**MINOR-2**

**Source Code**

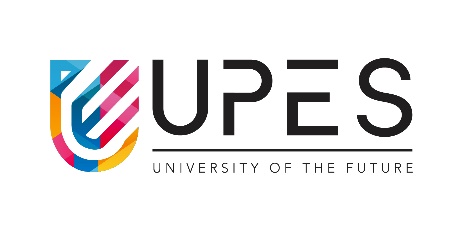
**on**

**Secure Chat Application**

Submitted By:

|  |  |  |
| --- | --- | --- |
| **Name** | **Roll No** | **Branch** |
| Aadeesh Jain | R2142220375 | BTech CSF |
| Deepanshu Chowdhury | R2142220474 | BTech CSF |
| Aman Anand | R2142220415 | BTech CSF |
| Abhinav Saini | R2142220392 | BTech CSF |

**Under the guidance of**

Dr. Avishek Majumder

School of Computer Science

**UNIVERSITY OF PETROLEUM AND ENERGY STUDIES**

**Dehradun 248007**

**Approved By**

**School of Computer Science**’

University of Petroleum & Energy Studies, Dehradun

**Source Code (2025)**

**Client.py**

import socket, hashlib, threading, os, struct, time

from aes import encrypt\_aes128, decrypt\_aes128

from dh import generate\_dh\_keys, compute\_shared\_key

from hmac\_util import generate\_hmac, verify\_hmac

HOST, PORT = '127.0.0.1', 8080

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

client.settimeout(60)

try:

    client.connect((HOST, PORT))

except Exception as e:

    print(f"[CLIENT] Connection failed: {e}")

    exit(1)

# Prompt for username and password

username = input("Enter your username: ").strip()

if not username:

    username = "anonymous"

password = input("Enter your password: ").strip()

if not password:

    print("[CLIENT] Password cannot be empty")

    client.close()

    exit(1)

print(f"[CLIENT] Proceeding as {username}")

# Send username and hashed password for authentication

try:

    # Send username

    username\_data = username.encode('utf-8')

    client.sendall(struct.pack("!I", len(username\_data)) + username\_data)

    print(f"[CLIENT] Sent username (length: {len(username\_data)} bytes): {username}")

    # Hash the password and send it

    password\_data = password.encode('utf-8')

    hashed\_password = hashlib.sha256(password\_data).hexdigest().encode('utf-8')

    client.sendall(struct.pack("!I", len(hashed\_password)) + hashed\_password)

    print(f"[CLIENT] Sent hashed password: {hashed\_password.decode()}")

    # Receive authentication result

    auth\_result = client.recv(7)

    if auth\_result != b"SUCCESS":

        print("[CLIENT] Authentication failed")

        client.close()

        exit(1)

    print("[CLIENT] Authentication successful")

except socket.timeout:

    print("[CLIENT] Authentication timed out")

    client.close()

    exit(1)

except socket.error as e:

    print(f"[CLIENT] Socket error during authentication: {e}")

    client.close()

    exit(1)

except Exception as e:

    print(f"[CLIENT] Authentication error: {e}")

    client.close()

    exit(1)

# Diffie-Hellman key exchange with logging

P, G = 23, 5

priv, pub = generate\_dh\_keys(P, G)

print(f"[CLIENT] Generated private key: {priv}, public key: {pub}")

try:

    pub\_data = str(pub).encode()

    client.sendall(struct.pack("!I", len(pub\_data)) + pub\_data)

    print(f"[CLIENT] Sent public key data (length: {len(pub\_data)} bytes): {pub\_data.decode()}")

    length\_data = client.recv(4)

    if not length\_data:

        print("[CLIENT] Failed to receive server public key length: no data")

        client.close()

        exit(1)

    length = struct.unpack("!I", length\_data)[0]

    server\_pub\_data = client.recv(length)

    if not server\_pub\_data:

        print(f"[CLIENT] Failed to receive server public key: expected {length} bytes, got 0")

        client.close()

        exit(1)

    server\_pub = int(server\_pub\_data.decode())

    print(f"[CLIENT] Received server public key: {server\_pub} (length: {length} bytes)")

except socket.timeout:

    print("[CLIENT] Key exchange timed out")

    client.close()

    exit(1)

except socket.error as e:

    print(f"[CLIENT] Socket error during key exchange: {e}")

    client.close()

    exit(1)

except ValueError as e:

