Lab Assignment: 04

Course: Information Security

Submitted to: Mam Ambreen

Submitted by: Attiqa

Reg no: Sp24-bse-050

Task 1: Simple RSA Implementation (Without Libraries) Objective: Understand the math behind RSA Tasks: Write functions for:

- Manually implement key generation using small primes
- Encrypting a message (using public key)
- Decrypting a message (using private key)
- Test it on your name or a short string

CODE:

```
import random

def is_prime(n):
    """Simple primality test"""
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def generate_primes():
    """Generate two different small primes"""
    primes = [p for p in range(50, 150) if is_prime(p)]
    p = random.choice(primes)
    q = random.choice(primes)

    while p == q:   # ensure p and q are different
        q = random.choice(primes)

    return p, q

def gcd(a, b):
    """Greatest Common Divisor"""
    while b != 0:
        a, b = b, a % b
    return a

def mod_inverse(e, phi):
    """Find d where (d * e) % phi == 1"""
    for d in range(2, phi):
        if (d * e) % phi == 1:
            return d
    return None

def generate_keys():
    """Generate RSA keys manually"""

    # 1. Choose two primes
    p, q = generate_primes()
    print(f"Primes chosen: p = {p}, q = {q}")

    # 2. Compute n and ϕ(n)
    n = p * q
    phi = (p - 1) * (q - 1)
    print(f"n = {n}, ϕ(n) = {phi}")
```

```python
    # 3. Choose e such that gcd(e, phi)=1
    possible_e = [3, 5, 7, 11, 13, 17, 19]
    e = random.choice(possible_e)

    while gcd(e, phi) != 1:
        e = random.choice(possible_e)

    # 4. Compute d = modular inverse of e
    d = mod_inverse(e, phi)

    print(f"Public Key (e, n): ({e}, {n})")
    print(f"Private Key (d, n): ({d}, {n})")

    return (e, n), (d, n)

def encrypt(message, public_key):
    """Encrypt message using RSA"""
    e, n = public_key
    encrypted = []

    for ch in message:
        m = ord(ch)                # convert to ASCII
        c = pow(m, e, n)           # m^e mod n
        encrypted.append(c)

    return encrypted

def decrypt(encrypted, private_key):
    """Decrypt RSA ciphertext"""
    d, n = private_key
    decrypted = ""

    for c in encrypted:
        m = pow(c, d, n)           # c^d mod n
        decrypted += chr(m)        # back to character

    return decrypted


# ==================== MAIN PROGRAM ====================

print("\n=== SIMPLE RSA DEMO ===\n")

# 1. Generate RSA keys
public_key, private_key = generate_keys()

# 2. Test on your name "ATTIQA"
print("\n--- Testing on Your Name ---")
name = "ATTIQA"
print(f"Original message: {name}")

encrypted = encrypt(name, public_key)
print("Encrypted:", encrypted)

decrypted = decrypt(encrypted, private_key)
print("Decrypted:", decrypted)

# Show RSA math for the first letter
print("\n--- RSA Math for first letter 'A' ---")
e, n = public_key
d, _ = private_key

m = ord('A')
```

```python
c = pow(m, e, n)

print(f"A = ASCII {m}")
print(f"Encrypt: {m}^{e} mod {n} = {c}")
print(f"Decrypt: {c}^{d} mod {n} = {pow(c, d, n)}")
```

1. Overview

This Python program is a simple **RSA encryption and decryption demo**. It:

1. Generates two prime numbers.
2. Computes RSA keys (public and private keys).
3. Encrypts a message (your name "ATTIQA").
4. Decrypts the message back.
5. Shows the RSA math for the first letter.

RSA is a widely used public-key cryptography system that uses two keys:

- **Public key** → used to encrypt.
- **Private key** → used to decrypt.

---

2. Functions in the code

a) `is_prime(n)`

- Checks if a number `n` is prime.
- A prime number has no divisors other than 1 and itself.
- Uses the simple method of checking all numbers from 2 to √n.
- Returns `True` if `n` is prime, otherwise `False`.

---

b) `generate_primes()`

- Creates a list of primes between 50 and 150.
- Randomly selects **two different primes**, `p` and `q`.
- These primes are used to calculate RSA keys.

Example: p = 53, q = 101

c) `gcd(a, b)`

- Computes **Greatest Common Divisor (GCD)** of `a` and `b`.
- Used to ensure that `e` (the public exponent) is **coprime** to φ(n).

