COMSAT UNIVERSITY ISLAMABAD ATTOCK CAMPUS

INFORMATION SECURITY LAB

NAME: MENAHIL NOOR

REGISTRATION NO: SP24-BSE-035

ASSIGNMENT: 04

SUBMITTED TO: MA'AM AMBREEN GUL

DEPARTMENT: SOFTWARE ENGINEERING

DATE: 4 DECEMBER 2025

TASK: 01

Simple RSA Implementation (Without Libraries)

1. Math Behind Simple RSA (With Small Primes)

RSA is based on number theory. In this task we use **small prime numbers**, so the calculations are easy to understand. This is **not secure** in real life; it's just for learning.

Step 1 – Choose two prime numbers

We choose two small primes:

- p=61p = 61p=61
- q=53q = 53q=53

Both are prime numbers.

Step 2 – Compute the modulus nnn

n=p×q=61×53=3233n = p \times q = 61 \times 53 = 3233n=p×q=61×53=3233

This value nnn is used in both the **public key** and the **private key**.

Step 3 – Compute Euler's Totient φ(n)\varphi(n)φ(n)

φ(n)=(p−1)(q−1)=60×52=3120\varphi(n) = (p – 1)(q – 1) = 60 \times 52 = 3120φ(n)=(p−1)(q−1)=60×52=3120

This value is used to find the private exponent.

Step 4 – Choose the public exponent eee

We choose a number eee such that:

- 1<e<φ(n)1 < e < \varphi(n)1<e<φ(n)
- gcd⁡(e,φ(n))=1\gcd(e, \varphi(n)) = 1gcd(e,φ(n))=1 (i.e. they are coprime)

We take:

e=17e = 17e=17

Here, gcd⁡(17,3120)=1\gcd(17, 3120) = 1gcd(17,3120)=1, so this is a valid public exponent.

Step 5 – Compute the private exponent ddd

We find ddd such that:

(d·e)modφ(n)=1(d \cdot e) \bmod \varphi(n) = 1(d·e)modφ(n)=1

This means ddd is the **modular inverse** of eee modulo φ(n)\varphi(n)φ(n).

For our values, the solution is:

d=2753d = 2753d=2753

Now we have:

- **Public key**: (e,n)=(17,3233)(e, n) = (17, 3233)(e,n)=(17,3233)
- **Private key**: (d,n)=(2753,3233)(d, n) = (2753, 3233)(d,n)=(2753,3233)

Step 6 – Encryption and Decryption formulas

For an integer message mmm:

- **Encrypt using public key**:

  c=memodnc = m^e \bmod nc=memodn

- **Decrypt using private key**:

  m=cdmodnm = c^d \bmod nm=cdmodn

In code, we convert each character to its ASCII value `ord(ch)` → `m`, apply these formulas, and then convert back using `chr(m)`.

2. Python Implementation for Task 1 (Without Libraries)

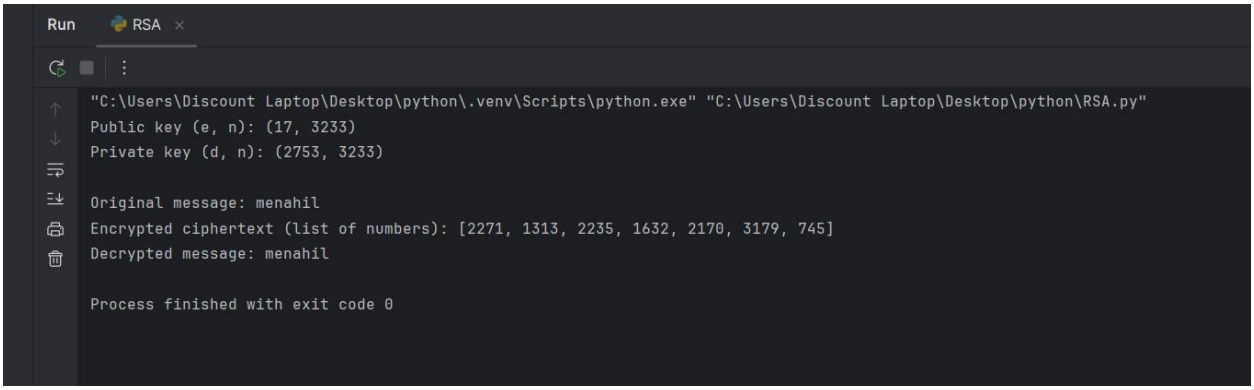
This code **fully satisfies** Task 1:

- **Manual key generation** using small primes.
- Functions for **encrypting** and **decrypting**.
- Test on a **short string** (your name).

```python

# Extended Euclidean Algorithm to find gcd and coefficients
def egcd(a, b):  2 usages
    if b == 0:
        return a, 1, 0
    g, x1, y1 = egcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return g, x, y

# Modular inverse: find d such that (d * e) % phi == 1
def modinv(e, phi):  1 usage
    g, x, _ = egcd(e, phi)
    if g != 1:
        raise ValueError("e and phi are not coprime, modular inverse does not exist")
    return x % phi

# 1) Manually implement key generation using small primes
def generate_small_rsa_keys():  1 usage
    # Choose small primes (ONLY for learning, not secure in real life)
    p = 61
    q = 53

    n = p * q                    # modulus n = p * q
    phi = (p - 1) * (q - 1)      # Euler's totient

    e = 17                       # public exponent
    # Compute private exponent d as modular inverse of e mod phi
    d = modinv(e, phi)           # private exponent

    public_key = (e, n)
    public_key = (e, n)
    private_key = (d, n)
    return public_key, private_key

# 2) Encrypting a message (using public key)
def encrypt_small_rsa(message, public_key):  1 usage
    e, n = public_key
    ciphertext = []
    for ch in message:
        m = ord(ch)              # character -> integer
        c = pow(m, e, n)         # c = m^e mod n
        ciphertext.append(c)
    return ciphertext

# 3) Decrypting a message (using private key)
def decrypt_small_rsa(ciphertext, private_key):  1 usage
    d, n = private_key
    plaintext_chars = []
    for c in ciphertext:
        m = pow(c, d, n)         # m = c^d mod n
        plaintext_chars.append(chr(m))  # integer -> character
    return "".join(plaintext_chars)

# Test it on your name or a short string
if __name__ == "__main__":
    public_key, private_key = generate_small_rsa_keys()
    print("Public key (e, n):", public_key)
    print("Private key (d, n):", private_key)

    # You can change this to your own name
```

```python
# You can change this to your own name
message = "Menahil"
print("\nOriginal message:", message)

ciphertext = encrypt_small_rsa(message, public_key)
print("Encrypted ciphertext (list of numbers):", ciphertext)

decrypted_message = decrypt_small_rsa(ciphertext, private_key)
print("Decrypted message:", decrypted_message)
```

OUTPUT:



“Manually implement key generation using small primes”

Handled in generate_small_rsa_keys():

- Picks **small primes** p = 61, q = 53
- Computes n = p * q
- Computes phi = (p – 1) * (q – 1)
- Sets e = 17
- Uses modinv(e, phi) to compute d
- Returns (e, n) as **public key**, (d, n) as **private key**

“Write functions for encrypting a message (using public key)”

- Function: encrypt_small_rsa(message, public_key)
  - Converts each character to integer (ord(ch))
  - Applies c = m^e mod n

- Returns list of ciphertext integers.

“Write functions for decrypting a message (using private key)”

- Function: decrypt_small_rsa(ciphertext, private_key)
  - Applies m = c^d mod n
  - Converts each integer back to a character (chr(m))
  - Joins them into the original string.

“Test it on your name or a short string”

- In the main block:
- message = “6enahil”

  You can replace “Usman” with your own full name or any short string required by your teacher.