    print(f"[CLIENT] Invalid server public key received: {e}")

    client.close()

    exit(1)

shared = compute\_shared\_key(server\_pub, priv, P)

print(f"[CLIENT] Computed shared secret: {shared}")

key = hashlib.sha256(str(shared).encode()).digest()[:16]

print(f"[CLIENT] Derived AES key: {key.hex()}")

def recv\_exact(sock, size):

    data = b''

    max\_attempts = 3

    attempt = 0

    while len(data) < size and attempt < max\_attempts:

        try:

            part = sock.recv(size - len(data))

            if not part:

                return None

            data += part

            print(f"[CLIENT] Received partial data: {len(data)}/{size} bytes")

        except socket.timeout:

            attempt += 1

            print(f"[CLIENT] Receive timed out after {len(data)} bytes (attempt {attempt}/{max\_attempts})")

            if attempt == max\_attempts or len(data) == 0:

                return None

            time.sleep(2)

        except socket.error as e:

            print(f"[CLIENT] Socket error in recv: {e}")

            return None

    return data if len(data) > 0 else None

def receive():

    while True:

        try:

            length\_data = recv\_exact(client, 4)

            if not length\_data:

                print("[CLIENT] Connection closed by server")

                break

            enc\_len = struct.unpack("!I", length\_data)[0]

            data = recv\_exact(client, enc\_len)

            iv = recv\_exact(client, 16)

            mac = recv\_exact(client, 32)

            if not data or not iv or not mac:

                print("[CLIENT] Incomplete data received")

                break

            print(f"[CLIENT] Received encrypted data: {data.hex()}")

            print(f"[CLIENT] Received IV: {iv.hex()}")

            print(f"[CLIENT] Received HMAC: {mac.hex()}")

            if not verify\_hmac(key, data, mac):

                print("[!] HMAC check failed")

                continue

            print(f"[CLIENT] Attempting decryption with key: {key.hex()}, IV: {iv.hex()}, data: {data.hex()}")

            msg = decrypt\_aes128(key, iv, data)

            if not msg:

                print("[CLIENT] Decryption failed: empty or invalid message")

                continue

            try:

                decoded\_msg = msg.decode('utf-8', errors='replace')

                print(f"\n[RECV]: {decoded\_msg}")

            except UnicodeDecodeError as e:

                print(f"[CLIENT] Decoding error: {e}, raw message: {msg.hex()}")

        except Exception as e:

            print(f"[CLIENT] Receive error: {e}")

            break

threading.Thread(target=receive, daemon=True).start()

while True:

    try:

        msg = input(f"{username}: ").strip().encode('utf-8')

        if not msg:

            continue

        iv = os.urandom(16)

        enc = encrypt\_aes128(key, iv, msg)

        mac = generate\_hmac(key, enc)

        print(f"[CLIENT] Sending encrypted data: {enc.hex()}")

        print(f"[CLIENT] Sending IV: {iv.hex()}")

        print(f"[CLIENT] Sending HMAC: {mac.hex()}")

        client.sendall(struct.pack("!I", len(enc)))

        client.sendall(enc)

        client.sendall(iv)

        client.sendall(mac)

    except KeyboardInterrupt:

        print("\n[CLIENT] Exiting")

        client.close()

        break

    except socket.error as e:

        print(f"[CLIENT] Socket error: {e}")

        client.close()

        break

    except Exception as e:

        print(f"[CLIENT] Error: {e}")

        client.close()

        break

**Server.py**

import socket, threading, hashlib, os, struct, time

from aes import encrypt\_aes128, decrypt\_aes128

from dh import generate\_dh\_keys, compute\_shared\_key

from hmac\_util import generate\_hmac, verify\_hmac

HOST, PORT = '127.0.0.1', 8080

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

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server.settimeout(60)

server.bind((HOST, PORT))

server.listen()

print(f"[SERVER] Listening on {HOST}:{PORT}")

