

COMP-8677

Networking and Data Security

LAB 5 Report

Professor

Dr. Shaoquan Jiang

Karan Vishavjit

Student Id: 110099867

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

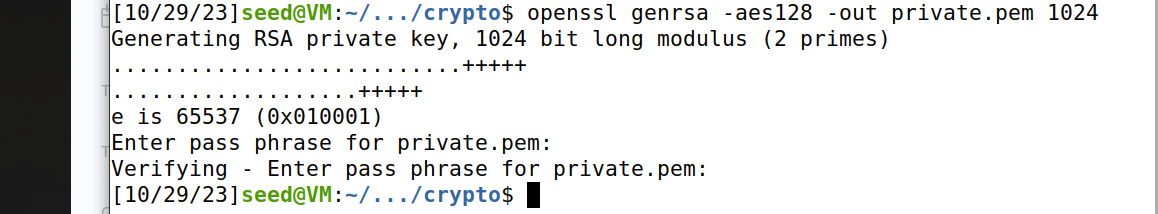
**$openssl rsa –in private.pem –text -noout**

You also can display public key using

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

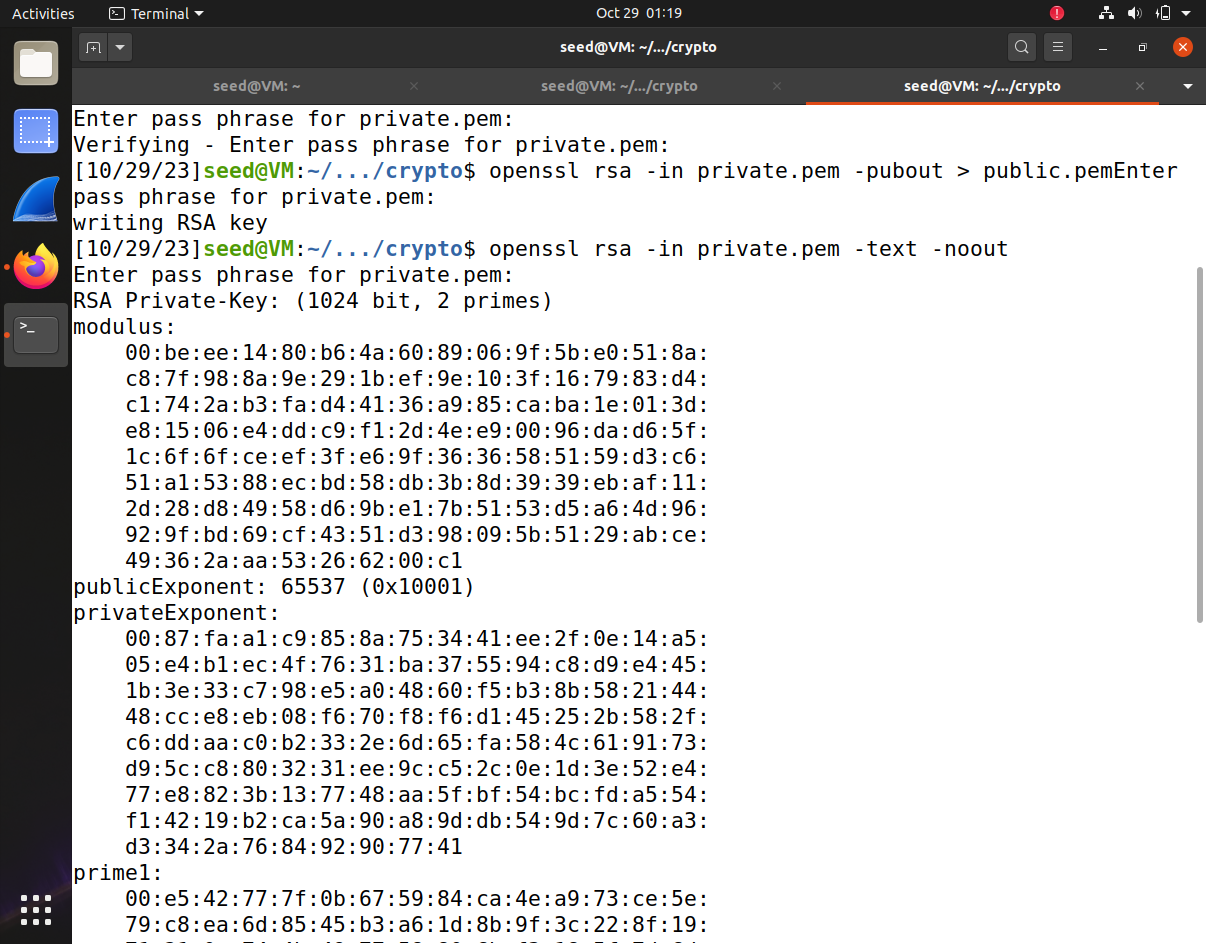
Take screen for the displays for these two files, as evidence of your work**.**

