**Project [20 Points]: Fall2024**

*Instructions: This project is Group-based, each group may consist of (4-5 students). Each team member is expected to submit programs that solve each of the following problems. Due date of submissions is on* ***Thursday. 5/12/2024****.*

**Team Members:**

1. Ahmed Amer 2. Hayat Mume

3. Maxim Timoshechkin 4. Selamawit Alemu

5. Wubrist Zewide

**Submission:** Each Team Leader submits the python client and server files and copy and paste the code of both client and server files into this document. *Files should be named as follow: teamleaderName\_client.py,teamleaderName\_server.py,teamleaderName\_projectDocument.docx*

**Rubric: (Each students might have different grade based on the discussions)**

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | Max Grade | Std. Grade | Feedback |
| Setup Phase | 5 |  |  |
| Closing Phase | 5 |  |  |
| Start-Packet | 2 |  |  |
| Encryption-Packet | 2 |  |  |
| Confirm-Connection-Packet | 2 |  |  |
| Using RSA including session key encryption | 5 |  |  |
| Using AES | 5 |  |  |
| Using Caesar | 5 |  |  |
| Prompt packets | 10 |  |  |
| openRead includes reading from files | 8 |  |  |
| openWrite incudes writing on files | 8 |  |  |
| Data Packet | 2 |  |  |
| Exception packets | 8 |  |  |
| Server Multithreads | 5 |  |  |
| Comments | 5 |  |  |
| Running code | 8 |  |  |
| Integrating C client part | 5 |  |  |
| Discussions | 10 |  |  |
| Total | /100 | /100 | /20 |

**Project Problem:**

Remote controlling protocols such as Telnet and SSH are used to remotely run commands on other devices without the need to physical direct interaction with these devices. In this project you will design and develop a simple application layer protocol that help folders and files management on servers.

You have to write Python3 code to implement the **Remote File Management Protocol (RFMP)** which is designed to serve a remote controlling application.

**RFMP** has three phases: ***setup phase***, ***operation phase***, and ***closing phase***.

**1. Setup-Phase:**

Once the client opens the application, the application sends a series of packets to the server device in the following order:

* **Start-Packet (from client to server):** contains the packet type, protocol name, protocol version, secured-communication example:
  + (**SS**,RFMP,v1.0,0), where S*S* is the packet type (start), RFMP is the protocol name, v1.0 is the version, 0 means no secured communication is required, in this case no encryption for the packet is required.
  + (**SS**, RFMP,v1.0,1), where S*S* is the packet type (start), RFMP is the protocol name, v1.0 is the version, 1 means secured communication is required, in this case the server expects to receive an encryption-packet (EC) immediately after the start-packet (SS).
* **Confirm-Connection-Packet** **(from server to client)**: this packet is sent with one field if security is not required which is the packet-type (**CC**) from the server to inform the client to start sending information packets or with two fields if security is required (CC, Server\_public\_key).

* **Encryption-Packet (from client to server):** contains information about encryption algorithm, and credentials used to secure communication. This packet is sent in case the Start-Packet contains **1** in the secure-communication field. This packet type contains the following fields (packet-type, Algorithm, credentials): (**EC**, ALgorithm, session\_key, username:Client\_public\_key).

The following table shows the options of the algorithms and credentials sent with each:

|  |  |
| --- | --- |
| **Fields** | **Credentials** |
| **AES Cipher** | Both will use AES to encrypt files contents |
| **Caesar Cipher** | Both will use Caesarto encrypt files contents |
| **Session\_key** | Either AES or Caesar key encrypted using RSA private key |
| **Public\_Key** | Public key generated by RSA algorithm, used to decrypt session key |