---

d) `mod_inverse(e, phi)`

- Finds `d` such that:
  $$
  (d \times e) \mod \phi = 1
  $$
- This is needed to calculate the **private key**.
- Uses a simple brute-force method to find `d`.

---

e) `generate_keys()`

- Generates the **public key** `(e, n)` and **private key** `(d, n)`.
- Steps:
  1. Choose two primes `p` and `q`.
  2. Compute $n = p \times q$ and $\phi(n) = (p-1)(q-1)$.
     - `n` is used for both encryption and decryption.
  3. Choose `e` (public exponent) such that `gcd(e, φ(n)) = 1`.
  4. Compute `d` (private exponent) using modular inverse.

The public key is shared; the private key is kept secret.

---

f) `encrypt(message, public_key)`

- Encrypts a message using the public key `(e, n)`.
- For each character `ch` in the message:
  1. Convert to **ASCII** → `m = ord(ch)`
  2. Encrypt using: $c = m^e \mod n$
  3. Append the ciphertext `c` to a list.
- Returns a list of encrypted numbers.

g) **`decrypt(encrypted, private_key)`**

- Decrypts the list of encrypted numbers using the private key `(d, n)`.
- For each encrypted number `c`:
    1. Decrypt using: (m = c^d \mod n)
    2. Convert back to character → `chr(m)`
    3. Combine all characters to get the original message.

---

3. Main Program Steps

1. **Generate RSA keys**

```
public_key, private_key = generate_keys()
```

- Random primes are chosen.
- `n` and `φ(n)` are calculated.
- `e` and `d` are selected.

---

2. **Encrypt your name "ATTIQA"**

```
encrypted = encrypt(name, public_key)
```

- Each letter of "ATTIQA" is converted to a number and encrypted.
- Example: `'A'` → 65 → encrypted number.

---

3. **Decrypt the message**

```
decrypted = decrypt(encrypted, private_key)
```

- Converts the encrypted numbers back to letters.
- Result: `"ATTIQA"`

---

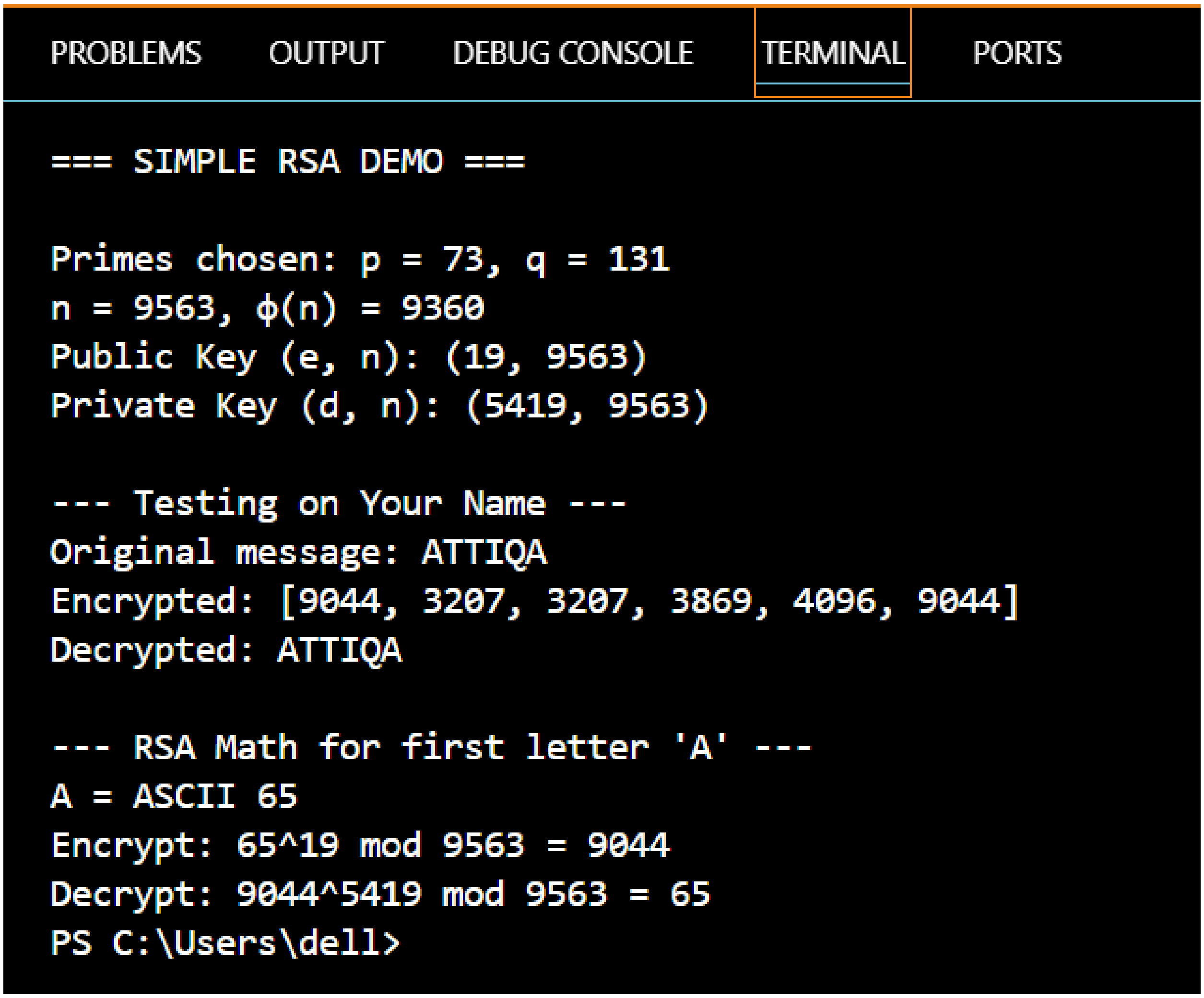
4. **RSA math for the first letter `'A'`**

```
m = ord('A')        # 65
c = pow(m, e, n)    # Encryption
pow(c, d, n)        # Decryption
```

