COMPE 560 Homework

Deadline May 5, 2025 7:00 AM

The goal of this assignment is to build a UDP-based chat application in Python that incorporates **secure communication** through a **hybrid key exchange** system. This homework will provide practical experience in both **network socket programming** and **applied cryptography**, specifically public-key encryption for key exchange and symmetric encryption for message confidentiality. We have provided examples of each in the previous lecture on socket programming.

Project Description

You will implement a **server** and multiple **clients** that communicate over **UDP sockets**. The server handles message broadcasting between clients and performs the **initial secure key exchange**. Each client generates an RSA public/private key pair and sends the public key to the server. The server, in turn, generates a unique symmetric key (e.g., AES) for the client, encrypts it with the client's public key, and sends it back.

Once the key exchange is complete, all subsequent messages between the client and server will be **encrypted using the symmetric key**, ensuring confidentiality over the connectionless UDP protocol.

Learning Outcomes

By completing this assignment, you will:

- Understand and apply UDP socket programming in Python.
- Implement a secure **hybrid encryption scheme** combining public-key and symmetric-key cryptography.
- Use cryptographic libraries (e.g., cryptography or pycryptodome) for key generation, encryption, and decryption.
- Handle message encoding and transmission in an insecure environment.
- Ability to self learn, research and improve yourself.

Requirements

You must implement the following components:

Server

- Accept public RSA keys from multiple clients.
- Generate a random AES key per client.
- Encrypt each AES key using the corresponding client's RSA public key.
- Send the encrypted AES key back to the client.
- Receive encrypted chat messages from clients.

- Decrypt and broadcast each message (after re-encryption) to all other connected clients.

Client

- Generate an RSA key pair (public/private) at startup.
- Send the public key to the server.
- Receive and decrypt the AES symmetric key using the private key.
- Use the AES key to encrypt outgoing messages and decrypt incoming ones.
- Display chat messages in real time.

Encryption Features

- Use RSA (2048-bit or higher) for key exchange.
- Use AES (128-bit or higher) for symmetric encryption.
- Base64-encode encrypted messages for safe transmission over UDP.
- Ensure proper padding and IV handling (e.g., CBC or GCM mode). (You will need to generate a fresh random Initial Vector for each encryption and use padding if needed to ensure that the plaintext is a multiple of block size)

Graduate Students

- Implement message authentication using HMAC.
- Build a terminal-based UI with curses or a simple GUI
- Add logging or error handling for packet loss and retransmissions (simulating reliable UDP).

Deliverables

1. server.py
2. client.py
3. README.md with:
    - Instructions to run the application
    - Summary of cryptographic design choices
    - Any assumptions or limitations
4. *Graduate Students* : Short report (1–2 pages) describing implementation and design decisions

Submission Instructions

Submit a zipped folder or Git repository link via *canvas* by *may 5, 2025 before 7:00 AM* . Include all source code and supporting documentation.

Allowed Libraries

- socket, base64, secrets, os, threading

- cryptography or pycryptodome for encryption

Source Code Interview

You will be asked to do a code review with the professor. This will require explaining your code and defending your design decisions. In the following pages, we provide a starter code with minimal error control. You are welcome to start with this code or develop your own from scratch.

File crypto_utils.py

```python
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP, AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
import base64

# --- RSA Operations ---
def generate_rsa_keypair():
    key = RSA.generate(2048)
    private_key = key.export_key()
    public_key = key.publickey().export_key()
    return private_key, public_key

def encrypt_with_rsa(public_key_bytes, message_bytes):
    pub_key = RSA.import_key(public_key_bytes)
    cipher_rsa = PKCS1_OAEP.new(pub_key)
    return cipher_rsa.encrypt(message_bytes)

def decrypt_with_rsa(private_key_bytes, encrypted_bytes):
    priv_key = RSA.import_key(private_key_bytes)
    cipher_rsa = PKCS1_OAEP.new(priv_key)
    return cipher_rsa.decrypt(encrypted_bytes)



# --- AES Operations ---
def generate_aes_key():
    return get_random_bytes(16)  # 128-bit key

def encrypt_with_aes(aes_key, plaintext):
    iv = get_random_bytes(16)
    cipher = AES.new(aes_key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))
    return base64.b64encode(iv + ciphertext).decode()

def decrypt_with_aes(aes_key, b64_ciphertext):
    raw = base64.b64decode(b64_ciphertext)
    iv = raw[:16]
    ciphertext = raw[16:]
    cipher = AES.new(aes_key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ciphertext), AES.block_size).decode()
```

File Server.py (overly simplified logic)

```
import socket
import threading
import base64
from crypto_utils import generate_aes_key, encrypt_with_rsa

clients = {}
client_keys = {}

def handle_messages(sock):
    while True:
        data, addr = sock.recvfrom(4096)
        if addr in clients:
            aes_key = clients[addr]
            # Broadcast the encrypted message to other clients
            for client_addr, key in clients.items():
                if client_addr != addr:
                    sock.sendto(data, client_addr)
        else:
            # First message is assumed to be client's RSA public key
            rsa_pub_key = base64.b64decode(data)
            aes_key = generate_aes_key()
            encrypted_key = encrypt_with_rsa(rsa_pub_key, aes_key)
            sock.sendto(base64.b64encode(encrypted_key), addr)
            clients[addr] = aes_key
            client_keys[addr] = rsa_pub_key
            print(f"Key exchanged with {addr}")

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(("localhost", 12345))
    print("Server started on port 12345")
    handle_messages(sock)

if __name__ == "__main__":
    main()
```

File : Client.py (overly simplified version)

```
import socket
import threading
import base64
from crypto_utils import (
    generate_rsa_keypair, decrypt_with_rsa,
    encrypt_with_aes, decrypt_with_aes
)

aes_key = None
def receive_messages(sock, private_key):
    global aes_key
    while True:
        data, _ = sock.recvfrom(4096)
        if aes_key is None:
            encrypted_key = base64.b64decode(data)
            aes_key = decrypt_with_rsa(private_key, encrypted_key)
            print("Received and decrypted AES key.")
        else:
            try:
                decrypted = decrypt_with_aes(aes_key, data.decode())
                print("Message:", decrypted)
            except:
                pass

def main():
    global aes_key
    server_addr = ("localhost", 12345)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Generate RSA keys and send public key
    private_key, public_key = generate_rsa_keypair()
    sock.sendto(base64.b64encode(public_key), server_addr)

    # Start thread to receive messages
    threading.Thread(target=receive_messages, args=(sock, private_key), daemon=True).start()
    while True:
        msg = input()
        if aes_key:
            encrypted = encrypt_with_aes(aes_key, msg)
            sock.sendto(encrypted.encode(), server_addr)
        else:
            print("Waiting for key exchange to complete...")

if __name__ == "__main__":
    main()
```