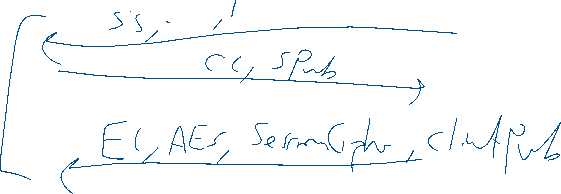
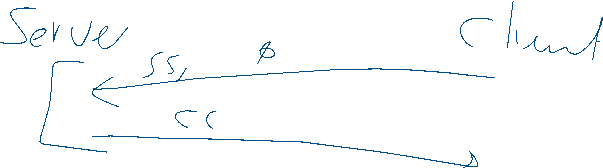
**In the Start-up phase the client prepares three keys:**

**1. Session Key that will be used later to encrypt data either by using Caesar or AES algorithms.**

**2. Public and Private Keys using RSA algorithm. That will be used later to encrypt and decrypt session key**

The server in the setup phase will generate both ***private key*** and ***public key*** using the *RSA* algorithm and send his public key to client in the confirmation packet (CC, Server\_publik\_Key).

The Client program will use the Server\_*public\_ key* to encrypt the *session\_key and send it to server*. The server will use his *private\_key* to decrypt the *session\_key,* then later in the operation phase the server will use the *session\_key* to decrypt data before saving the data into a file on server.



1. **Operation-Phase:**

This phase is started once the client starts typing commands and send them to the server. This packet has type filed (CM). The packet has only two fields the type and the command text.

Ex. (CM, ls), (CM, mkdir f1),.. See the below table for all commands.

You can send any system command to the server. Such as processor or memory status enquiry, or folder management commands. In this project you have to include the following commands for file and folder management:

|  |  |
| --- | --- |
| mkdir | Create new folder |
| cd | Change directory path command |
| rmdir or rd | To delete folders |
| del | To delete files |
| ren | To rename a folder |
| openRead | This command is not a prompt command, its keyword for the server program to use python file open command to create new file in write mode |
| openWrite | This command is not a prompt command, its keyword for the server program to use python file open command to create new file in read mode |
| Choose any 5 other system prompt commands you want to execute on the server |  |

* Each of the **prompt commands** requires argument which you should send in the packet (write full command text such as (CM,prompt,mkdir folder1), (CM,prompt,ren homework1 homework2). For these types of prompt commands you can use many python libraries to implement them:

os.system()

subprocess.run()

* The **openRead** and **openWrite** commands requires to send in the same packet file names you want to open in read mode or write mode for example: (CM,openRead, data.txt) or (CM, openWrite, data.txt)
* Data Packets are used to send data to be saved in the file created using openWrite command. This packet has the following format: (DP, text) where the text is any data you want the server to save.

If the communications are secured according to the setup phase then you have to encrypt the ***text*** field before sending. And the server will decrypt the text before saving it to the file.

* In the openRead, the server should open a file and send its contents to the client. Make sure if encryption is chosen in the setup phase then the server should use the session key to encrypt the text before sending it.
* After each command execution on the server, the server should returns either successful (SC) packet or Error packet(EE).

1. **Closing-Phase:**

Packet (**End**) is used in closing phase. In this phase a **Close-Packet** is sent to the server to confirm that the client has finished from using the application. And the server will expect no more messages from the client.

4. **Exception-Packets:** Your client should check each response from the server if the server sends an Exception Event (EE) packet-type message this means that there has an error occurred while processing client’s requests. Each EE packet contains the following fields: (EE, Error Code, Description). Error Code values are left to your implementation, you can suggest set of error codes (maximum 4 error codes).

**Note: You have to implement a client side with proper options that the user can choose from and remotely control the server and displays server responses.**

**5. Integrating with C client program:**

You have to integrate your project with a C client version in addition to the Python client version and Python server version. This C client part is simple version of the python client part, where you have to implement a nonsecured communication client in the C-part, (no encryption) and the C client program sends only one type command which is the openRead to save a text on server file.

