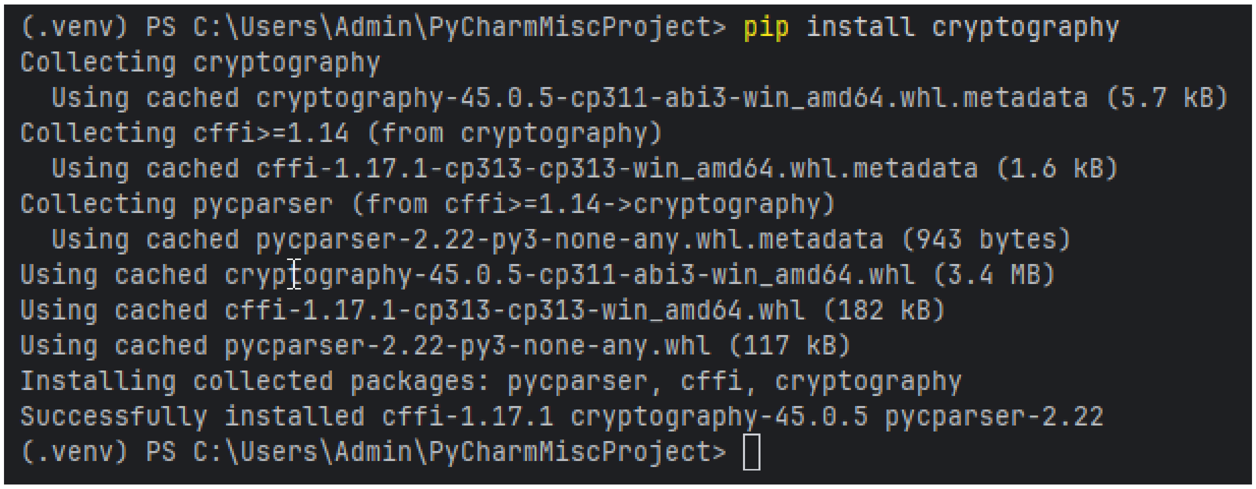
Final Exam – Applied Cryptography

Task 1: Encrypted Messaging App Prototype

User A generates an RSA key pair and shares the public key.

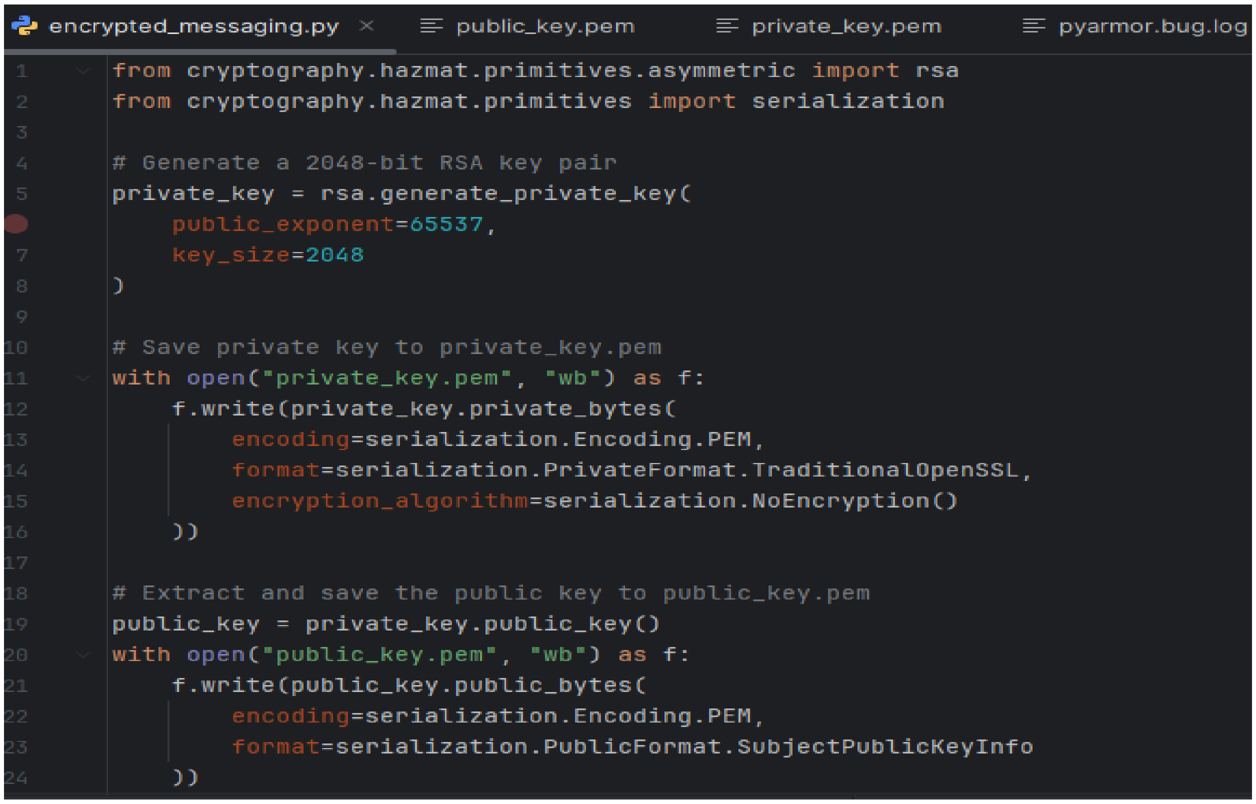
Goal: Implement a mini encrypted messaging system that supports both RSA and AES.

Step 1: Install the required library



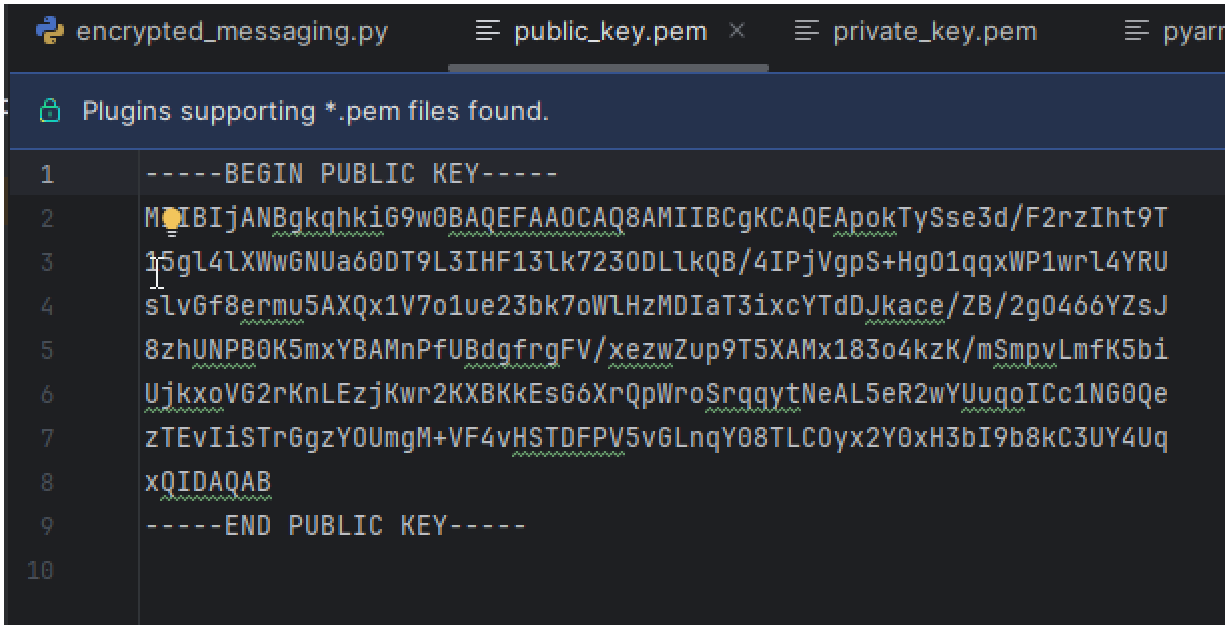
```
(.venv) PS C:\Users\Admin\PyCharmMiscProject> pip install cryptography
Collecting cryptography
  Using cached cryptography-45.0.5-cp311-abi3-win_amd64.whl.metadata (5.7 kB)
Collecting cffi>=1.14 (from cryptography)
  Using cached cffi-1.17.1-cp313-cp313-win_amd64.whl.metadata (1.6 kB)
Collecting pycparser (from cffi>=1.14->cryptography)
  Using cached pycparser-2.22-py3-none-any.whl.metadata (943 bytes)
Using cached cryptography-45.0.5-cp311-abi3-win_amd64.whl (3.4 MB)
Using cached cffi-1.17.1-cp313-cp313-win_amd64.whl (182 kB)
Using cached pycparser-2.22-py3-none-any.whl (117 kB)
Installing collected packages: pycparser, cffi, cryptography
Successfully installed cffi-1.17.1 cryptography-45.0.5 pycparser-2.22
(.venv) PS C:\Users\Admin\PyCharmMiscProject>
```

User A generates an RSA key pair and shares the public key.



