



University
of Windsor

Course Name

Networking and Data Security (COMP-8677)

Document Type

Lab Assignment 5

Professor

Dr. Shaoquan Jiang

Team - Members

Student ID

Manjinder Singh

110097177

NOTES: For simplicity questions and content from lab manual 5 are mentioned in a box.

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.

My Implementation of above:-

I have executed the following set of commands in sequence(Refer Screenshot 1-4):

(A) openssl genrsa -aes128 -out private.pem 1024

(B) openssl rsa -in private.pem -pubout >public.pem

(C) openssl rsa -in private.pem -text -noout

(D) openssl rsa -in public.pem -pubin -text -noout

Command (A) generates a 1024-bit RSA private key, encrypts it with AES-128, and saves it to "private.pem."

Command (B) extracts the public key from "private.pem" and saves it as "public.pem."

Command (C) displays detailed text information about the private key in "private.pem."

Command (D) prints the detailed text information about the public key in "public.pem."

Below are the screenshots for the evidence of work

```

[10/28/23]seed@VM:~/.../crypto$ openssl genrsa -aes128 -out private.pem 1024
Generating RSA private key, 1024 bit long modulus (2 primes)
.....+++++
.....+++++
e is 65537 (0x010001)
Enter pass phrase for private.pem:
Verifying - Enter pass phrase for private.pem:
[10/28/23]seed@VM:~/.../crypto$ openssl rsa -in private.pem -pubout >public.pem
Enter pass phrase for private.pem:
writing RSA key
[10/28/23]seed@VM:~/.../crypto$ openssl rsa -in private.pem -text -noout
Enter pass phrase for private.pem:
RSA Private-Key: (1024 bit, 2 primes)
modulus:
 00:ac:34:6a:28:87:2a:a3:65:1d:b2:f7:a9:29:1d:
e5:93:0a:0e:80:4b:a0:43:90:b9:ae:db:a0:af:ba:
df:b3:db:c5:46:2f:c5:37:b0:e6:83:ff:e4:a7:18:
57:66:8d:f3:2d:c6:c1:63:a5:a3:ac:39:64:46:24:
a0:76:ce:dc:19:0d:8b:e3:75:5d:20:f8:aa:80:3f:
e8:4e:72:30:1b:44:46:ef:e6:f7:81:1b:53:d8:18:
e8:fa:fd:d3:58:bd:ad:eb:cb:d5:ec:35:f5:8f:b2:
f2:b2:7e:82:22:6e:40:63:58:0d:a1:a0:23:c4:44:
fc:32:77:0a:d7:0b:f6:c6:6d
publicExponent: 65537 (0x10001)

```

Screenshot 1: Evidence of work for ques.1

```

privateExponent: 65537 (0x10001)
privateExponent:
 63:7b:c0:5c:7b:81:e9:75:50:0c:05:41:a7:ac:4a:
e5:80:68:d2:3b:5e:71:ca:19:4b:68:3c:53:69:2d:
35:35:e0:a8:e4:8e:15:d7:4f:c4:b0:3f:83:3c:ef:
b9:22:86:7b:4a:98:8c:9d:b3:89:9a:7a:50:7f:76:
68:4c:a7:1f:ab:3b:48:6d:47:ce:77:44:ea:fd:c6:
08:61:33:0c:8c:8a:1d:4d:80:5f:0e:a5:71:69:e7:
b2:c1:98:f6:b5:de:3c:45:5c:40:fa:ee:92:73:c9:
23:05:ac:5b:01:d8:f5:ea:9f:1e:d8:25:61:79:1c:
d0:ec:bc:14:c3:f4:8f:01
prime1:
 00:db:75:6d:d8:40:35:2a:c9:96:b2:1a:63:e2:bc:
20:f4:83:16:cf:96:ac:ef:14:5a:ba:75:b5:c8:ee:
8e:0e:69:3d:21:96:9f:8c:9d:58:6d:5d:b0:ac:04:
54:8d:c3:0e:b9:c1:e8:80:35:00:8d:98:50:b0:0e:
98:e7:94:52:41
prime2:
 00:c8:e0:c1:29:eb:d3:b8:0e:6d:f2:99:58:0e:45:
2e:c0:18:ec:58:18:71:3e:7e:7e:61:7e:02:90:02:
01:a4:d4:2b:32:18:e7:a9:9f:f5:db:9f:cb:0a:8e:
4c:3f:d7:6e:1c:c0:22:50:11:1a:35:04:ad:c0:00:
25:c5:ee:11:2d
exponent1:

```

Screenshot 2: Evidence of work for ques.1

```

exponent1:
  63:7c:d2:b5:2c:36:6e:15:41:39:b0:79:03:ba:a1:
  7d:f7:47:8a:08:16:91:b3:ba:85:b9:03:15:f9:ff:
  70:4d:09:27:86:ef:01:81:fb:2a:69:44:c4:56:9a:
  36:41:ed:15:0c:f0:e7:de:1f:48:11:7a:d2:b5:6b:
  5f:e0:7f:81
exponent2:
  6c:ab:9a:14:11:ae:21:be:3c:a0:a7:70:49:98:07:
  a8:88:53:23:7d:65:96:07:5c:5b:65:8e:01:55:f4:
  89:b4:f6:01:4e:13:d5:61:e1:e1:84:5a:95:45:51:
  de:9f:ae:c4:02:f5:0d:17:93:73:e7:2a:0f:da:84:
  94:c5:2b:a9
coefficient:
  00:bb:0b:83:7a:78:a8:51:6e:c5:c3:31:ad:85:27:
  ab:24:3b:8f:ea:70:4f:50:f3:db:39:3d:09:e8:12:
  d0:90:80:51:c6:f8:2a:6f:70:b0:c1:c6:ea:78:57:
  36:bb:e5:18:21:02:90:29:a9:31:2a:e9:60:61:c0:
  61:0a:c7:4a:7e

