COMSATS UNIVERSITY ISLAMABAD

Attock Campus

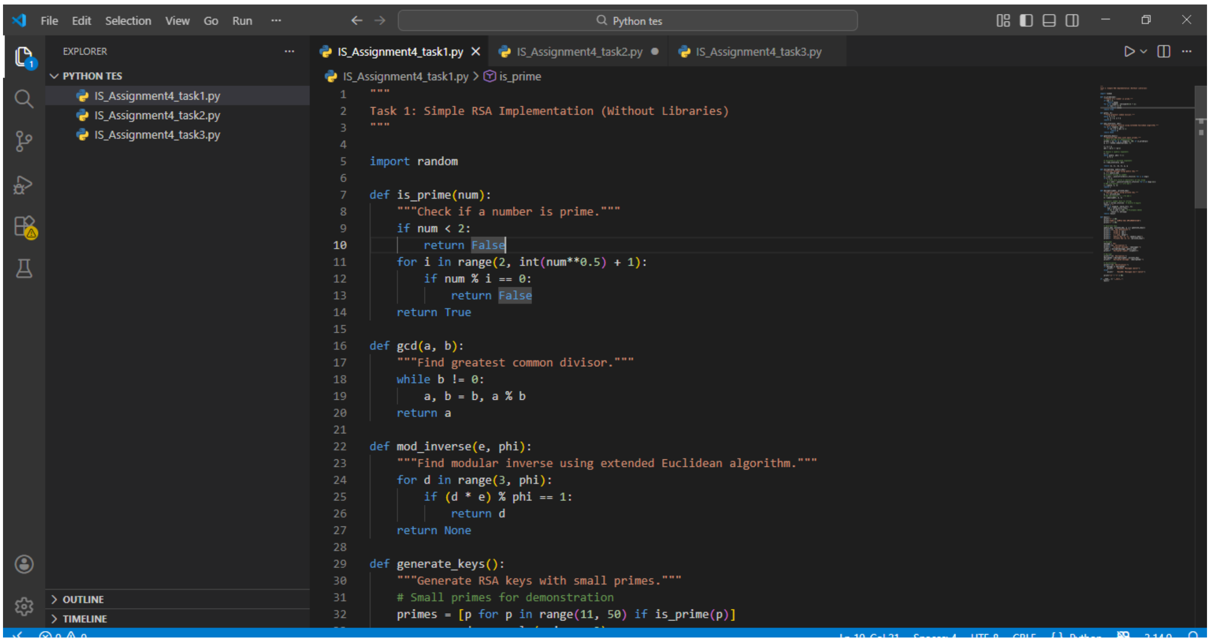
Department Of Computer Science

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

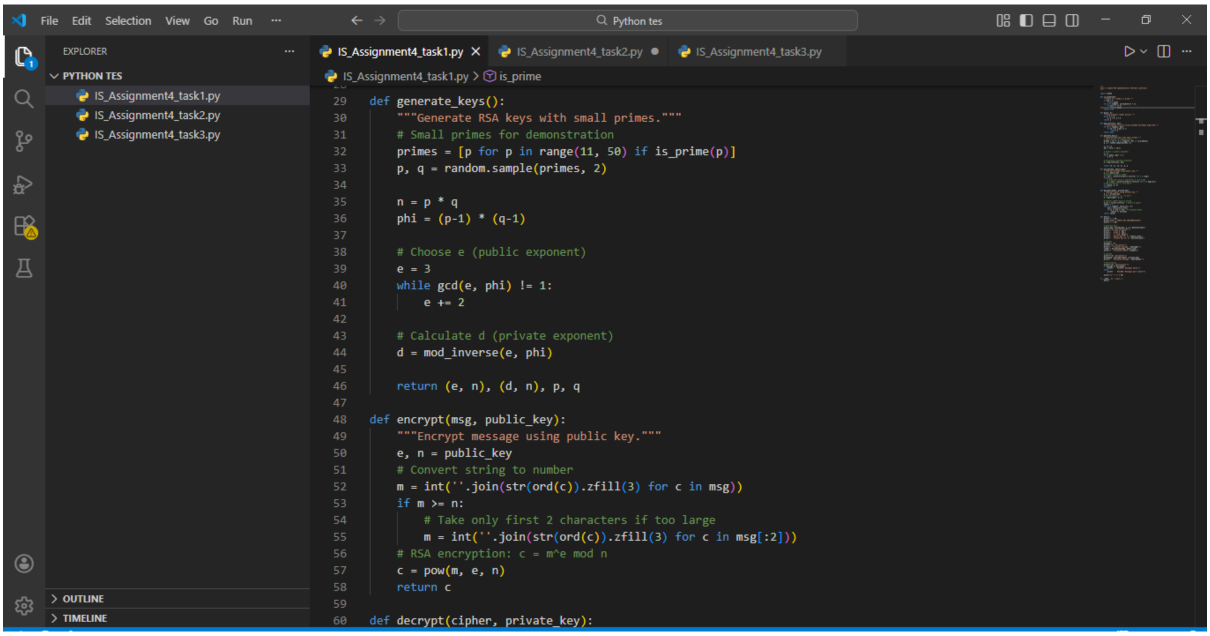
Submitted By:

Name	Registration No
Muhammad Yousaf Qazi	FA23-BSE-030

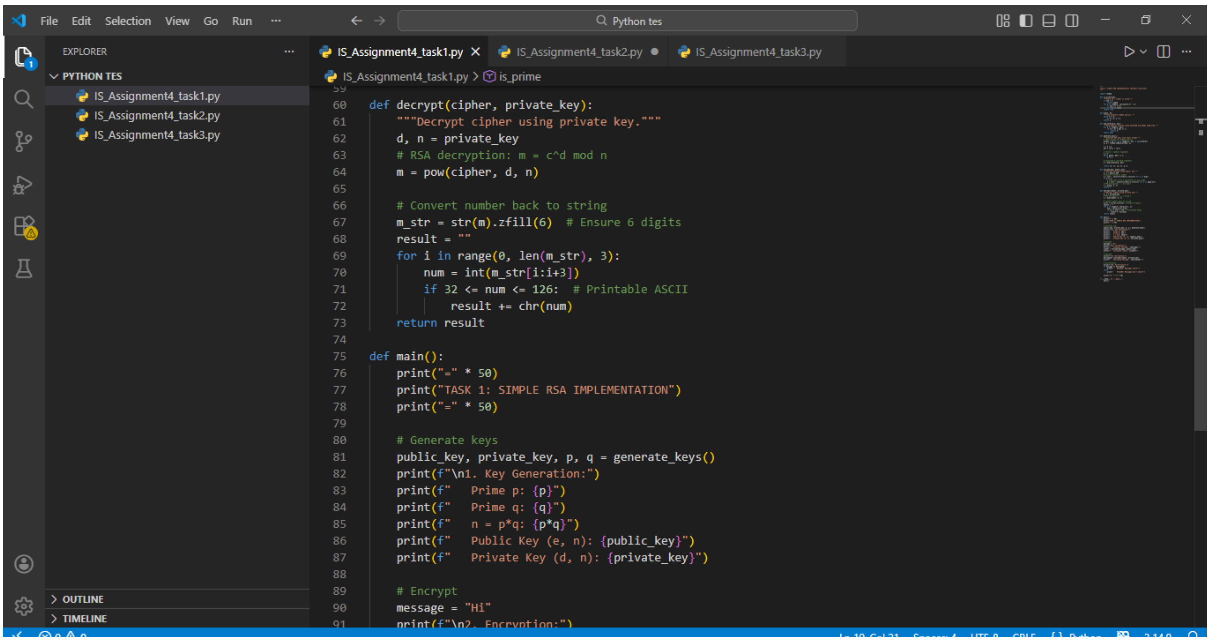
Code Explanation
Task No 1



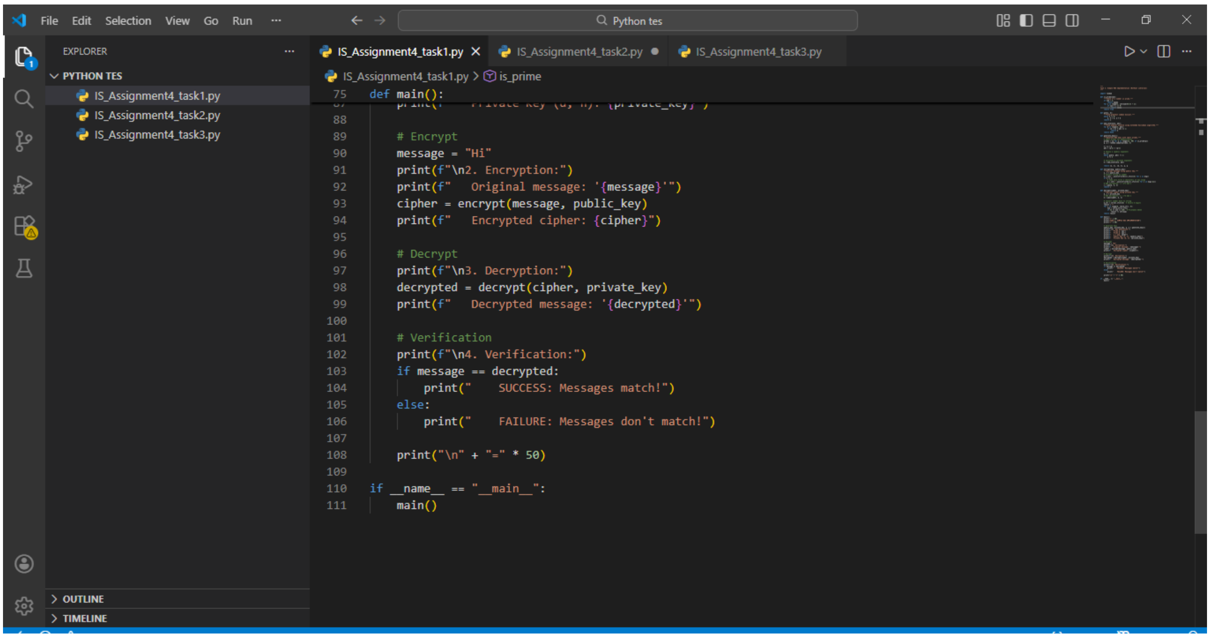
```python
"""
Task 1: Simple RSA Implementation (Without Libraries)
"""

import random

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

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

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

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)]
```



