ASSIGNMENT NO. 4

Subject: Information Security

Submitted To: Maam.Ambreen

Submitted By: Faiqa bibi

Registration Number: Sp24-Bse-019

Date: 7/12/2025

Task 1:

Simple RSA Implementation (Without Libraries)

Objective: Understand the math behind RSA Tasks:

• Manually implement key generation using small primes

• Write functions for: o Encrypting a message (using public key) o Decrypting a message (using private key)

• Test it on your name or a short string.

CODE:

```
# ----- Helper: Greatest Common Divisor -----
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


# ----- Helper: Extended Euclidean Algorithm -----
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    g, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return g, x, y


# ----- Modular Multiplicative Inverse -----
def mod_inverse(e, phi):
    g, x, _ = extended_gcd(e, phi)
    if g != 1:
        raise Exception("No modular inverse")
    return x % phi


# ============ RSA KEY GENERATION ============
def generate_keys():
    # Small primes for learning
    p = 101
    q = 113
```

```python
    n = p * q                      # Modulus
    phi = (p - 1) * (q - 1)        # Euler Totient

    # Choose e: 1 < e < phi and gcd(e, phi) = 1
    e = 17
    if gcd(e, phi) != 1:
        raise Exception("e and phi not co-prime")

    # Compute private key exponent d
    d = mod_inverse(e, phi)

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


# ============ ENCRYPTION ============
def encrypt_message(message, public_key):
    e, n = public_key
    encrypted = [(ord(ch) ** e) % n for ch in message]
    return encrypted


# ============ DECRYPTION ============
def decrypt_message(cipher, private_key):
    d, n = private_key
    decrypted = "".join([chr((c ** d) % n) for c in cipher])
    return decrypted


# ============ DEMO TEST ============
public_key, private_key = generate_keys()

name = "amna"   # change to your own name if needed
print("\nOriginal Message:", name)

cipher = encrypt_message(name, public_key)
print("Encrypted Numbers:", cipher)

plain = decrypt_message(cipher, private_key)
print("Decrypted Message:", plain)

print("\nPublic Key =", public_key)
print("Private Key =", private_key)
```

Detailed line-by-line explanation (with security analysis):

Below I explain your `Simple RSA Algorithm` code line-by-line. I'll present the code in small blocks and then explain every statement and its purpose. After the line-by-line section I give a focused **security analysis**.

Header / Title (for human readers)

```
# ========================

#   Simple RSA Algorithm

#   (No external libraries)

# ========================
```

Explanation: Comment block describing the file. No effect on execution; used for human readers and documentation.

GCD helper

```
# ----- Helper: Greatest Common Divisor -----

def gcd(a, b):

    while b != 0:

        a, b = b, a % b

    return a
```

Line-by-line:

1. def gcd(a, b): — defines a function gcd that returns the greatest common divisor of a and b.
2. while b != 0: — loop while b is non-zero.
3. a, b = b, a % b — Euclid's algorithm step: replace (a,b) with (b, a mod b).
4. return a — when b becomes zero, a is the gcd.

Purpose: Used later to check whether e is coprime to ϕ(n). This is fundamental to RSA key selection.

Extended Euclidean Algorithm

```
# ----- Helper: Extended Euclidean Algorithm -----

def extended_gcd(a, b):

    if b == 0:

        return a, 1, 0

    g, x1, y1 = extended_gcd(b, a % b)
    x = y1

    y = x1 - (a // b) * y1

    return g, x, y
```

Line-by-line:

1. def extended_gcd(a, b): — defines function that returns (g, x, y) where g = gcd(a,b) and a*x + b*y = g.
2. if b == 0: — base case: if b is 0 then gcd is a.
3. return a, 1, 0 — when b==0, a*1 + 0*0 = a.
4. g, x1, y1 = extended_gcd(b, a % b) — recursively compute g, x1, y1 for (b, a mod b).
5. x = y1 — update solutions per the algorithm.
6. y = x1 - (a // b) * y1 — compute y using quotient a // b.
7. return g, x, y — return gcd g and Bezout coefficients x, y.