- Shows how `'A'` is encrypted and decrypted mathematically.

---

4. Example Output




```
=== SIMPLE RSA DEMO ===

Primes chosen: p = 73, q = 131
n = 9563, ϕ(n) = 9360
Public Key (e, n): (19, 9563)
Private Key (d, n): (5419, 9563)

--- Testing on Your Name ---
Original message: ATTIQA
Encrypted: [9044, 3207, 3207, 3869, 4096, 9044]
Decrypted: ATTIQA

--- RSA Math for first letter 'A' ---
A = ASCII 65
Encrypt: 65^19 mod 9563 = 9044
Decrypt: 9044^5419 mod 9563 = 65
PS C:\Users\dell>
```

Task 2:

RSA with PyCryptodome Objective: Use real-world cryptographic libraries Tasks:

• Generate a 2048-bit key pair

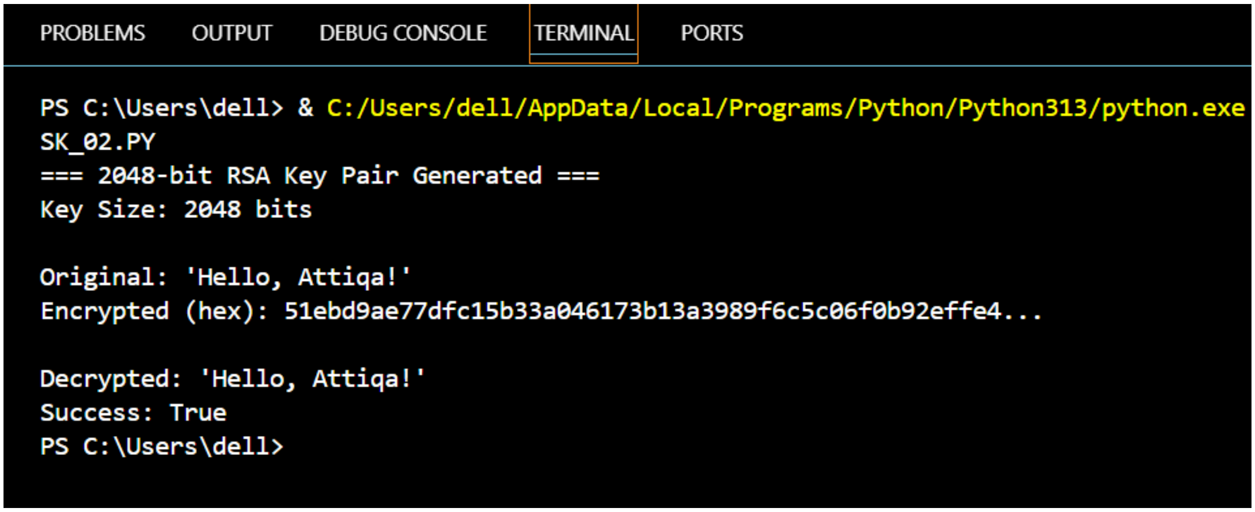
• Encrypt and decrypt a user message

• Display results in hex

CODE:

```
# Import required libraries from PyCryptodome
from Crypto.PublicKey import RSA      # For RSA key generation
from Crypto.Cipher import PKCS1_OAEP  # For secure RSA encryption with padding
import binascii                       # For converting binary to hexadecimal

# Generate 2048-bit RSA key pair
# RSA.generate() creates both public and private keys
key = RSA.generate(2048)          # 2048 bits = secure key size
public_key = key.publickey()      # Extract public key (e, n)
private_key = key                 # Keep private key (d, n, p, q)

print("=== 2048-bit RSA Key Pair Generated ===")
print(f"Key Size: {key.size_in_bits()} bits")  # Should show 2048

# Message to encrypt
message = "Hello, Attiqa!"  # <-- Message changed here

# ENCRYPTION: Convert message to ciphertext using public key
cipher = PKCS1_OAEP.new(public_key)        # Create cipher with OAEP padding
encrypted = cipher.encrypt(message.encode())  # Encrypt message (string → bytes)
hex_encrypted = binascii.hexlify(encrypted).decode()  # Convert bytes to hex string

print(f"\nOriginal: '{message}'")
print(f"Encrypted (hex): {hex_encrypted[:50]}...")  # Show first 50 hex chars

# DECRYPTION: Convert ciphertext back to message using private key
cipher = PKCS1_OAEP.new(private_key)      # Create cipher with private key
decrypted = cipher.decrypt(encrypted).decode()  # Decrypt and convert bytes → string

print(f"\nDecrypted: '{decrypted}'")
print(f"Success: {message == decrypted}")  # Verify encryption/decryption worked