# Store user credentials (username: hashed password)

users = {

    "alice": hashlib.sha256("alice123".encode()).hexdigest(),

    "bob": hashlib.sha256("bob123".encode()).hexdigest(),

}

clients = {}

keys = {}

usernames = {}

P, G = 23, 5

client\_counter = 0

def recv\_exact(sock, size):

    data = b''

    max\_attempts = 3

    attempt = 0

    while len(data) < size and attempt < max\_attempts:

        try:

            part = sock.recv(size - len(data))

            if not part:

                return None

            data += part

            print(f"[SERVER] Received partial data: {len(data)}/{size} bytes")

        except socket.timeout:

            attempt += 1

            print(f"[SERVER] Receive timed out after {len(data)} bytes (attempt {attempt}/{max\_attempts})")

            if attempt == max\_attempts or len(data) == 0:

                return None

            time.sleep(2)

        except socket.error as e:

            print(f"[SERVER] Socket error in recv: {e}")

            return None

    return data if len(data) > 0 else None

def handle\_client(conn, addr):

    global client\_counter

    # Authenticate the client

    try:

        # Receive username length and username

        username\_len\_data = recv\_exact(conn, 4)

        if not username\_len\_data:

            print(f"[SERVER] Failed to receive username length from {addr}: no data")

            conn.close()

            return

        username\_len = struct.unpack("!I", username\_len\_data)[0]

        username\_data = recv\_exact(conn, username\_len)

        if not username\_data:

            print(f"[SERVER] Failed to receive username from {addr}: expected {username\_len} bytes, got 0")

            conn.close()

            return

        username = username\_data.decode('utf-8')

        print(f"[SERVER] Received username from {addr}: {username}")

        # Receive hashed password length and hashed password

        hashed\_pwd\_len\_data = recv\_exact(conn, 4)

        if not hashed\_pwd\_len\_data:

            print(f"[SERVER] Failed to receive hashed password length from {addr}: no data")

            conn.close()

            return

        hashed\_pwd\_len = struct.unpack("!I", hashed\_pwd\_len\_data)[0]

        hashed\_pwd = recv\_exact(conn, hashed\_pwd\_len)

        if not hashed\_pwd:

            print(f"[SERVER] Failed to receive hashed password from {addr}")

            conn.close()

            return

        received\_hash = hashed\_pwd.decode('utf-8')

        print(f"[SERVER] Received hashed password from {username}: {received\_hash}")

        # Verify credentials

        if username not in users:

            print(f"[SERVER] Authentication failed for {username}: unknown user")

            conn.sendall(b"FAIL")

            conn.close()

            return

        print(f"[SERVER] Expected hash for {username}: {users[username]}")

        print(f"[SERVER] Received hash: {received\_hash}")

        if received\_hash != users[username]:

            print(f"[SERVER] Authentication failed for {username}: incorrect password")

            conn.sendall(b"FAIL")

            conn.close()

            return

        # Authentication successful

        print(f"[SERVER] Authentication successful for {username}")

        conn.sendall(b"SUCCESS")

    except Exception as e:

        print(f"[SERVER] Authentication error for {addr}: {e}")

        conn.sendall(b"FAIL")

        conn.close()

        return

    # Proceed with Diffie-Hellman key exchange

    client\_counter += 1

    print(f"[+] {addr} connected as {username}")

    conn.settimeout(60)

    priv, pub = generate\_dh\_keys(P, G)

    print(f"[SERVER] Generated private key: {priv}, public key: {pub}")

    try:

        length\_data = recv\_exact(conn, 4)

        if not length\_data:

            print(f"[SERVER] Failed to receive public key length from {username}@{addr}: no data")

            conn.close()

            return

        length = struct.unpack("!I", length\_data)[0]

        print(f"[SERVER] Expected public key length: {length} bytes")

        client\_pub\_data = recv\_exact(conn, length)

        if not client\_pub\_data:

            print(f"[SERVER] Failed to receive client public key from {username}@{addr}: expected {length} bytes, got 0")

            conn.close()

            return

        client\_pub = int(client\_pub\_data.decode())

        print(f"[SERVER] Received client public key: {client\_pub} (length: {length} bytes)")

        pub\_data = str(pub).encode()

        conn.sendall(struct.pack("!I", len(pub\_data)) + pub\_data)

        print(f"[SERVER] Sent public key data (length: {len(pub\_data)} bytes): {pub\_data.decode()}")

    except socket.timeout:

        print(f"[SERVER] Key exchange timed out for {username}@{addr}")

        conn.close()

        return

    except socket.error as e:

        print(f"[SERVER] Socket error during key exchange for {username}@{addr}: {e}")

        conn.close()

        return

    except ValueError as e:

        print(f"[SERVER] Invalid client public key received from {username}@{addr}: {e}")

        conn.close()