Purpose: Used to compute modular inverse (the private exponent d), via Bezout's identity.

Modular Inverse

```
# ----- Modular Multiplicative Inverse -----

def mod_inverse(e, phi):

    g, x, _ = extended_gcd(e, phi)

    if g != 1:

        raise Exception("No modular inverse")

    return x % phi
```

Line-by-line:

1. def mod_inverse(e, phi): — defines function to compute modular inverse of e modulo phi.
2. g, x, _ = extended_gcd(e, phi) — compute gcd and Bezout coefficients.
3. if g != 1: — if gcd != 1, inverse does not exist (not coprime).
4. raise Exception("No modular inverse") — throw error because d cannot be computed.
5. return x % phi — return the positive modular inverse d such that e*d ≡ 1 (mod phi).

Purpose: Calculates d, the private exponent for RSA.

Key generation

```
# ============ RSA KEY GENERATION ============
def generate_keys():
    # Small primes for learning
    p = 101
    q = 113

    n = p * q                    # Modulus
    phi = (p - 1) * (q - 1)      # Euler Totient

    # Choose e: 1 < e < phi and gcd(e, phi) = 1
    e = 17
    if gcd(e, phi) != 1:
        raise Exception("e and phi not co-prime")

    # Compute private key exponent d
    d = mod_inverse(e, phi)

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

Line-by-line:

1. def generate_keys(): — defines function to produce a public/private keypair.
2. p = 101 — first prime (small, educational).
3. q = 113 — second prime (small, educational).
4. n = p * q — modulus n: used by both public and private keys.
5. phi = (p - 1) * (q - 1) — Euler's totient $\phi(n)$ for RSA arithmetic.
6. e = 17 — choose public exponent e. (This code fixes e to 17.)
7. if gcd(e, phi) != 1: — verify e is coprime to $\phi(n)$.
8. raise Exception("e and phi not co-prime") — error if e and $\phi(n)$ not coprime.
9. d = mod_inverse(e, phi) — compute d such that $e * d \equiv 1 \pmod{\phi(n)}$.
10. return (e, n), (d, n) — return tuple (public_key, private_key) where public_key=(e,n) and private_key=(d,n).

Purpose: Produces RSA keys for encryption/decryption. **Note:** Choosing small fixed primes and a fixed small e is for education only — see security section.

Encryption function

```
# ============ ENCRYPTION ============

def encrypt_message(message, public_key):

    e, n = public_key

    encrypted = [(ord(ch) ** e) % n for ch in message]

    return encrypted
```

Line-by-line:

1. def encrypt_message(message, public_key): — defines encryption function that takes plaintext string and public key.
2. e, n = public_key — unpack public exponent and modulus.
3. encrypted = [(ord(ch) ** e) % n for ch in message] — for each character ch in the message:
   - ord(ch) converts character to integer code point,
   - ord(ch) ** e raises it to the power e,
   - % n reduces modulo n.
     This produces a list of integers: the ciphertext (character-by-character encryption).
4. return encrypted — return the list of ciphertext integers.

Purpose: Demonstrates RSA encryption. Note: encrypting each character separately is *not* how RSA is used in practice.

Implementation note: using pow(ord(ch), e, n) would be faster and more memory efficient than ord(ch) ** e % n because pow performs modular exponentiation without creating huge intermediate numbers.

Decryption function

```
# ============ DECRYPTION ============

def decrypt_message(cipher, private_key):

    d, n = private_key

    decrypted = "".join([chr((c ** d) % n) for c in cipher])

    return decrypted
```

Line-by-line:

1. def decrypt_message(cipher, private_key): — defines function to turn ciphertext integers back into a string.

2. d, n = private_key — unpack private exponent and modulus.
3. decrypted = "".join([chr((c ** d) % n) for c in cipher]) — for each ciphertext integer c:
    - compute c ** d % n, which yields the original message integer,
    - convert to character chr(...),
    - join characters into a plaintext string.
4. return decrypted — return the recovered plaintext string.

Purpose: Demonstrates RSA decryption. Same implementation note: use pow(c, d, n) to be efficient and safe from huge intermediates.

Demo execution

