print(f"[!] User '{username}' already exists!")
252             return False
253 
254         user = User(username)
255         self.users[username] = user
256 
257         # Display key information
258         self._display_key_info(username)
259         return True
260 
261     def _display_key_info(self, username):
262         """Display detailed RSA key information"""
263         user = self.users[username]
264         key_info = user.get_key_info()
265 
266         print(f"\n{'='*80}")
267         print(f"RSA KEY INFORMATION: {username}")
268         print(f"{'='*80}")
269 
```

```
270         # Convert large numbers to strings for display
271         modulus_str = str(key_info['modulus'])
272         private_exp_str = str(key_info['private_exponent'])
273 
274         print(f"Public Exponent (e): {key_info['public_exponent']}")
275         print(f"\nModulus (n):")
276         print(f"  Length: {len(modulus_str)} digits")
277         print(f"  First 50 digits: {modulus_str[:50]}...")
278         print(f"  Last 50 digits: ...{modulus_str[-50:]}")
279 
280         print(f"\nPrivate Exponent (d):")
281         print(f"  Length: {len(private_exp_str)} digits")
282         print(f"  First 50 digits: {private_exp_str[:50]}...")
283         print(f"  Last 50 digits: ...{private_exp_str[-50:]}")
284 
285         print(f"\nPrime 1 (p): {len(str(key_info['prime1']))} digits")
286         print(f"Prime 2 (q): {len(str(key_info['prime2']))} digits")
287 
288         print(f"\nPublic Key (PEM):")
289         pem = user.get_public_key_pem()
290         print(pem[:100] + "..." if len(pem) > 100 else pem)
291         print("="*80)
292 
293     def send_message(self, sender_username, recipient_username, message):
294         """Send an encrypted message from sender to recipient"""
295         if sender_username not in self.users:
296             print(f"[!] Sender '{sender_username}' not found!")
297             return False
298 
299         if recipient_username not in self.users:
300             print(f"[!] Recipient '{recipient_username}' not found!")
301             return False
302 
303         sender = self.users[sender_username]
304         recipient = self.users[recipient_username]
305 
306         # Check if recipient can receive messages (has private key)
307         if recipient.private_key is None:
308             print(f"[!] Cannot send message to '{recipient_username}'!")
309             print("Recipient was imported with public key only and cannot decrypt
                messages.")
310             print("The recipient must create their own account to receive
                messages.")
311             return False
312 
313         print(f"\n{'='*80}")
314         print(f"SENDING MESSAGE: {sender_username}     {recipient_username}")
315         print(f"{'='*80}")
316         print(f" Plaintext: {message}")
317 
318         # Check if session exists
319         session_established = False
320         if recipient_username not in sender.sessions:
321             print(f"\n No existing session found. Establishing new session...")
322             # Establish new session
323             encrypted_session_key, session_key = sender.establish_session(
324                 recipient_username,
325                 recipient.public_key
326             )
327 
328             # Recipient decrypts and stores the session key
329             recipient.decrypt_session_key(encrypted_session_key, sender_username)
330 
```

```
331             session_established = True
332             print(f" New session key established between {sender_username} and
                    {recipient_username}")
333         else:
334             print(f"\n Using existing session key")
335 
336         # Encrypt message with AES
337         encrypted_message = sender.encrypt_message(message, recipient_username)
338 
339         # Store message
340         msg_record = {
341             'from': sender_username,
342             'to': recipient_username,
343             'plaintext': message,
344             'encrypted': encrypted_message,
345             'session_established': session_established,
346             'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S')
347         }
348         self.messages.append(msg_record)
349 
350         print(f"\n Message encrypted and sent successfully!")
351         print(f"{'='*80}")
352 
353         return True
354 
355     def view_messages(self):
356         """Display all messages with detailed cryptographic information"""
357         if not self.messages:
358             print("\n[!] No messages yet.")
359             return
360 
361         print("\n" + "="*100)
362         print("MESSAGE HISTORY - CRYPTOGRAPHIC DETAILS")
363         print("="*100)
364 
365         for i, msg in enumerate(self.messages, 1):
366             print(f"\n{'   '*100}")
367             print(f"       MESSAGE #{i}")
368             print(f"{'   '*100}")
369             print(f"From: {msg['from']}     To: {msg['to']}")
370             print(f"Time: {msg['timestamp']}")
371 
372             if msg['session_established']:
373                 print(" [NEW SESSION KEY EXCHANGED VIA RSA]")
374 
375             print(f"\n PLAINTEXT:")
376             print(f"   {msg['plaintext']}")
377 
378             enc = msg['encrypted']
379             print(f"\n ENCRYPTED DATA:")
380             print(f"   IV (Base64): {enc['iv']}")
381             print(f"   IV (Hex):    {enc['iv_hex']}")
382             print(f"   Ciphertext (Base64): {enc['ciphertext'][:80]}...")
383             print(f"   Ciphertext (Hex):    {enc['ciphertext_hex'][:80]}...")
384             print(f"   Auth Tag (Base64): {enc['tag']}")
385             print(f"   Auth Tag (Hex):    {enc['tag_hex']}")
386 
387             print(f"\nENCRYPTION INFO:")
388             plaintext_len = len(msg['plaintext'])
389             ciphertext_len = len(enc['ciphertext'])
390             print(f"   Plaintext length:  {plaintext_len} characters")
391             print(f"   Ciphertext length: {ciphertext_len} Base64 characters")
392             print(f"   Expansion: {ciphertext_len/plaintext_len:.2f}x (due to
```