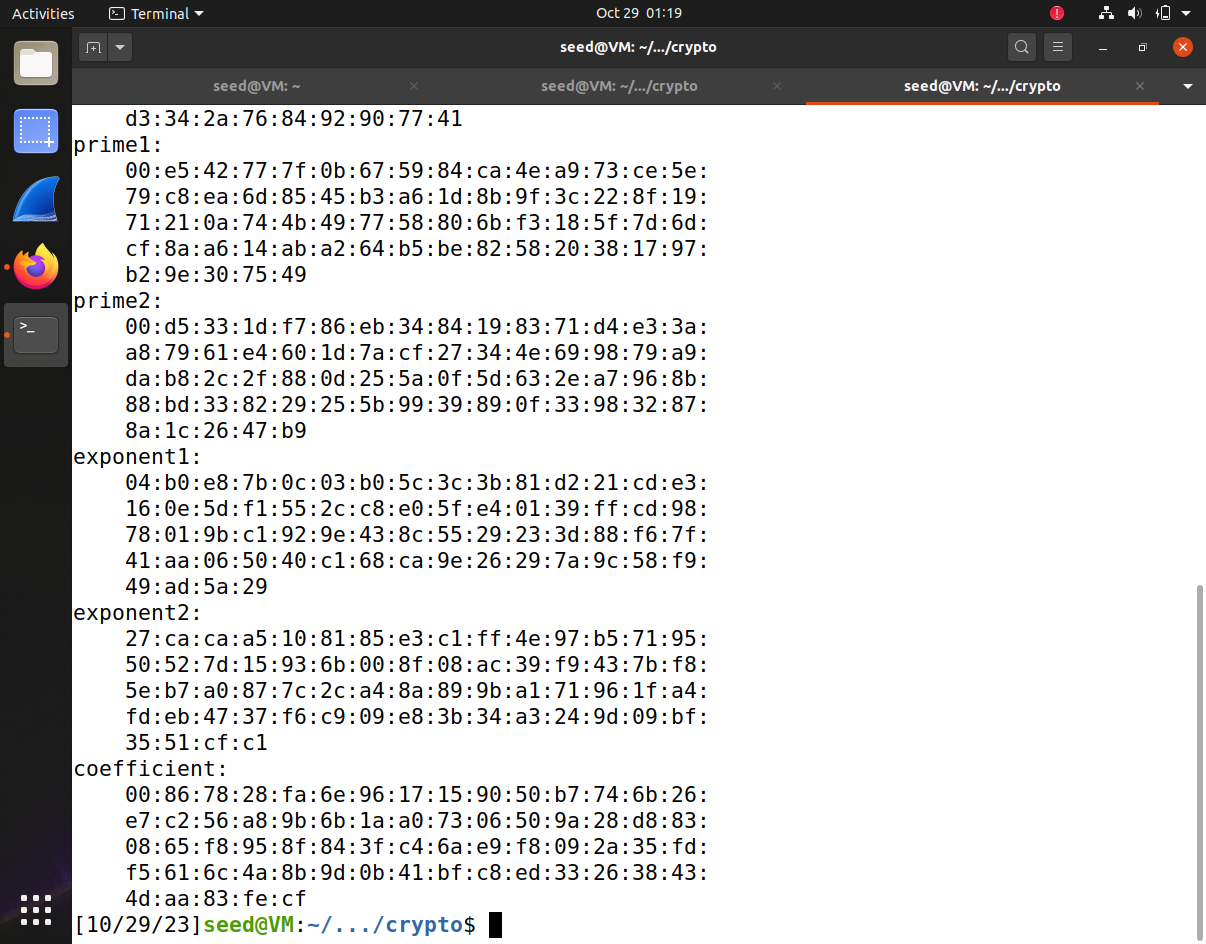
**Step 1:** $ openssl genrsa –aes128 -out private.pem 1024