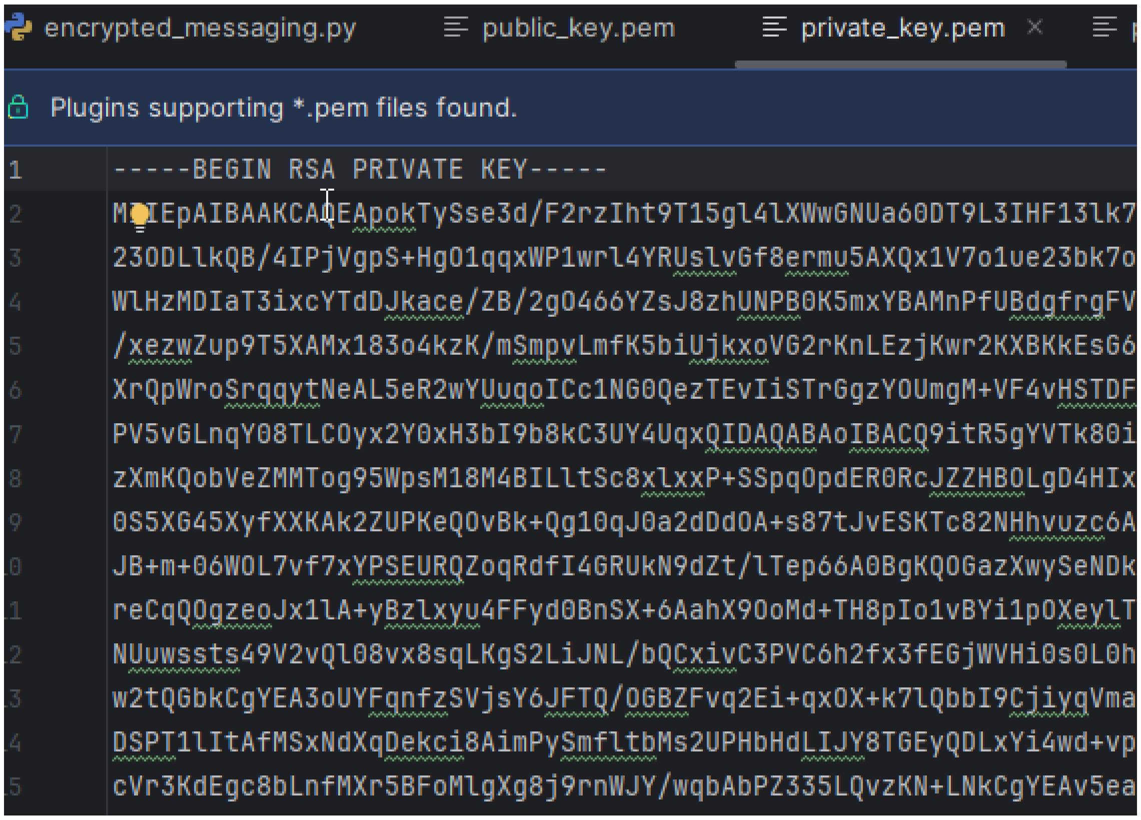
```python
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

# Generate a 2048-bit RSA key pair
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)

# Save private key to private_key.pem
with open("private_key.pem", "wb") as f:
    f.write(private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    ))

# Extract and save the public key to public_key.pem
public_key = private_key.public_key()
with open("public_key.pem", "wb") as f:
    f.write(public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ))
```

This screenshot shows public_key.pem is created successfully



```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApokTySse3d/F2rzIht9T
15gl4lXWwGNUa60DT9L3IHF13lk723ODLlkQB/4IPjVgpS+HgO1qqxWP1wrl4YRU
slvGf8ermu5AXQx1V7o1ue23bk7oWlHzMDIaT3ixcYTdDJkace/ZB/2gO466YZsJ
8zhUNPB0K5mxYBAMnPfUBdgfrgFV/xezwZup9T5XAMx183o4kzK/mSmpvLmfK5bi
UjkxoVG2rKnLEzjKwr2KXBKkEsG6XrQpWroSrqqytNeAL5eR2wYUuqoICc1NG0Qe
zTEvIiSTrGgzYOUmgM+VF4vHSTDFPV5vGLnqY08TLCOyx2Y0xH3bI9b8kC3UY4Uq
xQIDAQAB
-----END PUBLIC KEY-----
```

