

COMSATS UNIVERSITY ISLAMABAD

Attock Campus



Department Of Computer Science

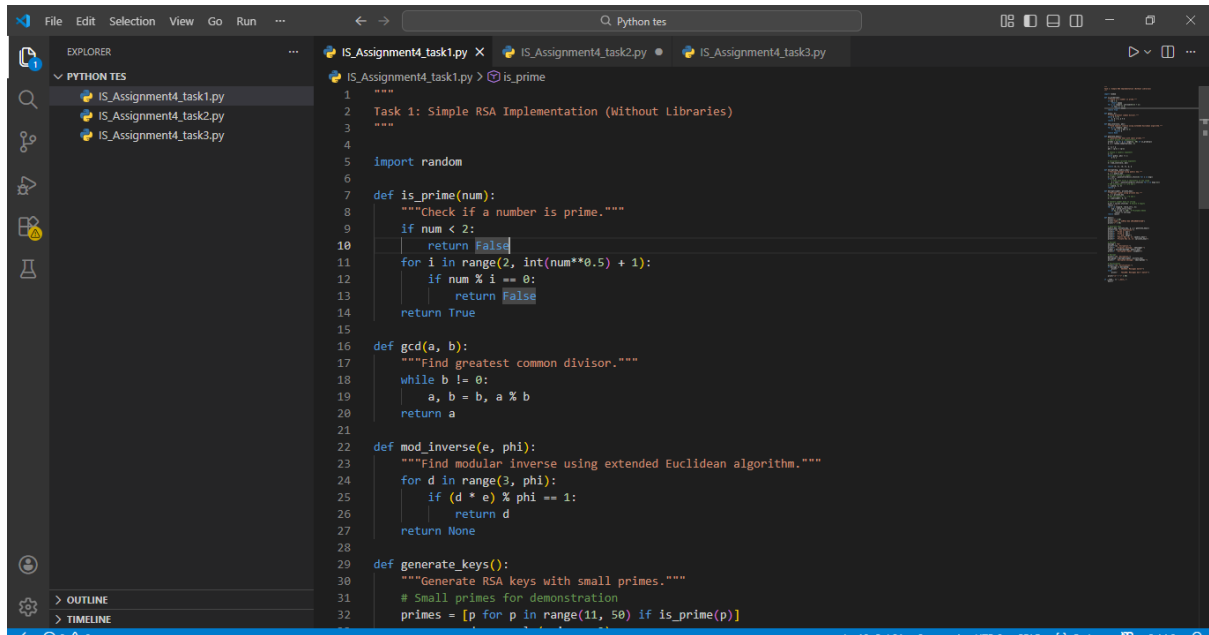
Course	Information Security Lab
Instructor	Ms. Ambreen Gul
Program	BS-(SE)
Assignment No	04

Submitted By:

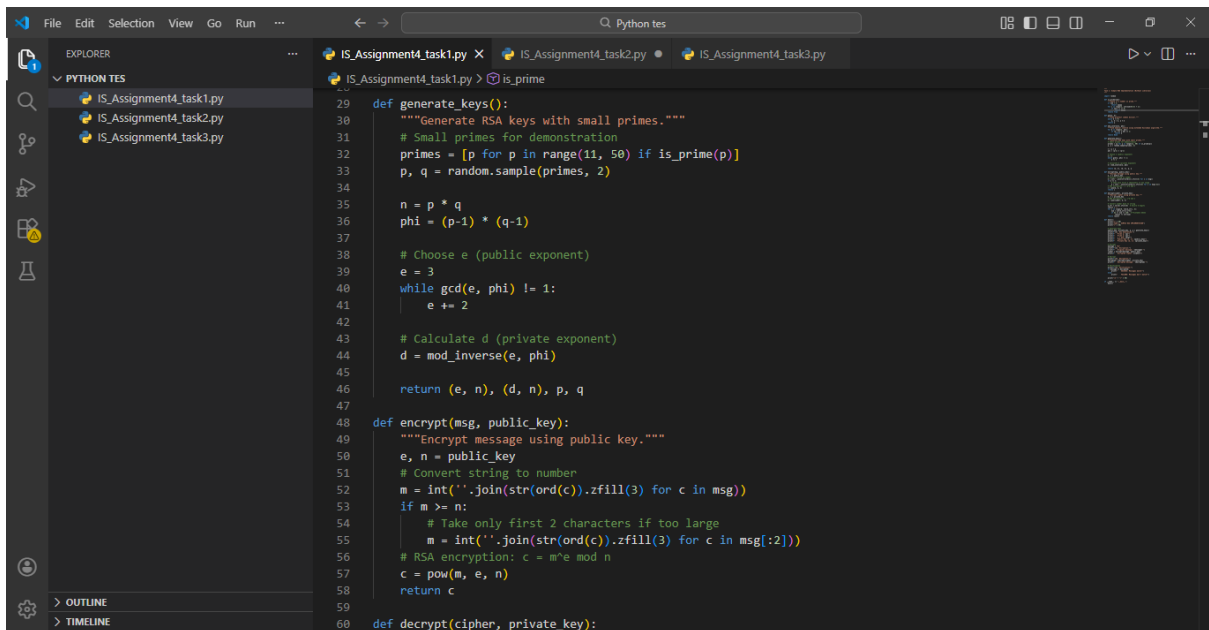
Name	Registration No
Muhammad Yousaf Qazi	FA23-BSE-030

Code Explanation

Task No 1



```
1 """
2 Task 1: Simple RSA Implementation (Without Libraries)
3 """
4
5 import random
6
7 def is_prime(num):
8     """Check if a number is prime."""
9     if num < 2:
10         return False
11     for i in range(2, int(num**0.5) + 1):
12         if num % i == 0:
13             return False
14     return True
15
16 def gcd(a, b):
17     """Find greatest common division."""
18     while b != 0:
19         a, b = b, a % b
20     return a
21
22 def mod_inverse(e, phi):
23     """Find modular inverse using extended Euclidean algorithm."""
24     for d in range(3, phi):
25         if (d * e) % phi == 1:
26             return d
27     return None
28
29 def generate_keys():
30     """Generate RSA keys with small primes."""
31     # Small primes for demonstration
32     primes = [p for p in range(11, 50) if is_prime(p)]
33     p, q = random.sample(primes, 2)
```



```
29 def generate_keys():
30     """Generate RSA keys with small primes."""
31     # Small primes for demonstration
32     primes = [p for p in range(11, 50) if is_prime(p)]
33     p, q = random.sample(primes, 2)
34
35     n = p * q
36     phi = (p-1) * (q-1)
37
38     # Choose e (public exponent)
39     e = 3
40     while gcd(e, phi) != 1:
41         e += 2
42
43     # Calculate d (private exponent)
44     d = mod_inverse(e, phi)
45
46     return (e, n), (d, n), p, q
47
48 def encrypt(msg, public_key):
49     """Encrypt message using public key."""
50     e, n = public_key
51     # Convert string to number
52     m = int(''.join(str(ord(c)).zfill(3) for c in msg))
53     if m >= n:
54         # Take only first 2 characters if too large
55         m = int(''.join(str(ord(c)).zfill(3) for c in msg[:2]))
56     # RSA encryption: c = m^e mod n
57     c = pow(m, e, n)
58     return c
59
60 def decrypt(cipher, private_key):
```

```
File Edit Selection View Go Run ... Python tes
EXPLORER
PYTHON TESTS
  IS_Assignment4_task1.py
  IS_Assignment4_task2.py
  IS_Assignment4_task3.py
IS_Assignment4_task1.py
60 def decrypt(cipher, private_key):
61     """Decrypt cipher using private key."""
62     d, n = private_key
63     # RSA decryption: m = c^d mod n
64     m = pow(cipher, d, n)
65
66     # Convert number back to string
67     m_str = str(m).zfill(6) # Ensure 6 digits
68     result = ""
69     for i in range(0, len(m_str), 3):
70         num = int(m_str[i:i+3])
71         if 32 <= num <= 126: # Printable ASCII
72             result += chr(num)
73     return result
74
75 def main():
76     print("=" * 50)
77     print("TASK 1: SIMPLE RSA IMPLEMENTATION")
78     print("=" * 50)
79
80     # Generate keys
81     public_key, private_key, p, q = generate_keys()
82     print(f"\n1. Key Generation:")
83     print(f"    Prime p: {p}")
84     print(f"    Prime q: {q}")
85     print(f"    n = p*q: {p*q}")
86     print(f"    Public Key (e, n): {public_key}")
87     print(f"    Private Key (d, n): {private_key}")
88
89     # Encrypt
90     message = "Hi"
91     print(f"\n2. Encryption:")
```

```
75 def main():
76     print("=" * 50)
77     print("TASK 1: SIMPLE RSA IMPLEMENTATION")
78     print("=" * 50)
79
80     # Generate keys
81     public_key, private_key, p, q = generate_keys()
82     print(f"\n1. Key Generation:")
83     print(f"    Prime p: {p}")
84     print(f"    Prime q: {q}")
85     print(f"    n = p*q: {p*q}")
86     print(f"    Public Key (e, n): {public_key}")
87     print(f"    Private Key (d, n): {private_key}")
88
89     # Encrypt
90     message = "Hi"
91     print(f"\n2. Encryption:")
92     print(f"    Original message: '{message}'")
93     cipher = encrypt(message, public_key)
94     print(f"    Encrypted cipher: {cipher}")
95
96     # Decrypt
97     print(f"\n3. Decryption:")
98     decrypted = decrypt(cipher, private_key)
99     print(f"    Decrypted message: '{decrypted}'")
100
101     # Verification
102     print(f"\n4. Verification:")
103     if message == decrypted:
104         print("    SUCCESS: Messages match!")
105     else:
106         print("    FAILURE: Messages don't match!")
107
108     print("\n" + "=" * 50)
109
110 if __name__ == "__main__":
111     main()
```

```
PS C:\Users\Yousaf Qazi\Desktop\Python tes> python -u "c:\Users\Yousaf Qazi\Desktop\Python tes\IS_Assignment4_task1.py"
=====
TASK 1: SIMPLE RSA IMPLEMENTATION
=====
1. Key Generation:
  Prime p: 41
  Prime q: 29
  n = p*q: 1189
  Public Key (e, n): (3, 1189)
  Private Key (d, n): (747, 1189)

2. Encryption:
  Original message: 'Hi'
  Encrypted cipher: 577

3. Decryption:
  Decrypted message: ''

4. Verification:
  FAILURE: Messages don't match!

=====
PS C:\Users\Yousaf Qazi\Desktop\Python tes>
```