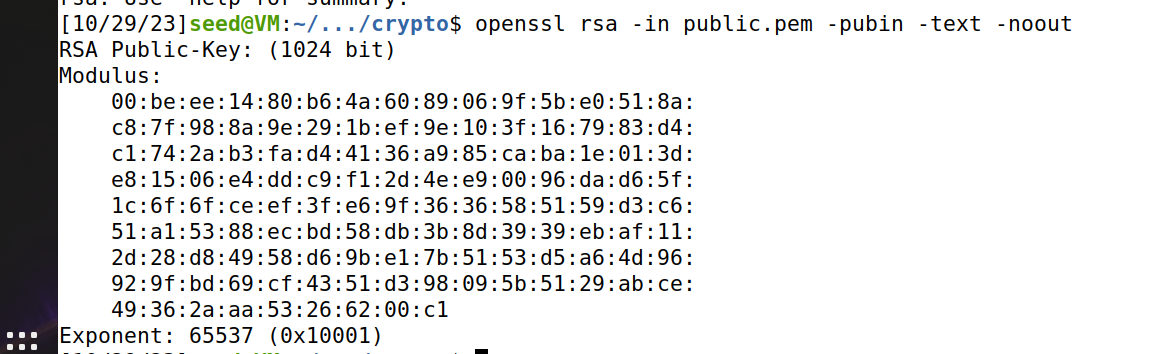
**Step 2:** $ openssl rsa –in private.pem –pubout >public.pem

**Step 3:** $ opnessl rsa –in private.pem –text -noout





**Step 4:** $openssl rsa –in public.pem –pubin –text –noout



**2.** In this problem, you need to practice RSA encryption and decryption.

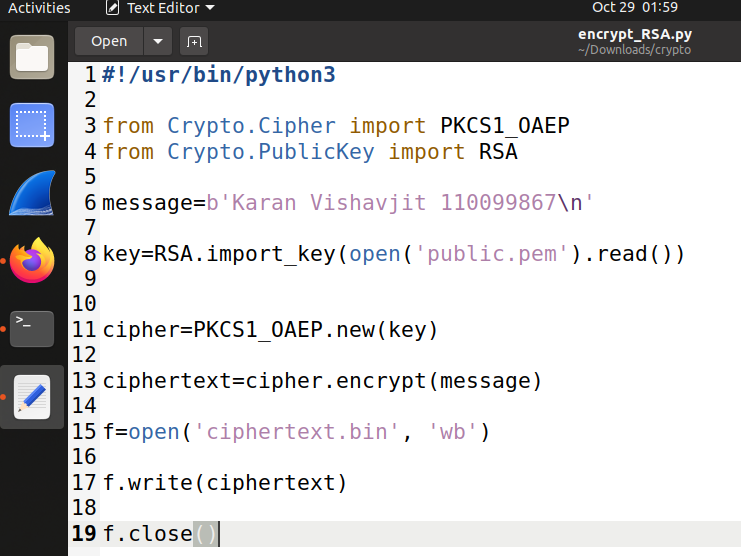
**a**. Encrypt messages using PKCS1\_OAEP, which is an implementation of RSA. Use the key **RsaKey** derived above to do the encryption. The functions are described as follow.