```
# ============ DEMO TEST ============
public_key, private_key = generate_keys()


name = "Faiqa"   # change to your own name if needed
print("\nOriginal Message:", name)


cipher = encrypt_message(name, public_key)
print("Encrypted Numbers:", cipher)


plain = decrypt_message(cipher, private_key)
print("Decrypted Message:", plain)


print("\nPublic Key =", public_key)
print("Private Key =", private_key)
```

Line-by-line:

1. public_key, private_key = generate_keys() — generate the keys and unpack.
2. name = "Faiqa" — sample plaintext (student name). Replace as needed.
3. print("\nOriginal Message:", name) — show original message for demonstration.
4. cipher = encrypt_message(name, public_key) — encrypt the name using the public key; result is a list of integers.

5. print("Encrypted Numbers:", cipher) — print ciphertext integers.
6. plain = decrypt_message(cipher, private_key) — decrypt ciphertext back to plaintext.
7. print("Decrypted Message:", plain) — print decrypted result; should match original.
8. print("\nPublic Key =", public_key) — display public key (e, n).
9. print("Private Key =", private_key) — display private key (d, n).

Purpose: Runs a full round-trip to show the RSA workflow works with the small-demo keys.

Security analysis — what's wrong or weak (detailed)

This section explains the security weaknesses of the current implementation and why it is **only** suitable for learning / demonstration.

1. Tiny primes (p, q)

- You use p = 101 and q = 113. These are extremely small primes.
- Real RSA requires *large* primes (2048-bit modulus is common today; 3072/4096 bits for higher security).
- With such small n (here n = 11413), an attacker can factor n quickly using trivial algorithms and recover p and q, then compute d. This completely breaks the scheme.

2. Character-by-character encryption

- The code encrypts each character individually: ord(ch).
- This leaks structure: repeated characters map to repeated ciphertext numbers. It is vulnerable to frequency analysis and other pattern attacks.
- Proper use of RSA encrypts blocks (or better: uses RSA to encrypt a symmetric key and uses symmetric cipher for bulk data).

3. No padding (textbook RSA)

- The code performs textbook RSA: m^e mod n with no padding or randomness.
- Textbook RSA is insecure: it is deterministic and vulnerable to chosen-plaintext, chosen-ciphertext attacks, and message recovery attacks.
- Standard secure padding schemes: **OAEP** for encryption and **PSS** for signatures. They add randomness and structure to prevent attacks.

4. Fixed small public exponent

- The code uses e = 17. While 17 is sometimes used and can be safe, common practice uses e = 65537 because it is prime and balances performance and security.

- Using small e can cause issues if message padding is inadequate (e.g., small-message attacks) — another reason to use OAEP and standard e.

5. Inefficient math and possible overflow

- The code uses ord(ch) ** e % n, which constructs a large intermediate integer ord(ch) ** e before the modulo. For large exponents this is inefficient and could be memory-heavy.
- Use pow(base, exponent, modulus) to perform modular exponentiation efficiently and safely without huge intermediate values.

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:

```
# ============================
#   Task 2 - RSA with PyCryptodome
# ============================

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

# ----------- Generate 2048-bit RSA Key Pair -----------
key_pair = RSA.generate(2048)

public_key = key_pair.publickey()
private_key = key_pair

print("\n----- Generated RSA Keys (2048-bit) -----")
print("Public Key:", public_key.export_key().decode())
print("Private Key:", private_key.export_key().decode())