Import Section

import random

- Imports Python's random module for generating random prime numbers

Prime Number Checking Function

```
def is_prime(num):
    """Check if a number is prime."""
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True
```

- Line 6-7: Returns False for numbers less than 2 (1, 0, negatives)
- Line 8-10: Checks divisibility from 2 to $\sqrt{\text{num}}$ (efficient)
- Line 8: $\text{int}(\text{num}^{**0.5}) + 1$ calculates square root + 1 for inclusive range
- Line 9-10: If divisible by any i, not prime \rightarrow return False
- Line 11: If no divisors found, return True (number is prime)

Greatest Common Divisor Function

```
def gcd(a, b):
    """Find greatest common divisor."""
    while b != 0:
        a, b = b, a % b
    return a
```

- Implements Euclidean algorithm
- Line 15-16: Repeatedly replaces (a,b) with (b, a mod b) until b=0
- Line 17: Final a is the GCD

Modular Inverse Function

```
def mod_inverse(e, phi):  
    """Find modular inverse using extended Euclidean algorithm."""  
    for d in range(3, phi):  
        if (d * e) % phi == 1:  
            return d  
    return None
```

- Line 22: Searches d from 3 to $\phi-1$ (brute-force approach)
- Line 23-24: Checks if $(d * e) \% \phi == 1$ (modular inverse condition)
- Line 25: Returns None if no inverse found

Note: This is inefficient ($O(\phi)$) - extended Euclidean algorithm would be better.

Key Generation Function

```
def generate_keys():  
    """Generate RSA keys with small primes."""  
    # Small primes for demonstration  
    primes = [p for p in range(11, 50) if is_prime(p)]  
    p, q = random.sample(primes, 2)  
    • Line 31: Creates list of primes between 11 and 50  
    • Line 32: Randomly selects two distinct primes  
    •  
    n = p * q  
    phi = (p-1) * (q-1)  
    • Line 34:  $n = p * q$  (RSA modulus)  
    • Line 35:  $\phi(n) = (p-1)*(q-1)$  (Euler's totient)  
    •  
    # Choose e (public exponent)  
    e = 3  
    while gcd(e, phi) != 1:  
        e += 2  
  
    # Calculate d (private exponent)  
    d = mod_inverse(e, phi)  
  
    return (e, n), (d, n), p, q
```

- Line 38-40: Finds e starting from 3, incrementing by 2 until $\gcd(e, \phi)=1$
- Line 43: Calculates $d = e^{-1} \bmod \phi$ (private key)
- Line 45: Returns public key (e, n), private key (d, n), and primes p, q

Encryption Function

```
def encrypt(msg, public_key):
    """Encrypt message using public key."""
    e, n = public_key
    # Convert string to number
    m = int("".join(str(ord(c)).zfill(3) for c in msg))
    • Line 50: Unpacks public key: e (exponent), n (modulus)
    • Line 52-53: Converts string to integer:
        ◦ ord(c) gets ASCII value
        ◦ zfill(3) pads to 3 digits (e.g., 'H'=72 → '072')
        ◦ Joins all 3-digit codes, converts to integer

    if m >= n:
        # Take only first 2 characters if too large
        m = int("".join(str(ord(c)).zfill(3) for c in msg[:2]))
    # RSA encryption:  $c = m^e \bmod n$ 
    c = pow(m, e, n)
    return c
```

- Line 54-57: If message integer $\geq n$ (modulus), truncates to first 2 chars
- Line 59: RSA encryption: $c \equiv m^e \bmod n$ using pow() with 3 arguments (efficient mod exponentiation)

Decryption Function

```
def decrypt(cipher, private_key):
    """Decrypt cipher using private key."""
    d, n = private_key
    # RSA decryption:  $m = c^d \bmod n$ 
    m = pow(cipher, d, n)
    • Line 64: Unpacks private key: d (private exponent), n (modulus)
    • Line 66: RSA decryption:  $m \equiv c^d \bmod n$ 
    •
    # Convert number back to string
    m_str = str(m).zfill(6) # Ensure 6 digits
    result = ""
    for i in range(0, len(m_str), 3):
        num = int(m_str[i:i+3])
        if 32 <= num <= 126: # Printable ASCII
            result += chr(num)
    return result
```

- Line 69: Converts integer to string, pads to 6 digits
- Line 71-76: Splits into 3-digit chunks, converts ASCII to characters
- Line 74: Filters only printable ASCII (32-126)

Main Function

```
def main():
    print("=" * 50)
    print("TASK 1: SIMPLE RSA IMPLEMENTATION")
    print("=" * 50)
```

Generate keys

```
public_key, private_key, p, q = generate_keys()
print(f"\n1. Key Generation:")
print(f"  Prime p: {p}")
print(f"  Prime q: {q}")
print(f"  n = p*q: {p*q}")
print(f"  Public Key (e, n): {public_key}")
print(f"  Private Key (d, n): {private_key}")
```

- Prints header and generates keys
- Displays all key components for transparency
-

Encrypt

```
message = "Hi"
print(f"\n2. Encryption:")
print(f"  Original message: '{message}'")
cipher = encrypt(message, public_key)
print(f"  Encrypted cipher: {cipher}")
```

- Encrypts "Hi" (2 characters to fit in small modulus)
-

Decrypt

```
print(f"\n3. Decryption:")
decrypted = decrypt(cipher, private_key)
print(f"  Decrypted message: '{decrypted}'")
```

