University of Windsor

Lab 5

Course: Networking and Data Security

COMP8677-1-R-2023F

Professor: Dr. Shaoquan Jiang

Prepared by

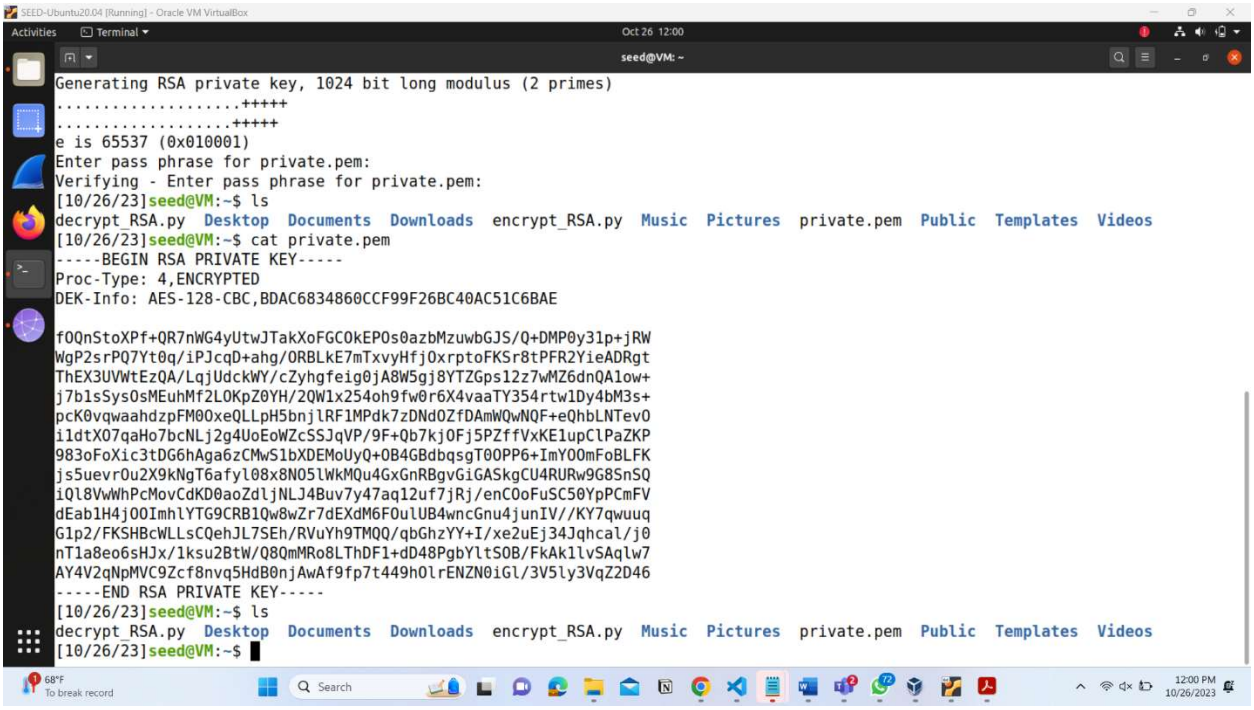
Harshil Hitendrabhai Panchal (110096129)

Due date: October 26, 2023

1. **Use openssl to generate RSA public/private key**

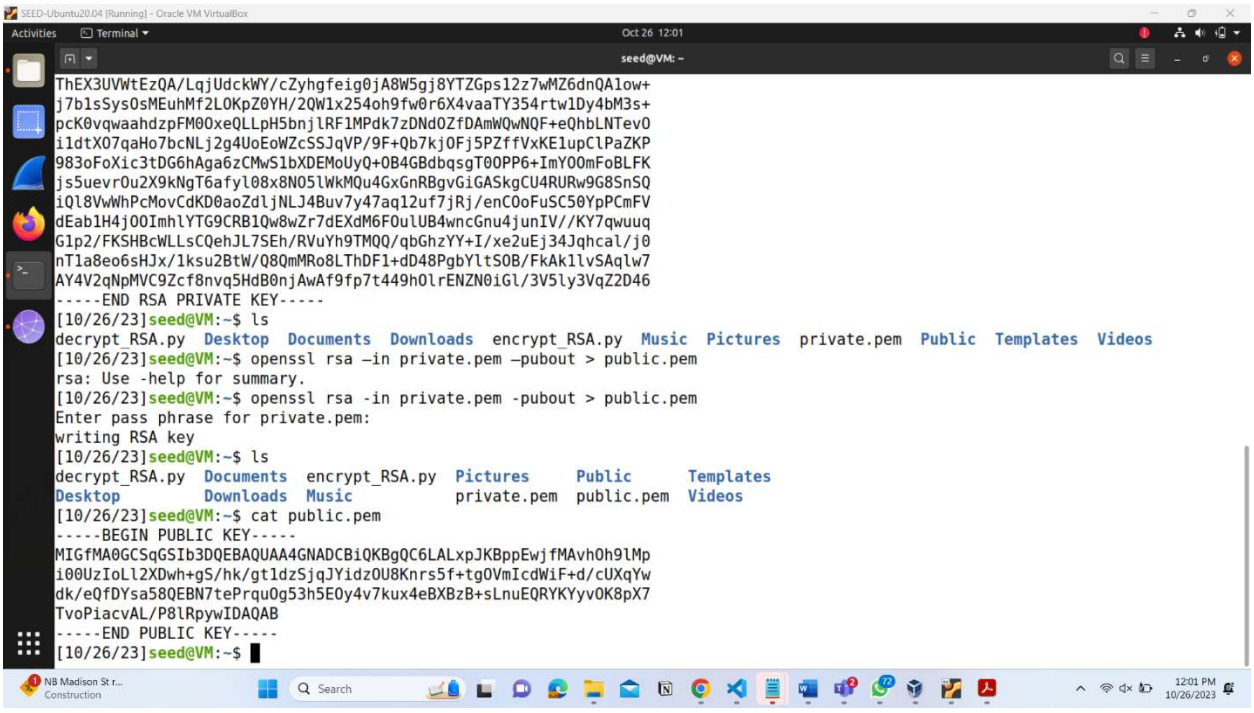
   We can generate RSA private key (p, q, d) using openssl:

   $ openssl genrsa –aes128 -out private.pem 1024



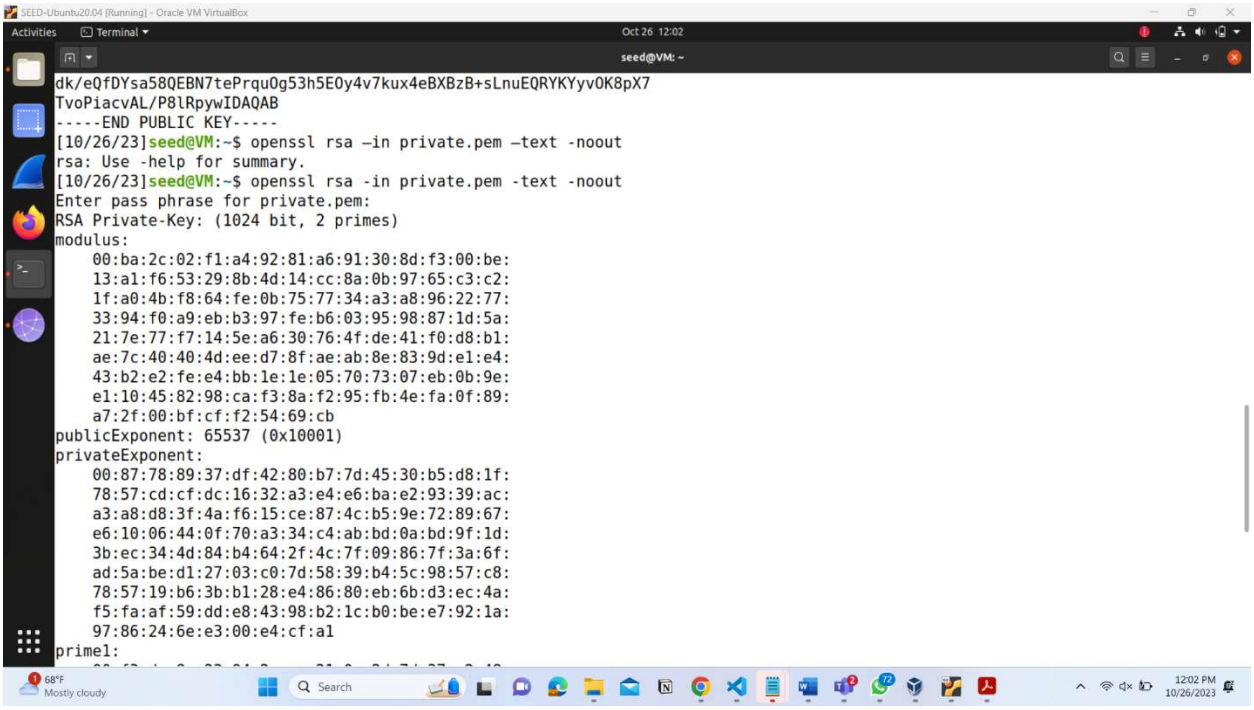
This will generate a rsa instance (p, q, d, e, n) with p, q of 1024 bits and to prevent leaking the private key, the output private.pem is encrypted by aes128 cipher with password you will be prompted to provide. Now use the above command to generate a rsa private key and save it in file private.pem. Then, extract the public key (e, n) in a file public.pem:

$ openssl rsa –in private.pem –pubout >public.pem



You can display private key using

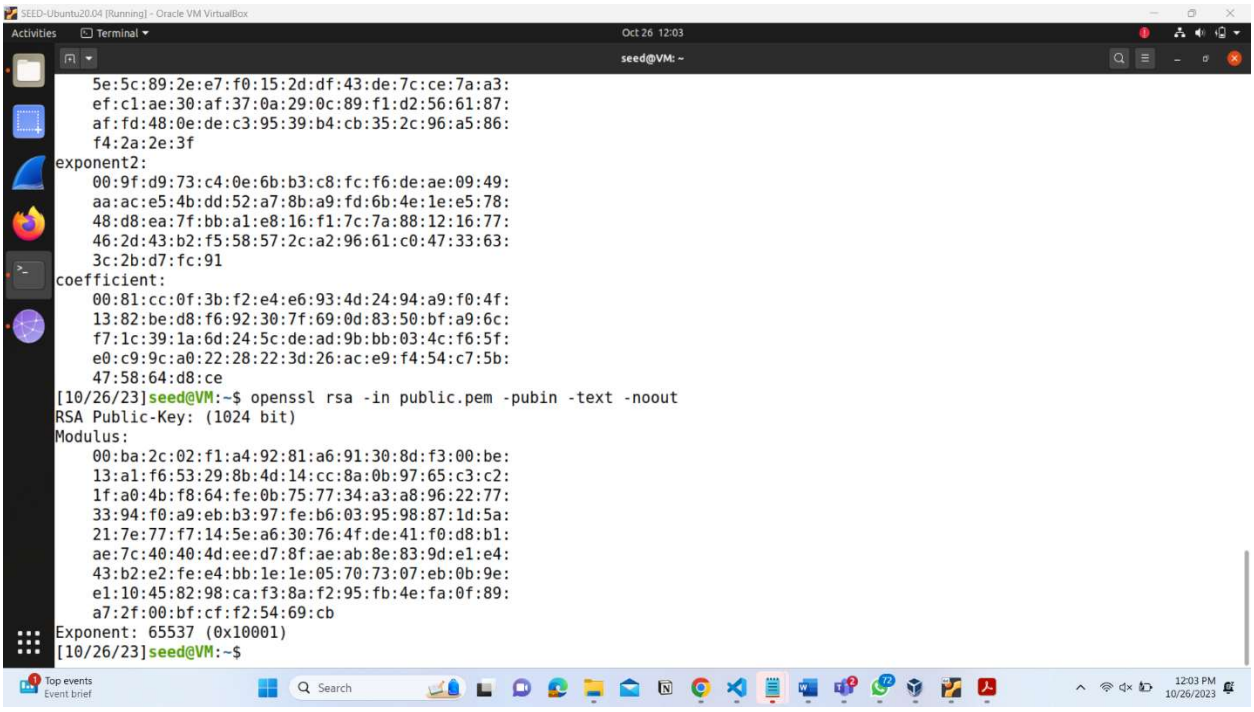
$openssl rsa –in private.pem –text -noout



