**Information Security COMP 421**

**Spring 2025 Section B**

**Assignment 3**

**Hafsah Shahbaz – 251684784**

**Task 1: Explain each line of the following Python’s implementation of AES**

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

These three lines import necessary components from the pycryptodome library:

* AES is the core class for performing AES encryption and decryption.
* get\_random\_bytes generates secure random values, critical for cryptographic operations like keys and IVs.
* pad and unpad manage message padding, since AES is a block cipher that requires inputs to match the block size.

key = get\_random\_bytes(32)

This line generates a random 256-bit encryption key. Since AES supports 128, 192, and 256-bit keys, here we're using the most secure 256-bit variant (32 bytes \* 8 = 256 bits).

iv = get\_random\_bytes(16)

Generates a 128-bit (16-byte) Initialization Vector (IV). Required for CBC mode to ensure identical plaintext blocks encrypt differently.

cipher = AES.new(key, AES.MODE\_CBC, iv)

Creates a new AES cipher object in CBC (Cipher Block Chaining) mode using the generated key and IV. CBC mode introduces dependency between ciphertext blocks, improving security.

plaintext = b"This is a secret message."

Defines the plaintext to be encrypted. The b prefix indicates it's a byte string, which is necessary because AES operates on bytes, not regular strings.

ciphertext = cipher.encrypt(pad(plaintext, AES.block\_size))

The plaintext is padded using PKCS#7 padding to ensure its length is a multiple of the AES block size (16 bytes), then encrypted.

print("Encrypted:", ciphertext)

Prints the resulting ciphertext, which will be a bytes object containing encrypted data.

decipher = AES.new(key, AES.MODE\_CBC, iv)

Recreates the cipher object using the same key and IV for decryption. This is necessary for AES decryption to work correctly.

decrypted = unpad(decipher.decrypt(ciphertext), AES.block\_size)

Decrypts the ciphertext and removes padding, revealing the original plaintext message.

print("Decrypted:", decrypted)

Prints the decrypted message. If encryption and decryption are done correctly, this will match the original plaintext.

**Task 2: Explain each line of the following Python’s implementation of RSA**

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

These imports allow you to work with RSA keys and perform encryption using the OAEP (Optimal Asymmetric Encryption Padding) scheme, which enhances security.

key = RSA.generate(2048)

Generates a new 2048-bit RSA key pair. This includes both the private and public keys. A 2048-bit key is standard for secure RSA operations.

private\_key = key.export\_key()

public\_key = key.publickey().export\_key()

Exports the private and public keys in PEM (Privacy Enhanced Mail) format, a base64 encoded format often used for key sharing and storage.

recipient\_key = RSA.import\_key(public\_key)

cipher\_rsa = PKCS1\_OAEP.new(recipient\_key)

Imports the public key and creates a cipher object for encryption. OAEP padding is used to secure against chosen ciphertext attacks.

ciphertext = cipher\_rsa.encrypt(b'This is a secret.')

Encrypts the plaintext using the RSA public key. The data must be small enough to fit into a single block after padding (a limitation of RSA).

print("Encrypted:", ciphertext)

Displays the RSA-encrypted message in binary format.

private\_key\_obj = RSA.import\_key(private\_key)

cipher\_rsa = PKCS1\_OAEP.new(private\_key\_obj)

Imports the private key and sets up a cipher object for decryption using the same OAEP scheme.

decrypted = cipher\_rsa.decrypt(ciphertext)

Decrypts the RSA-encrypted ciphertext and retrieves the original plaintext.

print("Decrypted:", decrypted)

Prints the decrypted message, which should exactly match the input message.

**Task 3: Error Detection and Suggestions**

Both the AES and RSA implementations are syntactically correct and function as expected without errors.

**Observations & Suggestions**

* **AES IV Reuse Risk**: Reusing the same IV with the same key compromises confidentiality in CBC mode. Use a unique IV for every encryption and transmit it securely (e.g., prefixing it to the ciphertext).

**iv = get\_random\_bytes(16) moved before encryption:** The IV is now generated *just before* the encryption process. This ensures it's fresh for each encryption.

* **RSA Data Size Limitation**: RSA encryption only works with very short messages. To handle large data, adopt hybrid encryption: encrypt data with AES, then encrypt the AES key with RSA.
* **Key Storage**: The encryption keys are stored in memory and are lost when the program ends. For real applications, keys should be saved securely using encrypted storage or hardware key modules.
* **No Authentication**: AES-CBC doesn’t provide integrity. For authenticated encryption, use AES in GCM mode or append an HMAC