```

Screenshot 3: Evidence of work for ques.1

```

[10/28/23]seed@VM:~/.../crypto$ openssl rsa -in public.pem -pubin -text -noout
RSA Public-Key: (1024 bit)
Modulus:
  00:ac:34:6a:28:87:2a:a3:65:1d:b2:f7:a9:29:1d:
  e5:93:0a:0e:80:4b:a0:43:90:b9:ae:db:a0:af:ba:
  df:b3:db:c5:46:2f:c5:37:b0:e6:83:ff:e4:a7:18:
  57:66:8d:f3:2d:c6:c1:63:a5:a3:ac:39:64:46:24:
  a0:76:ce:dc:19:0d:8b:e3:75:5d:20:f8:aa:80:3f:
  e8:4e:72:30:1b:44:46:ef:e6:f7:81:1b:53:d8:18:
  e8:fa:fd:d3:58:bd:ad:eb:cb:d5:ec:35:f5:8f:b2:
  f2:b2:7e:82:22:6e:40:63:58:0d:a1:a0:23:c4:44:
  fc:32:77:0a:d7:0b:f6:c6:6d
Exponent: 65537 (0x10001)

```

Screenshot 4: Evidence of work for ques.1

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):** ○ For the encryption, RsaKey is a public-key. Return an encryption object **Cipher**.
- **Cipher.encrypt(message):** ○ This returns ciphertext of message (byte string) under encryption object **Cipher**.

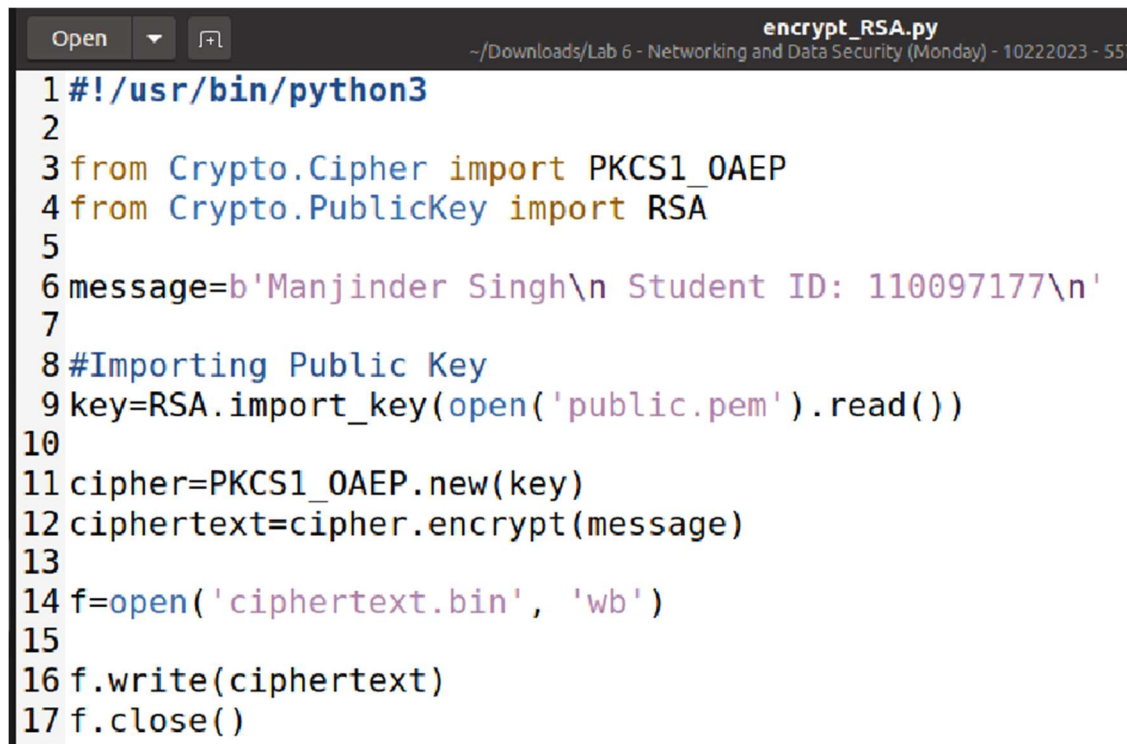
Encrypt message='your name and ID' and save ciphertext into a file. Take a screen shot for hexdump of your ciphertext (**\$hexdump -C filename**). Ref. **encrypt_RSA.py**.

My Implementation of above question 2(a) Part:-

Explanation of encrypt_RSA.py Program (Screenshot 5):-

1. Firstly, it loads a public key from the 'public.pem' file.
2. Then, it takes a text message ("Manjinder Singh\n Student ID: 110097177\n") and encrypts it using the RSA encryption algorithm with the PKCS1_OAEP padding scheme.
3. The encrypted message is written to a binary file named 'ciphertext.bin'.

To summarize, this script will encrypt a specific message using a public key, making the message unreadable to anyone who doesn't have the corresponding private key to decrypt it.