This screenshot shows that private_key.pem is created successfully

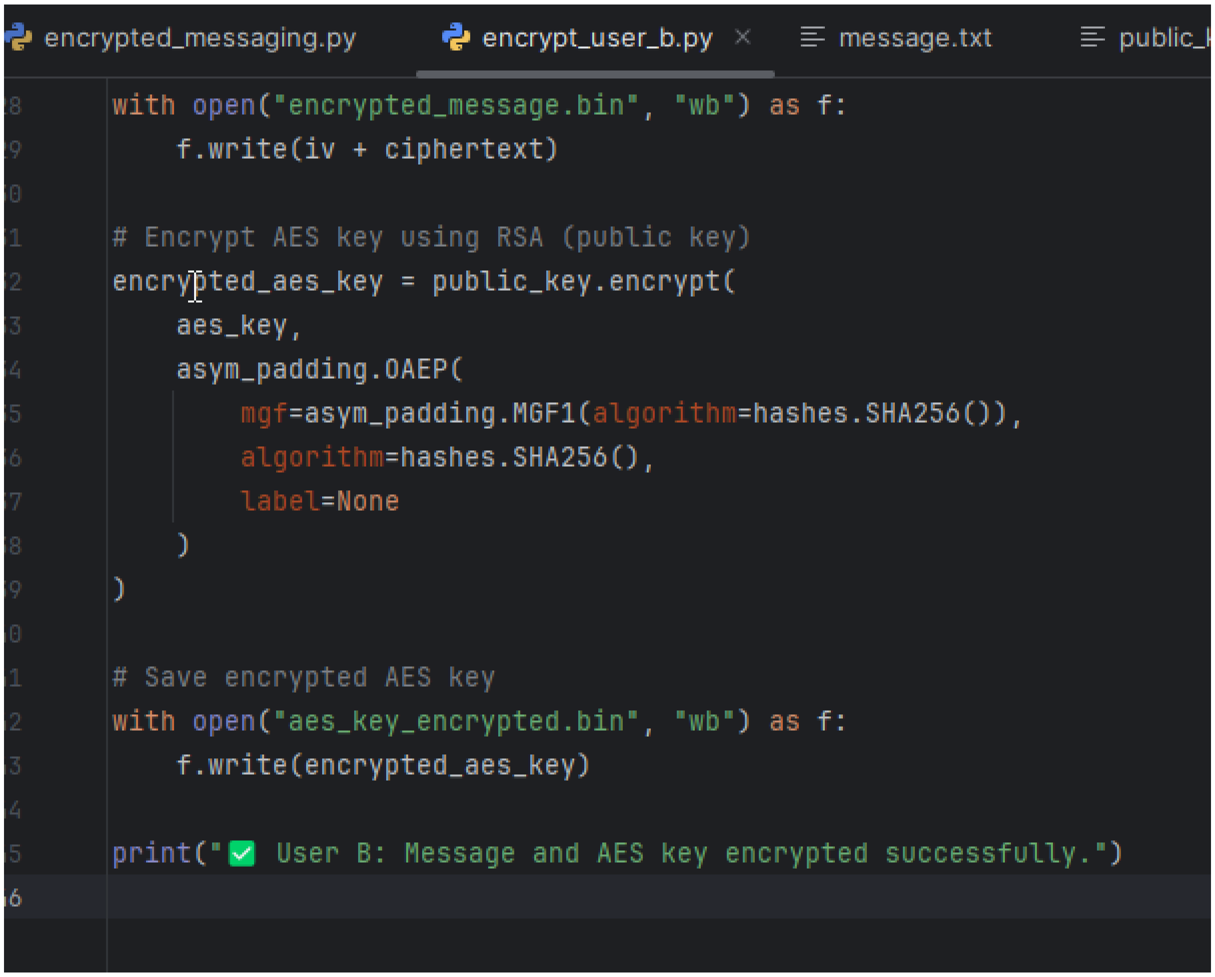


```
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEApokTySse3d/F2rzIht9T15gl4lXWwGNUa60DT9L3IHF13lk7
23ODLlkQB/4IPjVgpS+HgO1qqxWP1wrl4YRUslvGf8ermu5AXQx1V7o1ue23bk7o
WlHzMDIaT3ixcYTdDJkace/ZB/2gO466YZsJ8zhUNPB0K5mxYBAMnPfUBdgfrgFV
/xezwZup9T5XAMx183o4kzK/mSmpvLmfK5biUjkxoVG2rKnLEzjKwr2KXBKkEsG6
XrQpWroSrqqytNeAL5eR2wYUuqoICc1NG0QezTEvIiSTrGgzYOUmgM+VF4vHSTDF
PV5vGLnqY08TLCOyx2Y0xH3bI9b8kC3UY4UqxQIDAQABAoIBACQ9itR5gYVTk80i
zXmKQobVeZMMTog95WpsM18M4BILltSc8xlxxP+SSpqOpdER0RcJZZHBOLgD4HIx
0S5XG45XyfXXKAk2ZUPKeQOvBk+Qg10qJ0a2dDdOA+s87tJvESKTc82NHhvuzc6A
JB+m+06WOL7vf7xYPSEURQZoqRdfI4GRUkN9dZt/lTep66A0BgKQOGazXwySeNDk
reCqQOgzeoJx1lA+yBzlxyu4FFyd0BnSX+6AahX9OoMd+TH8pIo1vBYi1pOXeylT
NUuwssts49V2vQl08vx8sqLKgS2LiJNL/bQCxivC3PVC6h2fx3fEGjWVHi0s0L0h
w2tQGbkCgYEA3oUYFqnfzSVjsY6JFTQ/OGBZFvq2Ei+qxOX+k7lQbbI9CjiyqVma
DSPT1lItAfMSxNdXqDekci8AimPySmfltbMs2UPHbHdLIJY8TGEyQDLxYi4wd+vp
cVr3KdEgc8bLnfMXr5BFoMlgXg8j9rnWJY/wqbAbPZ335LQvzKN+LNkCgYEAv5ea
```

User B encrypts a secret message using:

○ AES-256 (with randomly generated key)

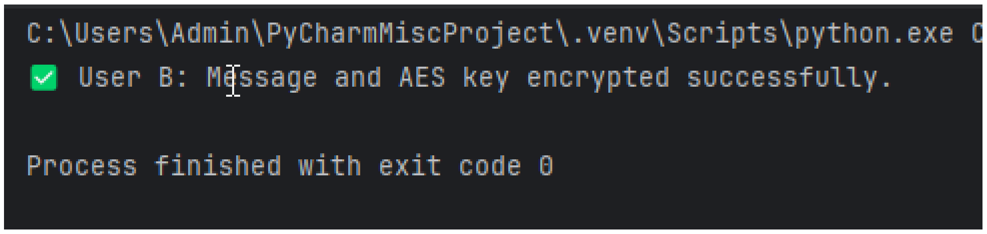
○ Encrypts AES key using RSA (User A’s public key)



encrypted_messaging.py | encrypt_user_b.py | message.txt | public_

```python
with open("encrypted_message.bin", "wb") as f:
    f.write(iv + ciphertext)

# Encrypt AES key using RSA (public key)
encrypted_aes_key = public_key.encrypt(
    aes_key,
    asym_padding.OAEP(
        mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Save encrypted AES key
with open("aes_key_encrypted.bin", "wb") as f:
    f.write(encrypted_aes_key)

print("✅ User B: Message and AES key encrypted successfully.")
```

Output shows that AES key encrypted successfully.

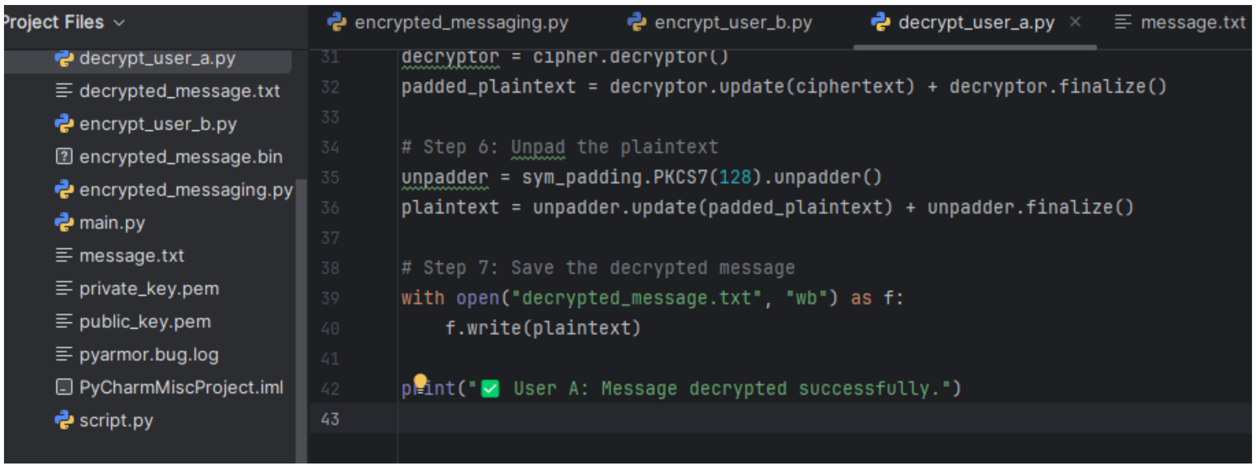


```
C:\Users\Admin\PyCharmMiscProject\.venv\Scripts\python.exe C
✅ User B: Message and AES key encrypted successfully.

Process finished with exit code 0
```

User A decrypts:

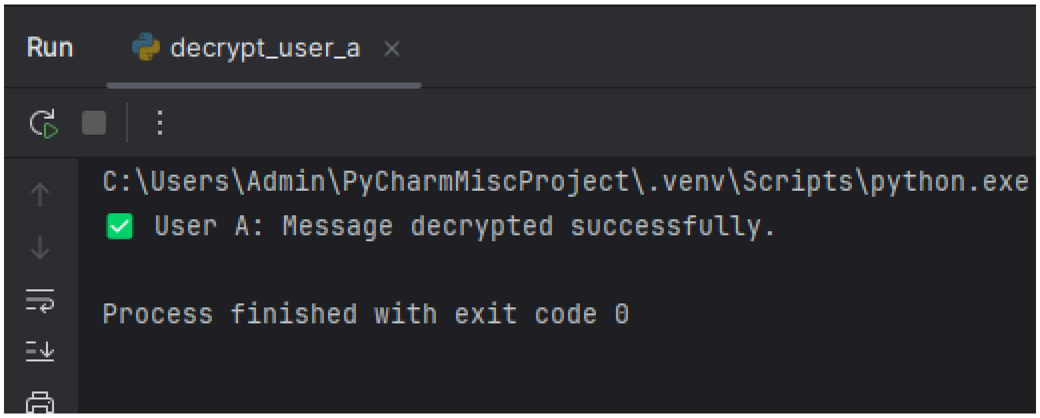
○ AES key using their private RSA key

○ Message using the decrypted AES key



```python
decryptor = cipher.decryptor()
padded_plaintext = decryptor.update(ciphertext) + decryptor.finalize()

# Step 6: Unpad the plaintext
unpadder = sym_padding.PKCS7(128).unpadder()
plaintext = unpadder.update(padded_plaintext) + unpadder.finalize()

# Step 7: Save the decrypted message
with open("decrypted_message.txt", "wb") as f:
    f.write(plaintext)

print("✅ User A: Message decrypted successfully.")
```