# ----------- Input Message from User -----------
message = input("\nEnter a message to encrypt: ").encode()
```

```python
# ----------- Encrypt using Public Key -----------
cipher = PKCS1_OAEP.new(public_key)
ciphertext = cipher.encrypt(message)

print("\nEncrypted Message (HEX):")
print(binascii.hexlify(ciphertext).decode())


# ----------- Decrypt using Private Key -----------
decrypt_cipher = PKCS1_OAEP.new(private_key)
decrypted_msg = decrypt_cipher.decrypt(ciphertext)

print("\nDecrypted Message:")
print(decrypted_msg.decode())
```

Detailed line-by-line explanation — Task 2 (RSA with PyCryptodome)

Header / imports

```
# ============================

#   Task 2 - RSA with PyCryptodome

# ============================


from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1_OAEP

import binascii
```

Explanation:

- Comment block: human-readable title, no effect on execution.
- from Crypto.PublicKey import RSA — imports PyCryptodome's RSA key-generation and key-handling utilities.
- from Crypto.Cipher import PKCS1_OAEP — imports the OAEP cipher wrapper (RSA-OAEP), which implements RSA encryption/decryption with OAEP padding.
- import binascii — imports Python's binascii module used for binary↔hex conversions (we'll use it to display ciphertext as hex).

Key generation

```
# ----------- Generate 2048-bit RSA Key Pair -----------

key_pair = RSA.generate(2048)
```

public_key = key_pair.publickey()

private_key = key_pair

Line-by-line:

- key_pair = RSA.generate(2048)
    - Generates an RSA key pair of size **2048 bits** (this builds p, q, n, e, d etc.). The PyCryptodome function uses a cryptographically secure random source and probabilistic primality tests.
- public_key = key_pair.publickey()
    - Extracts the public key from the generated key pair. public_key is an RSA key object containing (n, e).
- private_key = key_pair
    - Keeps the private key object that contains (n, d) (and usually CRT parameters). You could also call key_pair.export_key() to write it to PEM/DER.

Print keys (debug / demonstration)

print("\n----- Generated RSA Keys (2048-bit) -----")

print("Public Key:", public_key.export_key().decode())

print("Private Key:", private_key.export_key().decode())

Explanation:

- print("\n----- ... -----") — prints a header for readability.
- public_key.export_key().decode() — export_key() returns the key in PEM (bytes). .decode() converts bytes to string for printing. The printed PEM looks like:
- -----BEGIN PUBLIC KEY-----
- MIIBIj...
- -----END PUBLIC KEY-----
- private_key.export_key().decode() — same for the private key in PEM. **Important security note:** printing the private key to the console is only acceptable for demonstrations. In any real system you must never reveal or log a private key.

Input message

----------- Input Message from User -----------

message = input("\nEnter a message to encrypt: ").encode()

Explanation:

- input("\nEnter a message to encrypt: ") — reads a line of text from the user (a Python str).
- .encode() — converts that Python string to bytes using UTF-8 by default. RSA encryption works on bytes, not Python str, so you must encode before encrypting.

Behavioral notes:

- Encoding uses UTF-8; if the user types non-ASCII characters they will be correctly turned into a byte sequence.
- If this script is run in a non-interactive environment, input() will block or fail — for automation, pass a message via function argument or argument parsing.

Encryption

----------- Encrypt using Public Key -----------

cipher = PKCS1_OAEP.new(public_key)

ciphertext = cipher.encrypt(message)

Line-by-line:

- cipher = PKCS1_OAEP.new(public_key) — constructs an OAEP cipher object configured for encryption with the given public key. By default PyCryptodome's OAEP uses SHA-1 as the hash function unless you explicitly provide a different hashAlgo.
- ciphertext = cipher.encrypt(message) — encrypts the message bytes with RSA-OAEP, returning ciphertext bytes.

Important cryptographic notes:

- OAEP is a secure padding scheme that adds randomness and structure to the plaintext prior to RSA exponentiation; this prevents deterministic and some chosen-plaintext/ciphertext attacks that textbook RSA is vulnerable to.
- OAEP limits the maximum plaintext size per encryption operation (see Security Analysis below).

Display ciphertext in hex

print("\nEncrypted Message (HEX):")

print(binascii.hexlify(ciphertext).decode())

Explanation:

- binascii.hexlify(ciphertext) — converts the ciphertext bytes into a bytes object containing hexadecimal ASCII characters (e.g. b'4f2a...').
- .decode() — converts that bytes object to a Python str so it prints as readable hex.
- Printing hex is common for showing binary ciphertext in text reports and logs.

Decryption