A screenshot of a code editor window titled 'encrypt_RSA.py'. The window shows a Python script with 17 lines of code. The code imports the PKCS1_OAEP cipher and RSA public key functionality from the Crypto module. It defines a message string, imports a public key from 'public.pem', creates a cipher object, encrypts the message, and writes the resulting ciphertext to a file named 'ciphertext.bin'.

```
1#!/usr/bin/python3
2
3from Crypto.Cipher import PKCS1_OAEP
4from Crypto.PublicKey import RSA
5
6message=b'Manjinder Singh\n Student ID: 110097177\n'
7
8#Importing Public Key
9key=RSA.import_key(open('public.pem').read())
10
11cipher=PKCS1_OAEP.new(key)
12ciphertext=cipher.encrypt(message)
13
14f=open('ciphertext.bin', 'wb')
15
16f.write(ciphertext)
17f.close()
```

Screenshot 5: Encrypt RSA Python Program

In Screenshot 6,

- ➔ **ls** command lists the files in the current directory.
- ➔ The '**encrypt_RSA.py**' Python script is executed to encrypt a message using an RSA public key and save it as 'ciphertext.bin.'
- ➔ '**hexdump ciphertext.bin**' is used to display the hexadecimal and ASCII representation of the encrypted content in 'ciphertext.bin' for examination.


```

[10/28/23]seed@VM:~/.../crypto$ ls
decrypt_RSA.py  endec_AES.py  private.pem  sign_RSA.py
encrypt_RSA.py  hash_comp.py  public.pem   verify_RSA.py
[10/28/23]seed@VM:~/.../crypto$ python3 encrypt_RSA.py
[10/28/23]seed@VM:~/.../crypto$ hexdump ciphertext.bin
00000000 7510 750b f381 fba3 d113 3706 3d7e 4bfa
00000010 cece 5a13 f9b6 c53e 71bf a260 1c3f 0b69
00000020 0d46 e16b 77b9 9542 07df 6388 62c8 fe87
00000030 d2dd 7e58 17f8 75bd e73b 3424 db6c b085
00000040 9e4e 1c72 4c2a 0d1b 832f e2fa aa6c b0a5
00000050 e37d 467e 34bf 5a46 d41a cdce a850 11cf
00000060 b3c5 ad2b 92b9 5ac2 fd96 d117 c355 0d24
00000070 9ba7 c01b d2aa 26bd b57b 0cf9 8278 9b84
00000080

```

Screenshot 6: Execution of ls, script encrypt_RSA.py, hexdump ciphertext.bin

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

- Cipher=**PKCS1_OAEP.new**(RsaKey): ○ For the decryption, RsaKey is a private-key. Return an decryption object **Cipher**.
- Cipher.**decrypt**(ctxt): ○ This returns message=**'your name and ID'** under decryption object **Cipher**.

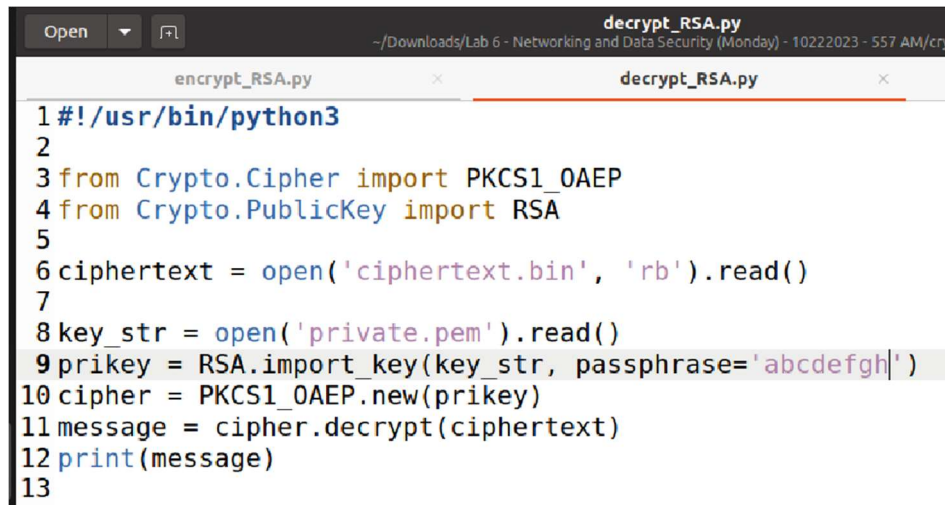
Take a screen shot for your decryption. Ref. **decrypt_RSA.py**.

My Implementation of above question 2(b) Part:-

Explanation of decrypt_RSA.py Script(Screenshot 7):-

1. First, it begins with reading an encrypted message from the '**ciphertext.bin**' file.
2. Then, it loads a private key from the '**private.pem**' file, using a passphrase '**abcdefgh**' for decryption.
3. Lastly, it uses the private key to decrypt the encrypted message and prints the original message to the console.

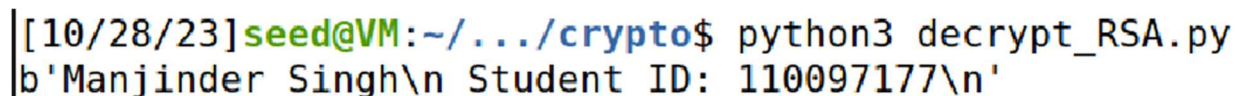
To summarize, **decrypt_RSA.py** script decrypts a previously encrypted message using a private key and displays the original, readable message. The passphrase 'abcdefgh' is used to unlock the private key for decryption.

A screenshot of a code editor window titled 'decrypt_RSA.py'. The editor shows a Python script with 13 lines of code. The code imports PKCS1_OAEP and RSA from the Crypto module, reads a ciphertext from 'ciphertext.bin', reads a private key from 'private.pem' using a passphrase 'abcdefgh', and then decrypts the ciphertext to print the message.

```
1#!/usr/bin/python3
2
3from Crypto.Cipher import PKCS1_OAEP
4from Crypto.PublicKey import RSA
5
6ciphertext = open('ciphertext.bin', 'rb').read()
7
8key_str = open('private.pem').read()
9prikey = RSA.import_key(key_str, passphrase='abcdefgh|')
10cipher = PKCS1_OAEP.new(prikey)
11message = cipher.decrypt(ciphertext)
12print(message)
13
```

Screenshot 7: Script decrypt_RSA.py

When the **decrypt_RSA.py** script is executed as per Screenshot 8, it decrypts the encrypted message and printed on the console.