Output shows that message decrypted successfully

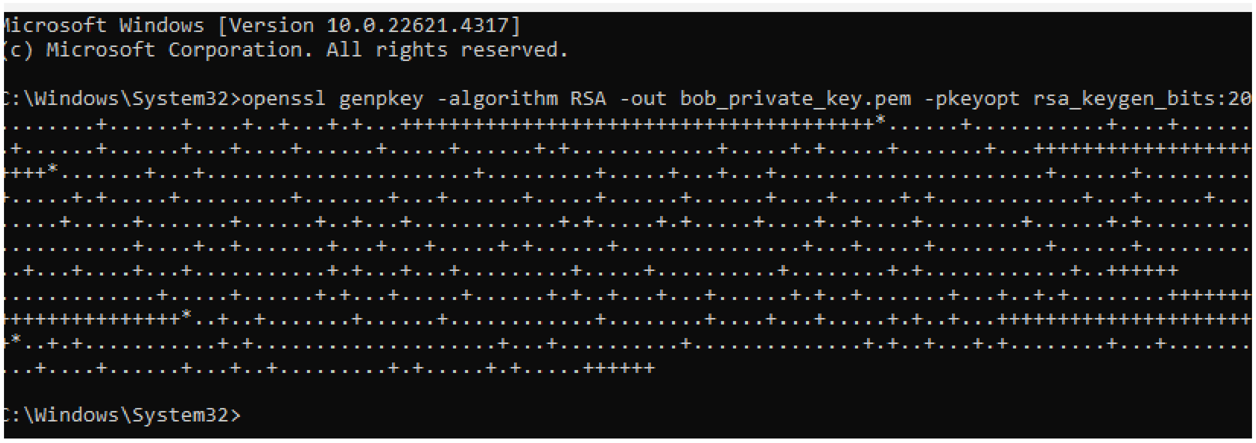


```
C:\Users\Admin\PyCharmMiscProject\.venv\Scripts\python.exe
✅ User A: Message decrypted successfully.

Process finished with exit code 0
```

Task 2: Secure File Exchange Using RSA + AES

Generate an RSA key pair for Bob (I used OpenSSL)

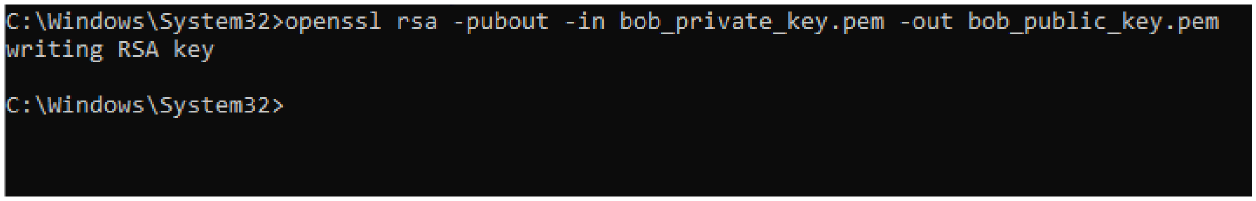
Step 1: Generate Private Key



```
licrosoft Windows [Version 10.0.22621.4317]
(c) Microsoft Corporation. All rights reserved.

:\Windows\System32>openssl genpkey -algorithm RSA -out bob_private_key.pem -pkeyopt rsa_keygen_bits:20
.......+.......+....+..+...+.+...+++++++++++++++++++++++++++++++++++++++*......+............+....+......
.+.......+.......+...+....+.......+.....+.......+.+...........+.....+.+.....+.......+...++++++++++++++++++
++++*.......+...+.....................+.........+.....+...+...+......................+......+........
+.....+.+.....+.........+.......+...+......+.....+......+.......+.....+.+............+...+.....+...
.....+.....+.......+......+..+...+...........+.+......+.+.....+....+..+...+......+.......+......+.+........
............+....+..+.......+...+...+.....+.+.........+.................+...+.....+.........+......+........
..+...+....+...+...........+.+...+...+...........+......+......+.........+.+............+..++++++
...............+.....+......+.+...+.....+........+.+..+...+...+.......+.+..+........+...+..+.+.........+++++++
+++++++++++++++*..+..+.......+......+...........+........+....+...+.....+.+..+...+++++++++++++++++++++
+*..+.+..........+.+...................+...+..........+.............+.+..+...+.+........+...+.......
...+....+......+...+..+.........+.+.....+.+.....++++++

:\Windows\System32>
```

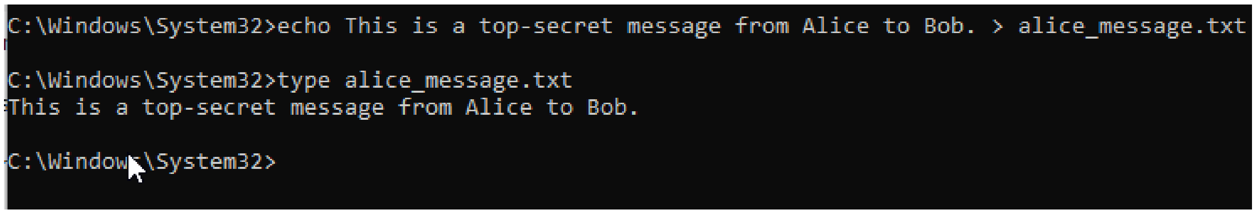
Step 2: Extract the Public Key from the Private Key

Output shows that successfully created Bob’s RSA public key



```
C:\Windows\System32>openssl rsa -pubout -in bob_private_key.pem -out bob_public_key.pem
writing RSA key

C:\Windows\System32>
```

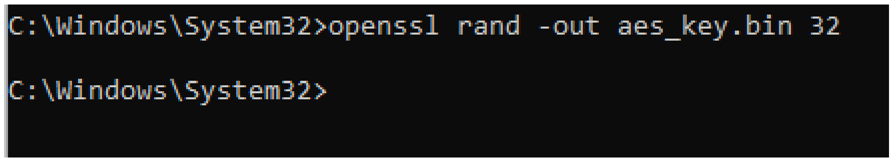
Alice creates a plaintext message in a file called alice_message.txt



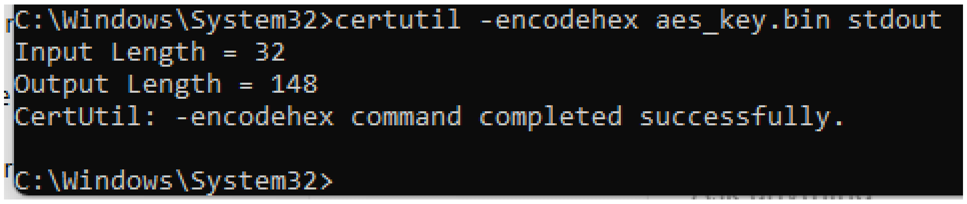
```
C:\Windows\System32>echo This is a top-secret message from Alice to Bob. > alice_message.txt

C:\Windows\System32>type alice_message.txt
This is a top-secret message from Alice to Bob.

C:\Windows\System32>
```