```
                Base64 encoding)")

    def verify_message(self, message_index):
        """Decrypt and verify a message"""
        if message_index < 0 or message_index >= len(self.messages):
            print("[!] Invalid message index!")
            return

        msg = self.messages[message_index]
        recipient = self.users[msg['to']]

        print(f"\n{'='*80}")
        print(f"VERIFYING MESSAGE #{message_index + 1}")
        print(f"{'='*80}")

        try:
            decrypted = recipient.decrypt_message(msg['encrypted'], msg['from'])
            print(f"\n Message decrypted successfully!")
            print(f"Decrypted text: {decrypted}")
            print(f"Original text:  {msg['plaintext']}")
            print(f"Match: {decrypted == msg['plaintext']}")
        except Exception as e:
            print(f"Decryption failed: {e}")

        print("="*80)

    def show_user_info(self, username):
        """Display user's cryptographic information"""
        if username not in self.users:
            print(f"[!] User '{username}' not found!")
            return

        user = self.users[username]

        print(f"\n{'='*80}")
        print(f"USER INFORMATION: {username}")
        print(f"{'='*80}")

        self._display_key_info(username)

        print(f"\nACTIVE SESSIONS:")
        if user.sessions:
            for peer, session_key in user.sessions.items():
                print(f"      {peer}: AES-256 key = {session_key.hex()[:32]}...")
        else:
            print("  No active sessions")
        print("="*80)

    def export_message_package(self, message_index, filepath):
        """Export an encrypted message with sender's public key for sharing"""
        if message_index < 0 or message_index >= len(self.messages):
            print("[!] Invalid message index!")
            return False

        msg = self.messages[message_index]
        sender = self.users[msg['from']]
        recipient = self.users[msg['to']]

        # Get the session key for this conversation
        session_key = sender.sessions.get(msg['to'])
        if not session_key:
            print("[!] No session key found for this conversation!")
            return False
```

```
455
456         # Encrypt session key with recipient's public key
457         encrypted_session_key = recipient.public_key.encrypt(
458             session_key,
459             padding.OAEP(
460                 mgf=padding.MGF1(algorithm=hashes.SHA256()),
461                 algorithm=hashes.SHA256(),
462                 label=None
463             )
464         )
465
466         # Create shareable package
467         package = {
468             'format_version': '1.1',
469             'sender_username': msg['from'],
470             'sender_public_key': sender.get_public_key_pem(),
471             'recipient_username': msg['to'],
472             'encrypted_message': msg['encrypted'],
473             'encrypted_session_key':
                    base64.b64encode(encrypted_session_key).decode('utf-8'),
474             'timestamp': msg['timestamp'],
475             'session_established': msg['session_established']
476         }
477
478         with open(filepath, 'w') as f:
479             json.dump(package, f, indent=2)
480
481         print(f"\n Message package exported to: {filepath}")
482         print(f"This file contains both the encrypted message AND the session
                key.")
483         print(f"It can be shared with {msg['to']} for complete decryption.")
484         return True
485
486     def import_message_package(self, filepath, recipient_username):
487         """Import and decrypt a shared message package"""
488         try:
489             with open(filepath, 'r') as f:
490                 package = json.load(f)
491
492             if recipient_username not in self.users:
493                 print(f"[!] Recipient '{recipient_username}' not found!")
494                 print("Please create the recipient user first.")
495                 return False
496
497             recipient = self.users[recipient_username]
498
499             # Check if recipient has private key (needed for decryption)
500             if recipient.private_key is None:
501                 print(f"[!] User '{recipient_username}' doesn't have a private
                        key!")
502                 print("You can only decrypt messages with your own private key.")
503                 return False
504
505             # Verify this message is for the correct recipient
506             if package['recipient_username'] != recipient_username:
507                 print(f"[!] Message is for '{package['recipient_username']}', not
                        '{recipient_username}'")
508                 return False
509
510             print(f"\n{'='*80}")
511             print(f"IMPORTING SHARED MESSAGE")
512             print(f"{'='*80}")
513             print(f"From: {package['sender_username']}")
```

