**Assigment#02**



**Name** : faiqa bibi

**Reg No**: sp24-bse-019

**Submitted To**:Maam Ambreen

**Course**:Information security

**Date**:12-oct-2025

Graded Task 1:

**You have implemented DES there is built in implemented DES in python in crypto cipher module use it for encryption/decryption and provide output sample example?**

**CODE:**

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

# DES key must be exactly 8 bytes long

key = get\_random\_bytes(8)

def des\_encrypt(data, key):

    cipher = DES.new(key, DES.MODE\_ECB)

    padded\_data = pad(data, DES.block\_size)

    encrypted\_data = cipher.encrypt(padded\_data)

    return encrypted\_data

def des\_decrypt(encrypted\_data, key):

    cipher = DES.new(key, DES.MODE\_ECB)

    decrypted\_data = unpad(cipher.decrypt(encrypted\_data), DES.block\_size)

    return decrypted\_data

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

    # Input data (must be bytes)

    data = b"Secret123"

    print(f"Original Data: {data}")

    # Encrypt the data

    encrypted\_data = des\_encrypt(data, key)

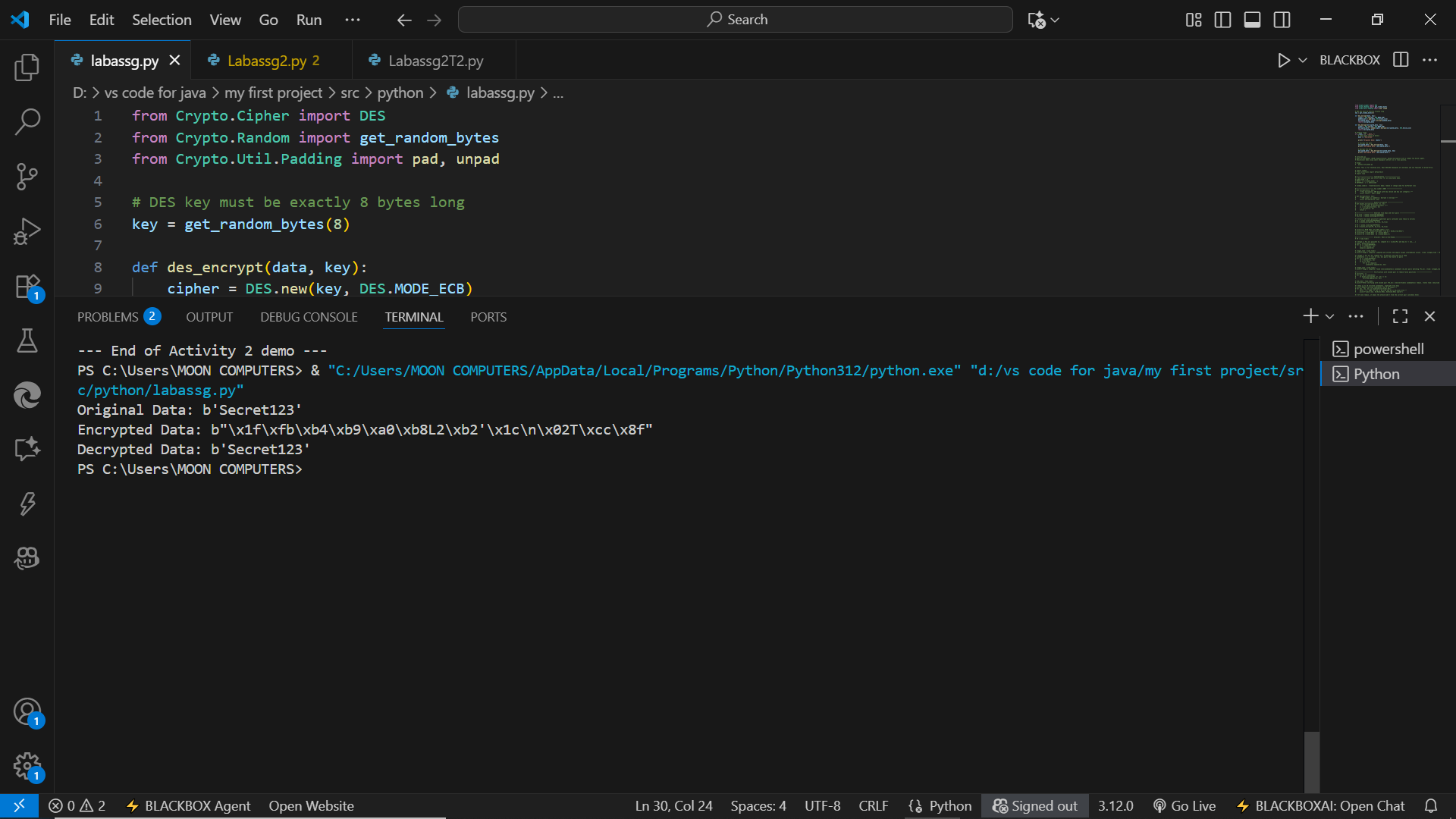
    print(f"Encrypted Data: {encrypted\_data}")