A screenshot of a terminal window showing the command 'python3 decrypt_RSA.py' being executed. The output is 'b'Manjinder Singh\n Student ID: 110097177\n'', which is displayed as 'Manjinder Singh\n Student ID: 110097177\n' on the terminal.

```
[10/28/23]seed@VM:~/.../crypto$ python3 decrypt_RSA.py
b'Manjinder Singh\n Student ID: 110097177\n'
```

Screenshot 8: Executing 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):** ○ This defines a signing object *signer* with RsaKey (imported from your RSA private key file).
- **Signer.sign(hashmessage):** ○ 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**

My Implementation of above question 3(a) Part:-

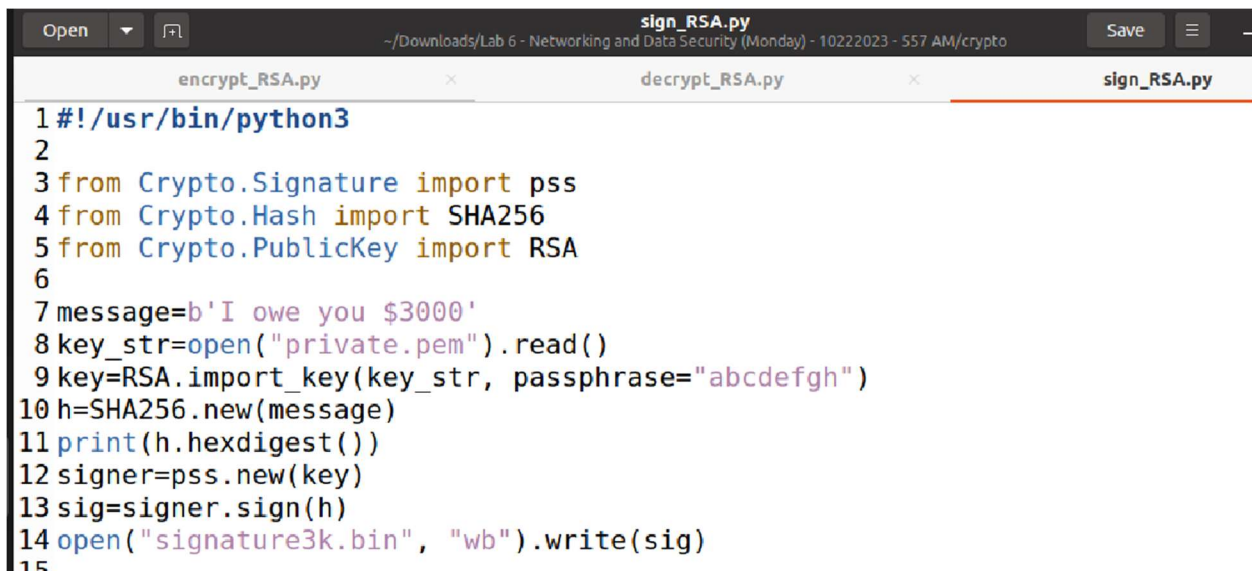
Explanation of sign_RSA.py Script(Screenshot 9):-

1. First, we created a digital signature for a message, in this case, "I owe you \$3000."
2. Then, we load the private key from the '**private.pem**' file, using the passphrase '**abcdefgh**' for security.

3. We calculated the SHA-256 hash of the message and prints the hexadecimal digest.
4. Lastly, we apply a digital signature to the hash of the message using the loaded private key and saves the signature in a file named 'signature3k.bin.'

To summarize, the script signs the message to prove its authenticity and integrity using a private key, and the resulting signature is stored for verification purposes. The SHA-256 hash provides a unique representation of the message.

In Screenshot 10, the execution of **sign_RSA.py** is displayed and then with the "**hexdump signature3k.bin**" command displays the hexadecimal representation of the contents of the "signature3k.bin" file, making it human-readable for examination.

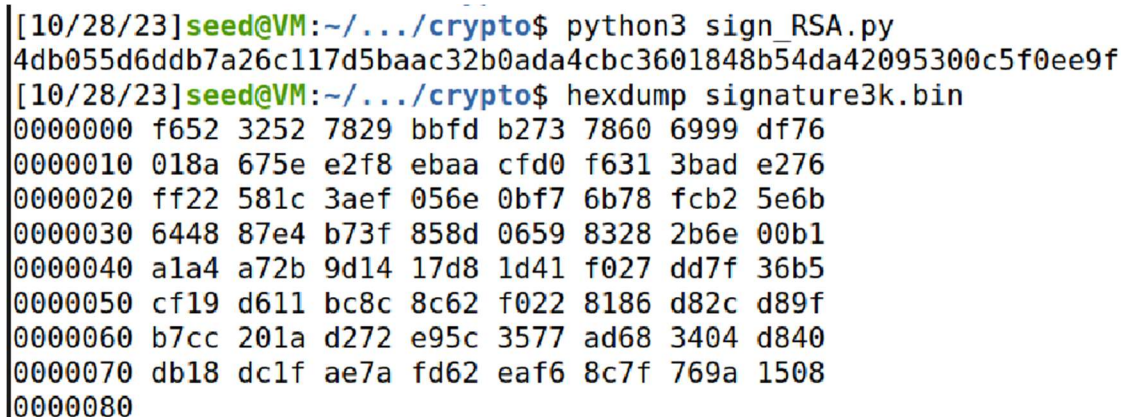


```

1#!/usr/bin/python3
2
3from Crypto.Signature import pss
4from Crypto.Hash import SHA256
5from Crypto.PublicKey import RSA
6
7message=b'I owe you $3000'
8key_str=open("private.pem").read()
9key=RSA.import_key(key_str, passphrase="abcdefgh")
10h=SHA256.new(message)
11print(h.hexdigest())
12signer=pss.new(key)
13sig=signer.sign(h)
14open("signature3k.bin", "wb").write(sig)
15