Alice generates a random AES-256 key and IV

Step 1: Generate a 256-bit AES key

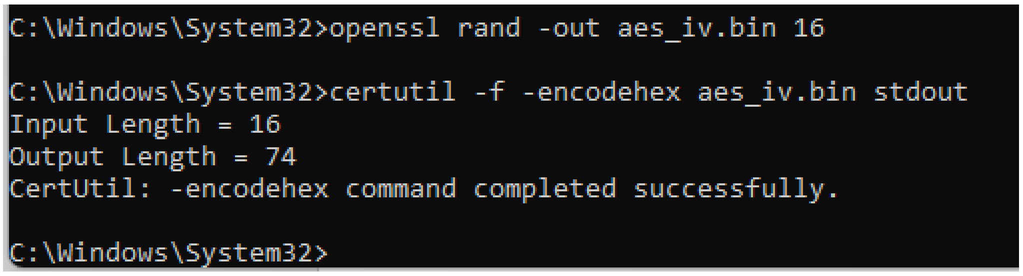


```
C:\Windows\System32>openssl rand -out aes_key.bin 32

C:\Windows\System32>
```



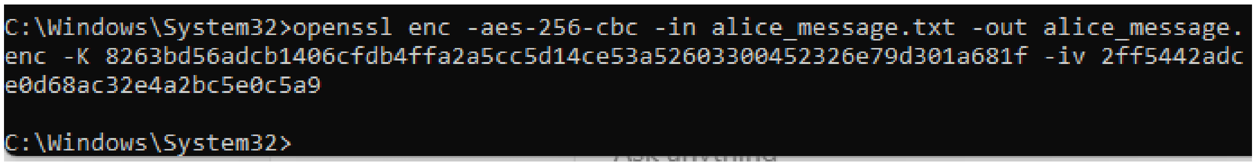
```
C:\Windows\System32>certutil -encodehex aes_key.bin stdout
Input Length = 32
Output Length = 148
CertUtil: -encodehex command completed successfully.

C:\Windows\System32>
```

Step 2: Generate a 128-bit IV



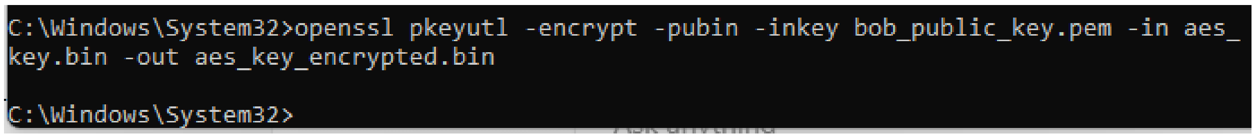
```
C:\Windows\System32>openssl rand -out aes_iv.bin 16

C:\Windows\System32>certutil -f -encodehex aes_iv.bin stdout
Input Length = 16
Output Length = 74
CertUtil: -encodehex command completed successfully.

C:\Windows\System32>
```

Encrypt the file (alice_message.txt) using AES-256 with the generated key and IV

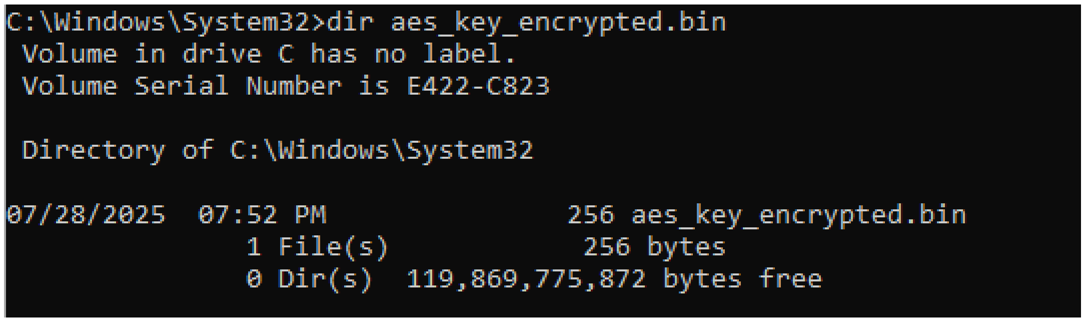


```
C:\Windows\System32>openssl enc -aes-256-cbc -in alice_message.txt -out alice_message.
enc -K 8263bd56adcb1406cfdb4ffa2a5cc5d14ce53a52603300452326e79d301a681f -iv 2ff5442adc
e0d68ac32e4a2bc5e0c5a9

C:\Windows\System32>
```

Encrypt the AES key using Bob’s RSA public key

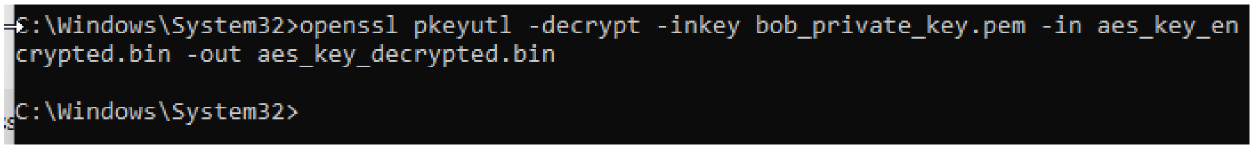


```
C:\Windows\System32>openssl pkeyutl -encrypt -pubin -inkey bob_public_key.pem -in aes_
key.bin -out aes_key_encrypted.bin

C:\Windows\System32>
```



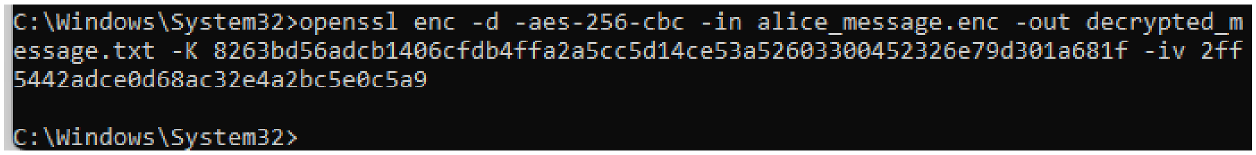
```
C:\Windows\System32>dir aes_key_encrypted.bin
 Volume in drive C has no label.
 Volume Serial Number is E422-C823

 Directory of C:\Windows\System32

07/28/2025  07:52 PM               256 aes_key_encrypted.bin
               1 File(s)            256 bytes
               0 Dir(s)  119,869,775,872 bytes free
```

Bob decrypts the AES key using his RSA private key

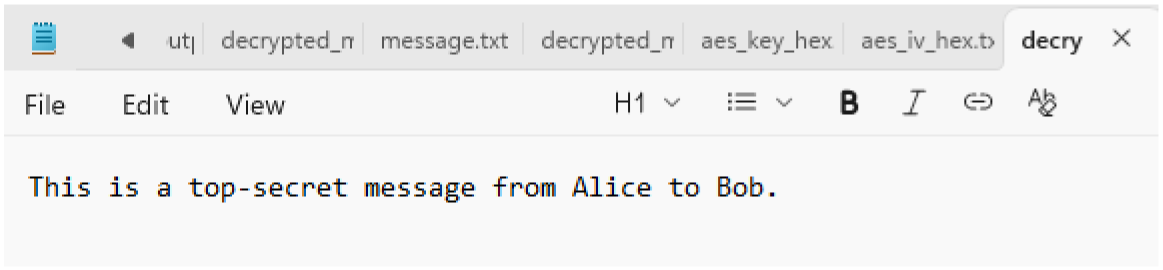


```
C:\Windows\System32>openssl pkeyutl -decrypt -inkey bob_private_key.pem -in aes_key_en
crypted.bin -out aes_key_decrypted.bin

C:\Windows\System32>
```

Bob uses the decrypted AES key and IV to decrypt the file and recover the original message



```
C:\Windows\System32>openssl enc -d -aes-256-cbc -in alice_message.enc -out decrypted_m
essage.txt -K 8263bd56adcb1406cfdb4ffa2a5cc5d14ce53a52603300452326e79d301a681f -iv 2ff
5442adce0d68ac32e4a2bc5e0c5a9

C:\Windows\System32>
```



```
This is a top-secret message from Alice to Bob.
```