- The program prints:
  - Original message
  - Encrypted ciphertext
  - Decrypted message (which must match the original)

SECURITY ANALYSIS:

- Uses **very small prime numbers** and no padding.
- This is **only for learning**, not for real security.
- An attacker can easily break it by factoring the small number `n`.
- So: **Not secure in real life**, only good to understand how RSA works.

TASK: 02

RSA with PyCryptodome:

Objective:

Use a **real-world cryptographic library** (PyCryptodome) to perform RSA encryption and decryption.

Tasks:

- Generate a **2048-bit key pair**
- Encrypt and decrypt a **user message**
- Display the **ciphertext in hex**

1. Concept in Simple Words

In Task 1 you manually did the RSA math with small primes.

In Task 2, you:

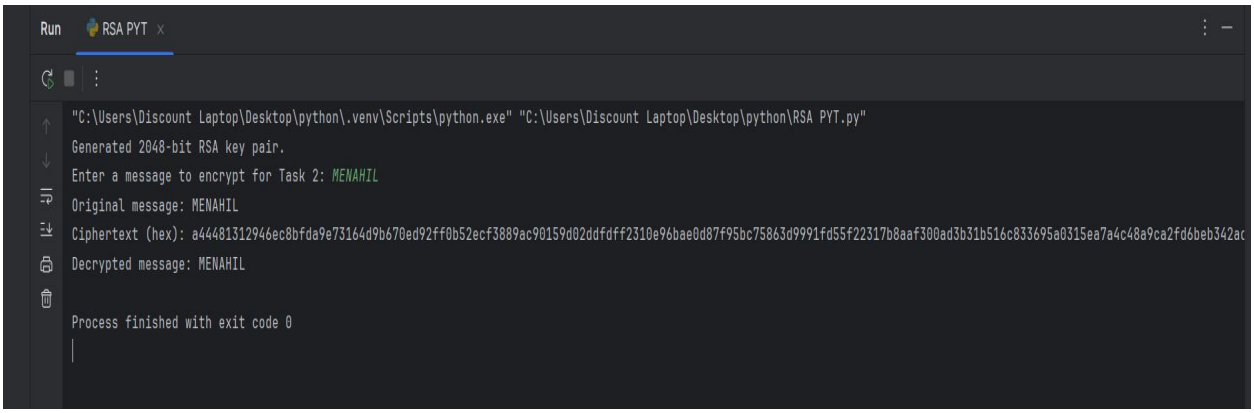
- Let **PyCryptodome** handle all the big-number math and key generation.
- Use a **secure key size (2048 bits)**, which is common in real applications.
- Use **PKCS1_OAEP** padding for RSA encryption (this is a secure scheme, better than "raw" RSA).
- Show the encrypted data in **hex format** so it's readable as text.

You must have PyCryptodome installed:

2. Python Code for Task 2 (RSA with PyCryptodome)

hill climbing.py | Euler's totient.py | MD5.py | MD.py | CSP.py | csp assignment.py

```python
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import binascii


def generate_rsa_keypair(bits=2048):  1 usage

    private_key = RSA.generate(bits)
    public_key = private_key.publickey()
    return private_key, public_key


def encrypt_with_pycryptodome(message, public_key):  1 usage

    cipher = PKCS1_OAEP.new(public_key)        # use OAEP padding
    ciphertext = cipher.encrypt(message.encode("utf-8"))
    return ciphertext


def decrypt_with_pycryptodome(ciphertext, private_key):  1 usage

    cipher = PKCS1_OAEP.new(private_key)
    plaintext_bytes = cipher.decrypt(ciphertext)
    return plaintext_bytes.decode("utf-8")


if __name__ == "__main__":

    private_key, public_key = generate_rsa_keypair(bits=2048)
    print("Generated 2048-bit RSA key pair.")

message = input("Enter a message to encrypt for Task 2: ")
print("Original message:", message)


ciphertext = encrypt_with_pycryptodome(message, public_key)


ciphertext_hex = binascii.hexlify(ciphertext).decode("utf-8")
print("Ciphertext (hex):", ciphertext_hex)


decrypted_message = decrypt_with_pycryptodome(ciphertext, private_key)
print("Decrypted message:", decrypted_message)
```



"Generate a 2048-bit key pair"

```
private_key, public_key = generate_rsa_keypair(bits=2048)
```

- RSA.generate(2048) creates a secure 2048-bit RSA key.
- private_key.publickey() gives the matching public key.