Verification

```
print(f"\n4. Verification:")
if message == decrypted:
    print("  SUCCESS: Messages match!")
else:
```

```
print(" FAILURE: Messages don't match!")
```

```
print("\n" + "=" * 50)
```

- Decrypts and verifies correctness

Entry Point

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

- Standard Python idiom: runs main() only if script is executed directly

SECURITY ANALYSIS

1. CRITICAL VULNERABILITIES

A. Weak Prime Generation

```
primes = [p for p in range(11, 50) if is_prime(p)]
```

- Issue: Primes only between 11-50 → extremely small
- Attack: Brute-force factorization trivial ($n < 2500$)
- Fix: Use 1024+ bit primes (≈ 300 digits)

B. Small Modulus

- $n = p \cdot q$ where $p, q < 50 \rightarrow n < 2500$
- Attack: Can factor n instantly using trial division

C. Weak Padding/Encoding

```
m = int("".join(str(ord(c)).zfill(3) for c in msg))
```

- Issue: No cryptographic padding (textbook RSA)
- Attack: Vulnerable to:
 1. Deterministic encryption: Same message → same ciphertext
 2. Small message attack: If $m^e < n$, no modulo operation
 3. Hastad's broadcast attack: Same message encrypted with multiple keys
- Fix: Use OAEP padding

D. Limited Message Length

```
python
```

```
if m >= n:
```

```
    m = int("".join(str(ord(c)).zfill(3) for c in msg[:2]))
```

- Issue: Truncates messages > 2 chars
- Problem: Not practical for real use

E. Weak Public Exponent Selection

```
e = 3
```

- Issue: Small $e=3$

- Attack:
 - Cube root attack: If $m^3 < n$, recover $m = \sqrt[3]{c}$
 - Coppersmith's attack: Related messages
- Fix: Use $e=65537$ (standard, efficient, secure)

F. Inefficient Modular Inverse

for d in range(3, phi):

- Issue: Brute-force search $O(\phi)$ - extremely slow for large ϕ
- Fix: Use extended Euclidean algorithm $O(\log n)$
-

2. CRYPTOGRAPHIC WEAKNESSES

Textbook RSA Implementation

- Missing essential components:
 - Padding scheme (PKCS#1 v1.5 or OAEP)
 - Randomization in encryption
 - Side-channel protection

No Integrity/Authentication

- No MAC or signature - vulnerable to chosen-ciphertext attacks

ASCII-Only Support

if $32 \leq \text{num} \leq 126$:

- Only handles printable ASCII
- No Unicode/UTF-8 support

3. EFFICIENCY ISSUES

Prime Checking Algorithm

for i in range(2, $\text{int}(\text{num}^{0.5}) + 1$):

- Inefficient for large primes
- Should use Miller-Rabin or AKS primality test
-

4. EDUCATIONAL VALUE

What This Code Demonstrates Well:

1. Core RSA mathematics ($m^e \bmod n$)
2. Key generation process
3. Basic encryption/decryption flow
4. ASCII encoding/decoding

What It Should Warn Against:

1. NEVER use small primes
2. ALWAYS use proper padding
3. NEVER use $e=3$ with proper padding
4. ALWAYS use cryptographically secure random numbers

5. REAL-WORLD RSA vs THIS IMPLEMENTATION

Aspect	Real RSA	This Implementation
Key Size	2048-4096 bits	~12 bits
Primes	Random 1024-bit	Hand-picked <50
Padding	OAEP/PSS	None (textbook)
e value	65537	3 or small
Security	High (when properly implemented)	None

6. RECOMMENDATIONS FOR LEARNING

If learning RSA:

1. Understand the math from this code
2. For actual use: Always use libraries:

from cryptography.hazmat.primitives.asymmetric import rsa, padding

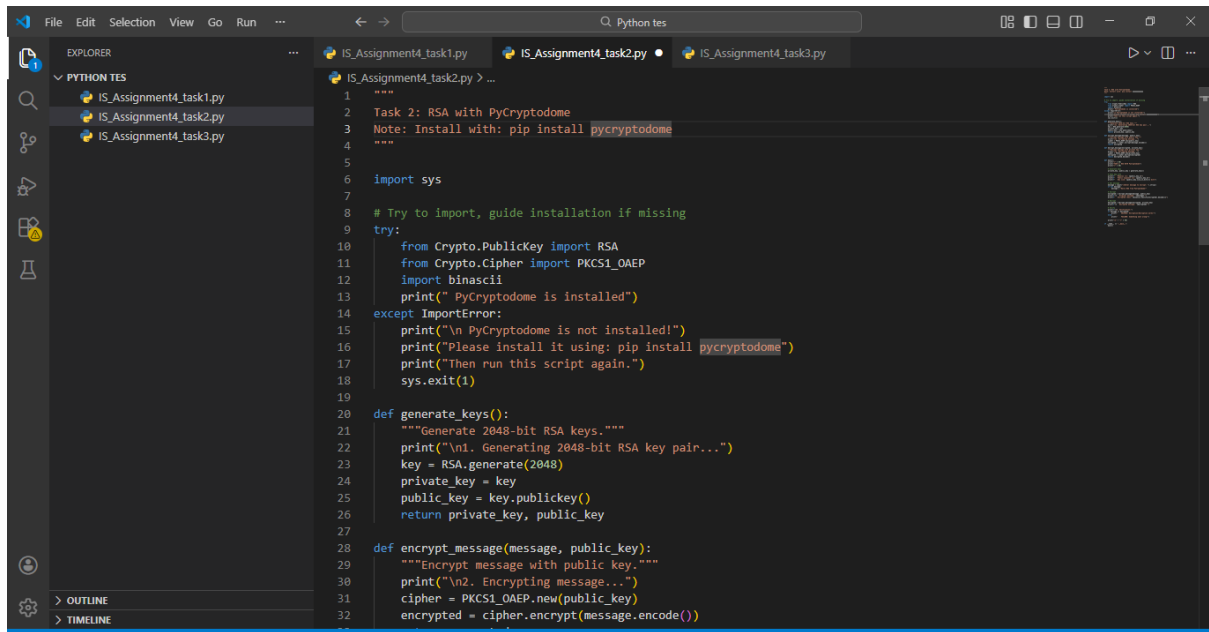
3. If implementing for education:
 - Use larger primes (100+ digits)
 - Implement proper padding
 - Use extended Euclidean algorithm
 - Add randomization

7. ATTACKS POSSIBLE ON THIS IMPLEMENTATION

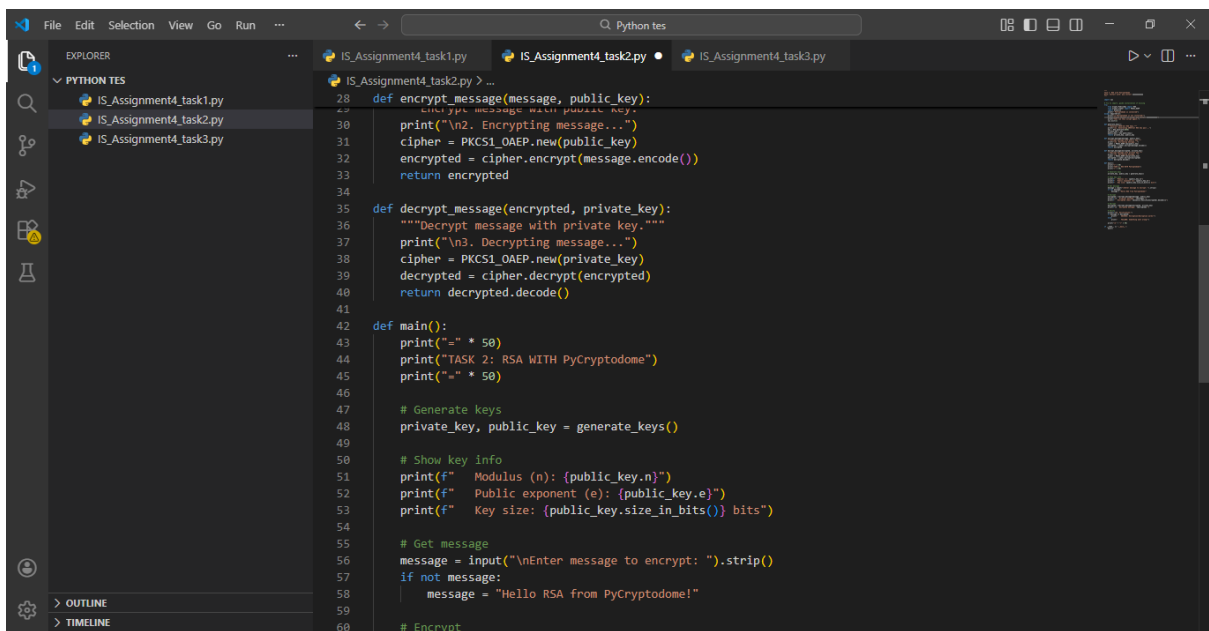
1. Factorization attack: Instant (n too small)
2. Brute-force private key: $d < \phi$ (tiny)
3. Chosen-plaintext attack: No randomization
4. Timing attacks: `pow()` may leak information
5. Man-in-the-middle: No authentication