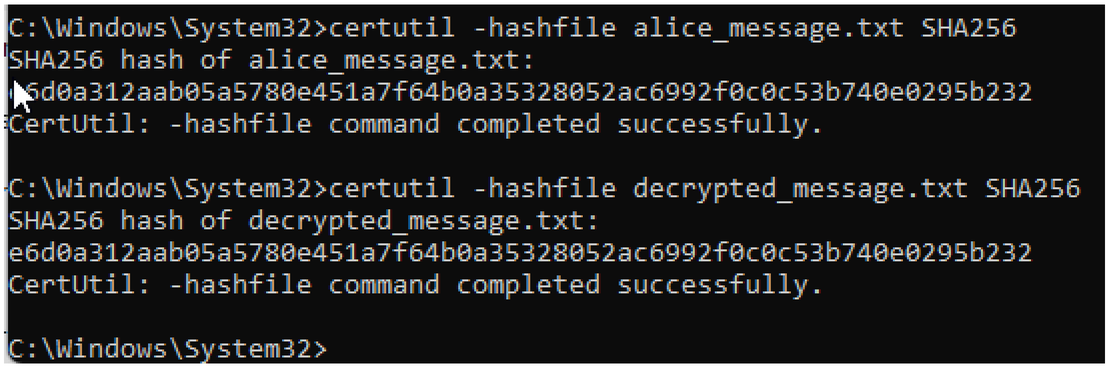
After decryption, compute the SHA-256 hash of the file and compare it to the original hash for integrity verification

Output confirms that:

The encrypted message was decrypted accurately

The message was not altered

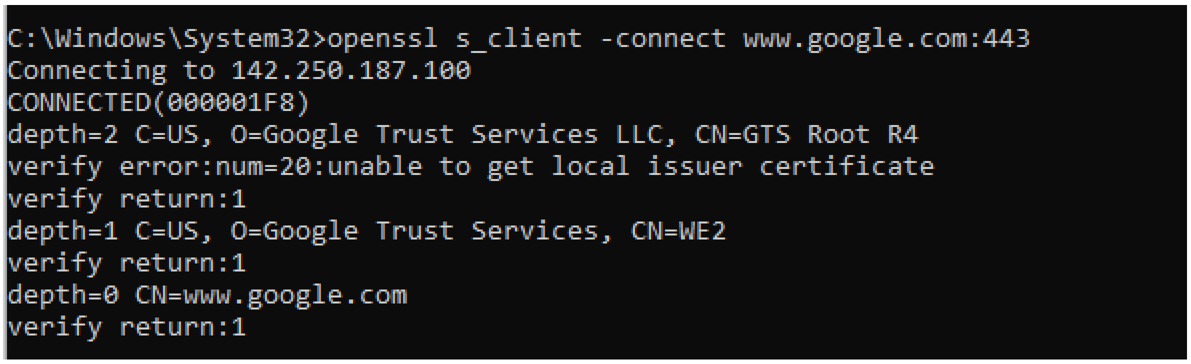
The hybrid encryption (RSA + AES) process was implemented correctly



```
C:\Windows\System32>certutil -hashfile alice_message.txt SHA256
SHA256 hash of alice_message.txt:
e6d0a312aab05a5780e451a7f64b0a35328052ac6992f0c0c53b740e0295b232
CertUtil: -hashfile command completed successfully.

C:\Windows\System32>certutil -hashfile decrypted_message.txt SHA256
SHA256 hash of decrypted_message.txt:
e6d0a312aab05a5780e451a7f64b0a35328052ac6992f0c0c53b740e0295b232
CertUtil: -hashfile command completed successfully.

C:\Windows\System32>
```

Task 3: TLS Communication Inspection & Analysis

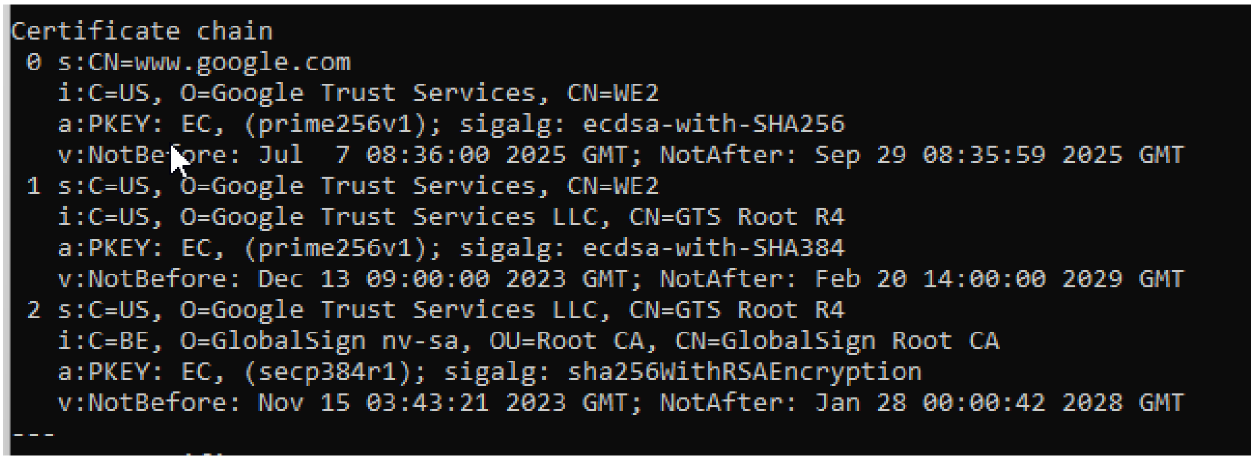
1. Connect to any HTTPS website using openssl s_client



```
C:\Windows\System32>openssl s_client -connect www.google.com:443
Connecting to 142.250.187.100
CONNECTED(000001F8)
depth=2 C=US, O=Google Trust Services LLC, CN=GTS Root R4
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=1 C=US, O=Google Trust Services, CN=WE2
verify return:1
depth=0 CN=www.google.com
verify return:1
```

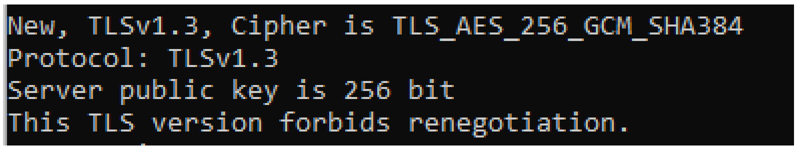
2. Extract and document:

○ Certificate chain (Root → Intermediate → Leaf)



```
Certificate chain
 0 s:CN=www.google.com
   i:C=US, O=Google Trust Services, CN=WE2
   a:PKEY: EC, (prime256v1); sigalg: ecdsa-with-SHA256
   v:NotBefore: Jul  7 08:36:00 2025 GMT; NotAfter: Sep 29 08:35:59 2025 GMT
 1 s:C=US, O=Google Trust Services, CN=WE2
   i:C=US, O=Google Trust Services LLC, CN=GTS Root R4
   a:PKEY: EC, (prime256v1); sigalg: ecdsa-with-SHA384
   v:NotBefore: Dec 13 09:00:00 2023 GMT; NotAfter: Feb 20 14:00:00 2029 GMT
 2 s:C=US, O=Google Trust Services LLC, CN=GTS Root R4
   i:C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA
   a:PKEY: EC, (secp384r1); sigalg: sha256WithRSAEncryption
   v:NotBefore: Nov 15 03:43:21 2023 GMT; NotAfter: Jan 28 00:00:42 2028 GMT
---
```

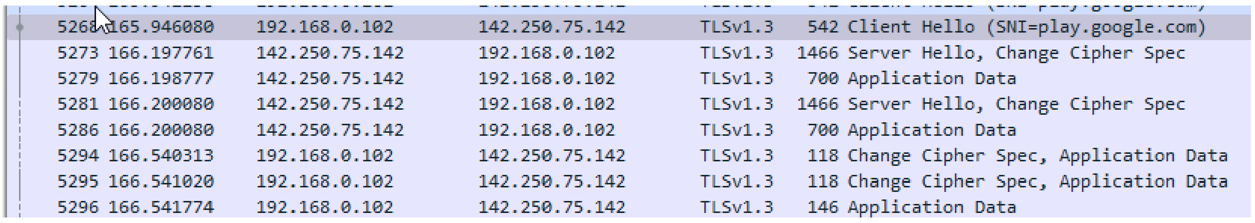
Cipher suite used and TLS version



```
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Protocol: TLSv1.3
Server public key is 256 bit
This TLS version forbids renegotiation.
```