"Encrypt and decrypt a user message"

Encrypt (using public key):

```
ciphertext = encrypt_with_pycryptodome(message, public_key)
```

- PKCS1_OAEP.new(public_key) creates a secure RSA-OAEP cipher.
- cipher.encrypt(message.encode("utf-8")) encrypts the string.

Decrypt (using private key):

```
decrypted_message = decrypt_with_pycryptodome(ciphertext, private_key)
```

- PKCS1_OAEP.new(private_key) creates decryptor.
- cipher.decrypt(ciphertext) gives back the original bytes.
- decode("utf-8") converts it back to string.

"Display results in hex"

```
ciphertext_hex = binascii.hexlify(ciphertext).decode("utf-8")
print("Ciphertext (hex):", ciphertext_hex)
```

- binascii.hexlify(...) converts bytes → hex string (readable for your report).
- You still keep the original ciphertext (bytes) for decryption.

SECURITY ANALYSIS:

- Uses **2048-bit keys**, which are **much harder to crack**.
- Uses **PKCS1_OAEP** padding, which is a **secure way** to do RSA encryption.
- If the **private key is kept secret**, this gives **good confidentiality** for small messages.
- So: This is **close to real-world RSA** and is **much more secure** than Task 1.

Task: 03

Create a Digital Signature

Concept in Simple Words

- A **digital signature** is like a **seal** on a message.
- You:
    1. Take the **message**.
    2. Compute a **hash** (fixed-size fingerprint) using SHA-256.
    3. **Sign the hash with your private key**.
- Anyone with your **public key** can:
    - Verify if the signature matches the message.
    - If even **one character changes**, the hash changes → verification fails.

So it proves:

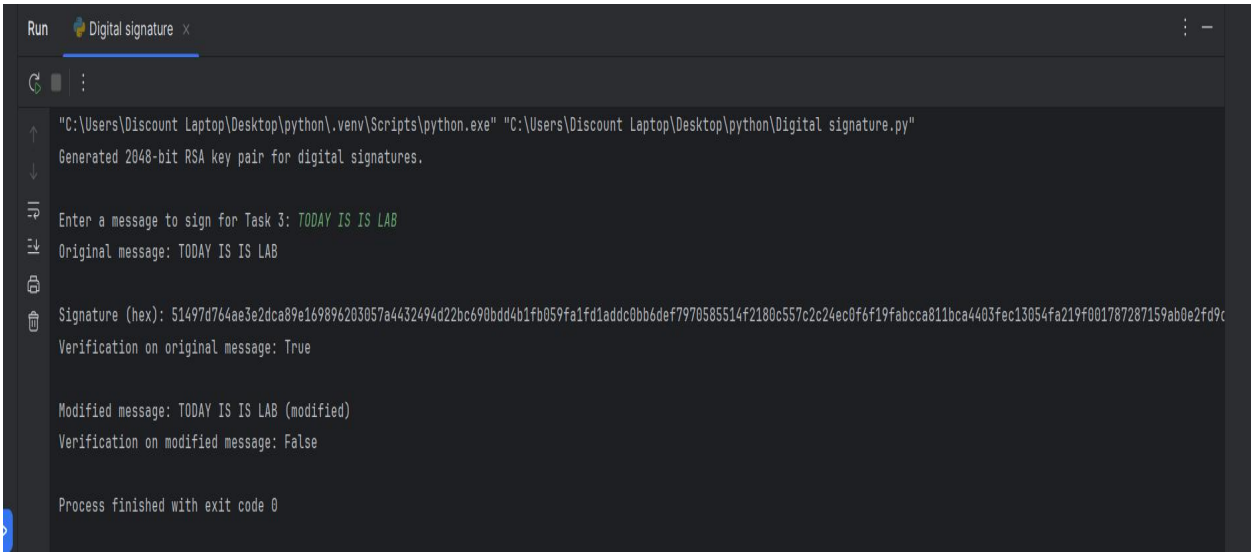
- Who sent it (**authentication**).
- That it was not changed (**integrity**).