**Python- Server Code:**

**<copy server code here>**

import socket

import os

import threading

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

import base64

import shutil

# Server configuration

HOST = '127.0.0.1'  # Server IP address

PORT = 12345       # Server port number

# RSA keys for server

server\_key = RSA.generate(2048)  # Generate RSA key pair

server\_private\_key = server\_key  # Private key for decrypting data

server\_public\_key = server\_key.publickey()  # Public key for encrypting data

# Define error codes for standardized responses

ERROR\_CODES = {

    1: "File not found",

    2: "Invalid Command",

    3: "Permission Denied",

    4: "Server Error"

}

# Encryption Utilities

def encrypt\_message\_aes(message, key):

    """Encrypts a message using AES encryption."""

    try:

        iv = os.urandom(16)  # Generate a random Initialization Vector (IV)

        cipher = AES.new(key, AES.MODE\_CBC, iv)  # Create AES cipher in CBC mode

        ct = cipher.encrypt(pad(message.encode(), AES.block\_size))  # Encrypt with padding

        return base64.b64encode(iv + ct).decode()  # Encode IV and ciphertext in Base64

    except Exception as e:

        print(f"Encryption error: {e}")

        return message

def decrypt\_message\_aes(encrypted\_message, key):

    """Decrypts a message using AES encryption."""

    try:

        raw\_data = base64.b64decode(encrypted\_message)  # Decode Base64 message

        iv, ct = raw\_data[:16], raw\_data[16:]  # Extract IV and ciphertext

        cipher = AES.new(key, AES.MODE\_CBC, iv)  # Create AES cipher in CBC mode

        return unpad(cipher.decrypt(ct), AES.block\_size).decode()  # Decrypt and remove padding

    except Exception as e:

        print(f"Decryption error: {e}")

        return encrypted\_message

def encrypt\_message\_caesar(message, shift):

    """Encrypts a message using Caesar Cipher."""

    return ''.join(chr((ord(char) + shift - 32) % 95 + 32) if 32 <= ord(char) <= 126 else char for char in message)

def decrypt\_message\_caesar(encrypted\_message, shift):

    """Decrypts a message using Caesar Cipher."""

    return ''.join(chr((ord(char) - shift - 32) % 95 + 32) if 32 <= ord(char) <= 126 else char for char in encrypted\_message)

def execute\_command(command\_type, args, encryption=None, key=None):

    """Executes commands for file and folder management."""

    try:

        # Decrypt command and arguments if encryption is enabled

        if encryption == "Caesar":

            command\_type = decrypt\_message\_caesar(command\_type, key)

            args = [decrypt\_message\_caesar(arg, key) for arg in args]

        elif encryption == "AES":

            command\_type = decrypt\_message\_aes(command\_type, key)

            args = [decrypt\_message\_aes(arg, key) for arg in args]

        # Handle various file and directory commands

        if command\_type == "mkdir":

            os.mkdir(args[0])  # Create directory

            response = "SC,Folder created successfully"

        elif command\_type == "cd":

            os.chdir(args[0])  # Change current directory

            response = "SC,Directory changed successfully"

        elif command\_type in ("rmdir", "rd"):

            os.rmdir(args[0])  # Remove directory

            response = "SC,Folder removed successfully"

        elif command\_type == "del":

            os.remove(args[0])  # Delete file

            response = "SC,File deleted successfully"

        elif command\_type == "ren":

            os.rename(args[0], args[1])  # Rename file or directory

            response = "SC,File renamed successfully"

        elif command\_type == "openWrite":

            with open(args[0], 'w') as file:  # Open file for writing

                response = "SC,File opened for writing"

        elif command\_type == "openRead":

            # Read content from the file

            with open(args[0], 'r') as file:  # Open file for reading

                content = file.read()

            # If the file is not empty, apply encryption (if needed)

            if content:  # If file has content

                if encryption == "Caesar":

                    content = encrypt\_message\_caesar(content, key)

                elif encryption == "AES":

                    content = encrypt\_message\_aes(content, key)

                response = f"SC,File contents: {content}"

            else:

                # If the file is empty, return an appropriate response without encryption

                response = "SC,File is empty"

        elif command\_type == "ls":

            files = os.listdir('.')  # List files in the current directory

            response = f"SC,{', '.join(files)}"

        elif command\_type == "pwd":

            cwd = os.getcwd()  # Get current working directory

            response = f"SC,Current directory: {cwd}"

        elif command\_type == "touch":

            with open(args[0], 'w') as file:  # Create an empty file

                pass

            response = "SC,File created successfully"

        elif command\_type == "filesize":

            size = os.path.getsize(args[0])  # Get size of the file

            response = f"SC,File size: {size} bytes"

        elif command\_type == "copy":

            shutil.copy(args[0], args[1])  # Copy file

            response = "SC,File copied successfully"

        else:

            response = f"EE,2,Command not found"  # Invalid command

        # Encrypt response if encryption is enabled

        if command\_type != "ls":

            if encryption == "Caesar":

                response = encrypt\_message\_caesar(response, key)

            if encryption == "AES":

                response = encrypt\_message\_aes(response, key)

        return response

    except FileNotFoundError:

        return f"EE,1,{ERROR\_CODES[1]}"  # Handle file not found

    except PermissionError:

        return f"EE,3,{ERROR\_CODES[3]}"  # Handle permission denied

    except Exception as e:

        return f"EE,4,{ERROR\_CODES[4]} - {str(e)}"  # Generic server error

def handle\_client(client\_socket):

    """Handles client communication."""

    encryption = None  # Encryption method (AES or Caesar)

    key = None  # Encryption key

    try:

        while True:

            packet = client\_socket.recv(1024).decode()  # Receive packet from client

            if not packet:

                break

            fields = packet.split(',')

            # Setup Phase: Handle Start Packet (SS)

            if fields[0] == "SS":

                secure\_comm = int(fields[3])  # Check if secure communication is requested

                if secure\_comm == 1:

                    # Send public key to client

                    client\_socket.send(f"CC,{server\_public\_key.export\_key().decode()}".encode())

                    # Receive encryption packet from client

                    encryption\_packet = client\_socket.recv(1024).decode().split(',')

                    rsa\_cipher = PKCS1\_OAEP.new(server\_private\_key)

                    encryption = encryption\_packet[1]  # Get encryption method

                    if encryption == "Caesar":

                        key = int(rsa\_cipher.decrypt(base64.b64decode(encryption\_packet[2])).decode())

                    elif encryption == "AES":

                        key = rsa\_cipher.decrypt(base64.b64decode(encryption\_packet[2]))

                    client\_socket.send("SC,Encryption setup successful".encode())

                else:

                    client\_socket.send("CC".encode())  # Confirm plain communication

            # Command Execution Phase: Handle Command Packet (CM)

            elif fields[0] == "CM":

                if len(fields) < 2 or not fields[1].strip():

                    response = "EE,2,Please enter a valid command"

                else:

                    command\_type = fields[1]  # Extract command type

                    args = fields[2:]  # Extract arguments

                    # General check for missing arguments (except ls, pwd)

                    if command\_type not in ("ls", "pwd") and not args:

                        response = f"EE,2,Please enter your argument for the '{command\_type}' command"

                    # Specific handling for openWrite

                    elif command\_type == "openWrite":

                         if len(args) < 1:

                              response = "EE,2,File name not provided"

                         else:

                         # Decrypt the file name if encryption is enabled

                                if encryption is not None and key is not None:

                                     file\_name = decrypt\_message\_aes(args[0], key) if encryption == "AES" else decrypt\_message\_caesar(args[0], key)

                                else:

                                    file\_name = args[0]  # Use raw file name if no encryption

                                print(f"File name after decryption (if applicable): {file\_name}")

                                # Confirm that the file is ready for writing

                                response = "SC,File opened for writing"