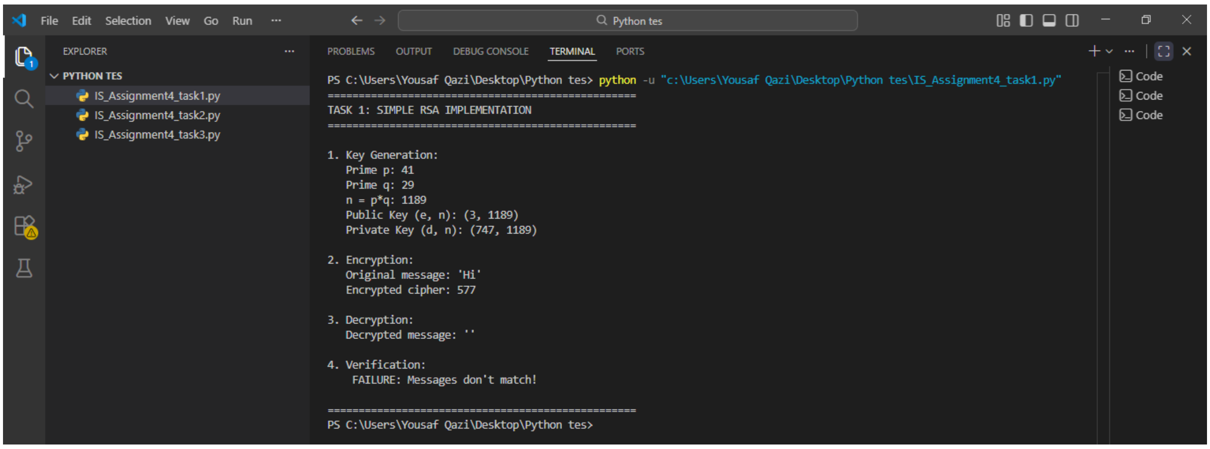
```python
def generate_keys():
    """Generate RSA keys with small primes."""
    # Small primes for demonstration
    primes = [p for p in range(11, 50) if is_prime(p)]
    p, q = random.sample(primes, 2)

    n = p * q
    phi = (p-1) * (q-1)

    # 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

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))
    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 mod n
    c = pow(m, e, n)
    return c

def decrypt(cipher, private_key):
```



```python
def decrypt(cipher, private_key):
    """Decrypt cipher using private key."""
    d, n = private_key
    # RSA decryption: m = c^d mod n
    m = pow(cipher, d, 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

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}")

    # Encrypt
    message = "Hi"
    print(f"\n2. Encryption:")
```



```python
def main():

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

    # 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)

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



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 √num (efficient)
- Line 8: int(num**0.5) + 1 calculates square root + 1 for inclusive range
- Line 9-10: If divisible by any i, not prime → 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 φ-1 (brute-force approach)
- Line 23-24: Checks if (d * e) % φ == 1 (modular inverse condition)
- Line 25: Returns None if no inverse found

Note: This is inefficient (O(φ)) - 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: φ(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, φ)=1
- Line 43: Calculates d = e^{-1} mod φ (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 mod n
    c = pow(m, e, n)
    return c
```

- Line 54-57: If message integer ≥ 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 mod 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:
```

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

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

- Decrypts and verifies correctness

Entry Point

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

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

SECURITY ANALYSIS

1. CRITICAL VULNERABILITIES

A. Weak Prime Generation

```python
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*q where p,q < 50 → n < 2500
- Attack: Can factor n instantly using trial division

C. Weak Padding/Encoding

```python
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

```python
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 $O(\varphi)$ - 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 <= num <= 126:
```

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

3. EFFICIENCY ISSUES

Prime Checking Algorithm