```

Screenshot 9: sign_RSA script



```

[10/28/23]seed@VM:~/.../crypto$ python3 sign_RSA.py
4db055d6ddb7a26c117d5baac32b0ada4cbc3601848b54da42095300c5f0ee9f
[10/28/23]seed@VM:~/.../crypto$ hexdump signature3k.bin
00000000 f652 3252 7829 bbfd b273 7860 6999 df76
00000100 018a 675e e2f8 ebba cfd0 f631 3bad e276
00000200 ff22 581c 3aef 056e 0bf7 6b78 fcb2 5e6b
00000300 6448 87e4 b73f 858d 0659 8328 2b6e 00b1
00000400 a1a4 a72b 9d14 17d8 1d41 f027 dd7f 36b5
00000500 cf19 d611 bc8c 8c62 f022 8186 d82c d89f
00000600 b7cc 201a d272 e95c 3577 ad68 3404 d840
00000700 db18 dc1f ae7a fd62 eaf6 8c7f 769a 1508
00000800

```

Screenshot 10: Executing sign_RSA script and hexadecimal representation of signature3k.bin

Explanation of sign_RSA_2k.py Script(Screenshot 11):-

1. First, we created a digital signature for a message, in this case, "I owe you \$2000."
2. Then, we load the private key from the '**private.pem**' file, using the passphrase '**abcdefgh**' for security.
3. We calculated the SHA-256 hash of the message and prints the hexadecimal digest.
4. Lastly, we apply a digital signature to the hash of the message using the loaded private key and saves the signature in a file named '**signature2k.bin**'.

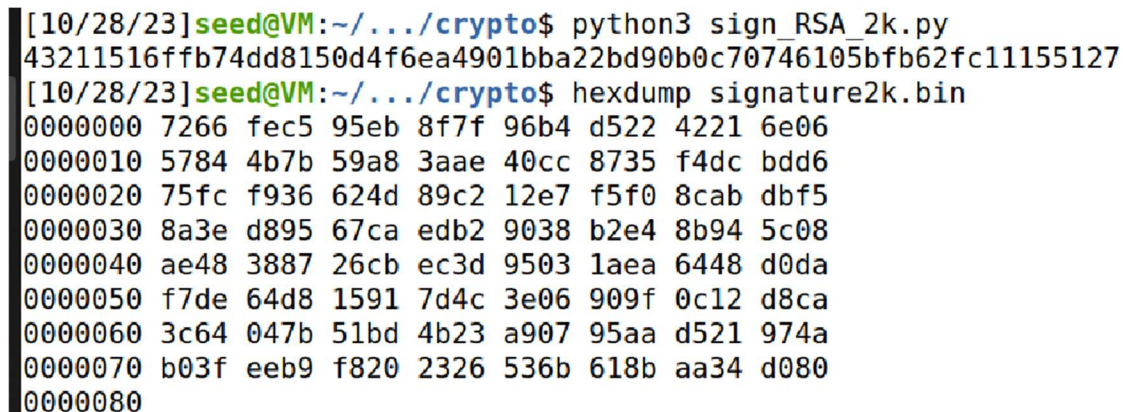
To summarize, the script signs the message to prove its authenticity and integrity using a private key, and the resulting signature is stored for verification purposes. The SHA-256 hash provides a unique representation of the message.

In Screenshot 12, the execution of **sign_RSA.py** is displayed and then with the "**hexdump signature2k.bin**" command displays the hexadecimal representation of the contents of the "signature3k.bin" file, making it human-readable for examination.



```
1 #!/usr/bin/python3
2
3 from Crypto.Signature import pss
4 from Crypto.Hash import SHA256
5 from Crypto.PublicKey import RSA
6
7 message=b'I owe you $2000'
8 key_str=open("private.pem").read()
9 key=RSA.import_key(key_str, passphrase="abcdefgh")
10 h=SHA256.new(message)
11 print(h.hexdigest())
12 signer=pss.new(key)
13 sig=signer.sign(h)
14 open("signature2k.bin", "wb").write(sig)
15
```

Screenshot 11: sign_RSA_2k.py script



```
[10/28/23]seed@VM:~/.../crypto$ python3 sign_RSA_2k.py
43211516ffb74dd8150d4f6ea4901bba22bd90b0c70746105bfb62fc11155127
[10/28/23]seed@VM:~/.../crypto$ hexdump signature2k.bin
00000000 7266 fec5 95eb 8f7f 96b4 d522 4221 6e06
00000010 5784 4b7b 59a8 3aae 40cc 8735 f4dc bdd6
00000020 75fc f936 624d 89c2 12e7 f5f0 8cab dbf5
00000030 8a3e d895 67ca edb2 9038 b2e4 8b94 5c08
00000040 ae48 3887 26cb ec3d 9503 laea 6448 d0da
00000050 f7de 64d8 1591 7d4c 3e06 909f 0c12 d8ca
00000060 3c64 047b 51bd 4b23 a907 95aa d521 974a
00000070 b03f eeb9 f820 2326 536b 618b aa34 d080
00000080 _
```

Screenshot 12: Executing sign_RSA_2k script and hexadecimal representation of signature2k.bin

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

- `Signer=pss.new(RsaKey)`: ○ This defines a signing object *signer* with *RsaKey* (imported from your RSA public key file).
- `Signer.verify(hashmessage, signature)`: ○ This verifies if *signature* is consistent with the *hashed message*.