3. Capture a TLS handshake using Wireshark and highlight:

○ Client Hello, Server Certificate, Key Exchange



```
5268 165.946080   192.168.0.102    142.250.75.142   TLSv1.3    542 Client Hello (SNI=play.google.com)
5273 166.197761   142.250.75.142   192.168.0.102    TLSv1.3   1466 Server Hello, Change Cipher Spec
5279 166.198777   142.250.75.142   192.168.0.102    TLSv1.3    700 Application Data
5281 166.200080   142.250.75.142   192.168.0.102    TLSv1.3   1466 Server Hello, Change Cipher Spec
5286 166.200080   142.250.75.142   192.168.0.102    TLSv1.3    700 Application Data
5294 166.540313   192.168.0.102    142.250.75.142   TLSv1.3    118 Change Cipher Spec, Application Data
5295 166.541020   192.168.0.102    142.250.75.142   TLSv1.3    118 Change Cipher Spec, Application Data
5296 166.541774   192.168.0.102    142.250.75.142   TLSv1.3    146 Application Data
```

How TLS Provides Confidentiality and Integrity

When I use TLS (Transport Layer Security), it acts like a secret code between my computer and the website I am visiting. TLS protects my data in two important ways:

Confidentiality:

TLS scrambles the messages I send so that only my computer and the website can read them. Even if someone tries to listen in, all they will see is a jumble of meaningless data. This happens because TLS encrypts the information using special keys that only my computer and the website share.

Integrity:

TLS also makes sure that the data does not get changed while it is traveling. It uses a kind of "digital fingerprint" called a hash. If someone tries to change the data, the fingerprint will not match, so my computer knows something is wrong and stops the connection.

Together, these features keep my online information private and trustworthy.

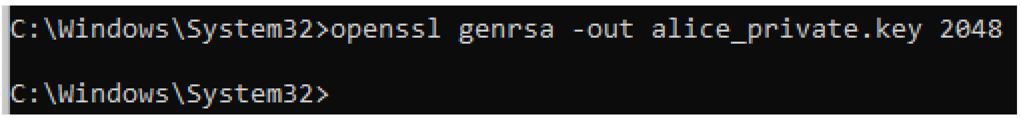
Task 4: Email Encryption and Signature Simulation (PGP or S/MIME)

Tasks:

1. Generate key pairs using GPG or OpenSSL for PGP/S/MIME

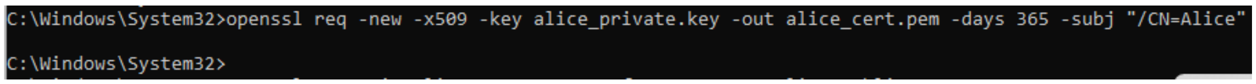
Step 1. Generate Key Pairs for Alice and Bob

Generate Alice's private key



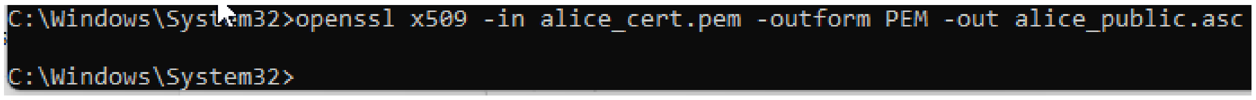
```
C:\Windows\System32>openssl genrsa -out alice_private.key 2048

C:\Windows\System32>
```

Generate Alice's public certificate



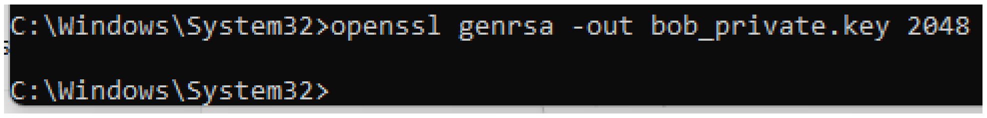
```
C:\Windows\System32>openssl req -new -x509 -key alice_private.key -out alice_cert.pem -days 365 -subj "/CN=Alice"

C:\Windows\System32>
```

Export Alice's public key in.asc format



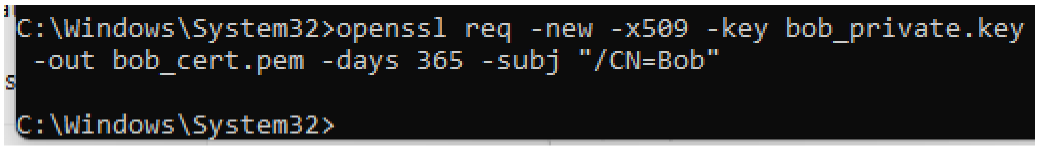
```
C:\Windows\System32>openssl x509 -in alice_cert.pem -outform PEM -out alice_public.asc

C:\Windows\System32>
```

Generate Bob's private key



```
C:\Windows\System32>openssl genrsa -out bob_private.key 2048

C:\Windows\System32>
```

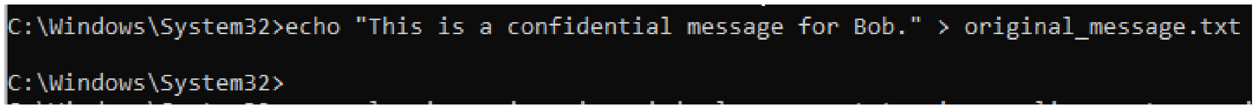
Generate Bob's public certificate



```
C:\Windows\System32>openssl req -new -x509 -key bob_private.key
 -out bob_cert.pem -days 365 -subj "/CN=Bob"

C:\Windows\System32>
```

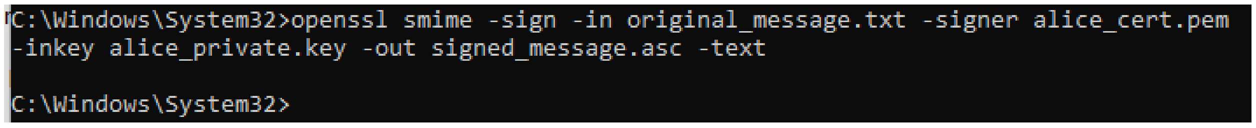
2. Sign and Encrypt a Text File as Alice

Create original message:

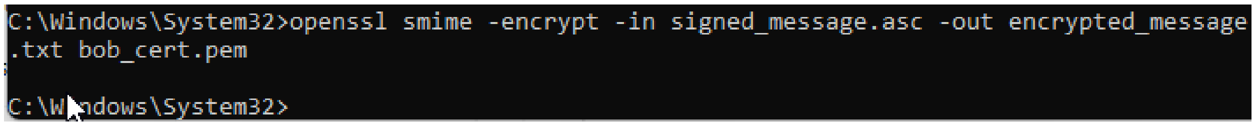


```
C:\Windows\System32>echo "This is a confidential message for Bob." > original_message.txt

C:\Windows\System32>
```

Then, sign and encrypt it (as Alice) for Bob:

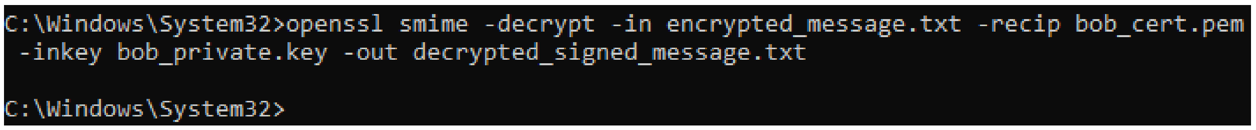


```
C:\Windows\System32>openssl smime -sign -in original_message.txt -signer alice_cert.pem
-inkey alice_private.key -out signed_message.asc -text

C:\Windows\System32>
```



```
C:\Windows\System32>openssl smime -encrypt -in signed_message.asc -out encrypted_message
.txt bob_cert.pem

C:\Windows\System32>
```