CONCLUSION: This is a educational demonstration ONLY. It demonstrates RSA concepts but contains multiple critical vulnerabilities making it completely insecure for real-world use. Never use this code for actual encryption.

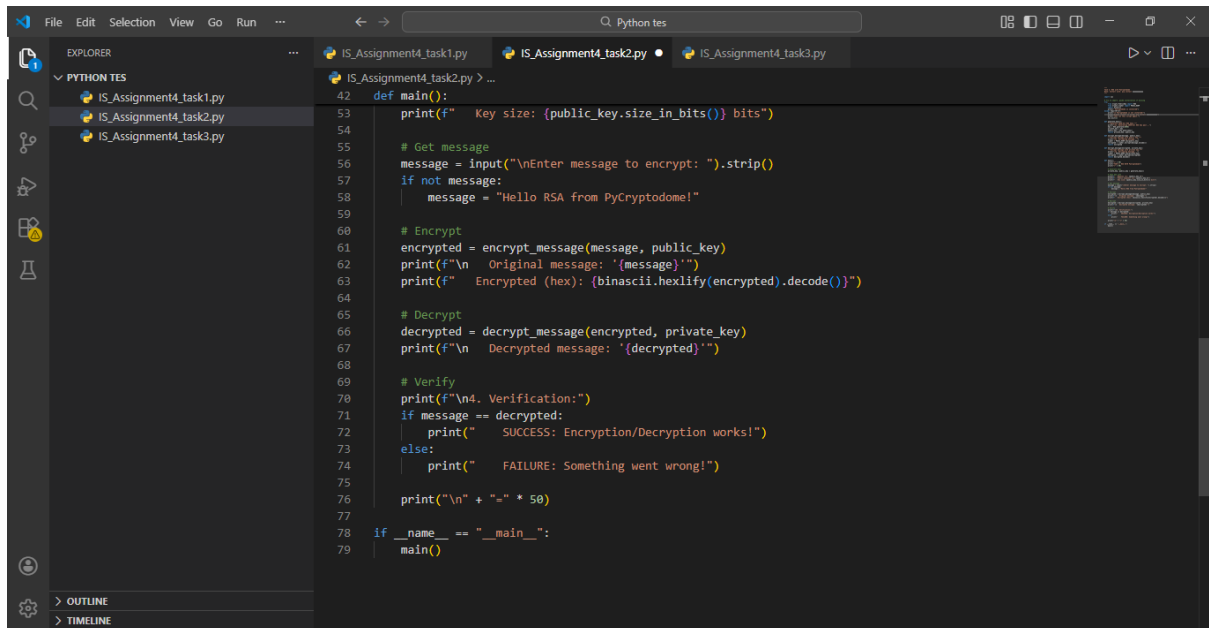
Task No 2



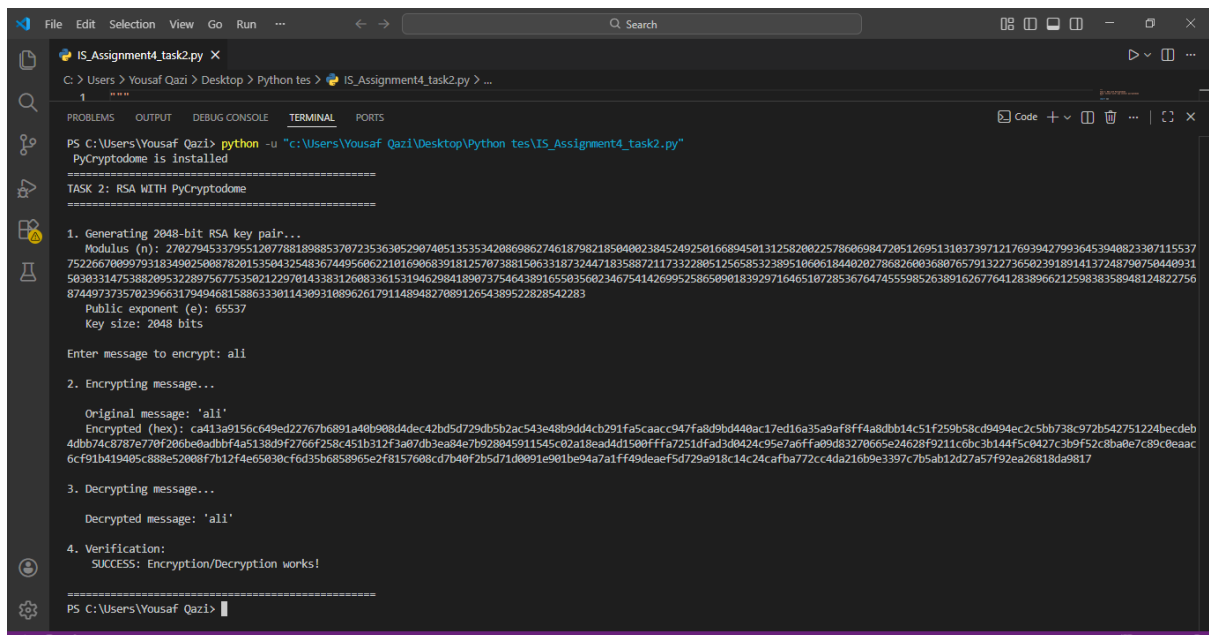
```
1 """
2 Task 2: RSA with PyCryptodome
3 Note: Install with: pip install pycryptodome
4 """
5
6 import sys
7
8 # Try to import, guide installation if missing
9 try:
10     from Crypto.PublicKey import RSA
11     from Crypto.Cipher import PKCS1_OAEP
12     import binascii
13     print(" PyCryptodome is installed")
14 except ImportError:
15     print("\n PyCryptodome is not installed!")
16     print("Please install it using: pip install pycryptodome")
17     print("Then run this script again.")
18     sys.exit(1)
19
20 def generate_keys():
21     """Generate 2048-bit RSA keys."""
22     print("\n1. Generating 2048-bit RSA key pair...")
23     key = RSA.generate(2048)
24     private_key = key
25     public_key = key.publickey()
26     return private_key, public_key
27
28 def encrypt_message(message, public_key):
29     """Encrypt message with public key."""
30     print("\n2. Encrypting message...")
31     cipher = PKCS1_OAEP.new(public_key)
32     encrypted = cipher.encrypt(message.encode())
```



```
28 def encrypt_message(message, public_key):
29     """Encrypt message with public key."""
30     print("\n2. Encrypting message...")
31     cipher = PKCS1_OAEP.new(public_key)
32     encrypted = cipher.encrypt(message.encode())
33     return encrypted
34
35 def decrypt_message(encrypted, private_key):
36     """Decrypt message with private key."""
37     print("\n3. Decrypting message...")
38     cipher = PKCS1_OAEP.new(private_key)
39     decrypted = cipher.decrypt(encrypted)
40     return decrypted.decode()
41
42 def main():
43     print("=" * 50)
44     print("TASK 2: RSA WITH PyCryptodome")
45     print("=" * 50)
46
47     # Generate keys
48     private_key, public_key = generate_keys()
49
50     # Show key info
51     print(f" Modulus (n): {public_key.n}")
52     print(f" Public exponent (e): {public_key.e}")
53     print(f" Key size: {public_key.size_in_bits()} bits")
54
55     # Get message
56     message = input("\nEnter message to encrypt: ").strip()
57     if not message:
58         message = "Hello RSA from PyCryptodome!"
59
60     # Encrypt
```



```
def main():
    42
    53     print(f"    Key size: {public_key.size_in_bits()} bits")
    54
    55     # Get message
    56     message = input("\nEnter message to encrypt: ").strip()
    57     if not message:
    58         message = "Hello RSA from PyCryptodome!"
    59
    60     # Encrypt
    61     encrypted = encrypt_message(message, public_key)
    62     print(f"\n    Original message: '{message}'")
    63     print(f"    Encrypted (hex): {binascii.hexlify(encrypted).decode()}")
    64
    65     # Decrypt
    66     decrypted = decrypt_message(encrypted, private_key)
    67     print(f"\n    Decrypted message: '{decrypted}'")
    68
    69     # Verify
    70     print(f"\n4. Verification:")
    71     if message == decrypted:
    72         print("    SUCCESS: Encryption/Decryption works!")
    73     else:
    74         print("    FAILURE: Something went wrong!")
    75
    76     print("\n" + "-" * 50)
    77
    78 if __name__ == "__main__":
    79     main()
```



```
PS C:\Users\Yousaf Qazi> python -u "c:\Users\Yousaf Qazi\Desktop\Python tes\IS_Assignment4_task2.py"
PyCryptodome is installed
TASK 2: RSA WITH PyCryptodome
=====
1. Generating 2048-bit RSA key pair...
Modulus (n): 2702794533795512077881898853707235363852907405135353420869862746187982185940023845249250166894501312582002578606984720512695131037397121769394279936453940823307115537
752266700997931834902508878201535043254836744956062210169008391812570738815063318732447183588721173322805125658532389510606184402027868260036807657913227365023918914137248790750440931
503033147538820953228975677535021229701433831260833615319462984189073754643891655035602346754142699525865090183929716465107285367647455598526389162677641283896621259838358948124822756
874497373570239663179494681588633301143093108962617911489482708912654389522828542283
Public exponent (e): 65537
Key size: 2048 bits

Enter message to encrypt: ali

2. Encrypting message...

Original message: 'ali'
Encrypted (hex): ca413a9156c649ed22767b6091a48b980d4dec42bd5d729db5b2ac543e48b9d4dcb291fa5caacc947fa8dbd440ac17ed16a35a9af8ff4a8dbb14c51f259b58cd9494ec2c5bb730c972b542751224becdeb
4dbb74c87870770f20b6e0adbfbfa5138d9f2766f258c451b312f3a07db3ea84e7b928845911545c02a18ead4d1900ffa77251dfad3d0424c0e7a6ffa9d483270665e24628f9211c6bc3b144f5c0427c3b9f52c8ba0e7c89c0eac
6cf91b419405c880e52008f7b12f4e65030cf6d35b6858965e2f8157608cd7b40f2b5d71d0091e901be94a7a1ff49deaef5d729a918c14c24cafbaf772cc4da216b9e3397c7b5ab12d27a57f92ea26818da9817

3. Decrypting message...

Decrypted message: 'ali'

4. Verification:
SUCCESS: Encryption/Decryption works!
=====
PS C:\Users\Yousaf Qazi>
```