    # Decrypt the data

    decrypted\_data = des\_decrypt(encrypted\_data, key)

    print(f"Decrypted Data: {decrypted\_data}")

**OUTPUT:**



**EXPLANATION:**

This Python program demonstrates DES encryption and decryption using the Crypto.Cipher module. A random 8-byte key is generated, and the plaintext is padded to match DES’s 8-byte block size. The data is then encrypted using ECB mode and later decrypted with the same key. The program shows how DES converts readable text into ciphertext and restores it back to the original form using symmetric key encryption.

**Graded Task 2**

**Visualization of MITM Attack Flow:**

**Attacker intercepts plaintext (P) and ciphertext (C)**

**1. Guess K1 P --[Encrypt with K1]--> Intermediate Value 1 (I1)**

**2. Guess K2 C --[Decrypt with K2]--> Intermediate Value 2 (I2)**

**3. If I1 == I2, then K1 and K2 are likely correct. Found the DES key: K = K1 + K2**

**CODE:**

import random

from collections import defaultdict

import time

# -------------------- Configuration --------------------

# Use 16-bit block and 16-bit keys for an executable demo.

BLOCK\_SIZE = 16

MASK = (1 << BLOCK\_SIZE) - 1

KEYSPACE = 1 << BLOCK\_SIZE

random.seed(1)  # deterministic demo; remove or change seed for different runs

# -------------------- Toy cipher (XOR) --------------------

def encrypt(block, key):

    """Toy encryption: XOR block with key (block and key are integers)."""

    return (block ^ key) & MASK

def decrypt(block, key):

    """XOR cipher is symmetric: decrypt == encrypt."""

    return encrypt(block, key)

# -------------------- Double encryption --------------------

def double\_encrypt(plaintext, k1, k2):

    """C = E\_{k2}( E\_{k1}(plaintext) )"""

    i1 = encrypt(plaintext, k1)

    c = encrypt(i1, k2)

    return c

# -------------------- Generate true keys and test pairs --------------------

K1\_true = random.randrange(KEYSPACE)

K2\_true = random.randrange(KEYSPACE)

# Create two known plaintext-ciphertext pairs (attacker uses these to verify)

P1 = random.randrange(KEYSPACE)

C1 = double\_encrypt(P1, K1\_true, K2\_true)

P2 = random.randrange(KEYSPACE)

C2 = double\_encrypt(P2, K1\_true, K2\_true)

print("=== MITM Demo (toy XOR cipher) ===")

print(f"True K1 = 0x{K1\_true:04X}, True K2 = 0x{K2\_true:04X}")

print(f"P1 = 0x{P1:04X}  C1 = 0x{C1:04X}")

print(f"P2 = 0x{P2:04X}  C2 = 0x{C2:04X}\n")

# -------------------- Attacker: Meet-in-the-Middle --------------------

t0 = time.time()