Take a screen shot for the output result. Ref. **verify_RSA.py**

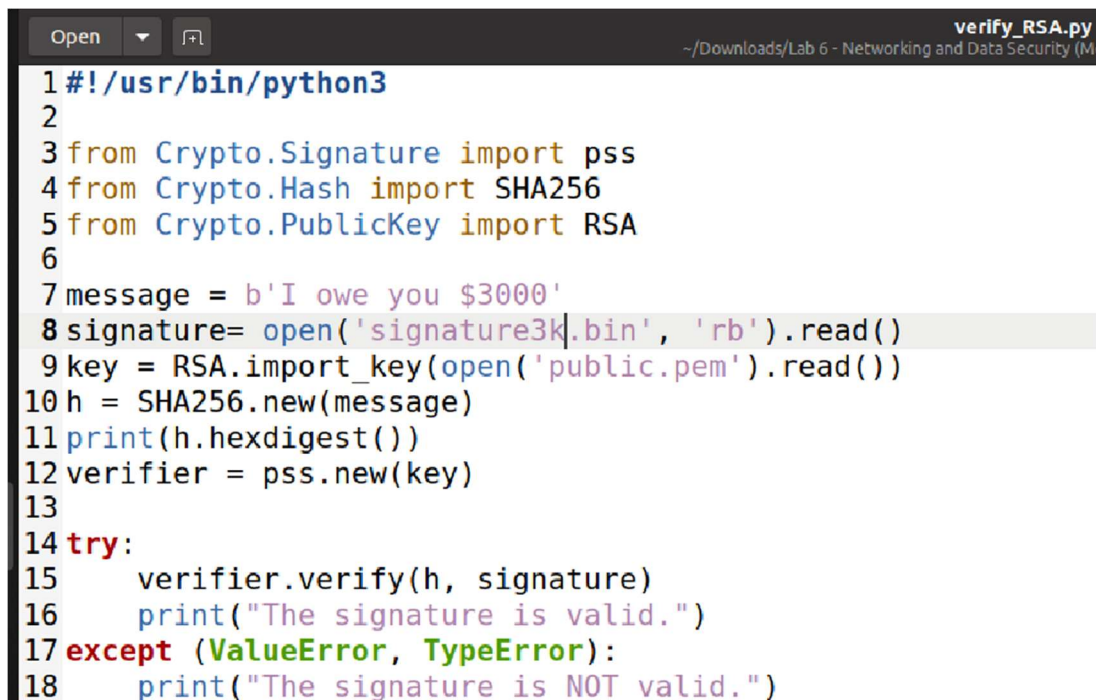
My Implementation of above question 3(b) Part:-

3.b.1 CONSIDERING VALID CASE:-

Explanation of verify_RSA.py Script(Screenshot 13):-

1. First, the script reads a message, "I owe you \$3000," and the associated digital signature from files.
2. Then, it loads a public key from 'public.pem.'
3. It calculates the SHA-256 hash of the message and prints its hexadecimal representation.
4. It uses the public key to verify the signature against the calculated hash.
5. Depending on the verification result, it prints either "The signature is valid" or "The signature is NOT valid."

To summarize, this script checks whether the provided digital signature matches the expected signature for the given message, using a public key. If the signature is valid, it confirms the authenticity and integrity of the message.



```
1#!/usr/bin/python3
2
3from Crypto.Signature import pss
4from Crypto.Hash import SHA256
5from Crypto.PublicKey import RSA
6
7message = b'I owe you $3000'
8signature= open('signature3k.bin', 'rb').read()
9key = RSA.import_key(open('public.pem').read())
10h = SHA256.new(message)
11print(h.hexdigest())
12verifier = pss.new(key)
13
14try:
15    verifier.verify(h, signature)
16    print("The signature is valid.")
17except (ValueError, TypeError):
18    print("The signature is NOT valid.")
```

Screenshot 13: verify_RSA.py script

In **screenshot 14**, we are executing the script **verify_RSA.py** to verify whether the provided digital signature matches the expected signature for the given message, using a public key. If the signature is valid, it confirms the authenticity and integrity of the message.

As we are referring to the respective digital signature of the message passed, then it will be valid case in this scenario, so it prints, "The signature is valid."

```
[10/28/23]seed@VM:~/.../crypto$ python3 verify_RSA.py  
4db055d6ddb7a26c117d5baac32b0ada4cbc3601848b54da42095300c5f0ee9f  
The signature is valid.
```