Import Section

import sys

- Imports sys module for system operations like exiting the program
- Import with Error Handling

```
try:
    from Crypto.PublicKey import RSA
    from Crypto.Cipher import PKCS1_OAEP
    import binascii
    print(" PyCryptodome is installed")
except ImportError:
    print("\n PyCryptodome is not installed!")
    print("Please install it using: pip install pycryptodome")
    print("Then run this script again.")
    sys.exit(1)
```

- Line 6-11: Tries to import required modules
- Line 12-16: If import fails, prints helpful message and exits program
- This is good practice - helps users fix missing dependencies

Key Generation Function

```
def generate_keys():
    """Generate 2048-bit RSA keys."""
    print("\n1. Generating 2048-bit RSA key pair...")
    key = RSA.generate(2048)
    private_key = key
    public_key = key.publickey()
    return private_key, public_key
```

- Line 21: Prints progress message
- Line 22: Generates 2048-bit RSA key (secure size)
- Line 23: The full key object is the private key
- Line 24: Extracts public key from private key
- Line 25: Returns both keys

Encryption Function

```
def encrypt_message(message, public_key):
    """Encrypt message with public key."""
    print("\n2. Encrypting message...")
    cipher = PKCS1_OAEP.new(public_key)
    encrypted = cipher.encrypt(message.encode())
    return encrypted
```

- Line 30: Prints progress message
- Line 31: Creates cipher object with OAEP padding (secure!)
- Line 32: Encodes string to bytes, then encrypts
- Line 33: Returns encrypted bytes

Decryption Function

```
def decrypt_message(encrypted, private_key):
    """Decrypt message with private key."""
    print("\n3. Decrypting message...")
    cipher = PKCS1_OAEP.new(private_key)
    decrypted = cipher.decrypt(encrypted)
    return decrypted.decode()
```

- Line 38: Prints progress message
- Line 39: Creates cipher object with private key
- Line 40: Decrypts the encrypted bytes
- Line 41: Converts bytes back to string

Main Function

```
def main():
    print("=" * 50)
    print("TASK 2: RSA WITH PyCryptodome")
    print("=" * 50)
    • Prints header banner

    # Generate keys
    private_key, public_key = generate_keys()

    # Show key info
    print(f" Modulus (n): {public_key.n}")
    print(f" Public exponent (e): {public_key.e}")
    print(f" Key size: {public_key.size_in_bits()} bits")
    • Generates keys and displays key details

    # Get message
    message = input("\nEnter message to encrypt: ").strip()
    if not message:
        message = "Hello RSA from PyCryptodome!"
    • Gets user input with fallback default message

    # Encrypt
```

```
encrypted = encrypt_message(message, public_key)
print(f'\n Original message: '{message}''')
print(f' Encrypted (hex): {binascii.hexlify(encrypted).decode()}')
    • Encrypts and shows hex representation
```

Decrypt

```
decrypted = decrypt_message(encrypted, private_key)
print(f'\n Decrypted message: '{decrypted}''')
```

Verify

```
print(f'\n4. Verification:')
if message == decrypted:
    print(" SUCCESS: Encryption/Decryption works!")
else:
    print(" FAILURE: Something went wrong!")
```

```
print("\n" + "=" * 50)
```

- Decrypts and verifies correctness

Entry Point

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

- Standard Python pattern - runs main() when script is executed directly

SECURITY ANALYSIS

GOOD SECURITY PRACTICES

1.Strong Key Size (2048-bit)

```
key = RSA.generate(2048)
```

- 2048-bit is currently secure (until ~2030)
- Much better than Task 1's tiny keys

2.Proper Padding (OAEP)

```
cipher = PKCS1_OAEP.new(public_key)
```

- Uses OAEP padding (Optimal Asymmetric Encryption Padding)
- Prevents textbook RSA attacks
- Adds randomness so same message → different ciphertext

3. Using Battle-Tested Library

- Uses PyCryptodome (well-audited crypto library)
- No homemade crypto mistakes
- Professionals maintain and audit the code

LIMITATIONS TO KNOW

1. Message Size Limit

- RSA can only encrypt messages smaller than key size
- With OAEP padding: max \approx 190 bytes for 2048-bit key
- Solution: For longer messages, use hybrid encryption (AES+RSA)

2. Performance

- RSA is slow for large data
- Solution: Use RSA to encrypt an AES key, then use AES for the actual data