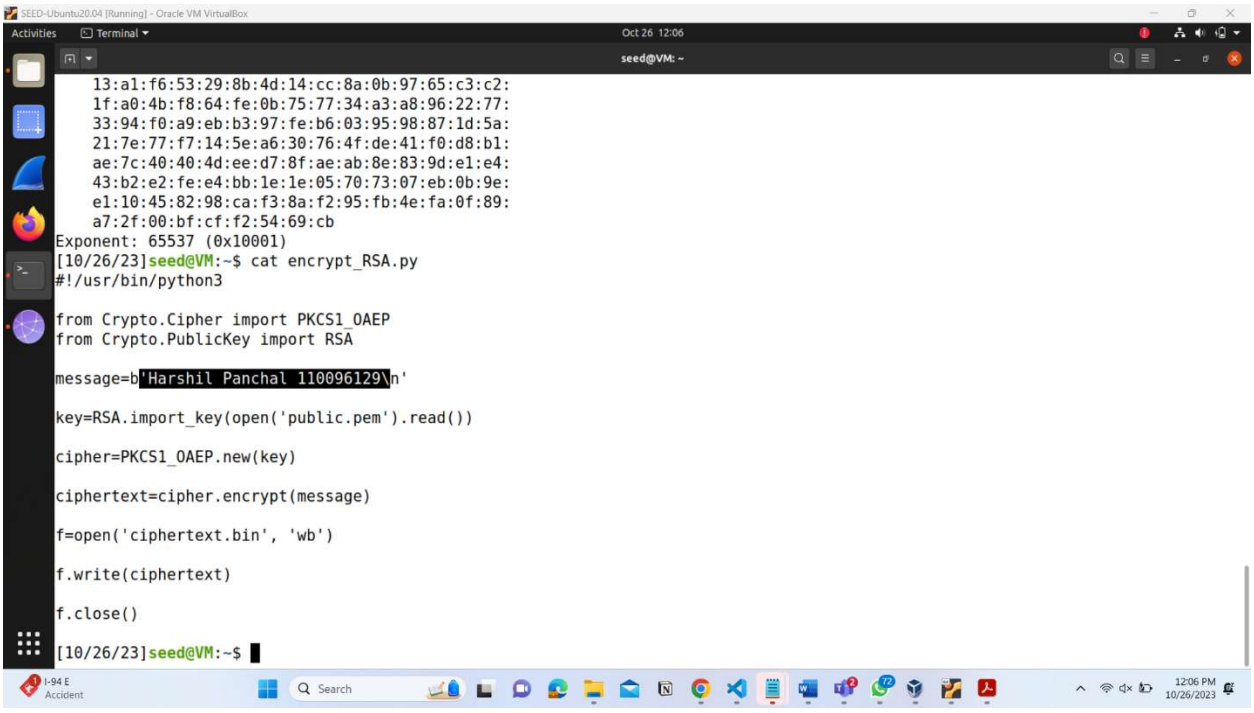
You also can display public key using

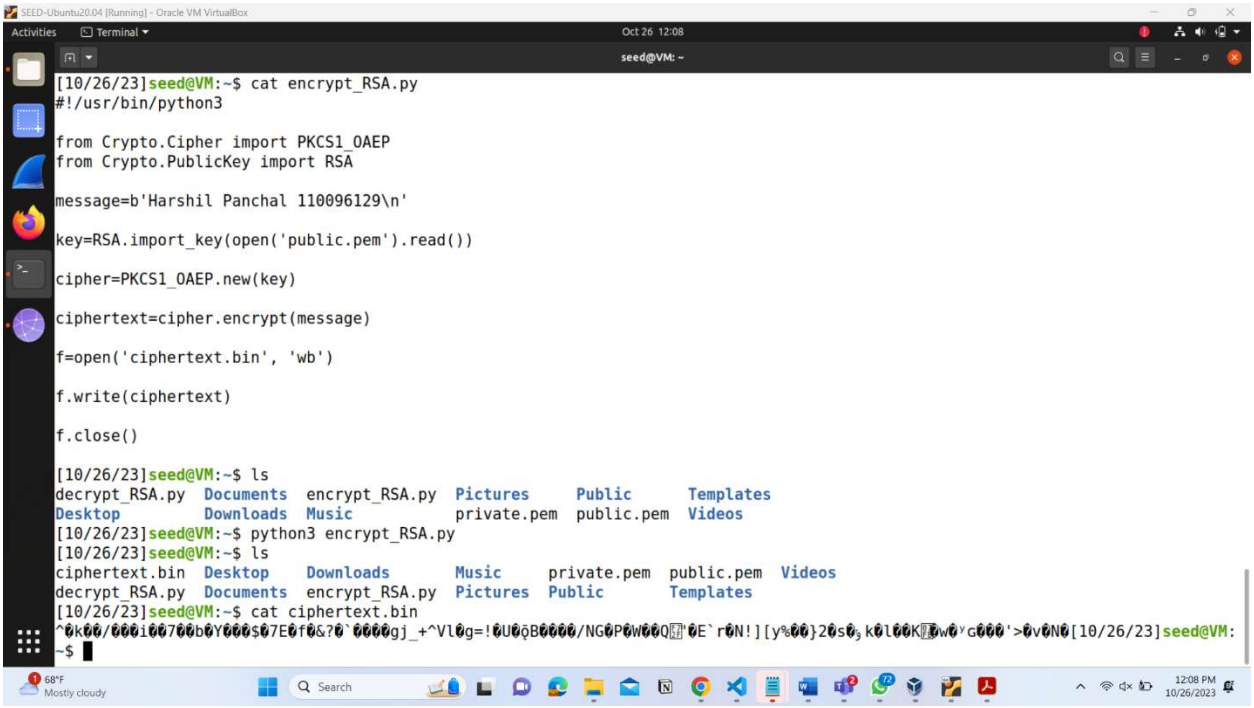
$openssl rsa –in public.pem –pubin –text -noout