Screenshot 14: Executing verify_RSA.py script

3.b.2 CONSIDERING INVALID CASE:-

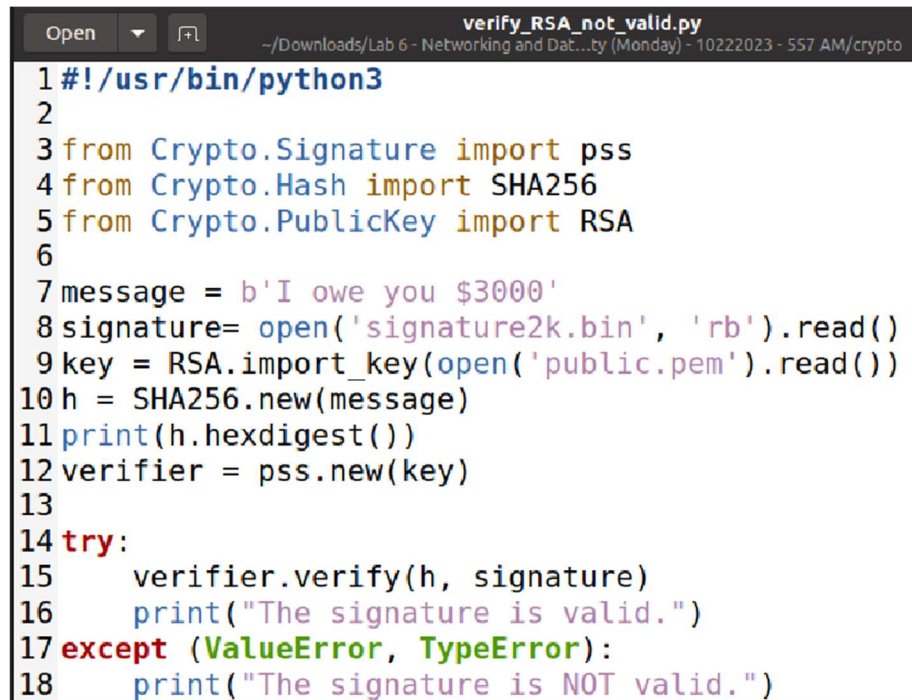
Explanation of verify_RSA_not_valid.py Script(Screenshot 15):-

1. First, the script reads a message, "I owe you \$3000," and the digital signature which is not associated with the message from files.
2. Then, it loads a public key from 'public.pem.'
3. It calculates the SHA-256 hash of the message and prints its hexadecimal representation.
4. It uses the public key to verify the signature against the calculated hash.
5. Depending on the verification result, it prints either "The signature is valid" or "The signature is NOT valid."

To summarize, this script checks whether the provided digital signature matches the expected signature for the given message, using a public key. If the signature is valid, it confirms the authenticity and integrity of the message, otherwise it will print the appropriate message.

In **screenshot 16**, we are executing the script **verify_RSA_not_valid.py** to verify whether the provided digital signature matches the expected signature for the given message, using a public key. If the signature is valid, it confirms the authenticity and integrity of the message, otherwise it will print the appropriate message.

As we are referring to the digital signature which is not related to the passed message, so it will be invalid case in this scenario, so it prints, "The signature is NOT valid."

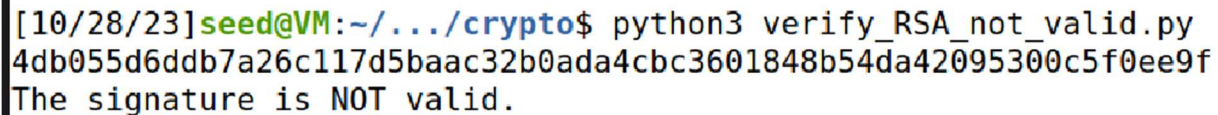


```

1#!/usr/bin/python3
2
3from Crypto.Signature import pss
4from Crypto.Hash import SHA256
5from Crypto.PublicKey import RSA
6
7message = b'I owe you $3000'
8signature= open('signature2k.bin', 'rb').read()
9key = RSA.import_key(open('public.pem').read())
10h = SHA256.new(message)
11print(h.hexdigest())
12verifier = pss.new(key)
13
14try:
15    verifier.verify(h, signature)
16    print("The signature is valid.")
17except (ValueError, TypeError):
18    print("The signature is NOT valid.")

```

Screenshot 15: verify_RSA_not_valid.py script



```

[10/28/23]seed@VM:~/.../crypto$ python3 verify_RSA_not_valid.py
4db055d6ddb7a26c117d5baac32b0ada4cbc3601848b54da42095300c5f0ee9f
The signature is NOT valid.

```

Screenshot 16: Executing verify_RSA_not_valid.py script

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

My Implementation of above question 4(a) Part:-

The python scripts **TCP_server.py** and **TCP_client.py**, implements a basic secure communication protocol using the **Diffie-Hellman key exchange** and **symmetric key encryption**.

Working of Script:

- ➔ Both the server and client generate a random private key (y for the server, and x for the client).
- ➔ They each calculate a public key based on their private key using the Diffie-Hellman formula ($g^{\text{private_key}} \bmod p$) and exchange these public keys.
- ➔ The shared secret key is then computed by both the server and client using the other party's public key and their own private key.
- ➔ The shared secret key is then hashed using SHA-256 to produce a symmetric key (sk) for encrypting and decrypting the communication.

- ➔ After establishing the symmetric key, both the client and server can send and receive messages over the network while encrypting and decrypting the messages with the shared symmetric key (sk).

To summarize, this script demonstrates a simple secure communication protocol that uses public and private keys to establish a shared secret key for encrypting and decrypting messages, providing confidentiality for the exchanged data. It is a simplified example and does not address all security aspects, but it showcases the fundamental principles of secure communication as per reference to **Screenshot 17**.

TCP_server.py

```
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 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'))
```


TCP_client.py

```
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'))
```

4 (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.

My Implementation of above question 4(b) Part:-

To summarize, this script demonstrates a simple secure communication protocol that uses public and private keys to establish a shared secret key for encrypting and decrypting messages, providing confidentiality for the exchanged data. It is a simplified example and does not address all security aspects, but it showcases the fundamental principles of secure communication as per reference to **Screenshot 17**.

In our case for 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 the below parameters:

$p=2582249878086908589655919172003011874329705792829223512830659356540647622016841194629645353280137831435903171972747559779$

$g=2$

```
[10/28/23]seed@VM:~/.../crypto$ python3 tcp_server.py
Server is listening...
Connected by ('127.0.0.1', 51488)
Secret Key: b'<\xea7\x1e\x8b\xe69\x1e\x1e\x00\xfa*\xa3s\x13\x83\x0eY\xb2w\xcc\x9\x13\xd3q\xe7\xcc\xae_u '
Client: I am Manjinder Singh - 110097177
Server: How are you, Manjinder?
Client: All good.
Server:
```

Screenshot 17: Client Server Communication using Diffie-Hellman.