                                client\_socket.send(response.encode())

                                print("Sent confirmation: File opened for writing")

                                # Receive data packet from the client in chunks and write to the file

                                try:

                                   with open(file\_name, 'wb') as file:

                                       while True:

                                           data\_packet = client\_socket.recv(1024)

                                           if not data\_packet:

                                                break

                                           print(f"Received data (before decryption): {data\_packet}")

                                           if encryption is not None and key is not None:

                                              if encryption == "AES":

                                                data\_packet = decrypt\_message\_aes(data\_packet.decode(), key).encode()

                                              elif encryption == "Caesar":

                                                data\_packet = decrypt\_message\_caesar(data\_packet.decode(), key).encode()

                                           file.write(data\_packet)

                                   print("Data written to the file successfully")

                                   client\_socket.send("SC,Data written to file successfully".encode())

                                except Exception as write\_error:

                                   client\_socket.send(f"EE,4,Error writing to file: {str(write\_error)}".encode())

                                   print(f"Error writing to file '{file\_name}': {write\_error}")

                                continue  # Skip further processing for this command

                    else:

                        # Handle other commands

                        response = execute\_command(command\_type, args, encryption, key)

                client\_socket.send(response.encode())  # Send response to client

            # Closing Phase: Handle End Packet

            elif fields[0] == "End":

                print("End packet received. Closing connection...")

                client\_socket.send("SC,Server shutting down.".encode())

                break

    except Exception as e:

        client\_socket.send(f"EE,4,{ERROR\_CODES[4]} - {str(e)}".encode())  # Send error packet

    finally:

        client\_socket.close()  # Close client connection

def start\_server():

    print(" \n\*\*\*\*\*\*\* Welcome to Remote File Management Protocol (RFMP)! \*\*\*\*\*\*\*\*\n")

    """Starts the server with multithreading."""

    server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    server.bind((HOST, PORT))

    server.listen(5)

    print(f"Server starting....\nServer listening on {HOST} : {PORT}")

    while True:

        client\_socket, addr = server.accept()

        print(f"Accepted connection from {addr}") # accepting client connection

        client\_thread = threading.Thread(target=handle\_client, args=(client\_socket,)) # creat new thread

        client\_thread.start() # starts new thread

if \_\_name\_\_ == "\_\_main\_\_":

    start\_server()

**Python- Client Code:**

**<copy client code here>**

import socket

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

import base64

import os

# Client configuration

SERVER\_HOST = '127.0.0.1'

SERVER\_PORT = 12345

AES\_SESSION\_KEY = b'sixteen\_byte\_key'  # 16-byte AES session key

CAESAR\_SHIFT = 5  # Caesar Cipher shift value

def decrypt\_message\_aes(encrypted\_message, key):

    """Decrypts a message using AES."""

    try:

        # Ensure the key is valid for AES (16, 24, or 32 bytes)

        assert len(key) in [16, 24, 32], "Invalid AES key length"

        # Decode the base64-encoded encrypted message

        raw\_data = base64.b64decode(encrypted\_message)

        # Extract the IV (first 16 bytes) and ciphertext

        iv, ct = raw\_data[:16], raw\_data[16:]

        # Initialize cipher with key and IV

        cipher = AES.new(key, AES.MODE\_CBC, iv)

        # Decrypt and unpad the ciphertext

        return unpad(cipher.decrypt(ct), AES.block\_size).decode()

    except Exception as e:

        print(f"Decryption error: {e}")

        return encrypted\_message  # Return raw message if decryption fails

def encrypt\_message\_aes(message, key):

    """Encrypts a message using AES."""

    try:

        # Ensure the key is valid for AES (16, 24, or 32 bytes)

        assert len(key) in [16, 24, 32], "Invalid AES key length"

        # Generate a random 16-byte IV

        iv = os.urandom(16)