Take screen for the displays for these two files, as evidence of your work

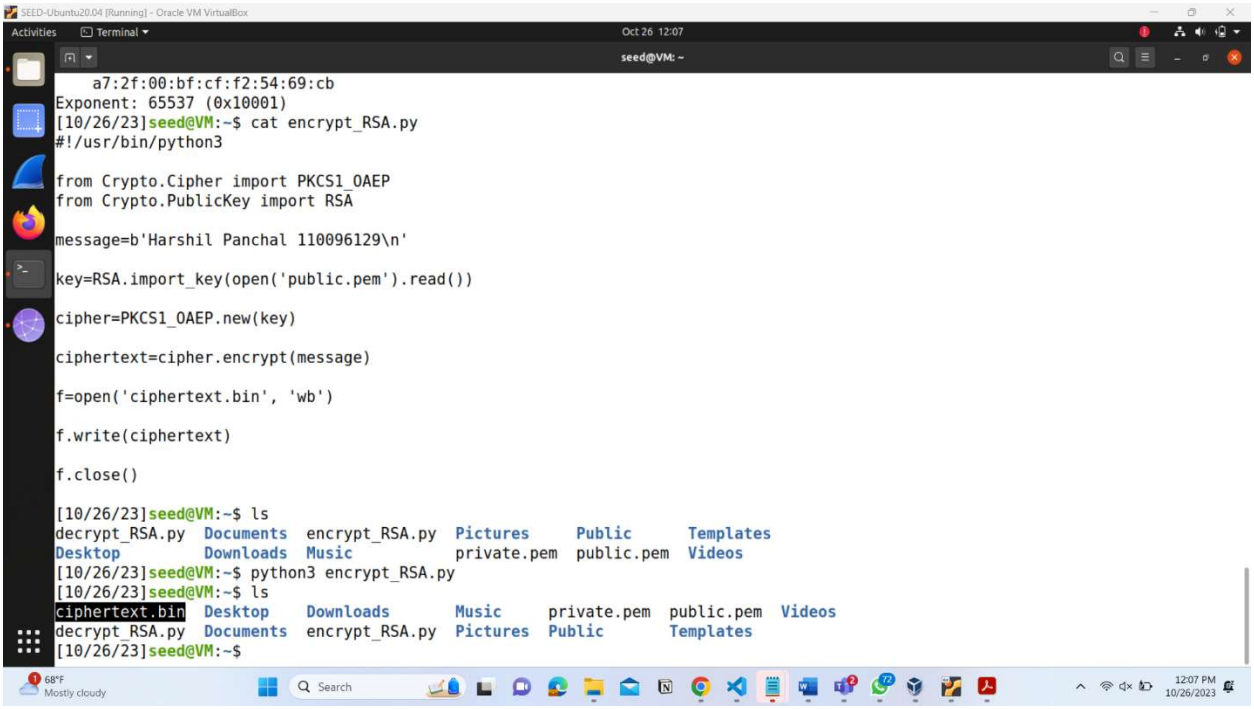


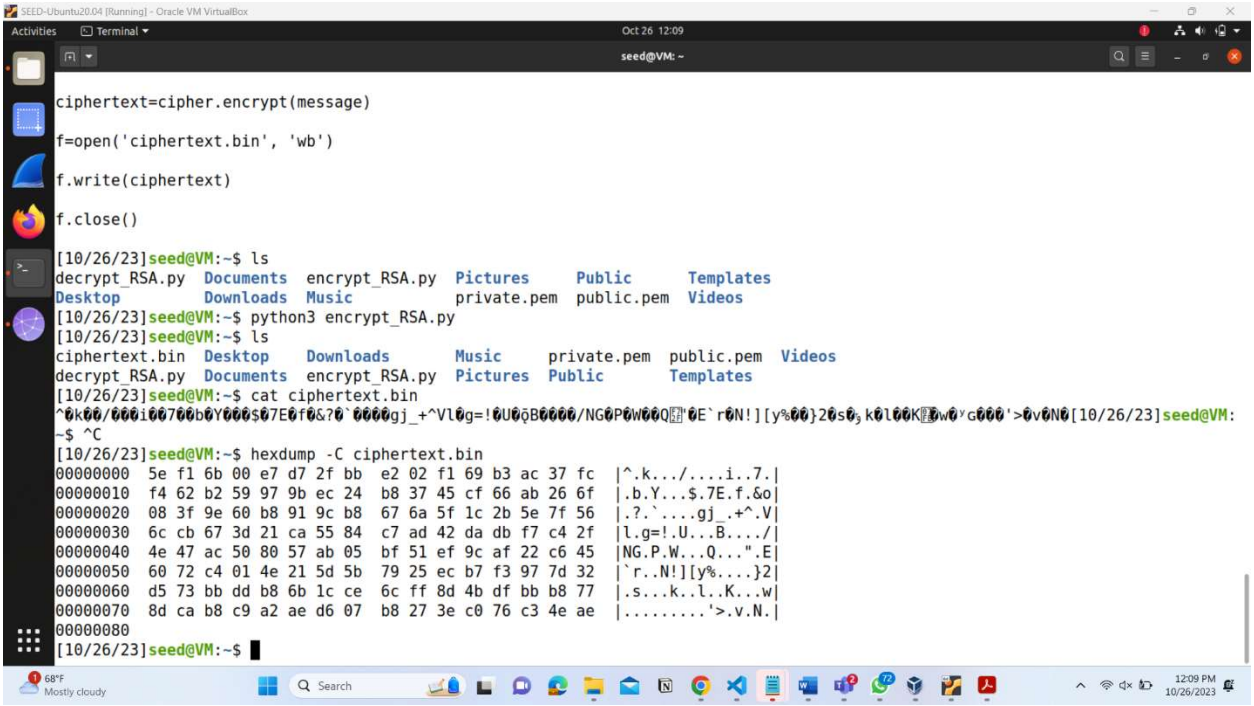
2. **In this problem, you need to practice RSA encryption and decryption.**
    a. Encrypt messages using PKCS1_OAEP, which is an implementation of RSA. Use the key **RsaKey** derived above to do the encryption. The functions are described as follow.
        i. Cipher=**PKCS1_OAEP.new**(RsaKey): ○ For the encryption, RsaKey is a public-key. Return an encryption object **Cipher**.
        ii. Cipher.**encrypt**(message): ○ This returns ciphertext of message (byte string) under encryption object **Cipher**.