4 (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 $\text{tag}=\text{sha256}(C)$. In (a), sender sends (C, tag), instead of plain chat message.

My Implementation of above question 4(c) Part:-

The client-server communication program(clientQ4PartC.py, serverQ4PartC.py) establishes a secure communication channel using the Diffie-Hellman key exchange and AES encryption

Working of Scripts(clientQ4PartC.py, serverQ4PartC.py):-

Client (clientQ4PartC.py):

1. First, the client connects to the server at the specified IP address and port.
2. Then, it generates a random private key and calculates a public key using the Diffie-Hellman algorithm.
3. The client sends its public key to the server.
4. Upon receiving the server's public key, it computes a shared key for AES encryption and derives a secret key using SHA-256.
5. The client can now securely exchange encrypted messages with the server.

Server (serverQ4PartC.py):

1. The server listens on a specific IP address and port for incoming connections.
2. It accepts a connection from a client.
3. Like the client, it generates a private key and computes a public key for Diffie-Hellman.

4. The server sends its public key to the client.
5. Upon receiving the client's public key, it calculates the shared key for AES encryption and derives a secret key using SHA-256.
6. The server can now securely exchange encrypted messages with the client.

Well, both the client and server use AES encryption with a shared secret key to encrypt and decrypt messages. They mainly include a SHA-256 tag to verify the integrity of the exchanged messages, ensuring that they haven't been tampered with during transmission. This ensures the confidentiality and integrity of the communication between the client and server.

clientQ4PartC.py

```
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

#Using below Diffie-Hellman parameters
p = 25822498780869085896559191720030118743297057928292235128306593565406476220168
41194629645353280137831435903171972747559779
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()

```

serverQ4PartC.py

```

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

#Using below Diffie-Hellman parameters
p = 25822498780869085896559191720030118743297057928292235128306593565406476220168
41194629645353280137831435903171972747559779
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()

```

4 (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.

My Implementation of above question 4(d) Part:-

It is observed at the Receiver (Server):

- The server receives the ciphertext (C) and tag from the client.
- It calculates the SHA-256 hash (tag) of the received ciphertext.
- The server then verifies whether the received tag matches the locally calculated tag (tag = SHA256(C)).

- If the verification fails (tags do not match), the server raises an exception.
- If the verification succeeds (tags match), the server uses the shared secret key (sk) as the AES secret to decrypt the ciphertext (C), recovering the chat message.
- The server then prints out the shared secret key (sk), ciphertext (C), tag, and the decrypted chat message for (one) chat message.

It ensures that the tag is used for integrity verification and that the shared secret key (sk) is used for decryption, successfully recovering the chat message.

```

seed@VM: ~/.../crypto seed@VM: ~/.../crypto
[10/28/23]seed@VM:~/.../crypto$ python3 serverQ4PartC.py
Server listening on 127.0.0.1:12349
Connected to client: ('127.0.0.1', 50880)
Key: 16169372563189496324481428929264142487167494039258845181394977427531354698360
Cipher Text (C): b'\x19\x86\xb2\xed\x08\xdc/L^\xc8PV\xda8F'\xf6\x81\xc4\xc6\xf3\xb3Q\xde\xb2\xbe0"
Tag: b' /\x9c\xb4\x01\xa6\xf0\xf0\xb1\x81H**R\xa6\xb3\xb5\x97~R\x19J\xb3\xabU\x8e\x1eP\x181\x14\x8cJ'
secret_key (sk): b'\x08\x84\xde\xa2d~I\xdb\xddt\t\xdb\xca\xe4\xb2\xbe\x96\x885CTT\xeb\xfb\x95\xe9=\xc28\x82a'
Decrypted Message: b'Manjinder Singh - 110097177'
Client: Manjinder Singh - 110097177

[10/28/23]seed@VM:~/.../crypto$ python3 clientQ4PartC.py
Secret Key: 16169372563189496324481428929264142487167494039258845181394977427531354698360
You: Manjinder Singh - 110097177
Cipher Text (C): b'\x19\x86\xb2\xed\x08\xdc/L^\xc8PV\xda8F'\xf6\x81\xc4\xc6\xf3\xb3Q\xde\xb2\xbe0"
Tag: b' /\x9c\xb4\x01\xa6\xf0\xf0\xb1\x81H**R\xa6\xb3\xb5\x97~R\x19J\xb3\xabU\x8e\x1eP\x181\x14\x8cJ'
Message: b'\x00\x00\x00\x1b\x00\x00\x00 \x19\x86\xb2\xed\x08\xdc/L^\xc8PV\xda8F'\xf6\x81\xc4\xc6\xf3\xb3Q\xde\xb2\xbe0/\x9c\xb4\x01\xa6\xf0\xf0\xb1\x81H**R\xa6\xb3\xb5\x97~R\x19J\xb3\xabU\x8e\x1eP\x181\x14\x8cJ\xa1\x0f\xfa\xf8\xf6\x11\xb6\xcd\x13X\x13\xd8\xca\xe9(\xa2"

```

Screenshot 18: Client Server Communication as per requirements

References: -

1. Lab Manual for Lab 5 from Brightspace
2. Lecture Notes for Lab 5 from Brightspace
3. Programs for Lab 5 from Brightspace
4. Python Libraries

One Drive Link for Python Program, Lab 5 Solution(Word File and PDF Document) for Lab 5 Work:-

[Networking and Data Security - Lab 5 - Submitted to Doc](#)