        # Initialize cipher with key and IV

        cipher = AES.new(key, AES.MODE\_CBC, iv)

        # Encrypt the message with padding

        ct = cipher.encrypt(pad(message.encode(), AES.block\_size))

        # Combine IV and ciphertext and return base64 encoded

        return base64.b64encode(iv + ct).decode()

    except Exception as e:

        print(f"Encryption error: {e}")

        return message  # Return raw message if encryption fails

def encrypt\_message\_caesar(message, shift):

    """Encrypts a message using Caesar Cipher."""

    encrypted = ''.join(

        chr((ord(char) + shift - 32) % 95 + 32) if 32 <= ord(char) <= 126 else char

        for char in message

    )

    return encrypted

def decrypt\_message\_caesar(encrypted\_message, shift):

    """Decrypts a message using Caesar Cipher."""

    decrypted = ''.join(

        chr((ord(char) - shift - 32) % 95 + 32) if 32 <= ord(char) <= 126 else char

        for char in encrypted\_message

    )

    return decrypted

def handle\_open\_read(file\_name, client, secure\_comm, session\_key):

    """Sends openRead command and handles the server response."""

    if secure\_comm:

        encrypted\_command = encrypt\_message\_aes("openRead", session\_key)

        encrypted\_file = encrypt\_message\_aes(file\_name, session\_key)

        client.send(f"CM,{encrypted\_command},{encrypted\_file}".encode())

    else:

        client.send(f"CM,openRead,{file\_name}".encode())

    response = client.recv(1024).decode()

    if secure\_comm:

        response = decrypt\_message\_aes(response, session\_key)

    print(f"Server Response: {response}")

def handle\_open\_write(file\_name, client, secure\_comm, session\_key):

    """Sends openWrite command and sends data packets."""

    # Send the openWrite command and file name to the server

    if secure\_comm:

        encrypted\_command = encrypt\_message\_aes("openWrite", session\_key)

        encrypted\_file = encrypt\_message\_aes(file\_name, session\_key)

        client.send(f"CM,{encrypted\_command},{encrypted\_file}".encode())

    else:

        client.send(f"CM,openWrite,{file\_name}".encode())

    # Receive confirmation from the server that the file has been opened for writing

    response = client.recv(1024).decode()

    print(f"Server Response: {response}")

    if response.startswith("SC,File opened for writing"):

        # Sample data to write to the file

        data = "Sample data sent for testing"

        # Send the data packet to the server

        if secure\_comm:

            encrypted\_data = encrypt\_message\_aes(data, session\_key)

            client.send(f"DP,{encrypted\_data}".encode())

        else:

            client.send(f"DP,{data}".encode())

        # Receive confirmation for the data packet from the server

        response = client.recv(1024).decode()

        print(f"Server Response: {response}")

def start\_client():

    print(" \*\*\*\*\*\*\*\*\* Welcome to Remote File Management Protocol (RFMP)! \*\*\*\*\*\*\*\*\n\n\n")

    """Starts the client."""

    client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

    client.connect((SERVER\_HOST, SERVER\_PORT))

    while True:

        secure\_comm\_input = input("Enable encryption? (yes/no): ").strip().lower()

        # Validate encryption enable input

        if secure\_comm\_input in ("yes", "no"):

            break

        else:

            print("Invalid input. Please enter 'yes' or 'no'.")

    secure\_comm = secure\_comm\_input == "yes"

    encryption\_method = None

    if secure\_comm:

        # Choose encryption method

        while True:

            encryption\_method = input("Choose encryption method (AES/Caesar): ").strip()

            if encryption\_method in ("AES", "Caesar"):

                break

            else:

                print("Invalid encryption method. Please choose either 'AES' or 'Caesar'.")

