Secure Encrypted Chat Prototype - Technical Report

Github Link: encrypted-chat-prototype

1. Introduction

1.1 Project Overview

This repository implements a threaded TCP chat with a hybrid cryptography scheme: clients generate RSA key pairs, send their public key to the server during a handshake (along with a plaintext server password in the prototype), the server encrypts a global Fernet symmetric key with each client's RSA public key (RSA-OAEP with SHA-256), and clients decrypt and use Fernet (AES + HMAC) for fast authenticated encryption of chat payloads. The server acts as a relay + authority (it generates and can read the symmetric key), user_manager handles connected clients, and the GUI uses Tkinter with careful thread-to-main-thread scheduling. The cryptography choices are standard hybrid encryption patterns, but the prototype contains several practical security caveats (password in plaintext, single shared symmetric key, no TLS) that should be addressed before production use.

1.2 Objectives

- ➢ Develop a secure, real-time chat application with GUI interface.
- ➢ Implement strong encryption based on industry-standard algorithms.
- ➢ Create a client-server architecture that supports multiple users concurrently.
- ➢ Allow password-protected access control to servers
- ➢ Ensure forward secrecy by using session-based key management

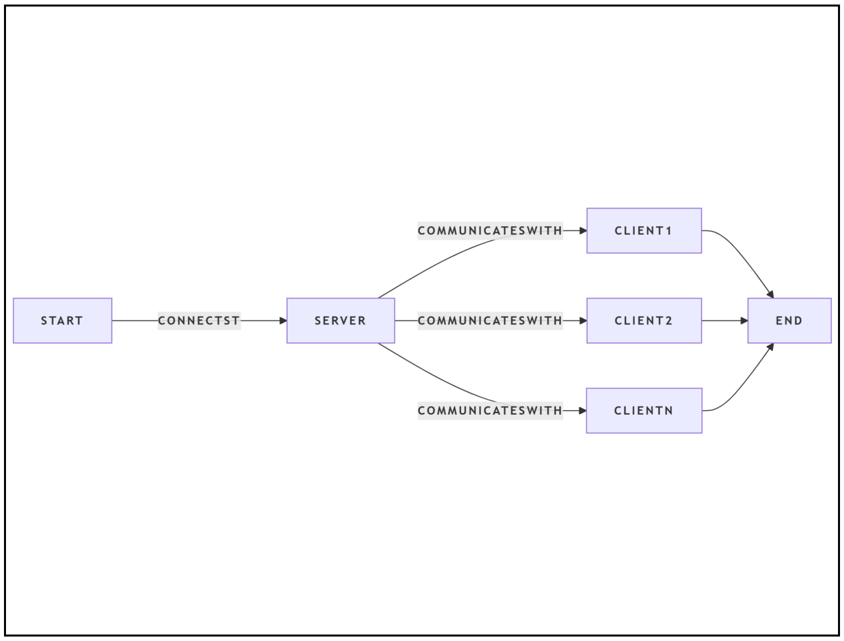
1.3 Scope

The prototype will demonstrate several of the core security principles: asymmetric key exchange, symmetric message encryption, and secure authentication. It is an educational tool to understand how cryptography can be implemented in networked applications.

2. System Architecture

2.1 High-Level Design

The system will be based on a client-server model, implemented with:



2.2 Component Architecture

2.2.1 Client Components

- ChatClient: Main client controller handling connection management
- ChatGUI: a Tkinter-based interface for messaging
- CryptoUtils: Cryptographic operations and key management

2.2.2 Server Components

- ChatServer: Primary server for maintaining client connections and routing messages.
- UserManager: Thread-safe user session management
- Protocol: Definitions of standardized message formats

2.3 Directory Structure

```
encrypted_chat/
├── client/
│   ├── client.py          # Client connection logic
│   ├── crypto_utils.py    # Encryption/decryption
│   └── gui.py             # User interface
├── server/
│   ├── server.py          # Server main loop
│   └── user_manager.py    # User session management
├── shared/
│   └── protocol.py        # Message protocol definitions
└── requirements.txt       # Dependencies
```

3. Technical Implementation

3.1 Encryption Implementation

3.1.1 Cryptographic Algorithms

- RSA-2048: Asymmetric encryption of key exchange
- AES-256: Symmetric encryption of message content
- Fernet: A high-level symmetric encryption implementation
- OAEP Padding: Optimal Asymmetric Encryption Padding for RSA
- SHA-256: Hash function for encryption operations

3.1.2 Key Management Process

Client key generation:

```
self.crypto.generate_rsa_keys()
public_key_pem = self.crypto.get_public_key_pem().decode()
```

Symmetric key generation at the server:

```
self.symmetric_key = Fernet.generate_key()
Key exchange encryption
encrypted_key = public_key.encrypt(
self.symmetric_key,
padding.OAEP(
mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(),
label=None
)
)
```