• Cipher**=PKCS1\_OAEP.new**(RsaKey):

O For the encryption, RsaKey is a public-key. Return an encryption object **Cipher**.

• Cipher.**encrypt**(message): o 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**.



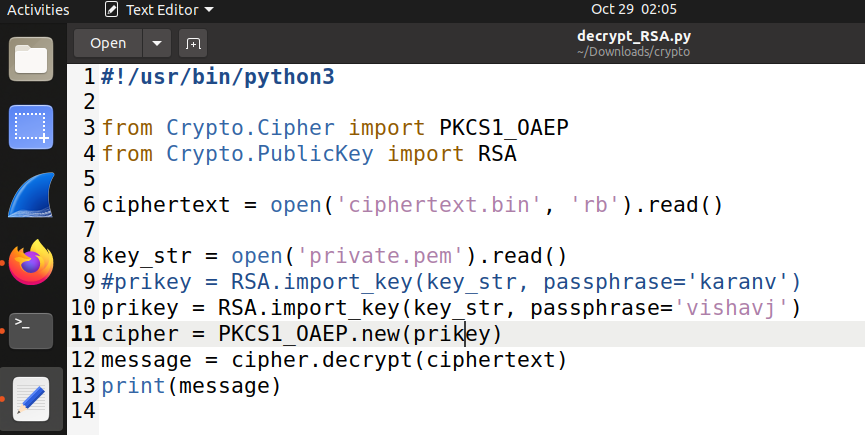


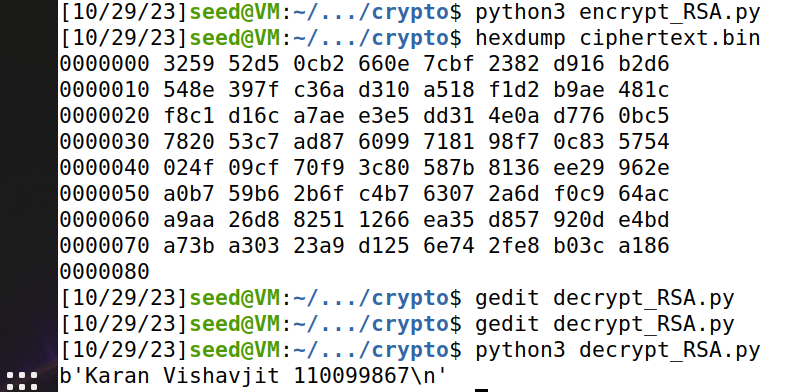
**b.** Decrypt the ciphertext in (a). The functions are described as follow.

• Cipher**=PKCS1\_OAEP.new**(RsaKey): o For the decryption, RsaKey is a private-key. Return an decryption object **Cipher**.

• Cipher.**decrypt**(ctxt): o This returns message=**’your name and ID’** under decryption object **Cipher**.

Take a screen shot for your decryption. Ref. **decrypt\_RSA.py**.