        return

    shared = compute\_shared\_key(client\_pub, priv, P)

    print(f"[SERVER] Computed shared secret: {shared}")

    key = hashlib.sha256(str(shared).encode()).digest()[:16]

    print(f"[SERVER] Derived AES key: {key.hex()}")

    keys[conn] = key

    clients[conn] = addr

    usernames[conn] = username

    while True:

        try:

            length\_data = recv\_exact(conn, 4)

            if not length\_data:

                print(f"[SERVER] Connection closed by {username}@{addr}")

                break

            enc\_len = struct.unpack("!I", length\_data)[0]

            data = recv\_exact(conn, enc\_len)

            iv = recv\_exact(conn, 16)

            mac = recv\_exact(conn, 32)

            if not data or not iv or not mac:

                print(f"[SERVER] Incomplete data received from {username}@{addr}")

                continue

            print(f"[SERVER] Received encrypted data from {username}: {data.hex()}")

            print(f"[SERVER] Received IV: {iv.hex()}")

            print(f"[SERVER] Received HMAC: {mac.hex()}")

            if not verify\_hmac(key, data, mac):

                print(f"[SERVER] HMAC check failed for {username}")

                continue

            try:

                print(f"[SERVER] Attempting decryption with key: {key.hex()}, IV: {iv.hex()}, data: {data.hex()}")

                msg = decrypt\_aes128(key, iv, data)

                if not msg:

                    print(f"[SERVER] Decryption failed: empty message for {username}")

                    continue

                decoded\_msg = msg.decode('utf-8', errors='replace')

                print(f"[{username}@{addr}] {decoded\_msg}")

            except UnicodeDecodeError as e:

                print(f"[SERVER] Decoding error for {username}: {e}, raw message: {msg.hex() if msg else 'None'}")

                continue

            except Exception as e:

                print(f"[SERVER] Decryption error for {username}: {e}, raw data: {data.hex()}")

                continue

            # Forward to other clients

            for c in list(clients.keys()):

                if c != conn:

                    try:

                        iv2 = os.urandom(16)

                        enc = encrypt\_aes128(keys[c], iv2, msg)

                        mac2 = generate\_hmac(keys[c], enc)

                        print(f"[SERVER] Forwarding to {usernames[c]}: encrypted data: {enc.hex()}")

                        print(f"[SERVER] Forwarding IV: {iv2.hex()}")

                        print(f"[SERVER] Forwarding HMAC: {mac2.hex()}")

                        c.sendall(struct.pack("!I", len(enc)))

                        c.sendall(enc)

                        c.sendall(iv2)

                        c.sendall(mac2)

                    except socket.error as e:

                        print(f"[SERVER] Error forwarding to {usernames[c]}: {e}")

                        c.close()

                        del clients[c], keys[c], usernames[c]

                        print(f"[-] {usernames[c]}@{clients[c]} disconnected due to error")

        except socket.error as e:

            print(f"[SERVER] Socket error for {username}@{addr}: {e}")

            break

        except Exception as e:

            print(f"[SERVER] Error for {username}@{addr}: {e}")

            break

    conn.close()

    if conn in clients:

        del clients[conn], keys[conn], usernames[conn]

    print(f"[-] {username}@{addr} disconnected")

while True:

    try:

        conn, addr = server.accept()

        print(f"[SERVER] Accepted connection from {addr}")

        threading.Thread(target=handle\_client, args=(conn, addr)).start()

    except socket.timeout:

        print("[SERVER] Accept timed out, continuing to listen")

    except Exception as e:

        print(f"[SERVER] Accept error: {e}")

**AES.py**

import os

# AES-128 parameters

Nb = 4  # Number of columns (32-bit words) in the state

Nk = 4  # Number of 32-bit words in the key (AES-128)

Nr = 10 # Number of rounds for AES-128

# S-box for SubBytes transformation

SBOX = [

    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,

    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,

    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,

    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,

    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,

    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,

    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,

    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,

    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,

    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16

]

# Inverse S-box for InvSubBytes transformation

INV\_SBOX = [

    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,

    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,

    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,

    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,

    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,

    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,

    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,

    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,

    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,

    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,

    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,

    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,

    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,

    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,

    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,

    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d

]