Encrypt message=**'your name and ID'** and save ciphertext into a file. Take a screen shot for hexdump of your ciphertext (**$hexdump -C filename**). Ref. **encrypt_RSA.py**.

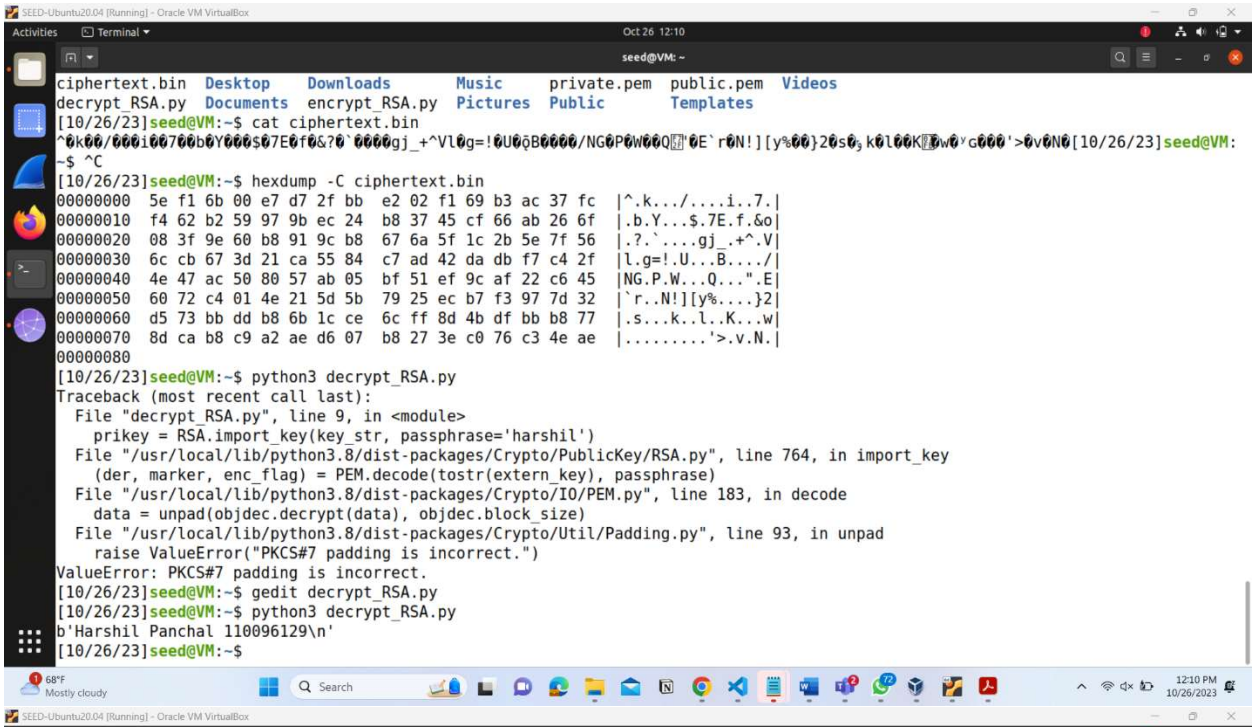


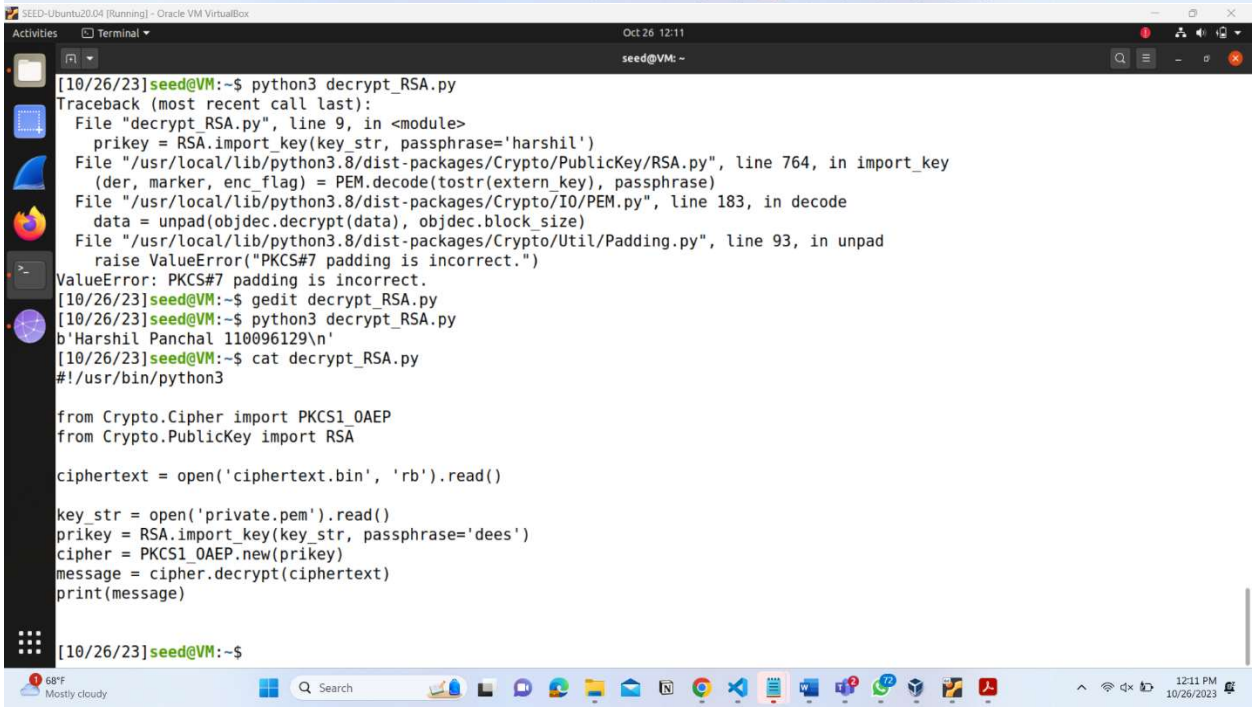


b. Decrypt the ciphertext in (a). The functions are described as follow.

i. Cipher=**PKCS1_OAEP.new**(RsaKey): ○ For the decryption, RsaKey is a private-key. Return an decryption object **Cipher**.