        # Send Start Packet

        client.send("SS,RFMP,v1.0,1".encode())

        response = client.recv(1024).decode().split(',')

        server\_public\_key = RSA.import\_key(response[1])

        # Encrypt session key

        rsa\_cipher = PKCS1\_OAEP.new(server\_public\_key)

        if encryption\_method == "AES":

            encrypted\_session\_key = base64.b64encode(rsa\_cipher.encrypt(AES\_SESSION\_KEY)).decode()

        elif encryption\_method == "Caesar":

            encrypted\_session\_key = base64.b64encode(rsa\_cipher.encrypt(str(CAESAR\_SHIFT).encode())).decode()

        # Send Encryption Packet

        client.send(f"EC,{encryption\_method},{encrypted\_session\_key}".encode())

        print(client.recv(1024).decode())  # Encryption confirmation

    else:

        client.send("SS,RFMP,v1.0,0".encode())

        print(client.recv(1024).decode())  # Connection confirmation

    try:

        while True:

            command = input("Enter command (or 'exit' to quit): ").strip()

            # Check for empty command

            if not command:

                print("Empty command. Please enter your command.")

                continue

            if command.strip().lower() == "exit":

                client.send("End".encode())

                print("Closing connection...")

                break

            command\_fields = command.split()

            valid\_commands = [

                "ls", "pwd", "mkdir", "cd", "rmdir", "rd", "del", "ren",

                "openWrite", "openRead", "touch", "filesize", "copy"

            ]

            # Check if the command is valid

            if command\_fields[0] not in valid\_commands:

               print("EE,2,Invalid command. Please enter a valid command.")

               continue

            # Check for commands that require arguments

            if len(command\_fields) == 1 and command\_fields[0] not in ("ls", "pwd"):

                print(f"Missing argument for the '{command\_fields[0]}' command. Please provide the necessary argument(s).")

                continue

            # Encrypt command if secure communication is enabled

            if secure\_comm:

                if encryption\_method == "AES":

                    encrypted\_command = encrypt\_message\_aes(command\_fields[0], AES\_SESSION\_KEY)

                    encrypted\_args = [encrypt\_message\_aes(arg, AES\_SESSION\_KEY) for arg in command\_fields[1:]]

                elif encryption\_method == "Caesar":

                    encrypted\_command = encrypt\_message\_caesar(command\_fields[0], CAESAR\_SHIFT)

                    encrypted\_args = [encrypt\_message\_caesar(arg, CAESAR\_SHIFT) for arg in command\_fields[1:]]

                client.send(f"CM,{encrypted\_command},{','.join(encrypted\_args)}".encode())

            else:

                client.send(f"CM,{command\_fields[0]},{','.join(command\_fields[1:])}".encode())

            # Receive response from server

            response = client.recv(1024).decode()

            # Skip decryption for `ls` command (plaintext response)

            if command\_fields[0] == "ls":

                print(f"Server Response: {response}")

            else:

                # Decrypt response if secure communication is enabled

                if secure\_comm:

                    if encryption\_method == "AES":

                        response = decrypt\_message\_aes(response, AES\_SESSION\_KEY)

                    elif encryption\_method == "Caesar":

                        response = decrypt\_message\_caesar(response, CAESAR\_SHIFT)

                print(f"Server Response: {response}")

    finally:

        client.close()

if \_\_name\_\_ == "\_\_main\_\_":

    start\_client()

**C- Client Code:**

**<copy client code here>**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <winsock2.h>

#pragma comment(lib, "ws2\_32.lib") // Link Winsock library

#define SERVER\_PORT 12345 // Server port to connect to

#define SERVER\_ADDRESS "127.0.0.1" // Server IP address (localhost)

#define BUFFER\_SIZE 1024 // Buffer size for receiving/sending data

// Function to handle the "openRead" command, which requests file content from the server

void handleOpenReadCommand(SOCKET clientSocket, const char \*fileName) {

char command[BUFFER\_SIZE]; // Buffer to hold the command to be sent

char buffer[BUFFER\_SIZE]; // Buffer to receive server response

int recvResult;