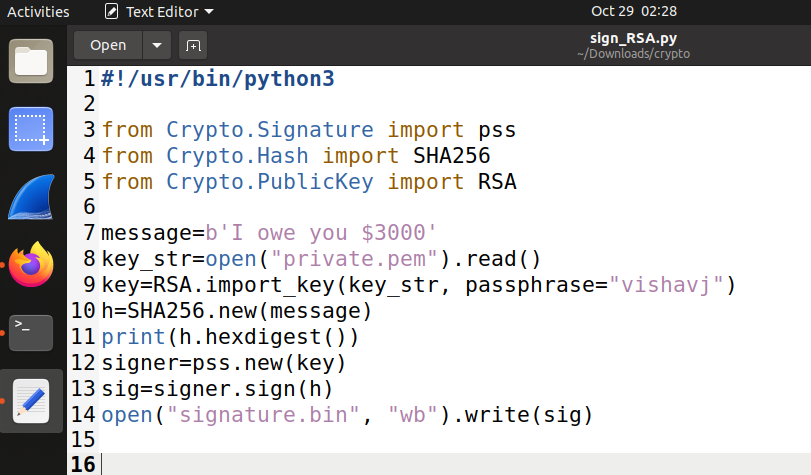
**3**. **(optional)** In this problem, you practice RSA signature: generation and verification.

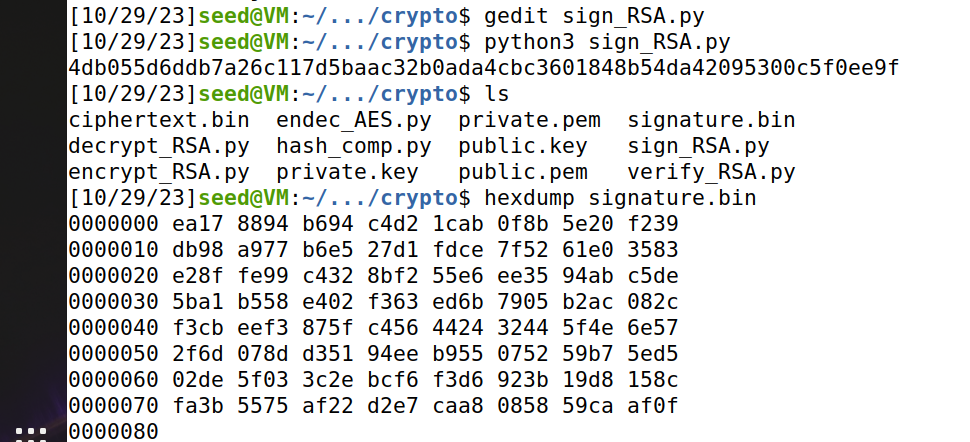
**a**. Generate RSA based signature. The functions are described as follows.

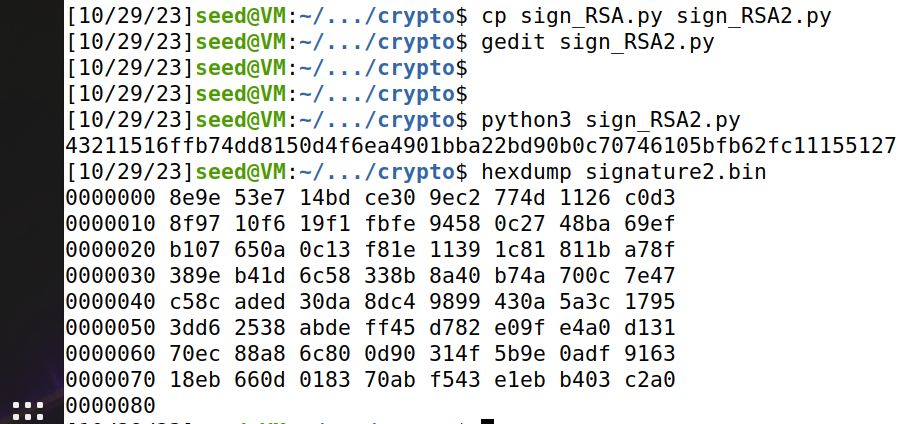
• Signer**=pss.new(**RsaKey): o This defines a signing object *signer* with RsaKey (imported from your RSA private key file).