ii. Cipher.**decrypt**(ctxt): ○ This returns message=**'your name and ID'** under decryption object **Cipher**.





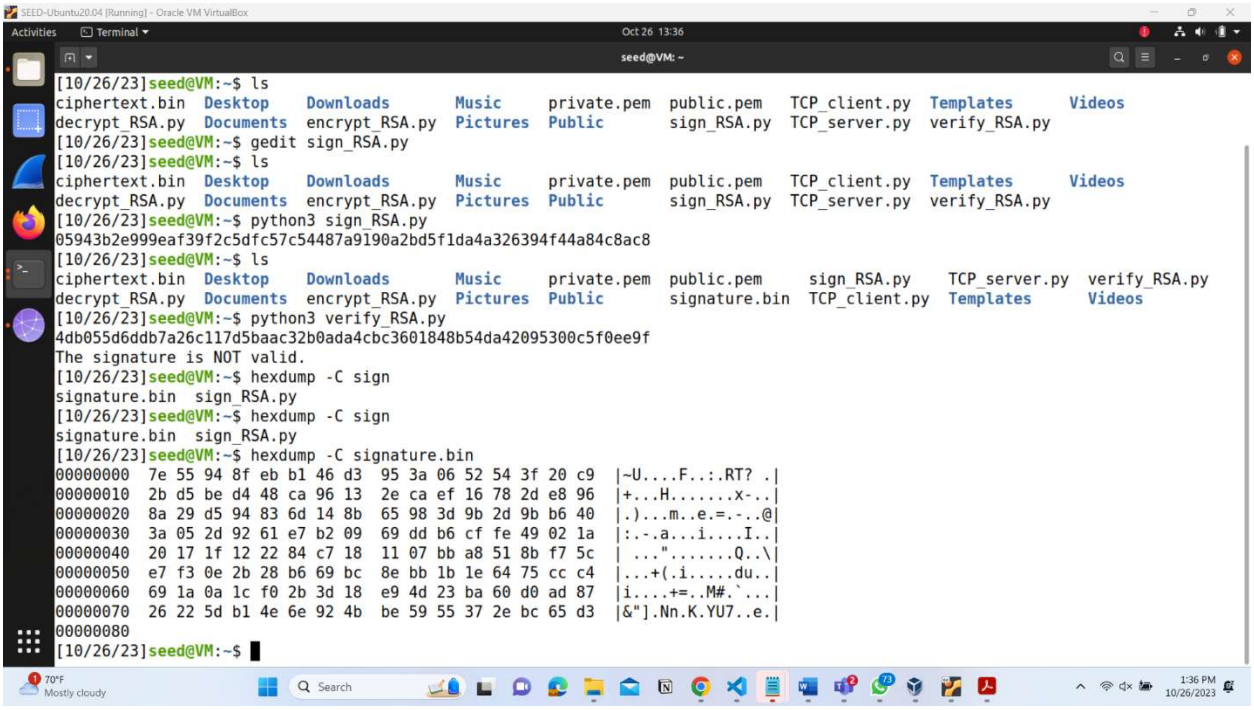
Take a screen shot for your decryption. Ref. **decrypt_RSA.py**.

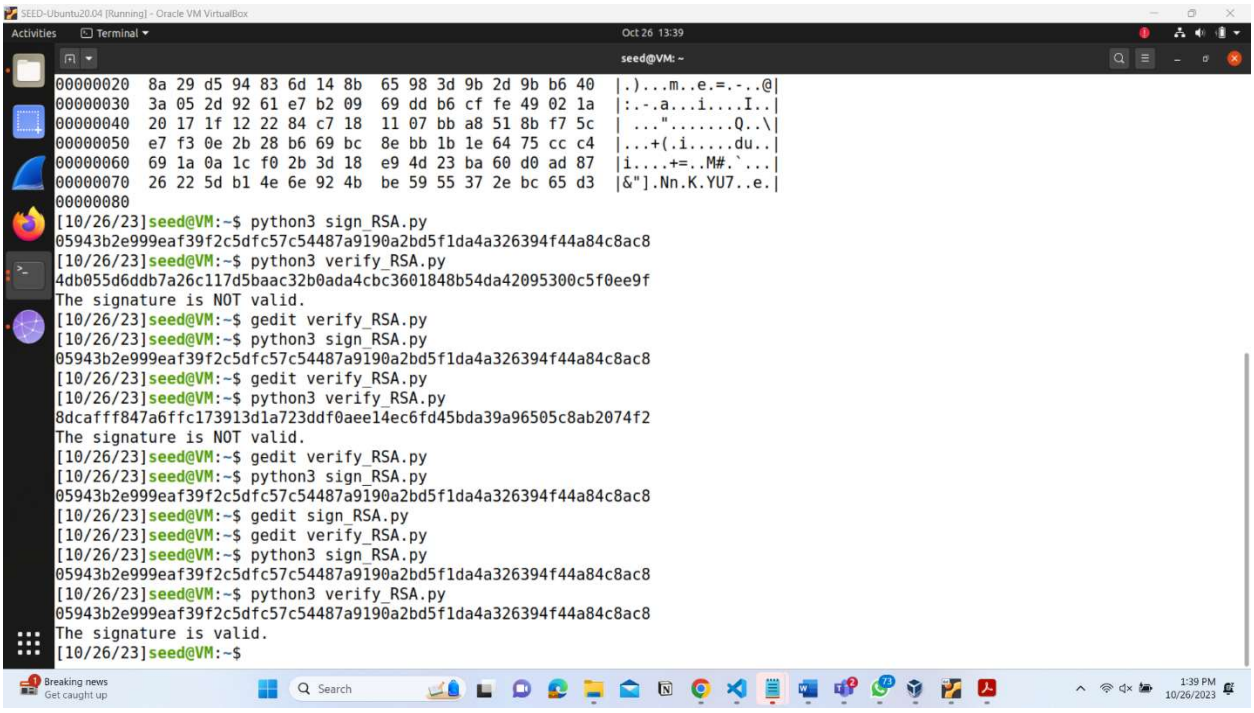
3. **In this problem, you practice RSA signature: generation and verification.**