3. Decrypt and Verify the Message as Bob



```
C:\Windows\System32>openssl smime -decrypt -in encrypted_message.txt -recip bob_cert.pem
 -inkey bob_private.key -out decrypted_signed_message.txt

C:\Windows\System32>
```

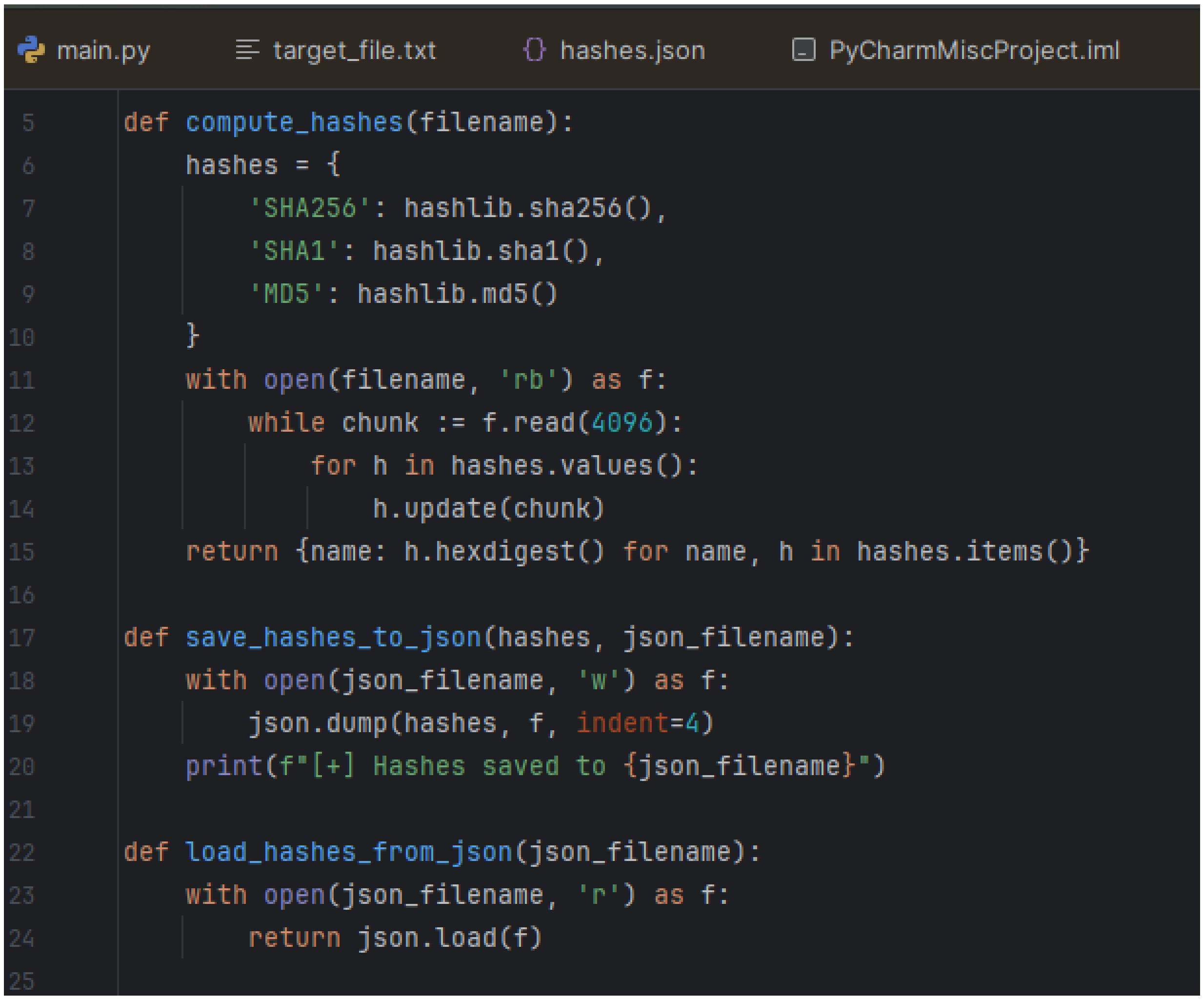
Task 5: Hashing & Integrity Check Utility

Tasks:

1. Create a Python script that:

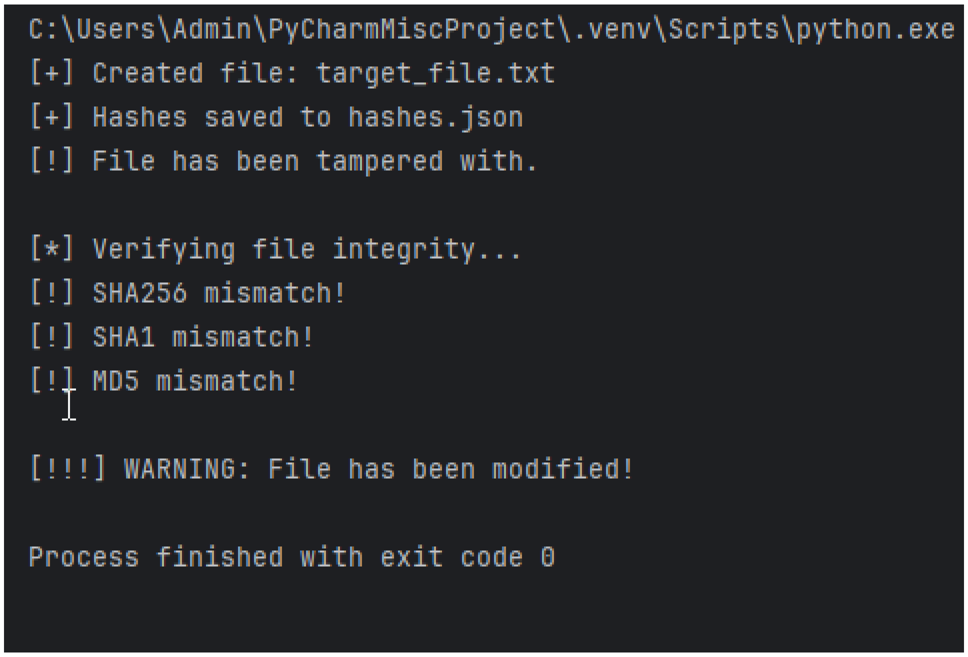
○ Takes a file and computes SHA-256, SHA-1, and MD5 hashes

○ Stores hashes in a .json file



```python
def compute_hashes(filename):
    hashes = {
        'SHA256': hashlib.sha256(),
        'SHA1': hashlib.sha1(),
        'MD5': hashlib.md5()
    }
    with open(filename, 'rb') as f:
        while chunk := f.read(4096):
            for h in hashes.values():
                h.update(chunk)
    return {name: h.hexdigest() for name, h in hashes.items()}

def save_hashes_to_json(hashes, json_filename):
    with open(json_filename, 'w') as f:
        json.dump(hashes, f, indent=4)
    print(f"[+] Hashes saved to {json_filename}")

def load_hashes_from_json(json_filename):
    with open(json_filename, 'r') as f:
        return json.load(f)

```

Output showing integrity check result (pass/fail)



```
C:\Users\Admin\PyCharmMiscProject\.venv\Scripts\python.exe
[+] Created file: target_file.txt
[+] Hashes saved to hashes.json
[!] File has been tampered with.

[*] Verifying file integrity...
[!] SHA256 mismatch!
[!] SHA1 mismatch!
[!] MD5 mismatch!

[!!!] WARNING: File has been modified!

Process finished with exit code 0
```

When I ran the script, it first created a file called target_file.txt with some secure content inside. Then, it calculated three types of hashes - SHA-256, SHA-1, and MD5 - which are like digital fingerprints for the file. These fingerprints were saved in a separate file called hashes.json.

Next, the script pretended someone tampered with the file by adding extra text to it. After that, it recalculated the file's fingerprints and compared them to the ones saved earlier.

Because the file had been changed, all three hashes were different. The script showed a warning saying the file was modified. This proves that the script can detect if someone changes a file without permission.