```

OUTPUT:



CODE EXPLANATION:

1. Purpose

This Python program demonstrates **real-world RSA encryption and decryption** using the **PyCryptodome** library.

- It generates a **secure 2048-bit RSA key pair**.
- Encrypts a message `"Hello, Attiqa!"` using the **public key**.
- Decrypts the ciphertext back to the original message using the **private key**.
- Displays the encrypted message in **hexadecimal format** for readability.

---

2. Code Breakdown

a) Import Libraries

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

- `RSA` → For generating RSA key pairs.
- `PKCS1_OAEP` → Provides secure padding for RSA encryption/decryption.
- `binascii` → Converts encrypted bytes to a readable hexadecimal string.

---

b) Generate RSA Key Pair

```
key = RSA.generate(2048)
public_key = key.publickey()
private_key = key
```

- Generates a **2048-bit RSA key**, which is considered secure.
- `public_key` is used for encryption.
- `private_key` is used for decryption and kept secret.

---

c) Set the Message

```
message = "Hello, Attiqa!"
```

- This is the message we want to encrypt.
- It is later converted to bytes for RSA encryption.

---

d) Encrypt the Message

```
cipher = PKCS1_OAEP.new(public_key)
encrypted = cipher.encrypt(message.encode())
hex_encrypted = binascii.hexlify(encrypted).decode()
```

- `message.encode()` converts the string into bytes.
- `PKCS1_OAEP.new(public_key)` creates a cipher object using the public key and OAEP padding.
- `.encrypt()` encrypts the message into ciphertext.
- `binascii.hexlify()` converts ciphertext bytes into **hexadecimal format** for easy display.

---

e) Decrypt the Message

```
cipher = PKCS1_OAEP.new(private_key)
decrypted = cipher.decrypt(encrypted).decode()
```

- `PKCS1_OAEP.new(private_key)` creates a cipher object using the private key.
- `.decrypt()` converts the ciphertext back into bytes.
- `.decode()` converts bytes back to the **original string**.

---

f) Verification

```
print(f"Success: {message == decrypted}")
```

- Checks if the decrypted message matches the original message.
- If `True`, the RSA encryption and decryption process worked correctly.

