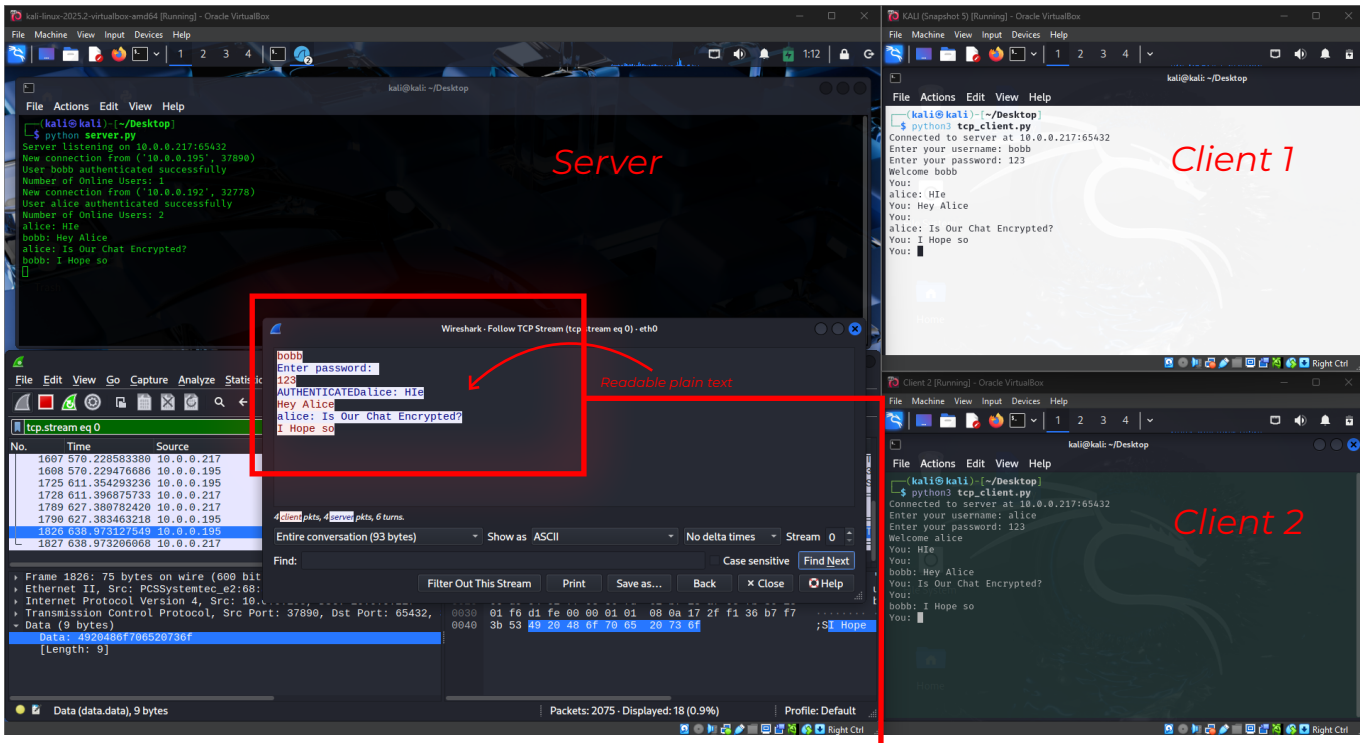
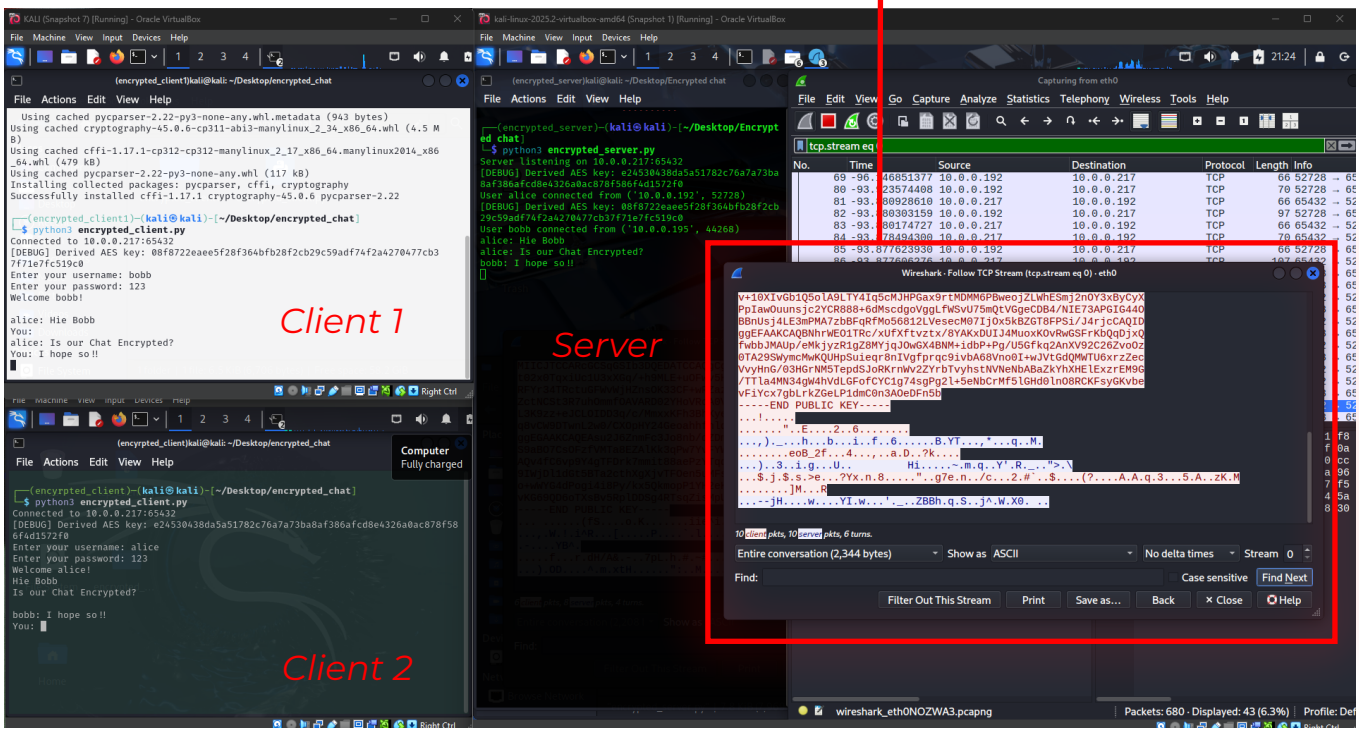
BEFORE ENCRYPTION