# Stage 1: For all possible k1, compute I1 = E\_{k1}(P1) and map I1 -> [k1,...]

imap = defaultdict(list)

for k1 in range(KEYSPACE):

    i1 = encrypt(P1, k1)

    imap[i1].append(k1)

stage1\_time = time.time()

print(f"Stage 1 complete: computed and stored {len(imap)} unique intermediate values. (time: {stage1\_time - t0:.3f}s)")

# Stage 2: For all k2, compute I2 = D\_{k2}(C1) and look up in imap

candidates = []  # list of (k1, k2) pairs that match on pair1

for k2 in range(KEYSPACE):

    i2 = decrypt(C1, k2)

    if i2 in imap:

        for k1 in imap[i2]:

            candidates.append((k1, k2))

stage2\_time = time.time()

print(f"Stage 2 complete: found {len(candidates)} candidate (k1,k2) pairs matching (P1,C1). (time: {stage2\_time - stage1\_time:.3f}s)")

# -------------------- Verification with second pair to reduce false positives --------------------

verified = []

for k1, k2 in candidates:

    if double\_encrypt(P2, k1, k2) == C2:

        verified.append((k1, k2))

end\_time = time.time()

print(f"After verifying with second pair (P2,C2): {len(verified)} candidate(s) remain. (total time: {end\_time - t0:.3f}s)\n")

# Show up to 10 verified candidates, highlight true keys

print("Sample verified candidates (up to 10 shown):")

for idx, (k1, k2) in enumerate(verified[:10]):

    mark = "<-- TRUE" if (k1 == K1\_true and k2 == K2\_true) else ""

    print(f"{idx+1:2d}. K1=0x{k1:04X}, K2=0x{k2:04X} {mark}")

# If none remain, it means the attack didn't find the correct pair (unlikely here)

if not verified:

    print("No verified key pairs found. (This can happen if randomness caused collision filtering.)")

else:

    found\_true = any(k1 == K1\_true and k2 == K2\_true for k1, k2 in verified)

    print(f"\nWas the true key pair recovered? {'Yes' if found\_true else 'No'}")

# Print some stats

print("\nStats:")

print(f"Keyspace per key: {KEYSPACE} keys (2^{BLOCK\_SIZE})")

print(f"Stage1 encrypt ops: {KEYSPACE}")

print(f"Stage2 decrypt ops: {KEYSPACE}")