# Round constants for key expansion

RCON = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36]

def bytes\_to\_matrix(b):

    return [list(b[i:i+4]) for i in range(0, len(b), 4)]

def matrix\_to\_bytes(matrix):

    return bytes(sum(matrix, []))

def add\_round\_key(state, round\_key):

    return [[s ^ k for s, k in zip(sr, kr)] for sr, kr in zip(state, round\_key)]

def sub\_word(word):

    return [SBOX[b] for b in word]

def rot\_word(word):

    return word[1:] + word[:1]

def key\_expansion(key):

    key\_words = [list(key[i:i+4]) for i in range(0, len(key), 4)]

    i = Nk

    while len(key\_words) < Nb \* (Nr + 1):

        word = key\_words[-1][:]

        if i % Nk == 0:

            word = sub\_word(rot\_word(word))

            word[0] ^= RCON[i // Nk - 1]

        word = [w ^ k for w, k in zip(word, key\_words[-Nk])]

        key\_words.append(word)

        i += 1

    round\_keys = []

    for r in range(Nr + 1):

        round\_key = key\_words[r\*Nb:(r+1)\*Nb]

        round\_keys.append(round\_key)

    return round\_keys

def gf\_multiply(a, b):

    p = 0

    for i in range(8):

        if b & 1:

            p ^= a

        hi\_bit = a & 0x80

        a = (a << 1) & 0xFF

        if hi\_bit:

            a ^= 0x1b

        b >>= 1

    return p

def mix\_columns(state):

    def mix\_single\_column(col):

        return [

            gf\_multiply(col[0], 2) ^ gf\_multiply(col[1], 3) ^ col[2] ^ col[3],

            col[0] ^ gf\_multiply(col[1], 2) ^ gf\_multiply(col[2], 3) ^ col[3],

            col[0] ^ col[1] ^ gf\_multiply(col[2], 2) ^ gf\_multiply(col[3], 3),

            gf\_multiply(col[0], 3) ^ col[1] ^ col[2] ^ gf\_multiply(col[3], 2)

        ]

    return [mix\_single\_column(list(col)) for col in zip(\*state)]

def inv\_mix\_columns(state):

    def inv\_mix\_column(col):

        return [

            gf\_multiply(col[0], 0x0e) ^ gf\_multiply(col[1], 0x0b) ^ gf\_multiply(col[2], 0x0d) ^ gf\_multiply(col[3], 0x09),

            gf\_multiply(col[0], 0x09) ^ gf\_multiply(col[1], 0x0e) ^ gf\_multiply(col[2], 0x0b) ^ gf\_multiply(col[3], 0x0d),

            gf\_multiply(col[0], 0x0d) ^ gf\_multiply(col[1], 0x09) ^ gf\_multiply(col[2], 0x0e) ^ gf\_multiply(col[3], 0x0b),

            gf\_multiply(col[0], 0x0b) ^ gf\_multiply(col[1], 0x0d) ^ gf\_multiply(col[2], 0x09) ^ gf\_multiply(col[3], 0x0e)

        ]

    return [inv\_mix\_column(list(col)) for col in zip(\*state)]

def sub\_bytes(state):

    return [[SBOX[b] for b in row] for row in state]

def inv\_sub\_bytes(state):

    return [[INV\_SBOX[b] for b in row] for row in state]

def shift\_rows(state):

    return [

        state[0],

        state[1][1:] + state[1][:1],

        state[2][2:] + state[2][:2],

        state[3][3:] + state[3][:3]

    ]

def inv\_shift\_rows(state):

    return [

        state[0],

        state[1][-1:] + state[1][:-1],

        state[2][-2:] + state[2][:-2],

        state[3][-3:] + state[3][:-3]

    ]

def encrypt\_block(plaintext, key\_schedule):

    state = bytes\_to\_matrix(plaintext)

    state = add\_round\_key(state, key\_schedule[0])

    for rnd in range(1, Nr):

        state = sub\_bytes(state)

        state = shift\_rows(state)

        state = mix\_columns(state)

        state = add\_round\_key(state, key\_schedule[rnd])

    state = sub\_bytes(state)

    state = shift\_rows(state)

    state = add\_round\_key(state, key\_schedule[Nr])

    return matrix\_to\_bytes(state)

def decrypt\_block(ciphertext, key\_schedule):

    if len(ciphertext) != 16:

        raise ValueError("Ciphertext block must be 16 bytes")

    state = bytes\_to\_matrix(ciphertext)

    state = add\_round\_key(state, key\_schedule[Nr])