```
for i in range(2, int(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
```

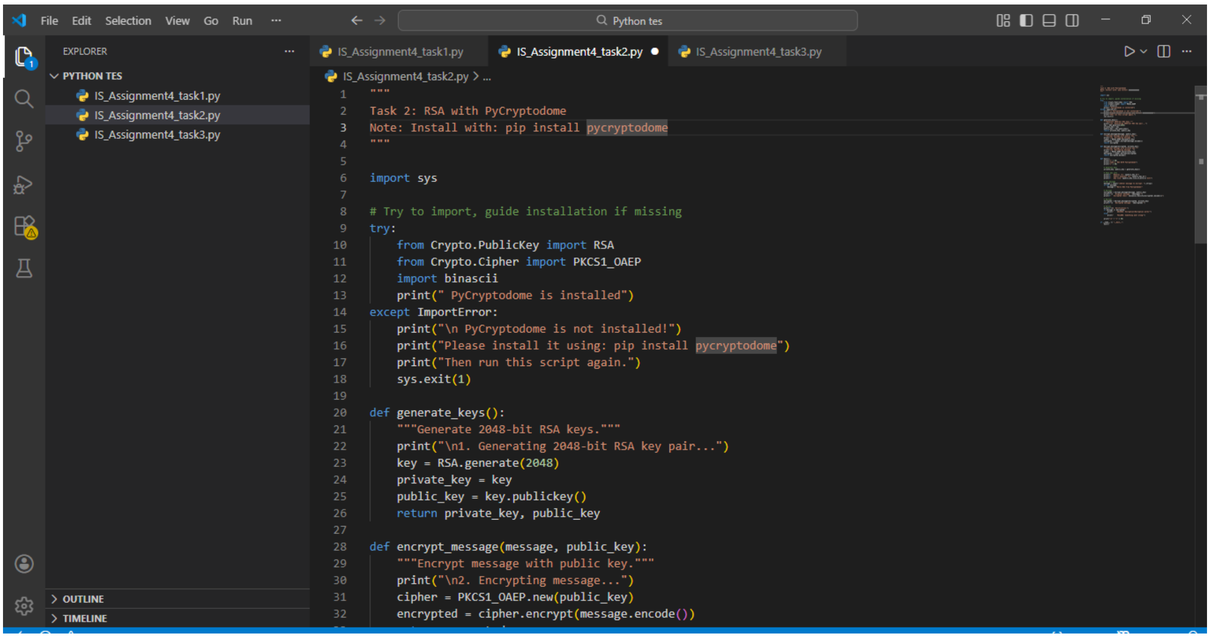
3. If implementing for education:
    - Use larger primes (100+ digits)
    - Implement proper padding
    - Use extended Euclidean algorithm
    - Add randomization

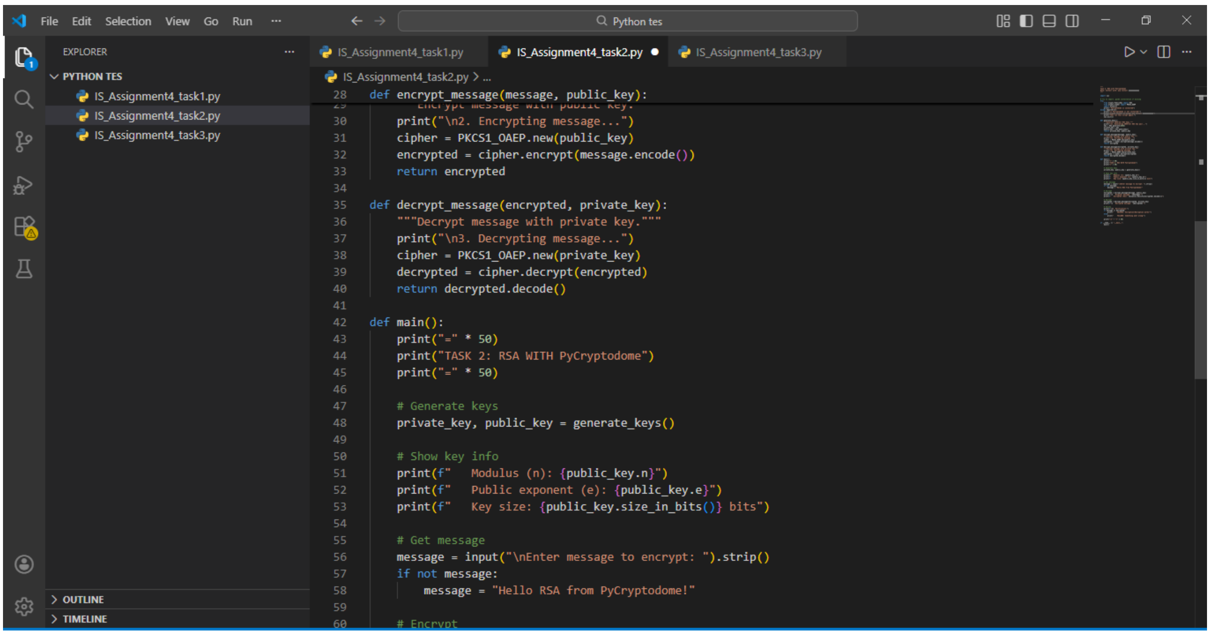
7. ATTACKS POSSIBLE ON THIS IMPLEMENTATION

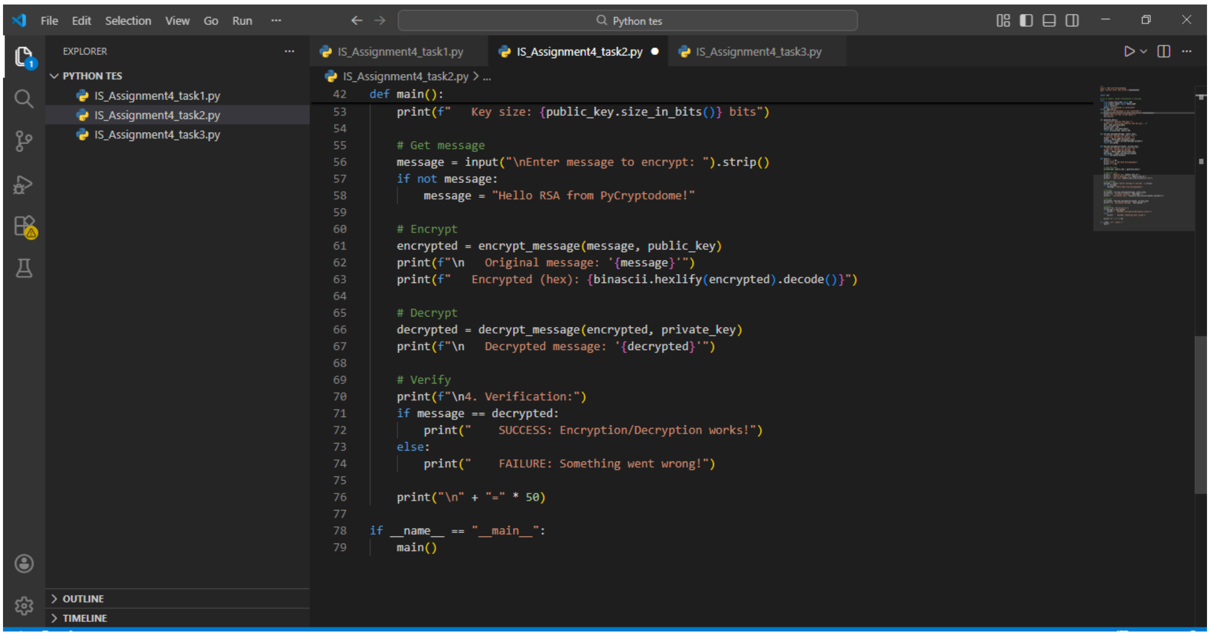
1. Factorization attack: Instant (n too small)
2. Brute-force private key: $d < \varphi$ (tiny)
3. Chosen-plaintext attack: No randomization
4. Timing attacks: pow() may leak information
5. Man-in-the-middle: No authentication

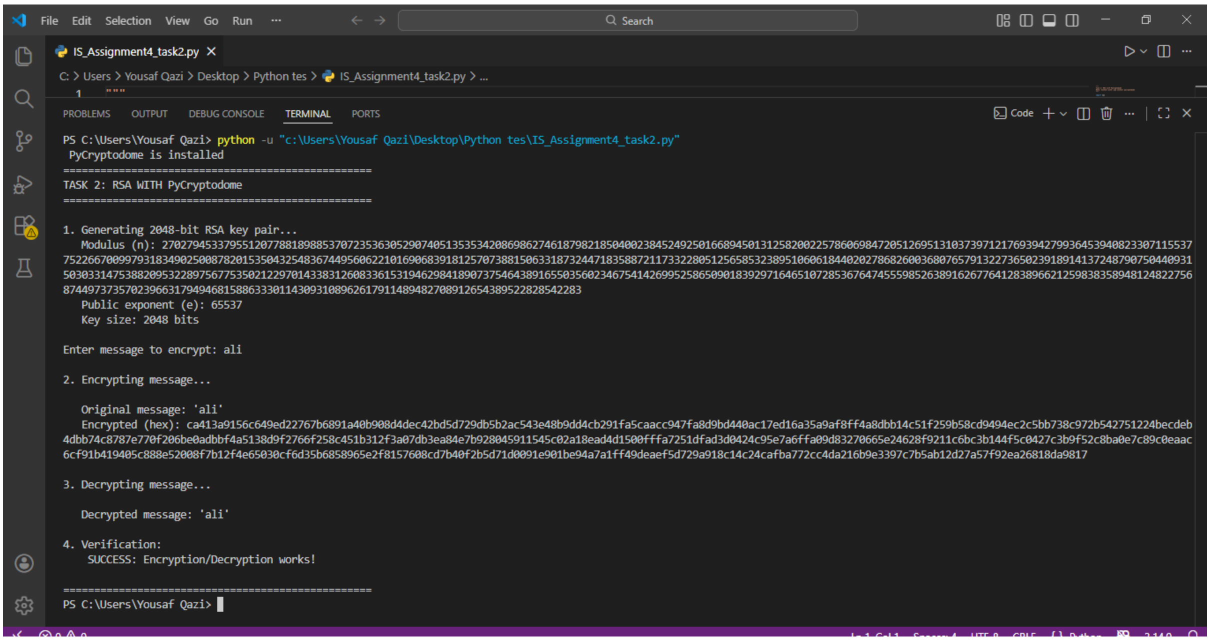
CONCLUSION: This is a educational demonstration ONLY. It demonstrates RSA concepts but contains multiple critical vulnerabilities making it completely insecure for real-world use. Never use this code for actual encryption.

Task No 2


File Edit Selection View Go Run
Python tes
EXPLORER
PYTHON TES
IS_Assignment4_task1.py
IS_Assignment4_task2.py
IS_Assignment4_task3.py
OUTLINE
TIMELINE
IS_Assignment4_task2.py > ...
"""
Task 2: RSA with PyCryptodome
Note: Install with: pip install pycryptodome
"""
import sys
# Try to import, guide installation if missing
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)
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
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())


File Edit Selection View Go Run
Python tes
EXPLORER
PYTHON TES
IS_Assignment4_task1.py
IS_Assignment4_task2.py
IS_Assignment4_task3.py
OUTLINE
TIMELINE
IS_Assignment4_task2.py > ...
def encrypt_message(message, public_key):
print("\n2. Encrypting message...")
cipher = PKCS1_OAEP.new(public_key)
encrypted = cipher.encrypt(message.encode())
return encrypted
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()
def main():
print("=" * 50)
print("TASK 2: RSA WITH PyCryptodome")
print("=" * 50)
# 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")
# Get message
message = input("\nEnter message to encrypt: ").strip()
if not message:
message = "Hello RSA from PyCryptodome!"





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
```

```python
    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