• 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**





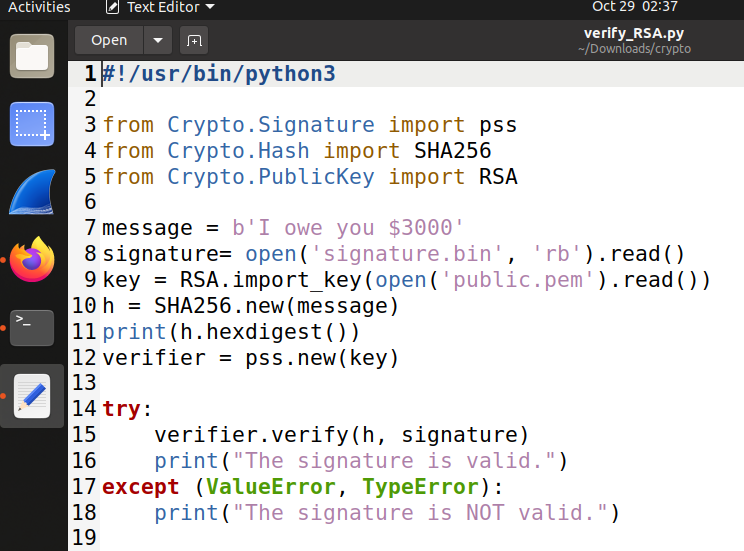


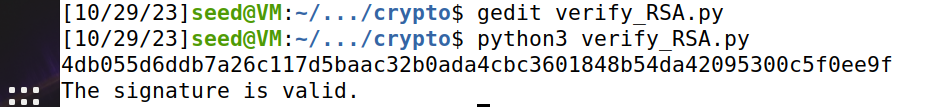
**b.** Verify the signature in (a). The functions are described as follows.

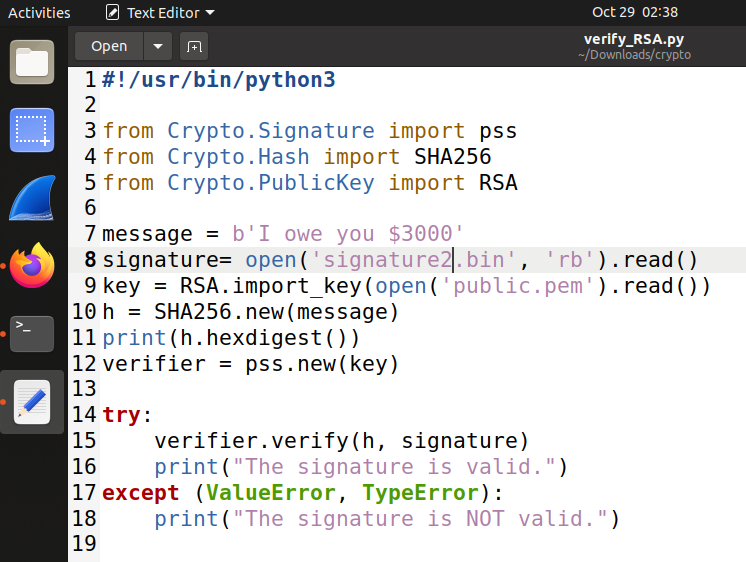
• Signer**=pss.new(**RsaKey): o This defines a signing object *signer* with RsaKey (imported from your RSA public key file).

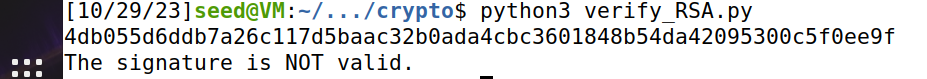
• Signer.**verify**(*hashedmessage*, *signature*): o This verifies if *signature* is consistent with the *hashed message*.

Take a screen shot for the output result. Ref. **verify\_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.