```
514             print(f"To: {package['recipient_username']}")
515             print(f"Timestamp: {package['timestamp']}")
516             print(f"Package format: {package.get('format_version', '1.0')}")
517
518             # Load sender's public key
519             sender_public_key = serialization.load_pem_public_key(
520                 package['sender_public_key'].encode('utf-8'),
521                 backend=default_backend()
522             )
523
524             # Create temporary sender user for session establishment
525             if package['sender_username'] not in self.users:
526                 temp_sender = User.__new__(User)
527                 temp_sender.username = package['sender_username']
528                 temp_sender.public_key = sender_public_key
529                 temp_sender.sessions = {}
530                 self.users[package['sender_username']] = temp_sender
531                 print(f"\n Created temporary user profile for
                        {package['sender_username']}")
532
533             # Handle session key based on package format
534             if 'encrypted_session_key' in package:
535                 # New format (v1.1+) - session key included in package
536                 print(f"\n Decrypting session key from package...")
537                 encrypted_session_key =
                        base64.b64decode(package['encrypted_session_key'])
538
539                 # Decrypt session key with recipient's private key
540                 session_key = recipient.private_key.decrypt(
541                     encrypted_session_key,
542                     padding.OAEP(
543                         mgf=padding.MGF1(algorithm=hashes.SHA256()),
544                         algorithm=hashes.SHA256(),
545                         label=None
546                     )
547                 )
548
549                 # Store session key
550                 recipient.sessions[package['sender_username']] = session_key
551                 print(f" Session key decrypted and stored for
                        {package['sender_username']}")
552
553             else:
554                 # Old format (v1.0) - check for existing session
555                 if package['session_established']:
556                     print("\n[!] This is an old format message that established a
                            new session.")
557                     print("The session key was not included in the package.")
558                     print("Please use the new export format or establish a session
                            first.")
559                     return False
560
561                 # Check if we have an existing session
562                 if package['sender_username'] not in recipient.sessions:
563                     print(f"\n[!] No session key found for
                            {package['sender_username']}")
564                     print("You need to establish a session first or use a newer
                            message package format.")
565                     return False
566
567             # Decrypt the message
568             decrypted = recipient.decrypt_message(
569                 package['encrypted_message'],
```

```
                package['sender_username']
            )

            print(f"\ n   MESSAGE DECRYPTED SUCCESSFULLY!")
            print(f"Decrypted message: {decrypted}")

            # Add to local message history
            msg_record = {
                'from': package['sender_username'],
                'to': package['recipient_username'],
                'plaintext': decrypted,
                'encrypted': package['encrypted_message'],
                'session_established': package['session_established'],
                'timestamp': package['timestamp'],
                'imported': True
            }
            self.messages.append(msg_record)

            print(f"Message added to local history.")
            print(f"{'='*80}")
            return True

        except FileNotFoundError:
            print(f"[!] File not found: {filepath}")
            return False
        except json.JSONDecodeError:
            print(f"[!] Invalid message package format")
            return False
        except Exception as e:
            print(f"[!] Error importing message: {e}")
            return False

    def export_public_key(self, username, filepath):
        """Export a user's public key for sharing"""
        if username not in self.users:
            print(f"[!] User '{username}' not found!")
            return False

        user = self.users[username]
        key_data = {
            'username': username,
            'public_key_pem': user.get_public_key_pem(),
            'exported_at': datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        }

        with open(filepath, 'w') as f:
            json.dump(key_data, f, indent=2)

        print(f"\n Public key for '{username}' exported to: {filepath}")
        return True

    def import_public_key(self, filepath):
        """Import someone's public key"""
        try:
            with open(filepath, 'r') as f:
                key_data = json.load(f)

            username = key_data['username']

            if username in self.users:
                print(f"[!] User '{username}' already exists!")
                return False

```

```
            # Create user with just public key
            user = User.__new__(User)
            user.username = username
            user.sessions = {}