```python
    # 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

```python
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)

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

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

2.Proper Padding (OAEP)

```python
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 ≈ 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	~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)

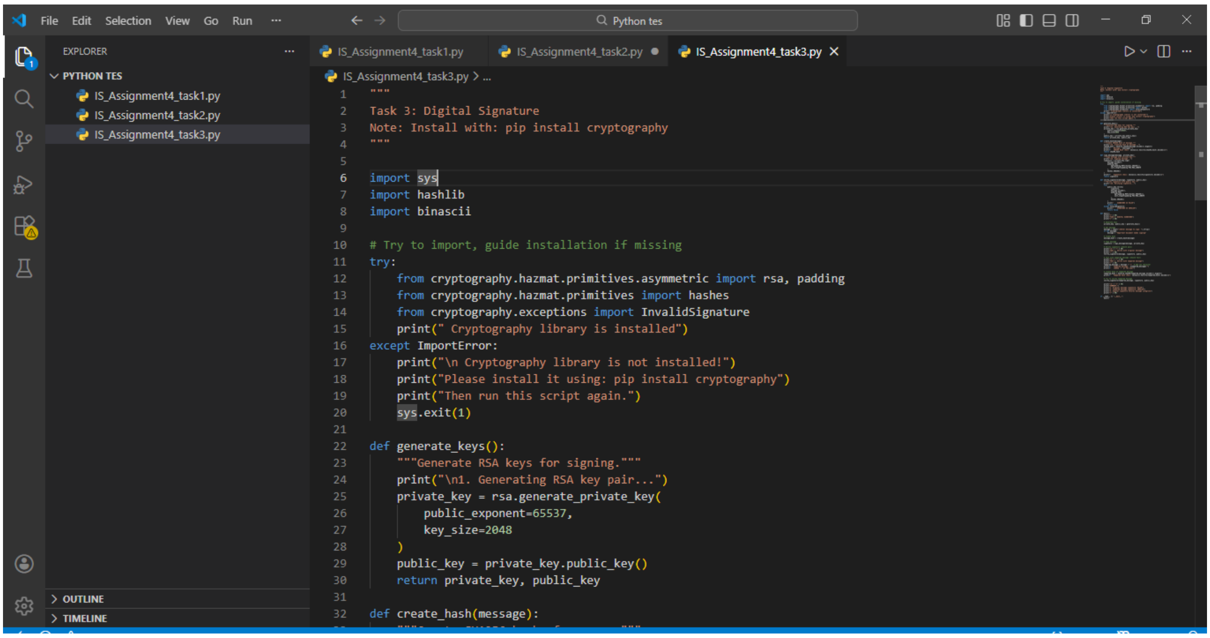
IMPORTANT WARNING

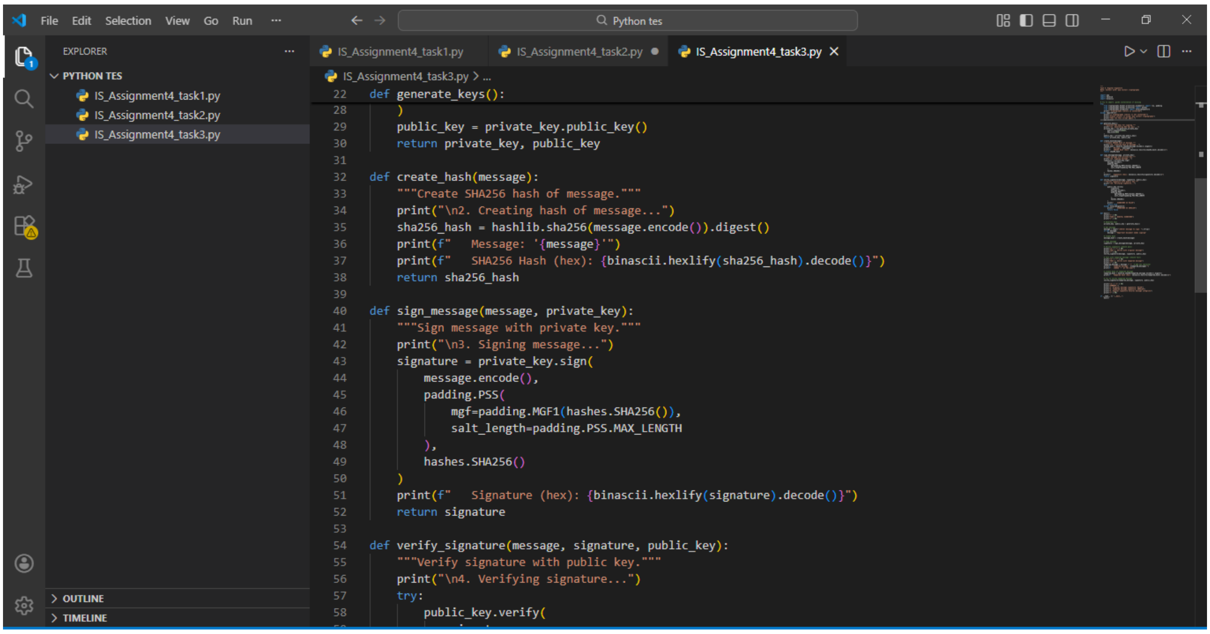
Even though this code is more secure than Task 1:

- 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

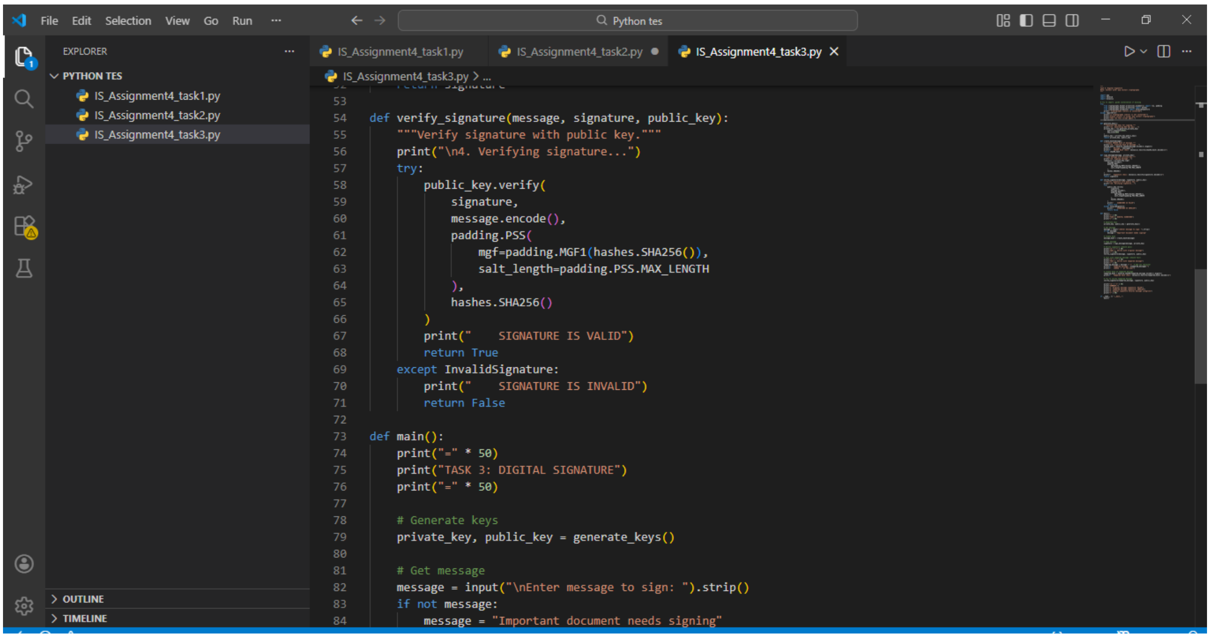



File Edit Selection View Go Run ···
Python tes
EXPLORER
PYTHON TES
IS_Assignment4_task1.py
IS_Assignment4_task2.py
IS_Assignment4_task3.py
OUTLINE
TIMELINE
IS_Assignment4_task3.py > ...
def generate_keys():
)
    public_key = private_key.public_key()
    return private_key, public_key