a. Generate RSA based signature. The functions are described as follows.

- **Signer=pss.new(RsaKey):**
  - **o This defines a signing object signer with RsaKey (imported from your RSA private key file).**
- **Signer.sign(hashedmessage):**
  - **o This generates the RSA signature of the hashed message. Here you can use SHA512 to generate the hash value of your message.**

M = "I owe you $2000". Change $2000 to $3000 and sign the modified message. Compare both signatures. Are they similar? Save your signature into a file. Take a screen shot for your file content (using hexdump). Ref. sign_RSA.py





4. **In this problem, you will use Diffie-Hellman with authentication to protect the client-server communication. Implement the following functionalities.**
    a. Create two files: TCP client and TCP server, capable to chat with each other using socket.

SERVER:

```
import socket

from Crypto.Random.random import getrandbits

from Crypto.PublicKey import DSA

from Crypto.Util.number import bytes_to_long, long_to_bytes

from Crypto.Hash import SHA256


p =
25822498780869085896559191720030118743297057928292235128306593565406476220168
411946296453532801378314359031719727475597 79

g = 2


def compute_shared_key(private_key, other_public_key):

    return pow(other_public_key, private_key, p)
```

```
def hash_shared_key(shared_key):
    return SHA256.new(long_to_bytes(shared_key)).digest()

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
    server_socket.bind(('127.0.0.1', 12345))
    server_socket.listen()

    print("Server is listening...")
    conn, addr = server_socket.accept()
    with conn:
        print("Connected by", addr)
        y = getrandbits(400)
        server_public_key = pow(g, y, p)
        conn.sendall(long_to_bytes(server_public_key))

        client_public_key = bytes_to_long(conn.recv(1024))

        shared_key = compute_shared_key(y, client_public_key)
        sk = hash_shared_key(shared_key)
        print("Secret Key:", sk)

        while True:
            data = conn.recv(1024)
            if not data:
                break
            print("Client:", data.decode('utf-8'))
            message = input("Server: ")
            conn.sendall(message.encode('utf-8'))
```

CLIENT:

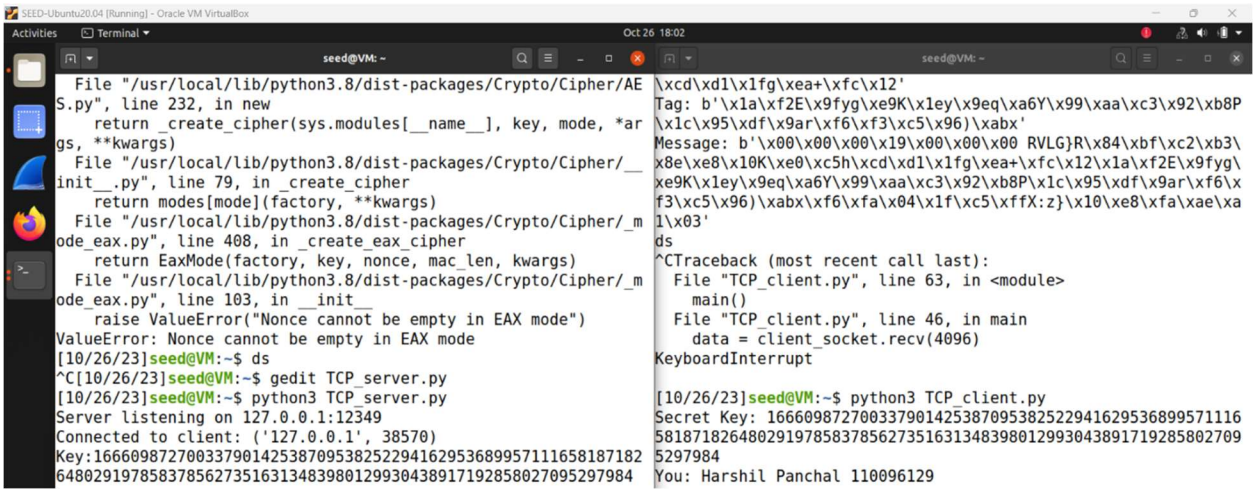
```
import socket
from Crypto.Random.random import getrandbits
from Crypto.PublicKey import DSA
```

```
from Crypto.Util.number import bytes_to_long, long_to_bytes
from Crypto.Hash import SHA256

p =
25822498780869085896559191720030118743297057928292235128306593565406476220168
41194629645353280137831435903171972747559779
g = 2

def compute_shared_key(private_key, other_public_key):
    return pow(other_public_key, private_key, p)

def hash_shared_key(shared_key):
    return SHA256.new(long_to_bytes(shared_key)).digest()

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
    client_socket.connect(('127.0.0.1', 12345))

    x = getrandbits(400)
    client_public_key = pow(g, x, p)
    client_socket.sendall(long_to_bytes(client_public_key))

    server_public_key = bytes_to_long(client_socket.recv(1024))

    shared_key = compute_shared_key(x, server_public_key)
    sk = hash_shared_key(shared_key)
    print("Secret Key:", sk)

    while True:
        message = input("Client: ")
        client_socket.sendall(message.encode('utf-8'))
        data = client_socket.recv(1024)
        print("Server:", data.decode('utf-8'))
```