// Create the command string to open the file for reading

snprintf(command, sizeof(command), "CM,openRead,%s", fileName);

// Send the command to the server

if (send(clientSocket, command, strlen(command), 0) < 0) {

printf("Failed to send openRead command. Error Code: %d\n", WSAGetLastError());

return;

}

// Receive the server's response (file content or error message)

recvResult = recv(clientSocket, buffer, BUFFER\_SIZE - 1, 0);

if (recvResult < 0) {

printf("Failed to receive response. Error Code: %d\n", WSAGetLastError());

return;

}

buffer[recvResult] = '\0'; // Null-terminate the received data

printf("File content: %s\n", buffer); // Print the file content received from the server

}

int main() {

WSADATA wsaData; // Holds information about the Windows Sockets implementation

SOCKET clientSocket; // Socket used for communication with the server

struct sockaddr\_in serverAddr; // Structure holding server address details

char buffer[BUFFER\_SIZE]; // Buffer for receiving data from the server

char userCommand[BUFFER\_SIZE]; // Buffer to hold user input commands

int recvResult;

// Initialize Winsock (required for socket programming on Windows)

if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {

printf("WSAStartup failed. Error Code: %d\n", WSAGetLastError());

return 1; // Exit if Winsock initialization fails

}

// Create a socket for communication (TCP socket)

if ((clientSocket = socket(AF\_INET, SOCK\_STREAM, 0)) == INVALID\_SOCKET) {

printf("Socket creation failed. Error Code: %d\n", WSAGetLastError());

WSACleanup(); // Clean up Winsock before exiting

return 1;

}

// Set up the server address structure

serverAddr.sin\_family = AF\_INET; // Address family (IPv4)

serverAddr.sin\_port = htons(SERVER\_PORT); // Port number (network byte order)

serverAddr.sin\_addr.s\_addr = inet\_addr(SERVER\_ADDRESS); // IP address (in network byte order)

// Connect to the server using the specified address and port

if (connect(clientSocket, (struct sockaddr\*)&serverAddr, sizeof(serverAddr)) < 0) {

printf("Connection failed. Error Code: %d\n", WSAGetLastError());

closesocket(clientSocket); // Close the socket on failure

WSACleanup(); // Clean up Winsock before exiting

return 1;

}

printf("Connected to server at %s:%d\n", SERVER\_ADDRESS, SERVER\_PORT);

// Main loop to interact with the user

while (1) {

// Ask the user for a command

printf("Enter command (e.g., openRead <filename> or exit): ");

fgets(userCommand, sizeof(userCommand), stdin); // Get user input

// Remove trailing newline character from the user input

userCommand[strcspn(userCommand, "\n")] = '\0';

// Parse the command and handle specific cases

if (strncmp(userCommand, "openRead ", 9) == 0) {

// If the command starts with "openRead", extract the file name

char \*fileName = userCommand + 9; // Extract file name after the "openRead " prefix

handleOpenReadCommand(clientSocket, fileName); // Call function to handle the openRead command

} else if (strcmp(userCommand, "exit") == 0) {

// If the user enters "exit", break the loop and close the connection

printf("Closing the connection...\n");

break;

} else {

// For any other command, send it to the server and wait for a response

if (send(clientSocket, userCommand, strlen(userCommand), 0) < 0) {

printf("Failed to send command. Error Code: %d\n", WSAGetLastError());

break;

}

// Receive the server's response to the command

recvResult = recv(clientSocket, buffer, BUFFER\_SIZE - 1, 0);

if (recvResult < 0) {

printf("Failed to receive response. Error Code: %d\n", WSAGetLastError());

break;

} else {

buffer[recvResult] = '\0'; // Null-terminate the received data

printf("Server response: %s\n", buffer); // Print the server's response

}

}

}

// Clean up resources and close the socket

closesocket(clientSocket);

WSACleanup(); // Clean up Winsock resources

return 0;

}