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

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

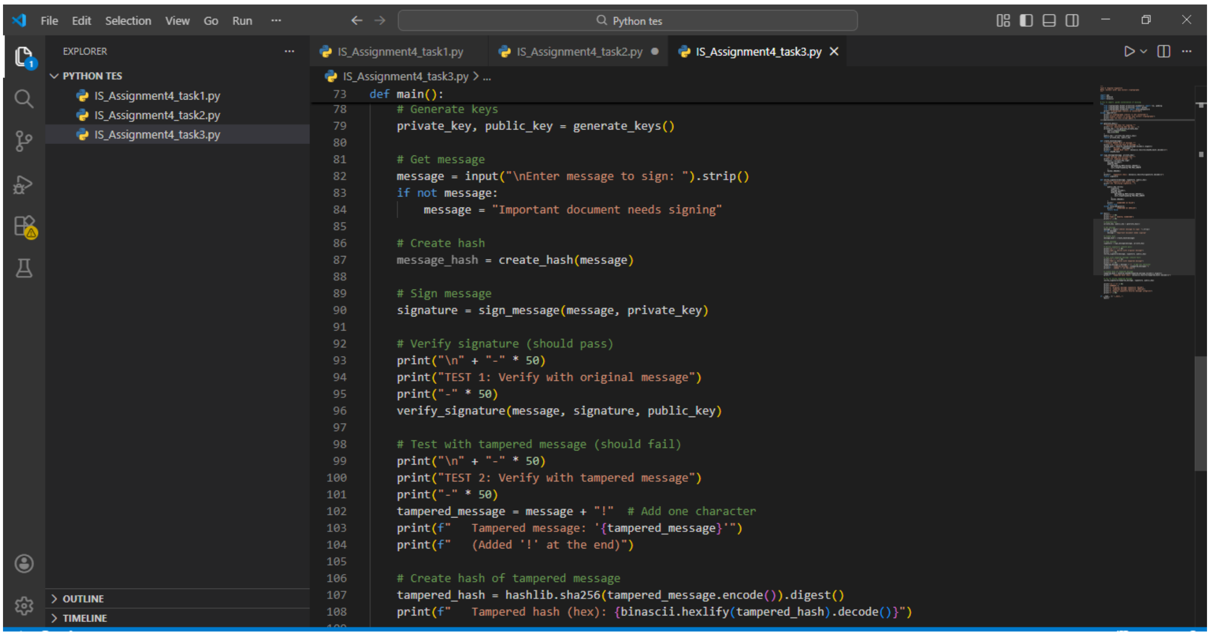
def verify_signature(message, signature, public_key):
    """Verify signature with public key."""
    print("\n4. Verifying signature...")
    try:
        public_key.verify(

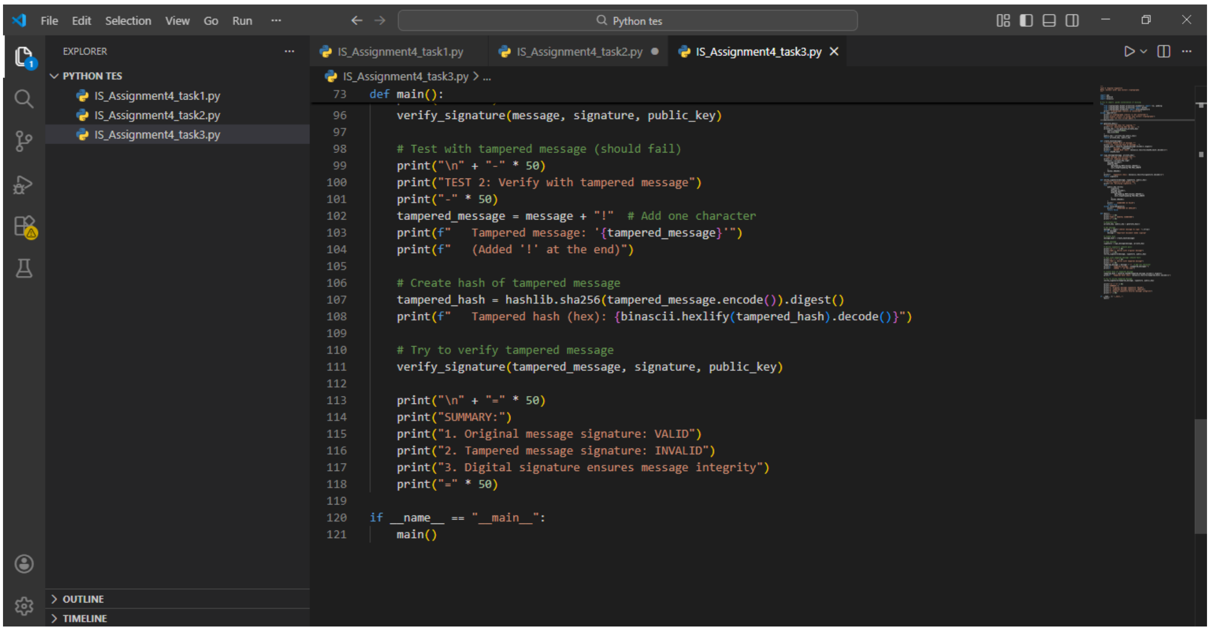

File Edit Selection View Go Run ···
Python tes
EXPLORER
PYTHON TES
IS_Assignment4_task1.py
IS_Assignment4_task2.py
IS_Assignment4_task3.py
OUTLINE
TIMELINE
IS_Assignment4_task3.py > ...
def verify_signature(message, signature, public_key):
    """Verify signature with public key."""
    print("\n4. Verifying 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

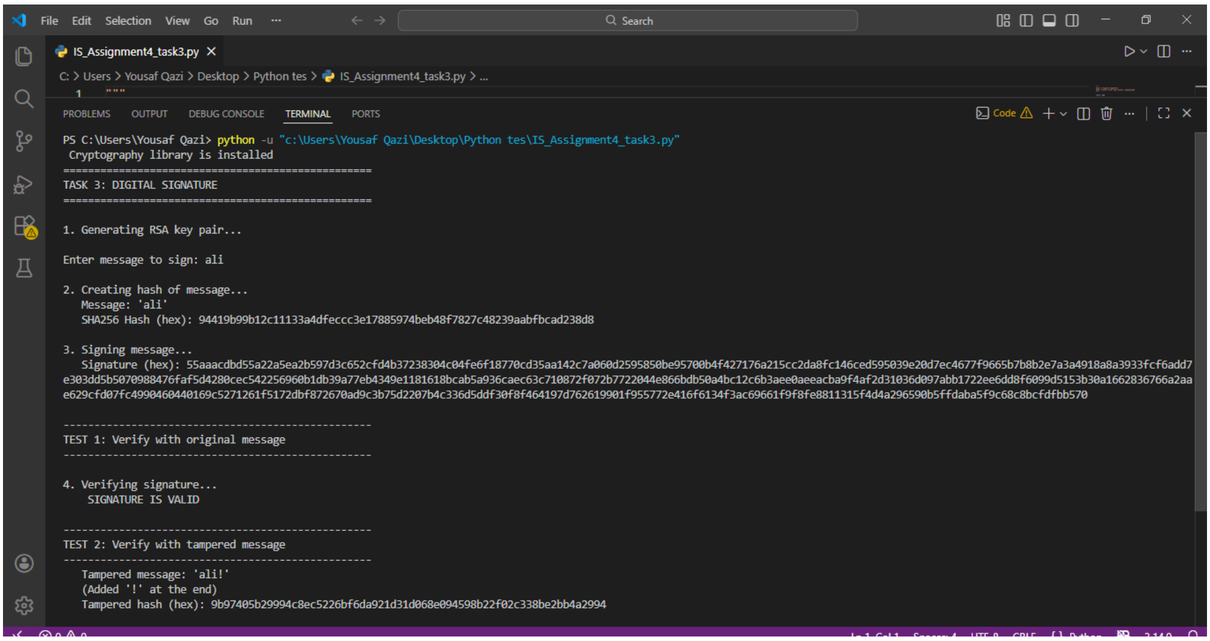
def main():
    print("=" * 50)
    print("TASK 3: DIGITAL SIGNATURE")
    print("=" * 50)