            # Load public key
            user.public_key = serialization.load_pem_public_key(
                key_data['public_key_pem'].encode('utf-8'),
                backend=default_backend()
            )

            # No private key for imported users
            user.private_key = None

            # Extract key components for display
            public_numbers = user.public_key.public_numbers()
            user.key_info = {
                'public_exponent': public_numbers.e,
                'modulus': public_numbers.n,
                'private_exponent': None,
                'prime1': None,
                'prime2': None
            }

            self.users[username] = user

            print(f"\ n   Public key imported for user: {username}")
            print(f"Exported at: {key_data['exported_at']}")
            print(f"You can now send encrypted messages to {username}")
            return True

        except FileNotFoundError:
            print(f"[!] File not found: {filepath}")
            return False
        except Exception as e:
            print(f"[!] Error importing public key: {e}")
            return False


def main():
    """Main function with interactive demo"""
    client = SecureMessagingClient()

    print("="*80)
    print("SECURE MESSAGING CLIENT")
    print("RSA-2048 Key Exchange + AES-256-GCM Encryption")
    print("="*80)

    # Interactive mode
    while True:
        print("\nOptions:")
        print("1. Create user")
        print("2. Send message")
        print("3. View messages")
        print("4. Verify message")
        print("5. Show user info")
        print("6. Demo mode")
        print("7. Save user keys")
        print("8. Load user keys")
        print("9. Export public key")
        print("10. Import public key")
        print("11. Export message package")
        print("12. Import message package")
```

```
696         print("13. Exit")
697
698         choice = input("\nEnter choice (1-13): ").strip()
699
700         if choice == '1':
701             username = input("Enter username: ").strip()
702             client.create_user(username)
703
704         elif choice == '2':
705             sender = input("Sender username: ").strip()
706             recipient = input("Recipient username: ").strip()
707             message = input("Message: ").strip()
708             client.send_message(sender, recipient, message)
709
710         elif choice == '3':
711             client.view_messages()
712
713         elif choice == '4':
714             try:
715                 idx = int(input("Message index (0-based): ").strip())
716                 client.verify_message(idx)
717             except ValueError:
718                 print("[!] Invalid index!")
719
720         elif choice == '5':
721             username = input("Username: ").strip()
722             client.show_user_info(username)
723
724         elif choice == '6':
725             print("\n Starting demo mode...")
726             # Create demo users
727             client.create_user("Alice")
728             client.create_user("Bob")
729
730             # Send demo messages
731             client.send_message("Alice", "Bob", "Hello Bob! This is our first
                    secure message.")
732             client.send_message("Bob", "Alice", "Hi Alice! The encryption is
                    working perfectly.")
733             client.send_message("Alice", "Bob", "Notice how the session key is
                    reused for efficiency.")
734
735             # View all messages
736             client.view_messages()
737
738         elif choice == '7':
739             username = input("Username to save: ").strip()
740             if username in client.users:
741                 filepath = input("Save to file (e.g., alice_keys.json): ").strip()
742                 client.users[username].save_keys_to_file(filepath)
743             else:
744                 print(f"[!] User '{username}' not found!")
745
746         elif choice == '8':
747             filepath = input("Load keys from file: ").strip()
748             try:
749                 user = User.load_keys_from_file(filepath)
750                 if user.username not in client.users:
751                     client.users[user.username] = user
752                     print(f"User '{user.username}' loaded successfully!")
753                 else:
754                     print(f"[!] User '{user.username}' already exists!")
755             except Exception as e:
```

```
                print(f"[!] Error loading keys: {e}")

        elif choice == '9':
            username = input("Username to export public key: ").strip()
            filepath = input("Export to file (e.g., alice_public.json): ").strip()
            client.export_public_key(username, filepath)

        elif choice == '10':
            filepath = input("Import public key from file: ").strip()
            client.import_public_key(filepath)

        elif choice == '11':
            try:
                idx = int(input("Message index to export (0-based): ").strip())
                filepath = input("Export to file (e.g., message_package.json):
                    ").strip()
                client.export_message_package(idx, filepath)
            except ValueError:
                print("[!] Invalid index!")

        elif choice == '12':
            filepath = input("Import message package from file: ").strip()
            recipient = input("Your username (recipient): ").strip()
            client.import_message_package(filepath, recipient)

        elif choice == '13':
            print("\n Secure Messaging Client shutdown complete!")
            break

        else:
            print("[!] Invalid choice!")


if __name__ == "__main__":
    main()
```

Listing 1: rsa.py