AFTER ENCRYPTION

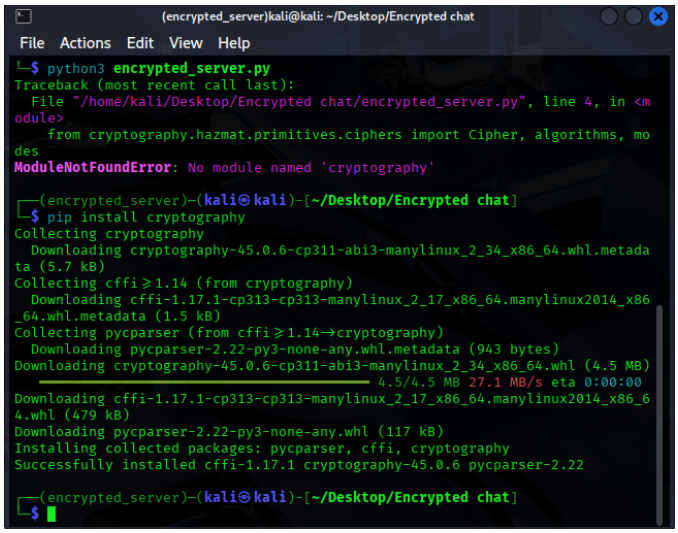


Steps To Achieving Encryption

Phase 1:
Cryptographic Setup

1: Import Required Libraries

```python
import socket
import threading
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.serialization import *
import os
```



2: Generate Diffie-Hellman Parameters

```python
class ChatServer:
    def __init__(self):
        self.online_users = {}  # {conn: username}
        self.users = {"bobb": "123", "alice": "123"}
        self.lock = threading.Lock()

        # Generate DH parameters (shared between server and clients)
        self.dh_parameters = dh.generate_parameters(generator=2, key_size=2048)
```

3: Add Key Derivation Function

```python
def derive_aes_key(self, shared_secret):
    # Derive a 32-byte (256-bit) AES key
    return HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'chat-app-key',
    ).derive(shared_secret)
```

Phase 2:
Secure Key Exchange

```python
if password == self.users[username]:
    # Perform Diffie-Hellman key exchange
    # Step 1: Server sends DH parameters to client
    conn.sendall(
        self.dh_parameters.parameter_bytes(
            Encoding.PEM,
            ParameterFormat.PKCS3
        )
    )

    # Step 2: Server generates its private key
    server_private_key = self.dh_parameters.generate_private_key()

    # Step 3: Server sends its public key to client
    server_public_key = server_private_key.public_key()
    conn.sendall(
        server_public_key.public_bytes(
            Encoding.PEM,
            PublicFormat.SubjectPublicKeyInfo
        )
    )

    # Step 4: Receive client's public key
    client_public_key_bytes = conn.recv(2048)
    client_public_key = load_pem_public_key(client_public_key_bytes)

    # Step 5: Generate shared secret
    shared_secret = server_private_key.exchange(client_public_key)

    # Step 6: Derive AES key
    aes_key = self.derive_aes_key(shared_secret)
```

Phase 3
Add Encryption and Decryption Function

```python
def encrypt_message(self, key, message):
    # Generate a random 12-byte nonce for GCM
    nonce = os.urandom(12)

    # Encrypt the message
    cipher = Cipher(
        algorithms.AES(key),
        modes.GCM(nonce),
    )
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(message.encode()) + encryptor.finalize()

    # Return nonce + ciphertext + tag
    return nonce + ciphertext + encryptor.tag
```

ENCRYPTION

```python
def decrypt_message(self, key, encrypted_data):
    # Split the data into nonce, ciphertext, and tag
    nonce = encrypted_data[:12]
    ciphertext = encrypted_data[12:-16]
    tag = encrypted_data[-16:]

    # Decrypt the message
    cipher = Cipher(
        algorithms.AES(key),
        modes.GCM(nonce, tag),
    )
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()

    return plaintext.decode()
```

DECRYPTION

Application Flow