    # 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"


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 > ...
def main():
    # 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)
    # 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)
    # 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()}")
OUTLINE
TIMELINE

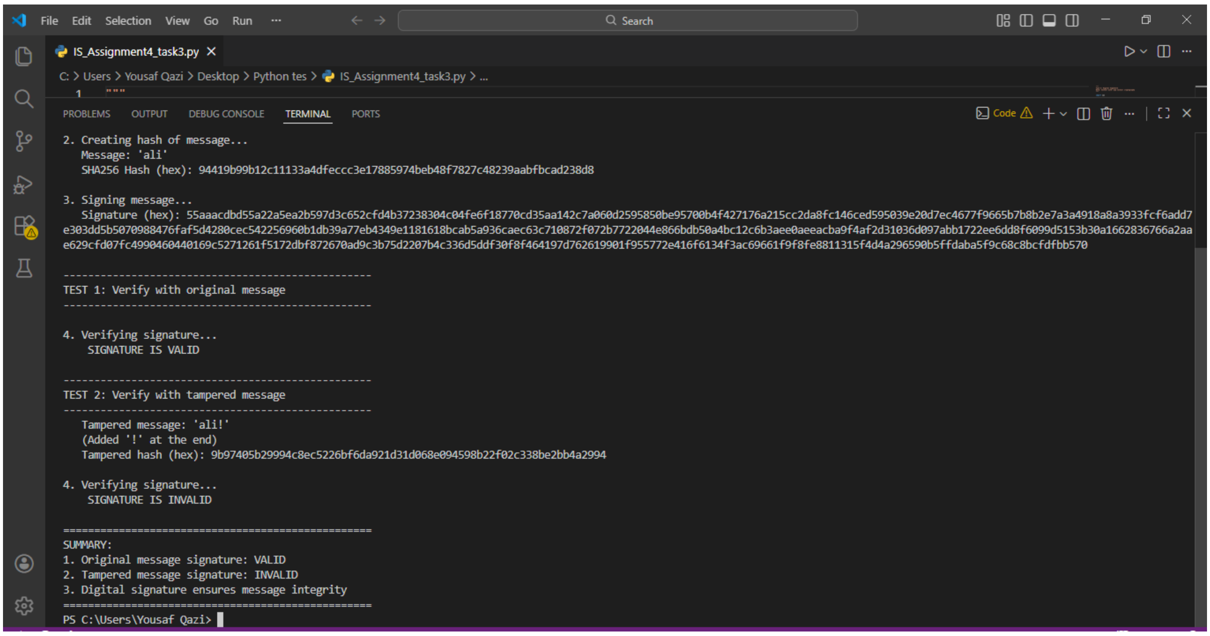

File Edit Selection View Go Run
Python tes
EXPLORER
PYTHON TES
IS_Assignment4_task1.py
IS_Assignment4_task2.py
IS_Assignment4_task3.py
IS_Assignment4_task3.py > ...
def main():
    verify_signature(message, signature, public_key)
    # Test with tampered message (should fail)
    print("\n" + "-" * 50)
    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)
    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)
if __name__ == "__main__":
    main()
OUTLINE
TIMELINE


File Edit Selection View Go Run
IS_Assignment4_task3.py
C: > Users > Yousaf Qazi > Desktop > Python tes > IS_Assignment4_task3.py > ...
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: ali
2. Creating hash of message...
Message: 'ali'
SHA256 Hash (hex): 94419b99b12c11133a4dfeccc3e17885974beb48f7827c48239aabfbcad238d8
3. Signing message...
Signature (hex): 55aaacdbd55a22a5ea2b597d3c652cfd4b37238304c04fe6f18770cd35aa142c7a060d2595850be95700b4f427176a215cc2da8fc146ced595039e20d7ec4677f9665b7b8b2e7a3a4918a8a3933fcf6add7
e303dd5b5070988476faf5d4280cec542256960b1db39a77eb4349e1181618bcab5a936caec63c710872f072b7722044e866bdb50a4bc12c6b3aee0aeeacba9f4af2d31036d097abb1722ee6dd8f6099d5153b30a1662836766a2aa
e629cfd07fc4990460440169c5271261f5172dbf872670ad9c3b75d2207b4c336d5ddf30f8f464197d762619901f955772e416f6134f3ac69661f9f8fe8811315f4d4a296590b5ffdaba5f9c68c8bcfdfbb570

TEST 1: Verify with original message
----
4. Verifying signature...
SIGNATURE IS VALID

TEST 2: Verify with tampered message

Tampered message: 'ali!'
(Added '!' at the end)
Tampered hash (hex): 9b97405b29994c8ec5226bf6da921d31d068e094598b22f02c338be2bb4a2994



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