Peer-to-Peer Chatting Application with Digital Signatures

Course Name and Code: MCA21206DCE

Teachers Name: Dr Gulnawaz Gani

Your Name: MAQSOOD HUSSAIN WANI

Enrolment Number: 23045112003

Date of Submission: 10 -Jun-2024

Table of Contents

1. Introduction

The Peer-to-Peer Chatting Application with Digital Signatures project aims to develop a secure messaging platform that leverages cryptographic techniques to ensure message integrity and authenticity. The application allows users to exchange messages in a secure manner, verifying the identity of participants through digital signatures. This ensures that the messages are not tampered with and the sender's identity is confirmed, addressing issues of data integrity and authenticity in peer-to-peer communication.

2. Objectives

- Develop a secure peer-to-peer chatting application.
- Implement user authentication with a database.
- Integrate digital signatures for message verification.
- Ensure the integrity and authenticity of messages.
- Provide a user-friendly interface for seamless communication.

3. System Design

3.1 Architecture

The system follows a client-server architecture where the server handles authentication and message verification, while the clients send and receive messages. The server also manages user credentials stored in a database and verifies digital signatures to ensure message authenticity.

4. Implementation

4.1 Technologies Used

- Python
- Socket Programming
- PyCryptoDome Library for Cryptographic Operations
- SQLite for User Credentials Database
-

4.2 Development Environment

- Operating System: Windows 11 pro
- IDE: VS Code
- Python Version: 3.8

4.3 Code Structure

- **main_server.py**: Contains the server-side logic for handling connections, user authentication, and message verification.
- **main_client.py**: Contains the client-side logic for connecting to the server, sending messages, and handling responses.
- **generateKeys.py**: Generates RSA key pairs for signing and verification.
- **database.py**: Handles database operations for user credentials.
- **digitalSignature.py**: Provides functions for signing and verifying messages.

4.4 Key Code Snippets

Server-Side Authentication and Message Verification

```
def verify_user():
    username = client.recv(1024).decode()
    password = client.recv(1024).decode()
    print(f" USERNAME: {username}, PASSWORD: {password}")

    result = connect.verify_credentials(username, password)
    return result

def con():
    res = verify_user()
    if res is True:
        client.sendall("ACCESS GRANTED..".encode())
        print("VALID USER ACCESSED")
    else:
        client.sendall("ACCESS DENIED..!".encode())
        print("UNAUTHORIZED USER TRY TO ACCESS")
        client.close()
        return

    sig = client.recv(1024)
    name = client.recv(1024)
    vr = digitalsig.verify(sig, name)

    print(f"Connected to {name.decode()} --{vr}--", end='\n')
    try:
        while True:
            print("wait...")
            sig = client.recv(1024)
            data = client.recv(1024)
            vr = digitalsig.verify(sig, data)
            if not data:
                break
            print(f"{name.decode()}: {data.decode()} --{vr}-- ")
            msg = input("You: ")
            client.sendall(msg.encode())
    except ConnectionRefusedError:
        print("Connection lost..!")
    finally:
        client.close()
        s.close()
```

Client-Side Message Signing and Sending

```
def sign_message(m):
    key = RSA.import_key(open('private.key').read())
    hash = SHA1.new(m.encode())

    signer = pkcs1_15.new(key)
    signature = signer.sign(hash)
    return signature

def con():
    send_userPwd()

    result = c.recv(1024).decode()
    print(f"{result}")

    if result == "ACCESS DENIED..!":
        return
    name = input("\nEnter Name: ")
    sig = sign_message(name)
    c.sendall(sig)
    c.sendall(name.encode())

    try:
        while True:
            msg = input("You: ")
            sig = sign_message(msg)
            c.sendall(sig)
            c.sendall(msg.encode())
            print("wait...")
            data = c.recv(1024)
            if not data:
                break
            print("Server: ", data.decode())
    except ConnectionResetError:
        print("Connection lost")
    finally:
        c.close()
```

Interface Screenshots

Server side

```
D:\GITHUB\Chat_App\FINAL 1.0\connection>python host1.py
Waiting for connection....
Connected to ('127.0.0.1', 59144)
 USERNAME: maqsood, PASSWORD: 123
VALID USER ACCESSED
Connected to maqsood    --VERIFIED--
wait...
maqsood:  hi --VERIFIED--
You: hlo how are you
wait...
```

Client Side

```
D:\GITHUB\Chat_App\FINAL 1.0\connection\HOST2>python host2.py
Connecting.....
Enter Username: maqsood
Enter Password: 123
ACCESS GRANTED..

Enter Name: maqsood
You: hi
wait...
```

4. Database Design

4.1 Schema

- **Users Table**: Stores user credentials (username and hashed password).

4.2 Tables and Relationships

- **Users (username TEXT PRIMARY KEY, password TEXT)**

5. Digital Signature Integration

5.1 Signing Messages

Messages are signed using RSA private keys. The signing process involves creating a SHA-1 hash of the message and then generating a signature using the private key.

5.2 Verifying Messages

Incoming messages are verified using the corresponding RSA public key. The verification process ensures that the message has not been tampered with and the sender is authentic.

5.3 Password Storage

Passwords are hashed before being stored in the database to enhance security. This ensures that even if the database is compromised, the passwords remain protected.

6. Testing

6.1 Test Cases

- **User Authentication**: Verify that valid users can log in and invalid users are denied access.
- **Message Signing and Verification**: Ensure that messages are correctly signed and verified.
- **Connection Handling**: Test the stability of the connection between the client and server.

6.2 Test Results

- All test cases passed successfully.
- Authentication mechanisms were validated.
- Message integrity and authenticity were confirmed through signature verification.

7. User Guide

7.1 Installation Instructions

1. Clone the repository.
2. Install the required Python libraries: **pip install pycryptodome.**
3. Generate RSA key pairs using **generateKeys.py.**
4. Set up the database using **database.py**.

7.2 How to Use the Application

1. We first give public key to client manually then server verify messages with own private key.
2. Start the server by running **main_server.py**.
3. Connect the client by running **main_client.py**.
4. Authenticate using your username and password.
5. Exchange messages securely with digital signatures.

8. Conclusion

This project successfully implements a secure peer-to-peer chatting application using digital signatures. It ensures message integrity and authenticity, providing a robust solution for secure communication.

9. Future Work

- Implement end-to-end encryption for message content.
- Sharing of keys securely
- Develop a graphical user interface for enhanced usability.
- Expand the application to support multiple clients and group chats.

10. References

- PyCryptoDome Documentation
- Python Socket Programming Tutorials
- SQLite Documentation
- Python documentation
- YouTube