    state = inv\_shift\_rows(state)

    state = inv\_sub\_bytes(state)

    for rnd in range(Nr - 1, 0, -1):

        state = add\_round\_key(state, key\_schedule[rnd])

        state = inv\_mix\_columns(state)

        state = inv\_shift\_rows(state)

        state = inv\_sub\_bytes(state)

    state = add\_round\_key(state, key\_schedule[0])

    return matrix\_to\_bytes(state)

def pad(plaintext):

    pad\_len = 16 - (len(plaintext) % 16)

    padded = plaintext + bytes([pad\_len] \* pad\_len)

    return padded

def unpad(plaintext):

    if not plaintext:

        raise ValueError("Cannot unpad empty plaintext")

    pad\_len = plaintext[-1]

    if not (1 <= pad\_len <= 16) or not all(b == pad\_len for b in plaintext[-pad\_len:]):

        raise ValueError("Invalid padding")

    return plaintext[:-pad\_len]

def encrypt\_aes128(key, iv, plaintext):

    if len(key) != 16 or len(iv) != 16:

        raise ValueError("Key and IV must be 16 bytes")

    plaintext = pad(plaintext)

    key\_schedule = key\_expansion(key)

    blocks = [plaintext[i:i+16] for i in range(0, len(plaintext), 16)]

    encrypted = b""

    prev = iv

    for block in blocks:

        xor\_block = bytes(a ^ b for a, b in zip(block, prev))

        encrypted\_block = encrypt\_block(xor\_block, key\_schedule)

        encrypted += encrypted\_block

        prev = encrypted\_block

    return encrypted

def decrypt\_aes128(key, iv, ciphertext):

    if len(key) != 16 or len(iv) != 16:

        raise ValueError("Key and IV must be 16 bytes")

    try:

        if len(ciphertext) % 16 != 0:

            raise ValueError(f"Ciphertext length ({len(ciphertext)}) must be a multiple of 16 bytes")

        key\_schedule = key\_expansion(key)

        blocks = [ciphertext[i:i+16] for i in range(0, len(ciphertext), 16)]

        decrypted = b""

        prev = iv

        for i, block in enumerate(blocks):

            decrypted\_block = decrypt\_block(block, key\_schedule)

            plain\_block = bytes(a ^ b for a, b in zip(decrypted\_block, prev))

            decrypted += plain\_block

            prev = block

        return unpad(decrypted)

    except Exception as e:

        print(f"[AES] Decryption error: {e}, ciphertext: {ciphertext.hex()}")

        return b""

**DH.py**

def generate\_dh\_keys(p, g):

    private\_key = 6

    public\_key = pow(g, private\_key, p)

    return private\_key, public\_key

def compute\_shared\_key(pub, priv, p):

    return pow(pub, priv, p)

**HMAC\_UTILS.py**

import hashlib, hmac

def generate\_hmac(key, data):

    return hmac.new(key, data, hashlib.sha256).digest()

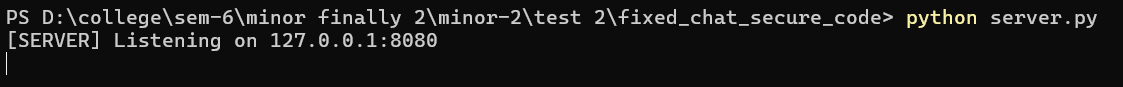
def verify\_hmac(key, data, mac):

    computed = hmac.new(key, data, hashlib.sha256).digest()

