A logo of a university of windsor

Description automatically generated with low confidence

|  |
| --- |
| **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):

1. openssl genrsa -aes128 -out private.pem 1024
2. openssl rsa -in private.pem -pubout >public.pem
3. openssl rsa -in private.pem -text -noout
4. 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

A screenshot of a computer

Description automatically generated

**Screenshot 1:** Evidence of work for ques.1

A screenshot of a computer code

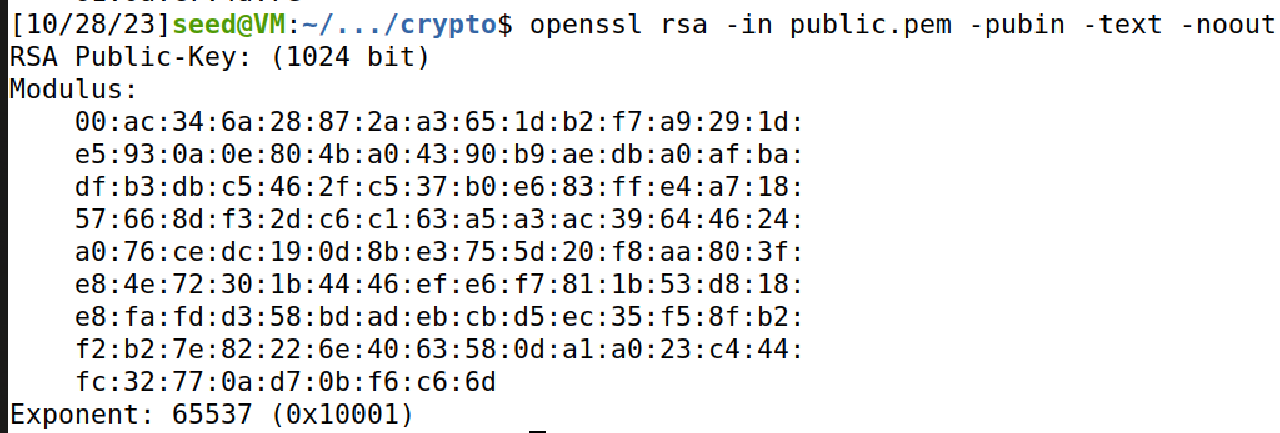
Description automatically generated

**Screenshot 2:** Evidence of work for ques.1

A screenshot of a computer code

Description automatically generated

**Screenshot 3:** Evidence of work for ques.1



**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): 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**. |

**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.

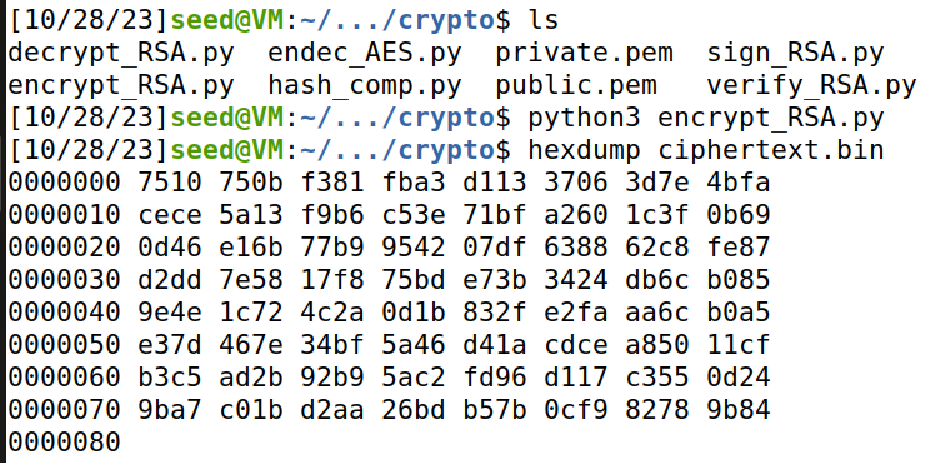
A screenshot of a computer

Description automatically generated

**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.



**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): 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**. |

**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.

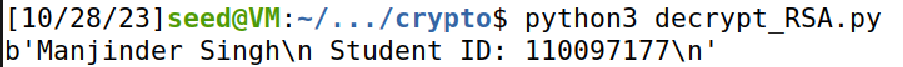
To summarize, **decrypt\_RSA.py**script decrypts a previously encrypted message using a private key and displays the original, readable message. The passphrase 'abcdefgh' is used to unlock the private key for decryption.

A screen shot of a computer

Description automatically generated

**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.



**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): 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** |

**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.

A screen shot of a computer

Description automatically generated

**Screenshot 9:** sign\_RSA script

A screenshot of a computer code

Description automatically generated

**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.

A screen shot of a computer code

Description automatically generated

**Screenshot 11:** sign\_RSA\_2k.py script

A screenshot of a computer code

Description automatically generated

**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): 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** |

**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.

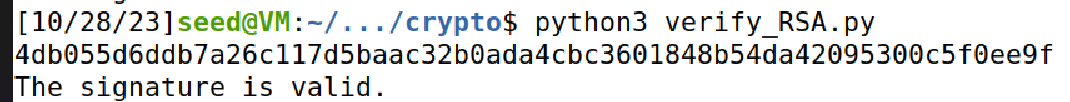
A screenshot of a computer program

Description automatically generated

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.”



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.

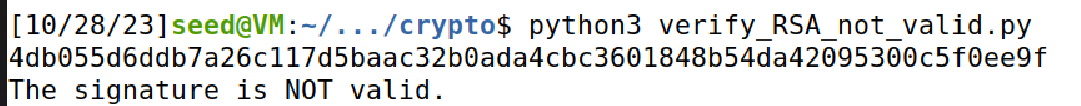
In **screenshot 16**, we are executing the script **verify\_RSA\_not\_valid.py** to verify whether the provided digital signature matches the expected signature for the given message, using a public key. If the signature is valid, it confirms the authenticity and integrity of the message, otherwise it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

A screenshot of a computer program

Description automatically generated

**Screenshot 15:** verify\_RSA\_not\_valid.py script



**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.  **a**. 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^private\_key mod 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

**A screen shot of a computer

Description automatically generated**

**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 tag=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.

**Top of Form**

|  |
| --- |
| **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.

**A screenshot of a computer

Description automatically generated**

**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](https://uwin365-my.sharepoint.com/:f:/g/personal/lnu8_uwindsor_ca/EuiNLkw0f3xEvJMj0GG62YcBi6_gL9zwlcZstSr8JNbYMw?email=lnu8%40uwindsor.ca&e=ITE6a4)