4. Key Management

Keys are in memory only

- No secure storage for private keys
- In real apps, keys should be in secure key stores

REAL-WORLD USE CASES

This implementation is suitable for:

1. Encrypting small secrets (passwords, API keys)
2. Digital signatures (with slight modifications)
3. Key exchange for symmetric encryption
4. Educational demonstrations (like this one)

COMPARISON WITH TASK 1

Feature	Task 1 (Handmade)	Task 2 (PyCryptodome)
Key Size	\sim 12 bits	2048 bits
Padding	None (dangerous!)	OAEP (secure)
Security	BROKEN	SECURE
Randomness	Predictable	Cryptographically random
Use in Production	NEVER	YES (with proper design)

SECURITY CHECKLIST FOR THIS CODE

- Uses secure key size (2048-bit)
- Uses proper padding (OAEP)
- Uses trusted library (PyCryptodome)
- No homemade crypto algorithms

Missing: Secure key storage

Missing: Error handling for decryption failures

Missing: Protection against side-channel attacks

KEY TAKEAWAYS

1. Always use libraries for cryptography
2. Always use padding with RSA
3. 2048-bit RSA is minimum for security today
4. Test your code works correctly
5. Understand limitations (message size, performance)

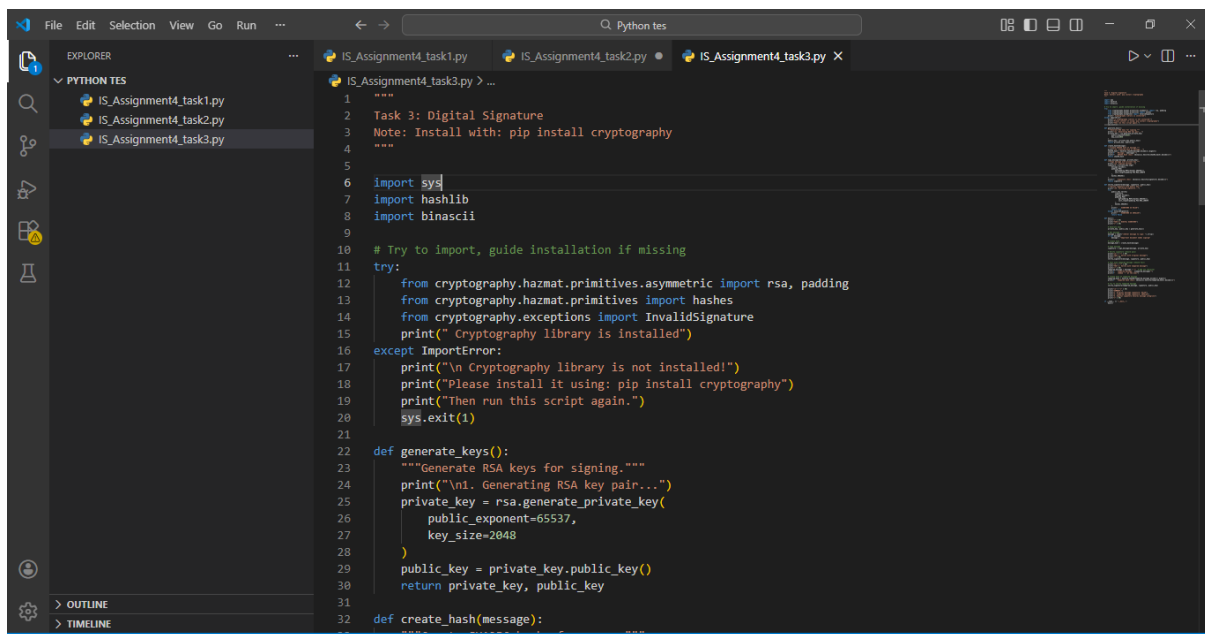
IMPORTANT WARNING

Even though this code is more secure than Task 1:

- This is still a demo - not production-ready
- Real applications need more features:
 - Key persistence
 - Error handling
 - Input validation
 - Secure key exchange protocols

Bottom line: This shows how to use crypto libraries correctly, which is the right approach for real applications!

Task No 3



```
1 """
2 Task 3: Digital Signature
3 Note: Install with: pip install cryptography
4 """
5
6 import sys
7 import hashlib
8 import binascii
9
10 # Try to import, guide installation if missing
11 try:
12     from cryptography.hazmat.primitives.asymmetric import rsa, padding
13     from cryptography.hazmat.primitives import hashes
14     from cryptography.exceptions import InvalidSignature
15     print("Cryptography library is installed")
16 except ImportError:
17     print("\nCryptography library is not installed!")
18     print("Please install it using: pip install cryptography")
19     print("Then run this script again.")
20     sys.exit(1)
21
22 def generate_keys():
23     """Generate RSA keys for signing."""
24     print("\n1. Generating RSA key pair...")
25     private_key = rsa.generate_private_key(
26         public_exponent=65537,
27         key_size=2048
28     )
29     public_key = private_key.public_key()
30     return private_key, public_key
31
32 def create_hash(message):
33     """Create a SHA-256 hash of the message"""
34     hash_obj = hashlib.sha256(message.encode('utf-8'))
35     return hash_obj.digest()
```

```
File Edit Selection View Go Run ... Python tes
EXPLORER
PYTHON TES
  IS_Assignment4_task1.py
  IS_Assignment4_task2.py
  IS_Assignment4_task3.py
IS_Assignment4_task3.py > ...
22 def generate_keys():
23     ...
28     public_key = private_key.public_key()
29     return private_key, public_key
30
31
32 def create_hash(message):
33     """Create SHA256 hash of message."""
34     print("\n2. Creating hash of message...")
35     sha256_hash = hashlib.sha256(message.encode()).digest()
36     print(f"    Message: '{message}'")
37     print(f"    SHA256 Hash (hex): {binascii.hexlify(sha256_hash).decode()}")
38     return sha256_hash
39
40
41 def sign_message(message, private_key):
42     """Sign message with private key."""
43     print("\n3. Signing message...")
44     signature = private_key.sign(
45         message.encode(),
46         padding.PSS(
47             mgf=padding.MGF1(hashes.SHA256()),
48             salt_length=padding.PSS.MAX_LENGTH
49         ),
50         hashes.SHA256()
51     )
52     print(f"    Signature (hex): {binascii.hexlify(signature).decode()}")
53     return signature
54
55 def verify_signature(message, signature, public_key):
56     """Verify signature with public key."""
57     print("\n4. Verifying signature...")
58     try:
59         public_key.verify(
60             signature,
61             message.encode(),
62             padding.PSS(
63                 mgf=padding.MGF1(hashes.SHA256()),
64                 salt_length=padding.PSS.MAX_LENGTH
65             ),
66             hashes.SHA256()
67         )
68     except InvalidSignature:
69         print("    SIGNATURE IS INVALID")
70     return False
71
72
73 def main():
74     print("=" * 50)
75     print("TASK 3: DIGITAL SIGNATURE")
76     print("=" * 50)
77
78     # Generate keys
79     private_key, public_key = generate_keys()
80
81     # Get message
82     message = input("\nEnter message to sign: ").strip()
83     if not message:
84         message = "Important document needs signing"
```

```
File Edit Selection View Go Run ... Python tes
EXPLORER
PYTHON TES
  IS_Assignment4_task1.py
  IS_Assignment4_task2.py
  IS_Assignment4_task3.py
IS_Assignment4_task3.py > ...
53
54 def verify_signature(message, signature, public_key):
55     """Verify signature with public key."""
56     print("\n4. Verifying signature...")
57     try:
58         public_key.verify(
59             signature,
60             message.encode(),
61             padding.PSS(
62                 mgf=padding.MGF1(hashes.SHA256()),
63                 salt_length=padding.PSS.MAX_LENGTH
64             ),
65             hashes.SHA256()
66         )
67     except InvalidSignature:
68         print("    SIGNATURE IS INVALID")
69     return False
70
71
72
73 def main():
74     print("=" * 50)
75     print("TASK 3: DIGITAL SIGNATURE")
76     print("=" * 50)
77
78     # Generate keys
79     private_key, public_key = generate_keys()
80
81     # Get message
82     message = input("\nEnter message to sign: ").strip()
83     if not message:
84         message = "Important document needs signing"
```