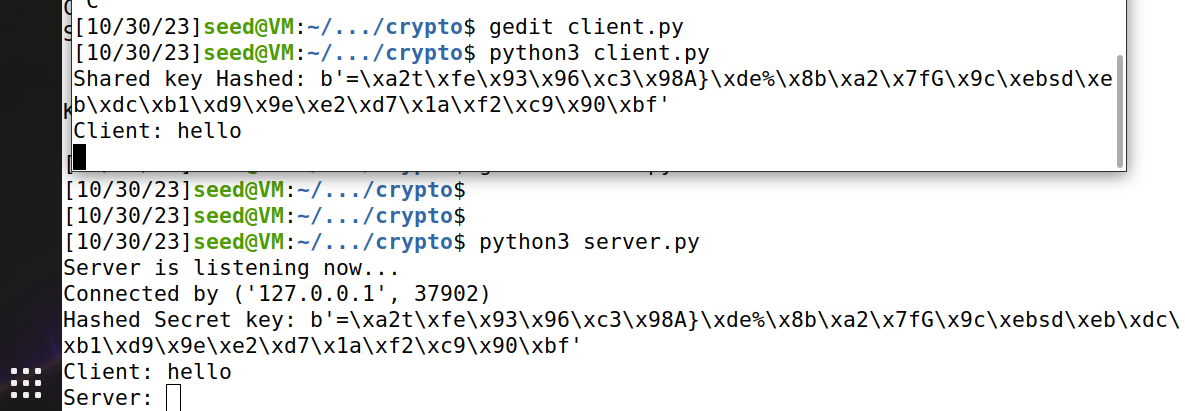
**b.** Client and Server execute Diffie-Hellman to generate a shared key and use sha256 to hash this shared key to 32-byte secret sk. Diffie-Hellman uses parameters:

p=2582249878086908589655919172003011874329705792829223512830659356540647622016841194629645353280137831435903171972747559779

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.



**Server.py** File

import socket

from Crypto.Random.random import getrandbits

from Crypto.Hash import SHA256

from Crypto.Util.number import bytes\_to\_long, long\_to\_bytes

p = 25822498780869085896559191720030118743297057928292235128306593565406476220168

41194629645353280137831435903171972747559779

g = 2

def generate\_keypair():

# Generates private and public key

private\_key = getrandbits(400)

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

return private\_key, public\_key

def sharedKey(private\_key, public\_key):

# Computes shared key using private and public keys

return pow(public\_key, private\_key, p)

def hash(key):

# Hashes the shared key

return SHA256.new(long\_to\_bytes(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 now...")

conn, addr = server\_socket.accept()

with conn:

print("Connected by", addr)

# Generate keypair

private\_key, public\_key = generate\_keypair()

# Send public key

conn.sendall(long\_to\_bytes(public\_key))

# Receive client public key

client\_public\_key = bytes\_to\_long(conn.recv(1024))

# Compute shared key

shared\_key = sharedKey(private\_key, client\_public\_key)

# Hash shared key

key = hash(shared\_key)

print("Hashed Secret key:", key)

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.py** file

import socket

from Crypto.Hash import SHA256

from Crypto.Util.number import bytes\_to\_long, long\_to\_bytes

from Crypto.Random.random import getrandbits

p =25822498780869085896559191720030118743297057928292235128306593565406476220168

41194629645353280137831435903171972747559779

g = 2

def generate\_keypair():

# Generate private and public keys

private\_key = getrandbits(400)

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

return private\_key, public\_key

def sharedKey(private\_key, public\_key):

# Compute shared key

return pow(public\_key, private\_key, p)

def hash(key):

# Hash the shared key

return SHA256.new(long\_to\_bytes(key)).digest()

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as client\_socket:

client\_socket.connect(("127.0.0.1", 12345))

# Generate keypair

private\_key, public\_key = generate\_keypair()

# Send public key

client\_socket.sendall(long\_to\_bytes(public\_key))

# Receive server public key

server\_public\_key = bytes\_to\_long(client\_socket.recv(1024))

# Compute shared key

shared\_key = sharedKey(private\_key, server\_public\_key)