3.2 Client Implementation

3.2.1 Connection Management

The ChatClient class manages server connections using sockets:

- ➢ Establishes the TCP connection to the server
- ➢ Handles connection timeouts and errors
- ➢ Manages receive thread for incoming messages
- ➢ graceful disconnection: Implemented

3.2.2 Message Processing

```
def _receive_messages(self):
"""Continuous message reception loop"""
buffer = ""
while self.receiving:
data = self.socket.recv(1024).decode()
buffer += data
while '
' in buffer:
line, buffer = buffer.split('
', 1)
self._process_message(line)
```

3.2.3 GUI Implementation

The ChatGUI class provides:

- ➢ Real-time message display with scrollable text area
- ➢ Online users list dynamically updating
- ➢ Connection status indicators
- ➢ Message input with keyboard support (Enter key)

3.3 Server Implementation

3.3.1 Client Handling

Threading allows the server to handle multiple clients at once:

```
client_thread = threading.Thread(
target=self.handle_client,
args=(client_socket, address),
daemon=True
)
client_thread.start()
```

3.3.2 User Management

The UserManager class provides thread-safe user operations:

- ➢ Add/remove users with locking mechanism
- ➢ Broadcast messages to all connected clients
- ➢ Maintain user session state
- ➢ Handle disconnected clients cleanup

3.3.3 Message Broadcasting

```
def broadcast(self, message, exclude_user=None):
"""Send message to all users except specified one"""
with self.lock:
disconnected_users = []
for username, user_info in self.users.items():
if username == exclude_user:
continue
# Send message implementation
```

3.4 Communication Protocol

3.4.1 Message Types

The protocol defines five message types:

- handshake: Setup of initial connection
- key_exchange: Secure symmetric key delivery
- message: Chat message transmission
- user_list: Online users update
- system: Server notifications

3.4.2 Message Format

All messages use JSON format with following type-based structure:

```
{
"type": "message",
"sender": "username",
"message": "encrypted_content",
"encrypted": true
}
```

4. Security Analysis

4.1 Cryptographic Security

4.1.1 Key Strength

- RSA-2048: 112-bit security, enough for the current standards
- AES-256: Military-grade encryption, considered quantum-resistant
- Key Generation: Cryptographically secure random number generation

4.1.2 Key Exchange Security

The hybrid encryption approach provides:

- Forward Secrecy: Session keys are ephemeral
- Authentication: Verification of client credentials by the server
- Confidentiality: Keys are sent encrypted with RSA-OAEP

4.2 Application Security

4.2.1 Authentication Mechanism

- Password-protected server access
- Username uniqueness enforcement
- Session-based authentication without persistent tokens

4.2.2 Data Protection

- Messages encrypted before transmission

- ➢ Encryption status visible by the users
- ➢ No plaintext message storage on server

4.2.3 Network Security

- ➢ TCP-based reliable communication
- ➢ Buffer management prevents overflow
- ➢ Proper termination of connections

4.3 Limitations and Considerations

4.3.1 Security Limitations

- ➢ No protection against man-in-the-middle attacks
- ➢ Server knows symmetric key (not pure end-to-end)
- ➢ No verification of message integrity other than by encryption
- ➢ No protection against replay attacks

4.3.2 Operational Considerations

- ➢ Single server creates a single point of failure
- ➢ No persistence or message history
- ➢ Basic error handling without recovery mechanisms

5. Conclusion and Future Work

5.1 Project Achievements

The Secure Encrypted Chat Prototype successfully demonstrates:

1. Practical implementation of cryptographic principles
2. Real-time multi-user communication with GUI
3. Robust client-server architecture
4. Educational value in secure application development
5. Working hybrid encryption system
6. Thread-safe multi-client support
7. Clean separation of concerns in code architecture
8. Comprehensive error handling and recovery

5.2 Future Enhancements

5.2.1 Security Enhancements

- ➢ Implement SSL/TLS for Transport Security
- ➢ Add digital signatures for message authentication.
- ➢ Implement perfect forward secrecy using Diffie-Hellman
- ➢ Add message integrity checks with HMAC

5.2.2 Adding Features

- ➢ File transfer capability with encryption
- ➢ Private messaging between users
- ➢ Message history and persistence
- ➢ Mobile client applications
- ➢ Web-based client interface

5.2.3 Operational Improvements

- ➢ Database integration for user management
- ➢ Admin interface for server management
- ➢ Logging and audit capabilities

- Configuration file support

5.3 Educational Value

This project is an excellent learning resource for:

- Cryptographic algorithm implementation
- Network programming using sockets
- Python Multi-threading
- GUI application development
- Security protocol design
- The modular architecture is suitable for further experimentation and extension in academic or research settings.