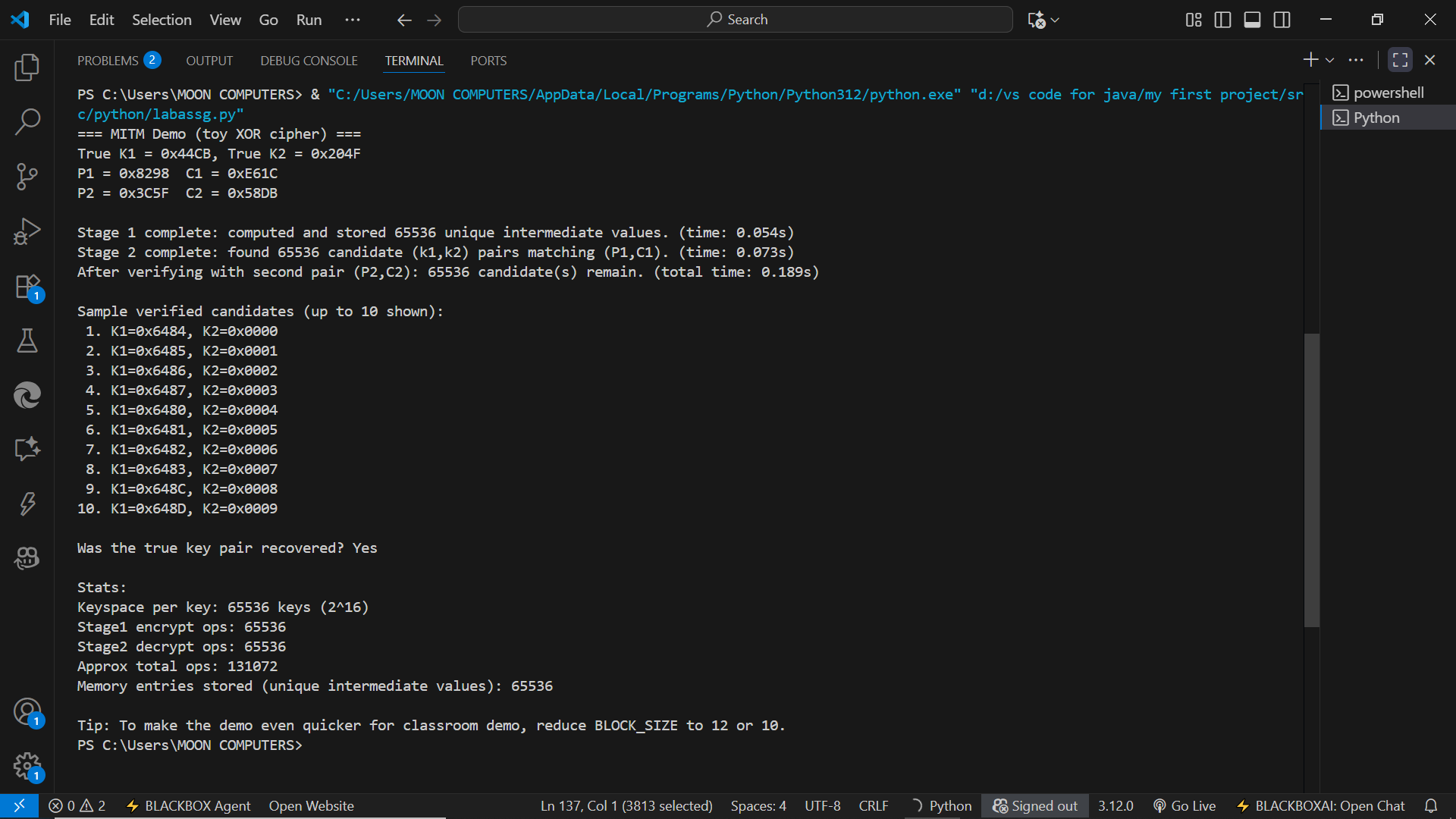
print(f"Approx total ops: {KEYSPACE \* 2}")

print(f"Memory entries stored (unique intermediate values): {len(imap)}")

# Optional note:

print("\nTip: To make the demo even quicker for classroom demo, reduce BLOCK\_SIZE to 12 or 10.")

**OUTPUT:**



**EXPLANATION:**

This script demonstrates a Meet-in-the-Middle (MITM) attack on a toy double-encryption scheme. It uses a tiny XOR-based block cipher with 16-bit blocks and 16-bit keys (easy to run in class).

* **Setup**: Random true keys K1\_true, K2\_true are chosen. Two known plaintext/ciphertext pairs (P1,C1) and (P2,C2) are generated using double encryption C = E\_{K2}(E\_{K1}(P)).
* **Toy cipher**: encrypt(block,key) is block XOR key (symmetric, so decrypt == encrypt).
* **MITM Stage 1**: For every possible k1 in the keyspace, compute I1 = E\_{k1}(P1) and store mapping I1 -> k1. This costs 2^n encryptions and memory entries.
* **MITM Stage 2**: For every possible k2, compute I2 = D\_{k2}(C1) and look up I2 in the stored map. Matches produce candidate (k1,k2) pairs. This costs another 2^n operations.
* **Verification**: Candidate pairs are checked against the second known pair (P2,C2) to remove false positives; true (K1\_true,K2\_true) should remain.
* **Output & complexity**: The script prints found candidates and stats (ops ≈ 2 \* 2^n, memory ≈ 2^n). With 16-bit keys this runs instantly; with real DES-sized keys MITM still reduces effort to about 2^{57} for double-DES (i.e., much less than 2^{112}).
* Note: The code includes a tip to reduce BLOCK\_SIZE (e.g., to 12 or 10) for even faster classroom demos.