```
# ----------- Decrypt using Private Key -----------

decrypt_cipher = PKCS1_OAEP.new(private_key)

decrypted_msg = decrypt_cipher.decrypt(ciphertext)


print("\nDecrypted Message:")

print(decrypted_msg.decode())
```

Line-by-line:

- decrypt_cipher = PKCS1_OAEP.new(private_key) — constructs an OAEP cipher object for decryption using the private key (PyCryptodome detects whether the key object has the private exponent).
- decrypted_msg = decrypt_cipher.decrypt(ciphertext) — decrypts the ciphertext back into plaintext bytes.
- print(decrypted_msg.decode()) — decodes the plaintext bytes back into a Python string using UTF-8 and prints it.

Notes:

- If ciphertext is corrupted or tampered, decrypt() will raise a ValueError (or return incorrect data) — OAEP includes integrity checks and decryption should fail if padding is invalid.
- Always wrap decrypt() in a try/except block in production to handle errors gracefully.

Security analysis (what it does well, what to watch out for)

✅ **Good security aspects of this script**

- **Uses PyCryptodome** — a widely used cryptographic library that implements tested primitives.
- **Key size 2048 bits** — currently considered acceptable for many use cases (NIST recommends 2048 as a baseline; 3072 is stronger for longer-term security).
- **OAEP padding** — using PKCS1_OAEP is correct for RSA encryption (much safer than textbook RSA).
- **Secure random** — RSA.generate() uses a cryptographically secure RNG and proper primality testing.

⚠️ **Limitations and risks (must be mentioned in your report)**

1. **Printing the private key**

   - The script prints the private key PEM. In real systems, the private key must be kept secret (no printing, no plain-file storage without encryption, prefer HSM or OS-protected key stores).

2. **Plain encryption of arbitrary-length messages**
   - RSA (even with OAEP) can only encrypt messages up to a maximum length: max_message_length = k - 2*hLen - 2, where k is modulus length in bytes and hLen is the hash length used by OAEP.
     - For a 2048-bit key k = 256 bytes. If OAEP uses SHA-1 (hLen = 20) then max_message_length = 256 - 40 - 2 = 214 bytes.
     - If you switch OAEP to SHA-256 (hLen = 32) then max_message_length = 256 - 64 - 2 = 190 bytes.
   - If the user's message is longer than allowed, cipher.encrypt() will raise ValueError about message too long. For arbitrary-length data you must use hybrid encryption (see recommendations).
3. **Default OAEP hash = SHA-1**
   - PyCryptodome's PKCS1_OAEP.new() uses SHA-1 by default. SHA-1 is considered weak for collision resistance (but OAEP uses it for a different purpose). Prefer explicitly setting hashAlgo=SHA256 to use SHA-256:
4. from Crypto.Hash import SHA256
5. cipher = PKCS1_OAEP.new(public_key, hashAlgo=SHA256)
   - Using stronger hash algorithms is recommended in modern code.
6. **No authentication for ciphertext**
   - RSA-OAEP provides some integrity via padding checks, but for authenticated encryption workflows you often use a hybrid approach with an AEAD symmetric cipher (e.g., AES-GCM) that guarantees confidentiality + integrity.
7. **Performance / practicality**
   - RSA is relatively slow and not suitable for encrypting large payloads. The standard approach is hybrid:
     - Generate a random symmetric key (e.g., 256-bit AES key).
     - Encrypt the message with AES-GCM (fast, supports large messages).
     - Encrypt the AES key with RSA-OAEP.
     - Send RSA-encrypted AES key + AES-GCM ciphertext + GCM tag.

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