b. Client and Server execute Diffie-Hellman to generate a shared key and use sha256 to hash this shared key to 32-byte secret **sk.** Diffie-Hellman uses parameters:

p=25822498780869085896559191720030118743297057928292235128306593565406476220168411946296453532801378314359031719727475597 79

g=2



Note: x, y in Diffie-Hellman can be obtained with **`Crypto.Random.random.getrandbits`(400);** see https://pycryptodome.readthedocs.io/en/latest/src/random/random.html if necessary.

c. Sender (Client or Server) uses **sk** as a secret key of AES to encrypt your chat message in (a).
This results in ciphertext C and computes tag=sha256(C). In (a), sender sends (C, tag), instead of plain chat message.

SERVER:

```
import socket
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Random.random import getrandbits
from Crypto.Hash import SHA256

# Diffie-Hellman parameters
p =
25822498780869085896559191720030118743297057928292235128306593565406476220168411946296453532801378314359031719727475597 79
g = 2

def generate_dh_key():
    private_key = getrandbits(400)
    public_key = pow(g, private_key, p)
    return private_key, public_key

def compute_shared_key(private_key, other_public_key):
    return pow(other_public_key, private_key, p)

def main():
    server_ip = '127.0.0.1'
    server_port = 12349

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((server_ip, server_port))
```

```
    server_socket.listen(1)

    print("Server listening on {}:{}".format(server_ip, server_port))

    conn, addr = server_socket.accept()
    print("Connected to client:", addr)

    private_key, public_key = generate_dh_key()

    # sending the public key
    conn.send(str(public_key).encode())

    # receive the clients public key
    client_public_key = int(conn.recv(4096).decode())

    # create a shared key on both the sides
    shared_key = compute_shared_key(private_key, client_public_key)
    print(f"Key:{shared_key}")
    secret_key = SHA256.new(str(shared_key).encode()).digest()

    aes_cipher = AES.new(secret_key, AES.MODE_EAX,nonce=b'\x00' * 16)

    while True:
        data = conn.recv(4096)
        ciphertext_length = int.from_bytes(data[:4], 'big')
        tag_length = int.from_bytes(data[4:8], 'big')
        ciphertext = data[8:8+ciphertext_length]
        print(f"Cipher Text (C): {ciphertext}")
        received_tag =
data[8+ciphertext_length:8+ciphertext_length+tag_length]
        print(f"Tag: {received_tag}")
        nonce=data[8+ciphertext_length+tag_length:]
        print(f"secret_key (sk): {secret_key}")
        aes_cipher = AES.new(secret_key, AES.MODE_EAX, nonce=nonce)
        decrypted_message = aes_cipher.decrypt(ciphertext)
        print(f"Decrypted Message: {decrypted_message}")
        # Verify the tag
        new_tag = SHA256.new(ciphertext).digest()
        if new_tag != received_tag:
            print("Tag verification failed. Message might be tampered.")
        else:
            print("Client:", decrypted_message.decode())  # Convert to string
for display

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

CLIENT:

```
import socket
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Random.random import getrandbits
from Crypto.Hash import SHA256

# Diffie-Hellman parameters
```

```
p =
25822498780869085896559191720030118743297057928292235128306593565406476220168
411946296453532801378314359031719727475597 79
g = 2

def generate_dh_key():
    private_key = getrandbits(400)
    public_key = pow(g, private_key, p)
    return private_key, public_key

def compute_shared_key(private_key, other_public_key):
    return pow(other_public_key, private_key, p)

def main():
    server_ip = '127.0.0.1'
    server_port = 12349

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((server_ip, server_port))

    private_key, public_key = generate_dh_key()
    client_socket.send(str(public_key).encode())

    server_public_key = int(client_socket.recv(4096).decode())
    shared_key = compute_shared_key(private_key, server_public_key)
    print(f"Secret Key: {shared_key}")
    secret_key = SHA256.new(str(shared_key).encode()).digest()
    aes_cipher = AES.new(secret_key, AES.MODE_EAX)

    while True:
        message = input("You: ")

        # Encrypt the message and compute the tag
        ciphertext = aes_cipher.encrypt(message.encode('utf-8'))
        print(f"Cipher Text (C): {ciphertext}")
        tag=SHA256.new(ciphertext).digest()
        message_to_send = len(ciphertext).to_bytes(4, 'big') +
len(tag).to_bytes(4, 'big') + ciphertext + tag+aes_cipher.nonce
        print(f"Tag: {tag}")
        client_socket.send(message_to_send)
        print(f"Message: {message_to_send}")
        data = client_socket.recv(4096)
        ciphertext_length = int.from_bytes(data[:4], 'big')
        tag_length = int.from_bytes(data[4:8], 'big')
        ciphertext = data[8:8+ciphertext_length]
        received_tag = data[8+ciphertext_length:]

        aes_cipher = AES.new(secret_key, AES.MODE_EAX,
nonce=aes_cipher.nonce)
        decrypted_message = aes_cipher.decrypt(ciphertext)

        # Verify the tag
        new_tag = SHA256.new(ciphertext).digest()
        if new_tag != received_tag:
            print("Tag verification failed. Message might be tampered.")
        else:
```

```
            print("Server:", decrypted_message.decode())  # Convert to string
for display

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

d. At the receiver, when receiving (C, tag), verify whether tag=sha256(C) holds. If it fails, raise exception; otherwise, use sk as the AES secret to decrypt C. This will recover your chat message.
Paste your client and server programs in your submission file. Print out sk, C, tag and decrypted *chat message* in (d) for one *chat message*.