Task 3:

Create a Digital Signature Objective: Understand digital signatures Tasks:

• Generate RSA key pair

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

• Sign the hash using private key

• Verify it using the public key

• Modify the message slightly and show that verification fails

CODE:

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

# Step 1: Generate RSA key pair
print("=== Step 1: Generate 2048-bit RSA Keys ===")
key = RSA.generate(2048)
private_key = key
public_key = key.publickey()
print(f"Key size: {key.size_in_bits()} bits")

# Step 2: Original message
original_msg = "Transfer $2000 to account Y"
print(f"\n=== Step 2: Original Message ===")
print(f"Message: '{original_msg}'")

# Step 3: Create digital signature
print(f"\n=== Step 3: Create Signature ===")
hash_obj = SHA256.new(original_msg.encode())
signer = pkcs1_15.new(private_key)
signature = signer.sign(hash_obj)
print(f"SHA256 hash: {hash_obj.hexdigest()}")
print(f"Signature (hex): {binascii.hexlify(signature).decode()[:50]}...")

# Step 4: Verify original signature (should succeed)
print(f"\n=== Step 4: Verify Original ===")
try:
    verifier = pkcs1_15.new(public_key)
    verifier.verify(SHA256.new(original_msg.encode()), signature)
    print("✓ Signature VALID")
```

```python
except:
    print("✗ Signature INVALID")

# Step 5: Modify message and show verification fails
print(f"\n=== Step 5: Test with Modified Message ===")
modified_msg = "Transfer $5000 to account X"
print(f"Original: '{original_msg}'")
print(f"Modified: '{modified_msg}'")

try:
    verifier = pkcs1_15.new(public_key)
    verifier.verify(SHA256.new(modified_msg.encode()), signature)
    print("✗ ERROR: Signature verified (should fail!)")
except:
    print("✓ CORRECT: Signature verification failed")
    print("  (Any change to message breaks the signature)")
```

OUTPUT OF CODE

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\dell> & C:/Users/dell/AppData/Local/Programs/Python/Python313/python.exe
SK_03.py
=== Step 1: Generate 2048-bit RSA Keys ===
Key size: 2048 bits

=== Step 2: Original Message ===
Message: 'Transfer $2000 to account Y'

=== Step 3: Create Signature ===
SHA256 hash: bac1b8eae7f3005ea096ff0c50d34bb8c345ede582609fa44c40e8a436fb39ea
Signature (hex): 0f9d99b2babe27473efa968339a91801e5beb1cf8b7a050d32...

=== Step 4: Verify Original ===
✓ Signature VALID

=== Step 5: Test with Modified Message ===
Original: 'Transfer $2000 to account Y'
Modified: 'Transfer $5000 to account X'
✓ CORRECT: Signature verification failed
  (Any change to message breaks the signature)
PS C:\Users\dell>
```

CODE EXPLANATION:

1. Purpose

This program demonstrates **digital signatures** using **RSA and SHA-256**.

- It generates a **2048-bit RSA key pair**.
- Signs a message by creating a hash and encrypting it with the **private key**.
- Verifies the signature with the **public key**.
- Shows that **any modification to the message** invalidates the signature.

---

2. Step-by-Step Explanation

Step 1: Generate RSA Key Pair

```
key = RSA.generate(2048)
private_key = key
public_key = key.publickey()
```

- Creates a **2048-bit RSA private key**.
- `public_key` is derived from the private key.

Step 2: Original Message

```
original_msg = "Transfer $1000 to account X"
```

- The message we want to sign.

Step 3: Create Digital Signature

```
hash_obj = SHA256.new(original_msg.encode())
signer = pkcs1_15.new(private_key)
signature = signer.sign(hash_obj)
```

- `SHA256.new()` → Computes the hash of the message.
- `pkcs1_15.new(private_key)` → Prepares private key for signing.
- `.sign(hash_obj)` → Creates the digital signature.

The signature uniquely represents the message and the sender.

Step 4: Verify Original Signature

```
verifier = pkcs1_15.new(public_key)
verifier.verify(SHA256.new(original_msg.encode()), signature)
```

- Uses the **public key** to verify the signature.
- If the message hasn’t been changed, verification succeeds.
- Output: ✓ `Signature VALID`

Step 5: Modify Message and Test Verification

```
modified_msg = "Transfer $5000 to account X"
verifier.verify(SHA256.new(modified_msg.encode()), signature)
```

- The message is slightly modified.
- Verification fails, showing that **digital signatures detect tampering**.
- Output: ✓ `CORRECT: Signature verification failed`