```python
# ================================
#   Task 3 - Digital Signature RSA
# ================================

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

# ------------- Generate RSA Key Pair (2048-bit) -------------
key_pair = RSA.generate(2048)
public_key = key_pair.publickey()
private_key = key_pair


print("\n----- RSA Key Pair Generated -----")
print("Public Key:", public_key.export_key().decode())
print("Private Key:", private_key.export_key().decode())

# ------------- Original Message -------------
message = "Data Security Assignment - RSA Digital Signature"
print("\nOriginal Message:", message)

# Create SHA256 hash
hash_obj = SHA256.new(message.encode())

# Sign the hashed message using PRIVATE KEY
signature = pkcs1_15.new(private_key).sign(hash_obj)
print("\nDigital Signature (Hex):", signature.hex())

# ------------- Verification Step -------------
try:
    pkcs1_15.new(public_key).verify(hash_obj, signature)
    print("\nVerification Successful: Signature is VALID")
except (ValueError, TypeError):
    print("\nVerification Failed!!")
```

```
# ------------- Tamper the Message -------------
fake_message = "Data Security Assignment - RSA Digital Signatures"  # small
change (extra 's')
print("\nModified Message:", fake_message)

# New hash for modified message
fake_hash = SHA256.new(fake_message.encode())

# Verify again with original signature (should fail)
try:
    pkcs1_15.new(public_key).verify(fake_hash, signature)
    print("Verification Successful: Signature is VALID (Unexpected!)")
except (ValueError, TypeError):
    print("Verification Failed: Message has been modified ❌")
```

Line-by-line explanation

Header / imports

from Crypto.PublicKey import RSA

from Crypto.Signature import pkcs1_15

from Crypto.Hash import SHA256

RSA — class/object to create and manage RSA key pairs (generation, import/export).

pkcs1_15 — implementation of the PKCS#1 v1.5 signature scheme (RSASSA-PKCS1-v1_5) for signing/verification.

SHA256 — hash object implementing SHA-256; you will hash the message before signing.

Key generation

python

Copy code

key_pair = RSA.generate(2048)

public_key = key_pair.publickey()

private_key = key_pair

RSA.generate(2048) — creates a fresh RSA key pair with a 2048-bit modulus. Internally generates primes p,q, computes n, e, d, and CRT parameters.

public_key = key_pair.publickey() — extracts the public key object (contains n, e).

private_key = key_pair — private key object (contains n, d, and CRT values). You could instead store/export to PEM.

Key printing (demo)

```
print("\n----- RSA Key Pair Generated -----")

print("Public Key:", public_key.export_key().decode())

print("Private Key:", private_key.export_key().decode())
```

export_key() returns PEM bytes. .decode() converts to string for printing.

Security note: printing the private key is only acceptable for educational/demo purposes. Never print or store private keys in plaintext in production.

Message setup

```
message = "Data Security Assignment - RSA Digital Signature"

print("\nOriginal Message:", message)
```

Sets the message string that will be signed and prints it for demonstration.

Hash creation

```
hash_obj = SHA256.new(message.encode())
```

Creates a SHA-256 hash object for the message bytes (UTF-8 encoding by default).

pkcs1_15 expects a hash object when signing/verifying.

Signing

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

print("\nDigital Signature (Hex):", signature.hex())
```

pkcs1_15.new(private_key) builds a signer object bound to the private key.

.sign(hash_obj) calculates the RSA signature over the hash (using PKCS#1 v1.5 formatting), returning raw signature bytes.

signature.hex() prints the signature in hexadecimal for human-readable output.