# Hash shared key

key = hash(shared\_key)

print("Shared key Hashed:", key)

while True:

message = input("Client: ")

client\_socket.sendall(message.encode("utf-8"))

data = client\_socket.recv(1024)

print("Server:", data.decode("utf-8"))

**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.

**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*.

**AES encrypted Client.py file**

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 = 258224987808690858965591917200301187432970579282922351283065935654064762201684119462965453280137831435903171972747559779

g = 2

def compute\_shared\_key(private\_key, other\_public\_key):

# Compute the shared key using public and private key

return pow(other\_public\_key, private\_key, p)

def generate\_dh\_key():

# Generate a private and publid key

private\_key = getrandbits(400)

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

return private\_key, public\_key

# Server configuration

server\_ip = '127.0.0.1'

server\_port = 12349

# client socket to connect to the server

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

client\_socket.connect((server\_ip, server\_port))

# Generate the client's private and public key

client\_private\_key, client\_public\_key = generate\_dh\_key()

# Send client's public key to the server

client\_socket.send(str(client\_public\_key).encode())

# Receive the server's public key

server\_public\_key = int(client\_socket.recv(4096).decode())

# Calculate the shared key using the client's private key and server's public key

shared\_key = compute\_shared\_key(client\_private\_key, server\_public\_key)

print(f"Shared Key: {shared\_key}")

# Derive a secret key from the shared key using a secure hash function

secret\_key = SHA256.new(str(shared\_key).encode()).digest()

# Initialize the AES cipher for encryption and decryption

aes\_cipher = AES.new(secret\_key, AES.MODE\_EAX)

while True:

# Get user input for the message to send

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()

# The message to send

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 Sent: {message\_to\_send}")

# Receive the server's response

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:]

# Reinitialize the AES cipher with the received nonce

aes\_cipher = AES.new(secret\_key, AES.MODE\_EAX, nonce=aes\_cipher.nonce)

# Decrypt the server's response

decrypted\_message = aes\_cipher.decrypt(ciphertext)

# Verify the tag by recomputing it and comparing with the received 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

**AES encrypted serverAes.py file**

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 = 258224987808690858965591917200301187432970579282922351283065935654064762201684119462965453280137831435903171972747559779

g = 2

def generate\_dh\_key():

# Generate a private key and its corresponding public 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):

# Compute the shared key using the received public key and own private key

return pow(other\_public\_key, private\_key, p)

def main():

# Server configuration

server\_ip = '127.0.0.1'

server\_port = 12349

# Create a server socket and bind it to the specified IP and port

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))

# Accept incoming client connection

conn, addr = server\_socket.accept()

print("Connected to client:", addr)

# Generate the server's private and public key

server\_private\_key, server\_public\_key = generate\_dh\_key()

# Send the server's public key to the client

conn.send(str(server\_public\_key).encode())

# Receive the client's public key

client\_public\_key = int(conn.recv(4096).decode())

# Calculate the shared key using the server's private key and client's public key

shared\_key = compute\_shared\_key(server\_private\_key, client\_public\_key)

print(f"Shared Key: {shared\_key}")

# Derive a secret key from the shared key using a secure hash function

secret\_key = SHA256.new(str(shared\_key).encode()).digest()

# Initialize the AES cipher for encryption and decryption

aes\_cipher = AES.new(secret\_key, AES.MODE\_EAX, nonce=b'\x00' \* 16)

while True:

# Receive data from the client

data = conn.recv(4096)

# Extract the length of ciphertext, tag, and the ciphertext itself

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:8 + ciphertext\_length + tag\_length]

nonce = data[8 + ciphertext\_length + tag\_length:]

# Reinitialize the AES cipher with the received nonce

aes\_cipher = AES.new(secret\_key, AES.MODE\_EAX, nonce=nonce)

# Decrypt the ciphertext

decrypted\_message = aes\_cipher.decrypt(ciphertext)

# Verify the tag by recomputing it and comparing with the received 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()