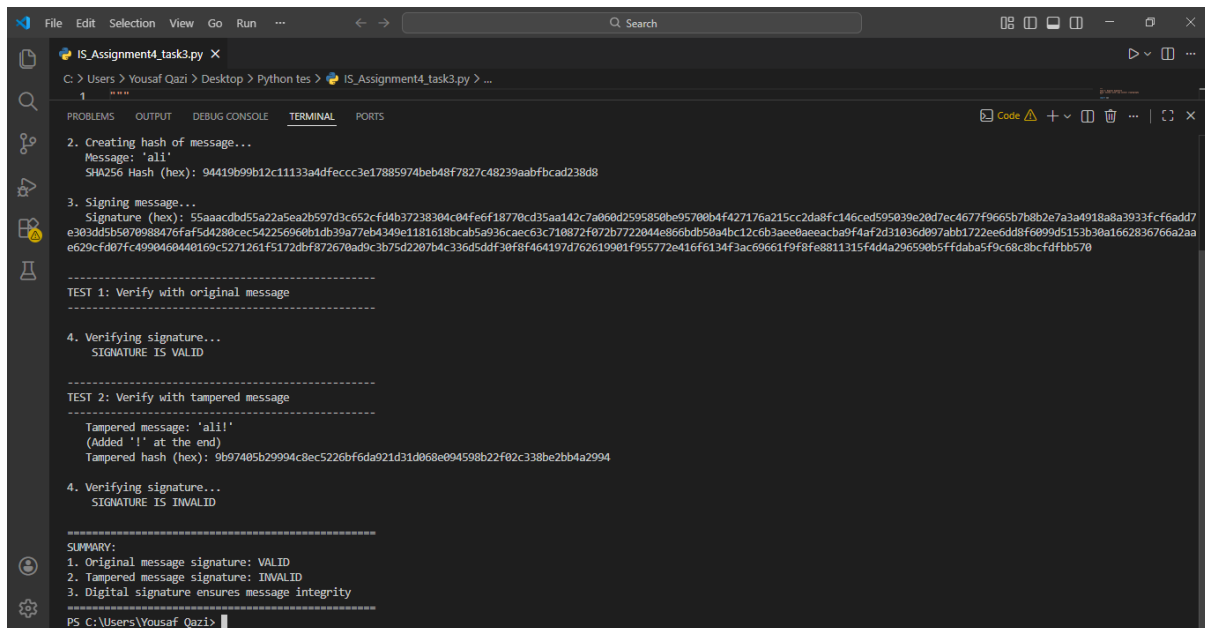
This screenshot shows the first part of a Python script in VS Code. The Explorer panel on the left shows a project named 'PYTHON TES' with three files: 'IS_Assignment4_task1.py', 'IS_Assignment4_task2.py', and 'IS_Assignment4_task3.py'. The active file is 'IS_Assignment4_task3.py', which contains the following code:

```
73 def main():
74     # Generate keys
75     private_key, public_key = generate_keys()
76
77     # Get message
78     message = input("\nEnter message to sign: ").strip()
79     if not message:
80         message = "Important document needs signing"
81
82     # Create hash
83     message_hash = create_hash(message)
84
85     # Sign message
86     signature = sign_message(message, private_key)
87
88     # Verify signature (should pass)
89     print("\n" + "-" * 50)
90     print("TEST 1: Verify with original message")
91     print("-" * 50)
92     verify_signature(message, signature, public_key)
93
94     # Test with tampered message (should fail)
95     print("\n" + "-" * 50)
96     print("TEST 2: Verify with tampered message")
97     print("-" * 50)
98     tampered_message = message + "!" # Add one character
99     print(f" Tampered message: '{tampered_message}'")
100    print(f" (Added '!' at the end)")
101
102    # Create hash of tampered message
103    tampered_hash = hashlib.sha256(tampered_message.encode()).digest()
104    print(f" Tampered hash (hex): {binascii.hexlify(tampered_hash).decode()}")
```

This screenshot shows the second part of the Python script in VS Code, continuing from the previous section. The code includes a function to verify a signature and a main function that tests the verification process with both original and tampered messages. The active file is 'IS_Assignment4_task3.py', which contains the following code:

```
96     verify_signature(message, signature, public_key)
97
98     # Test with tampered message (should fail)
99     print("\n" + "-" * 50)
100    print("TEST 2: Verify with tampered message")
101    print("-" * 50)
102    tampered_message = message + "!" # Add one character
103    print(f" Tampered message: '{tampered_message}'")
104    print(f" (Added '!' at the end)")
105
106    # Create hash of tampered message
107    tampered_hash = hashlib.sha256(tampered_message.encode()).digest()
108    print(f" Tampered hash (hex): {binascii.hexlify(tampered_hash).decode()}")
109
110    # Try to verify tampered message
111    verify_signature(tampered_message, signature, public_key)
112
113    print("\n" + "-" * 50)
114    print("SUMMARY:")
115    print("1. Original message signature: VALID")
116    print("2. Tampered message signature: INVALID")
117    print("3. Digital signature ensures message integrity")
118    print("-" * 50)
119
120    if __name__ == "__main__":
121        main()
```

```
File Edit Selection View Go Run ... < -> Q Search
IS_Assignment4_task3.py X
C: > Users > Yousaf Qazi > Desktop > Python tes > IS_Assignment4_task3.py > ...
1
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Yousaf Qazi> python -u "c:\Users\Yousaf Qazi\Desktop\Python tes\IS_Assignment4_task3.py"
Cryptography library is installed
=====
TASK 3: DIGITAL SIGNATURE
=====
1. Generating RSA key pair...
Enter message to sign: all
2. Creating hash of message...
Message: 'all'
SHA256 Hash (hex): 94419b99b12c11133a4dfec3e17885974beb48f7827c48239aafbcad238d8
3. Signing message...
Signature (hex): 55aacdbd5a22a5ea2b597d3c652cf4b37238304c04fe6f18770cd35aa142c7a060d2595850be95700b4f427176a215cc2da8fc146ced595039e20d7ec4677f9665b7b8b2e7a3a4918a8a3933fcf6add7
e303dd5b5070988476faf5d4280cec54225690bb1db39a77eb4349e1181618bcab5a936caec63c718872f072b7722044e866bdb50a4bc12c0b3aee0aeeacba9f4af2d31036d097abb1722eedd8f6099d5153b30a1662836766a2aa
e629cf087fc499040440169c5271261f5172dbf872670ad9c3b75d2207b4c336d5ddf30f8f464197d762619901f955772e416f6134f3ac69661f9f8fe8811315f44a296590b5ffda5a5f9c68c8bcfd0fbb570
-----
TEST 1: Verify with original message
-----
4. Verifying signature...
SIGNATURE IS VALID
-----
TEST 2: Verify with tampered message
-----
Tampered message: 'all!'
(Added '!' at the end)
Tampered hash (hex): 9b97405b29994c8ec5226bf0da921d31d068e094598b22f02c338be2bb4a2994
```



```
IS_Assignment4_task3.py X
C: > Users > Yousaf Qazi > Desktop > Python tes > IS_Assignment4_task3.py > ...

1
2. Creating hash of message...
   Message: 'all!'
   SHA256 Hash (hex): 94419b99b12c11133a4dfeccc3e17885974beb48f7827c48239aafbcbad238d8

3. Signing message...
   Signature (hex): 55aaacd5d55a22a5ea2b597d3c652cf4b37238304c04f6f18770cd35aa142c7a060d2595850be95700b4f427176a215cc2da9fc146ced595930c20d7ec4677f9665b7b0b2e7a3a4918a8a3033fc-f6add7
   e303d5d5070908047f9f5d4280cc542256968b1db30a77eb4349e1181618bcab5a936caec63c718872f072b7722044e866bdb50a4bc12c8b3aee0aeacba9f4af2d3103c6d07abb1722ee6d48f6099d5153b30a1662830766a2aa
   e629cf0d7fc4990460440169c5271261f5172dbf872670ad9c3b75d2207b4c336d5ddf30f8f464197d762619901f955772e416f6134f3ac69661f9f8fe8811315f4d4a296590b5ffddaba5f9c68c8bcdfbb570

-----
TEST 1: Verify with original message
-----

4. Verifying signature...
   SIGNATURE IS VALID

-----
TEST 2: Verify with tampered message
-----
   Tampered message: 'all!'
   (Added '!' at the end)
   Tampered hash (hex): 9b97405b29994c8ec5226bf6da921d31d068e094598b22f02c338be2bb4a2994

4. Verifying signature...
   SIGNATURE IS INVALID

-----
SUMMARY:
1. Original message signature: VALID
2. Tampered message signature: INVALID
3. Digital signature ensures message integrity
-----
PS C:\Users\Yousaf Qazi>
```