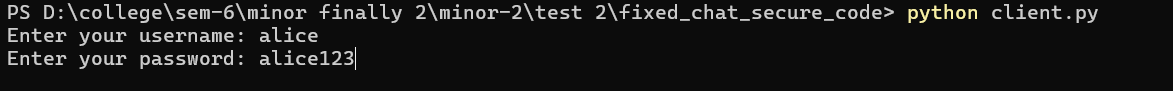
    return computed == mac

**Output**

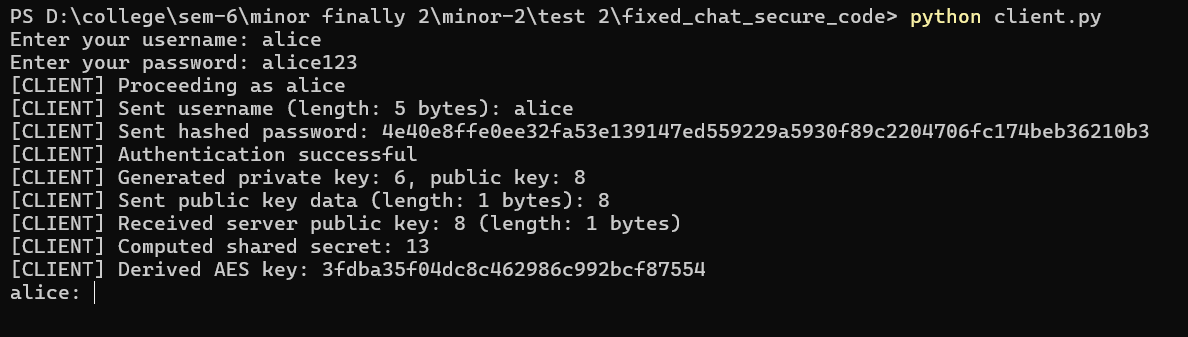
**Starting Server:**

****

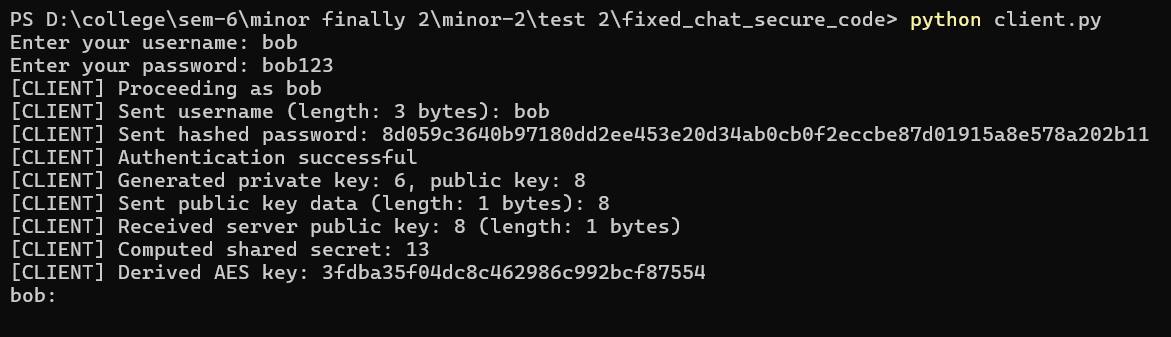