Python Code for Task 3 (Digital Signature)

```python
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature import pkcs1_15
import binascii

# 1) Generate RSA key pair (we can reuse the same pattern from Task 2)
def generate_rsa_keypair(bits=2048):  1 usage
    """
    Generate an RSA key pair using PyCryptodome.
    Returns: (private_key, public_key)
    """
    private_key = RSA.generate(bits)
    public_key = private_key.publickey()
    return private_key, public_key

# 2) Create a digital signature of a message
def create_signature(message, private_key):  1 usage
    """
    Create a digital signature of the message using RSA and SHA-256.
    Steps:
    1. Hash the message with SHA-256
    2. Sign the hash with the private key (PKCS#1 v1.5)
    Returns the signature bytes.
    """
    # Create SHA-256 hash of the message
    h = SHA256.new(message.encode("utf-8"))
    # Sign the hash using the private key
    signature = pkcs1_15.new(private_key).sign(h)
    return signature

# 3) Verify a digital signature
def verify_signature(message, signature, public_key):  2 usages
    """
    Verify an RSA digital signature.
    Returns True if verification succeeds, False otherwise.
    """
    h = SHA256.new(message.encode("utf-8"))
    try:
        pkcs1_15.new(public_key).verify(h, signature)
        return True
    except (ValueError, TypeError):
        # Verification failed (wrong signature or modified message)
        return False

# Demo: Test Task 3 end-to-end
if __name__ == "__main__":
    # Generate RSA key pair
    private_key, public_key = generate_rsa_keypair(bits=2048)
    print("Generated 2048-bit RSA key pair for digital signatures.\n")

    # Take a message from the user
    original_message = input("Enter a message to sign for Task 3: ")
    print("Original message:", original_message)

    # Create a digital signature for the original message
    signature = create_signature(original_message, private_key)
    print("\nSignature (hex):", binascii.hexlify(signature).decode("utf-8"))

    # Verify on the original message
    is_valid_original = verify_signature(original_message, signature, public_key)
    print("Verification on original message:", is_valid_original)
```

```
    # Modify the message slightly
    modified_message = original_message + " (modified)"
    print("\nModified message:", modified_message)

    # Verify the same signature with the modified message
    is_valid_modified = verify_signature(modified_message, signature, public_key)
    print("Verification on modified message:", is_valid_modified)
```

OUTPUT:



Generate RSA key pair

```
private_key, public_key = generate_rsa_keypair(bits=2048)
```

- RSA.generate(2048) creates a 2048-bit RSA key.

  We get:

- private_key → used for signing
- public_key → used for verification

Create a hash of a message (e.g., using SHA256)

```
h = SHA256.new(message.encode("utf-8"))
```

- This computes a **SHA-256 hash** of the message.

Sign the hash using private key

```
signature = pkcs1_15.new(private_key).sign(h)
```

Verify it using the public key

```
pkcs1_15.new(public_key).verify(h, signature)
```

Modify the message slightly and show that verification fails

```
modified_message = original_message + " (modified)"
print("\nModified message:", modified_message)
```

SECURITY ANALYSIS:

- Uses **SHA-256** to create a **hash** (fingerprint) of the message.
- Signs the hash with the **private key** and verifies with the **public key**.
- If you change the message even a little, **verification fails**.

This gives:

- **Integrity** → message not changed
- **Authentication** → who signed it
- So: Task 3 shows a **secure basic idea** of how real digital signatures work.