Import Section

```
import sys
import hashlib
import binascii
```

- sys: For system operations (exiting program)
- hashlib: For creating cryptographic hashes
- binascii: For converting binary to hexadecimal display

Import with Error Handling

```
try:
    from cryptography.hazmat.primitives.asymmetric import rsa, padding
    from cryptography.hazmat.primitives import hashes
    from cryptography.exceptions import InvalidSignature
    print(" Cryptography library is installed")
except ImportError:
    print("\n Cryptography library is not installed!")
    print("Please install it using: pip install cryptography")
    print("Then run this script again.")
    sys.exit(1)
```

- Tries to import crypto library components
- Shows helpful message if library missing
- InvalidSignature exception is used for signature verification failures

Key Generation Function

```
def generate_keys():
    """Generate RSA keys for signing."""
    print("\n1. Generating RSA key pair...")
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()
    return private_key, public_key
```

- Generates 2048-bit RSA key pair (secure size)
- Uses standard public exponent 65537 (secure and efficient)
- Returns both private key (for signing) and public key (for verification)

Hash Creation Function

```
def create_hash(message):
    """Create SHA256 hash of message."""
    print("\n2. Creating hash of message...")
    sha256_hash = hashlib.sha256(message.encode()).digest()
    print(f" Message: '{message}'")
    print(f" SHA256 Hash (hex): {binascii.hexlify(sha256_hash).decode()}")
    return sha256_hash
```

- Creates SHA-256 hash of the message
- Shows message and its hash for demonstration
- Returns the hash as bytes

Signing Function

```
def sign_message(message, private_key):
    """Sign message with private key."""
    print("\n3. Signing message...")
    signature = private_key.sign(
        message.encode(),
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    print(f" Signature (hex): {binascii.hexlify(signature).decode()}")
    return signature
```

- Signs the message with the private key
- Uses PSS padding with SHA-256 (secure signing scheme)
- MGF1 is the Mask Generation Function
- Returns the signature as bytes

Verification Function

```
def verify_signature(message, signature, public_key):
    """Verify signature with public key."""
    print("\n4. Verifying signature...")
    try:
        public_key.verify(
            signature,
            message.encode(),
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        print(" SIGNATURE IS VALID")
        return True
    except InvalidSignature:
        print(" SIGNATURE IS INVALID")
        return False
```

- Tries to verify the signature
- If signature matches message: returns True
- If signature doesn't match: catches InvalidSignature exception, returns False

Main Function – Demonstration

```
def main():
    print("=" * 50)
    print("TASK 3: DIGITAL SIGNATURE")
    print("=" * 50)

    # Generate keys
    private_key, public_key = generate_keys()

    # Get message
    message = input("\nEnter message to sign: ").strip()
```

```

if not message:
    message = "Important document needs signing"

# Create hash
message_hash = create_hash(message)
    • Sets up and gets user input
# Sign message
signature = sign_message(message, private_key)

# Verify signature (should pass)
print("\n" + "-" * 50)
print("TEST 1: Verify with original message")
print("-" * 50)
verify_signature(message, signature, public_key)
    • Signs original message
    • Verifies it (should succeed)
# Test with tampered message (should fail)
print("\n" + "-" * 50)
print("TEST 2: Verify with tampered message")
print("-" * 50)
tampered_message = message + "!" # Add one character
print(f"  Tampered message: '{tampered_message}'")
print(f"  (Added '!' at the end)")

# Create hash of tampered message
tampered_hash = hashlib.sha256(tampered_message.encode()).digest()
print(f"  Tampered hash (hex): {binascii.hexlify(tampered_hash).decode()}")

# Try to verify tampered message
verify_signature(tampered_message, signature, public_key)
    • Shows tampering demonstration
    • Even small change ("!" added) completely changes hash
    • Verification should fail dramatically

print("\n" + "=" * 50)
print("SUMMARY:")
print("1. Original message signature: VALID")
print("2. Tampered message signature: INVALID")
print("3. Digital signature ensures message integrity")
print("=" * 50)
    • Shows final summary

```

SECURITY ANALYSIS

WHAT'S GOOD ABOUT THIS CODE

1. Uses Proper Signing Scheme

```
padding.PSS(  
    mgf=padding.MGF1(hashes.SHA256()),  
    salt_length=padding.PSS.MAX_LENGTH  
)
```

- Uses PSS padding (Probabilistic Signature Scheme)
- Adds randomness to signatures (same message → different signature)
- Protects against certain attacks

2. Strong Hash Function

```
hashes.SHA256()
```

- Uses SHA-256 (currently secure)
- Produces 256-bit hash (very hard to find collisions)

3. Secure Key Size

```
key_size=2048
```

- 2048-bit RSA is secure for signatures
- Standard for current applications

4. Proper Error Handling

```
except InvalidSignature:
```

```
    print(" SIGNATURE IS INVALID")
```

```
    return False
```

- Gracefully handles invalid signatures
- No crashes on verification failure

5. Good Demonstration

- Shows both success and failure cases
- Demonstrates how tampering is detected

LIMITATIONS TO KNOW

1. Signature Size

- RSA signatures are large (256 bytes for 2048-bit)
- Not efficient for many small messages

2. Performance

- RSA signing/verification is slower than symmetric crypto
- For high-volume systems, consider ECDSA (Elliptic Curve)

3. What Signatures DON'T Do

- Don't encrypt the message (signature is separate)
- Don't hide the message (message is visible)
- Don't prevent replay attacks (need timestamps/nonces)

REAL-WORLD USE CASES

Digital Signatures are used for:

1. Software updates (verify publisher)
2. Digital documents (PDF signatures)
3. SSL/TLS certificates (website security)

4. Email signing (PGP/GPG)
5. Blockchain transactions

HOW IT WORKS - SIMPLE ANALOGY

Think of a wax seal on an old letter:

1. Signing = Pressing your unique ring into hot wax
2. Message = The letter content
3. Signature = The wax seal imprint
4. Verification = Checking if imprint matches your ring
5. Tampering = If letter changed, wax breaks

SECURITY PROPERTIES PROVIDED

1. Authentication
 - Proves who created the message
 - "This message came from Alice's private key"
2. Integrity
 - Proves message wasn't changed
 - "This message is exactly what Alice signed"
3. Non-repudiation
 - Signer cannot deny they signed it
 - "Alice cannot claim she didn't sign this"

SIGNATURE vs ENCRYPTION

Digital Signature	Encryption
Proves source	Hides content
Anyone can verify	Only receiver can decrypt
Uses private key to sign	Uses public key to encrypt
Uses public key to verify	Uses private key to decrypt
Message is visible	Message is hidden

KEY TAKEAWAYS

1. Signatures prove authenticity - who sent it
2. Signatures prove integrity - not tampered with
3. Always use padding schemes (like PSS)
4. Use strong hash functions (SHA-256 or higher)
5. Public key verifies, private key signs

COMMON MISTAKES TO AVOID

1. Never sign untrusted data (could be malicious)
2. Never reuse keys for different purposes
3. Always verify before trusting
4. Store private keys securely