**Starting Client:**



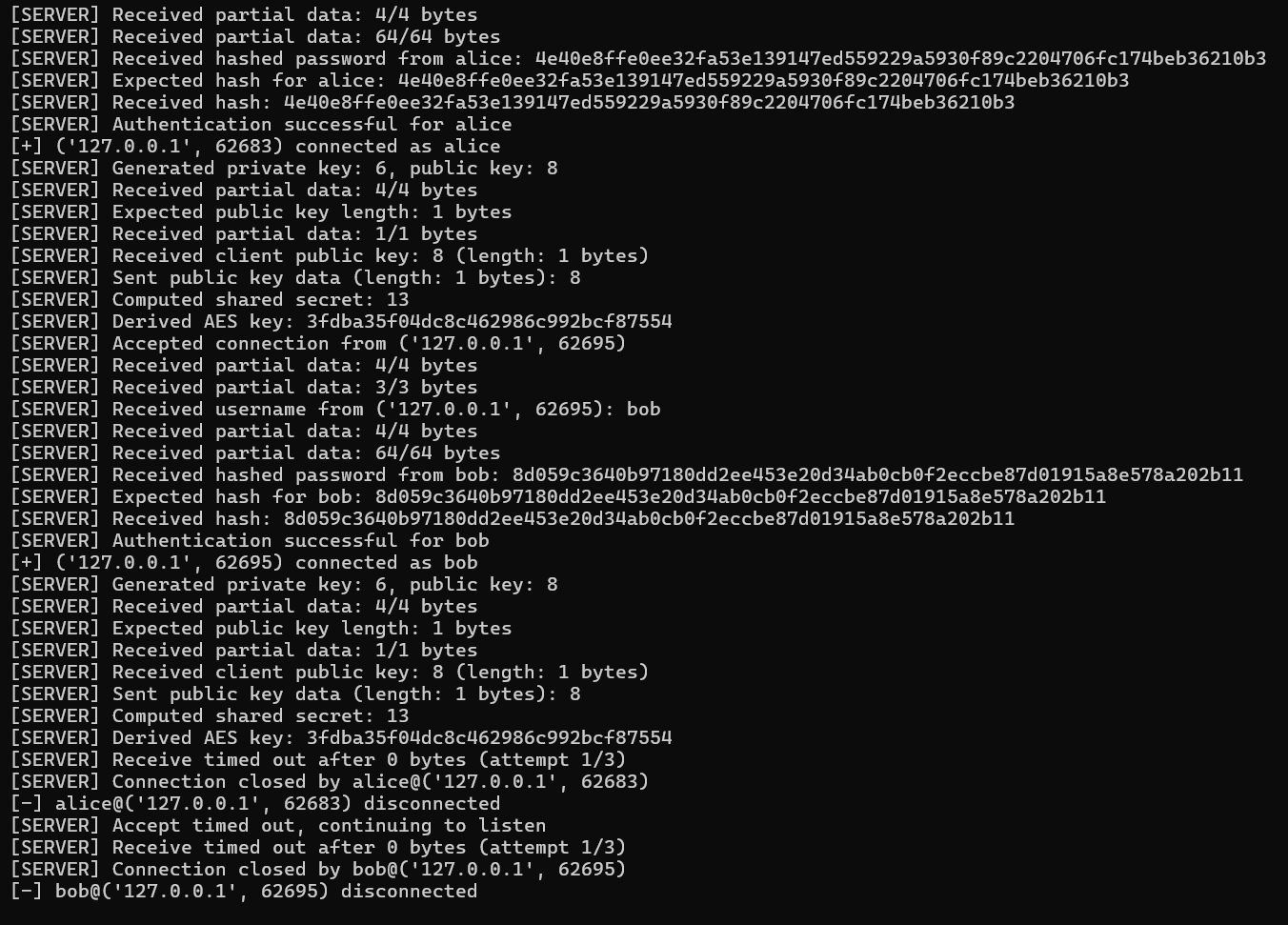
**Login for client 1:**

****

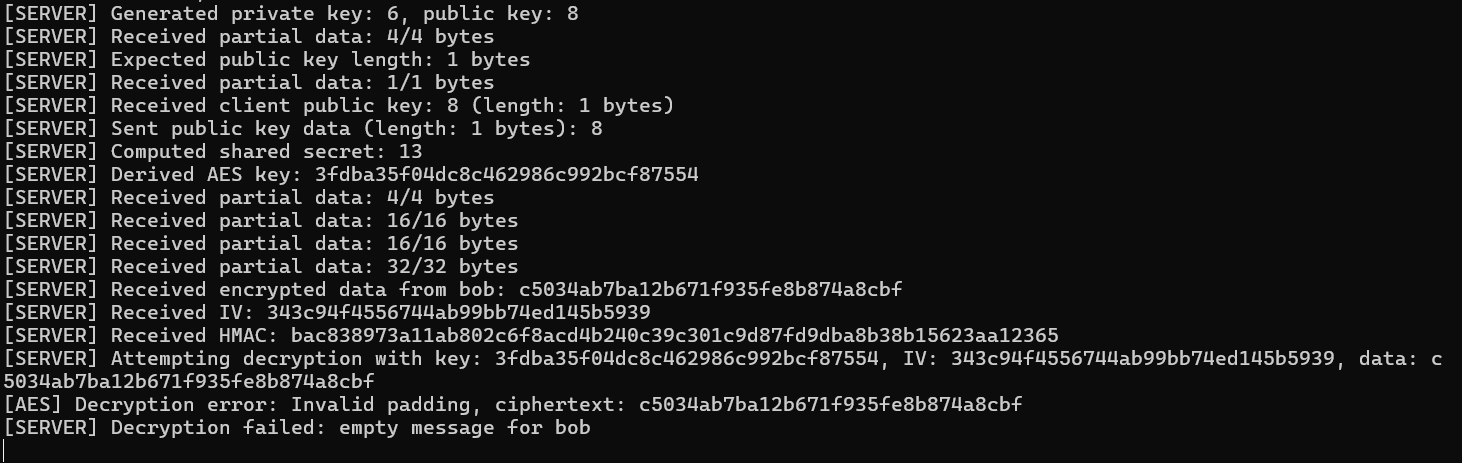
**Login for client 2:**

****

**Server Logs:**

****

**Server Logs on sending messages:**

****