Verification (original message)

```
try:

    pkcs1_15.new(public_key).verify(hash_obj, signature)

    print("\nVerification Successful: Signature is VALID")

except (ValueError, TypeError):

    print("\nVerification Failed!!")
```

pkcs1_15.new(public_key).verify(hash_obj, signature) attempts to verify the signature against the provided hash and public key.

If valid, it returns None and you print success; if invalid (signature mismatch, tampered message), it raises ValueError or TypeError, so the except prints failure.

Tampering test

```
fake_message = "Data Security Assignment - RSA Digital Signatures"  # small change (extra 's')

print("\nModified Message:", fake_message)

fake_hash = SHA256.new(fake_message.encode())
```

Modify the message (add an 's') to simulate tampering. Compute the hash of the modified message.

Verification (tampered message)

```
try:
    pkcs1_15.new(public_key).verify(fake_hash, signature)
    print("Verification Successful: Signature is VALID (Unexpected!)")
except (ValueError, TypeError):
    print("Verification Failed: Message has been modified ❌")
```

Attempts to verify the original signature against the hash of the tampered message. This should fail, causing an exception and printing the expected failure message.

What the script demonstrates (short)

How to generate RSA keys, hash a message with SHA-256, create a signature using the RSA private key (PKCS#1 v1.5), and verify it with the public key.

That even a small change to the message causes verification to fail (showing integrity and authenticity properties).

Security analysis — strengths, weaknesses, and recommendations

Strong / correct choices in your script

Uses a vetted library (PyCryptodome) — avoids the pitfalls of rolling your own crypto.

Uses SHA-256 — strong hashing algorithm suitable for signatures.

Performs verification — demonstrates the correct verify flow and that tampering is detected.

Uses 2048-bit RSA — currently acceptable baseline for many applications (short-term security).

Weaknesses / risks & recommended changes

1. **PKCS#1 v1.5 (pkcs1_15) vs RSASSA-PSS**

- You are using **RSASSA-PKCS1-v1_5** (pkcs1_15). It is widely supported and historically common but **PSS (Probabilistic Signature Scheme)** is recommended by modern standards for new designs because PSS provides stronger security proofs and better resistance to certain attacks.
- **Recommendation:** use Crypto.Signature.pss with SHA-256:
- from Crypto.Signature import pss
- signature = pss.new(private_key).sign(hash_obj)
- pss.new(public_key).verify(hash_obj, signature) # raises ValueError if invalid

2. **Private key exposure**
   - The script prints the private key PEM to the console. In real use, never expose private keys. Store them encrypted, in an HSM, or in a secure vault/KMS. If you must export, protect it with a passphrase:
   - private_key.export_key(passphrase="password", pkcs=8, protection="scryptAndAES128-CBC")
3. **Key size and future-proofing**
   - 2048-bit is acceptable today, but for long-term protection (decades) use **3072-bit** or **4096-bit**, or consider elliptic-curve signatures (ECDSA with curve P-384 or Ed25519) for better efficiency and equivalent security.
4. **No timestamping or context**
   - A signature proves *message origin* at some point, but not necessarily *when* it was created or whether the signer intended that exact message in that context.
   - **Recommendation:** include metadata (timestamps, document IDs, signer identity) in the signed payload or use timestamping services (RFC 3161) for non-repudiation.
5. **No canonicalization / encoding rules**
   - If the message comes from structured data (JSON, XML), different canonicalizations can lead to signature verification failures. Always define and apply canonicalization before hashing.
   - Example: sign the UTF-8 encoded *canonical* byte sequence of the message.
6. **No signature format / envelope**
   - You output raw signature bytes. In many real systems you wrap signatures in a standard (CMS/PKCS#7, JOSE / JWS for web APIs) that includes signer info, algorithm, and certificate chain.
7. **No certificate/PKI**

- The verification step only uses a raw public key. In practice, public keys are distributed via certificates (X.509) and verified via PKI/CA chains, or via a trusted key distribution mechanism.