

LAB ASSIGNMENT: 02

SUBMITTED BY:

Muhammad Hassan

SP24-BSE-008

SUBMITTED TO: MA'AM AMBREEN

DATE: 12 OCT, 2025

COURSE: IS

TASK1:

rom 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}")

### ****Explanation:****

key = get\_random\_bytes(8)  
DES requires an 8-byte key (64 bits).

**Padding:**  
Since DES works on 8-byte blocks, we use pad() to make sure data length is a multiple of 8.

**Mode:**  
DES.MODE\_ECB means we’re using the simplest DES mode (Electronic Codebook).

**Unpadding:**  
After decryption, unpad() removes the extra padding bytes.

\_\_name\_\_ == "\_\_main\_\_"  
The correct syntax has **double underscores**, not single ones.

**TASK 2:**

mport os, binascii, base64

from Crypto.Cipher import DES, AES

from Crypto.Util.Padding import pad, unpad

def mitm\_demo():

    PREF = b'\x00'\*6

    P = b'EXAMPLE!'

    # secrets chosen for demo (attacker doesn't know these)

    true1, true2 = 0x1A2B, 0x3C4D

    K1 = PREF + true1.to\_bytes(2,'big')

    K2 = PREF + true2.to\_bytes(2,'big')

    # double-DES

    C = DES.new(K2, DES.MODE\_ECB).encrypt(DES.new(K1, DES.MODE\_ECB).encrypt(P))

    # build table E\_{K1}(P) -- reduced search space for demo speed

    table = {}

    SEARCH = 1 << 12   # 4096 candidates (change to 1<<16 for fuller demo if you have time)

    for k1 in range(SEARCH):

        mid = DES.new(PREF + k1.to\_bytes(2,'big'), DES.MODE\_ECB).encrypt(P)

        table.setdefault(mid, []).append(k1)

    # try decrypting with K2 candidates and look for matches

    found = []

    for k2 in range(SEARCH):

        mid2 = DES.new(PREF + k2.to\_bytes(2,'big'), DES.MODE\_ECB).decrypt(C)

        if mid2 in table:

            for k1c in table[mid2]:

                if DES.new(PREF + k2.to\_bytes(2,'big'), DES.MODE\_ECB).encrypt(

                   DES.new(PREF + k1c.to\_bytes(2,'big'), DES.MODE\_ECB).encrypt(P)) == C:

                    found.append((k1c, k2))

    # print results

    print("=== MITM demo ===")

    print("P:", P)

    print("C(hex):", binascii.hexlify(C).decode())

    print("candidates:", len(found))

    print("true short keys:", format(true1,'04x'), format(true2,'04x'))

    if (true1, true2) in found:

        print("Recovered true pair (demo).")

    else:

        if found:

            print("Some candidates (up to 6):")

            for a,b in found[:6]:

                print(" ", format(a,'04x'), format(b,'04x'))

        else:

            print("No candidates found in this reduced search (expected).")

def aes\_demo():

    key = os.urandom(16)                # AES-128

    msg = b"Playing with AES"

    iv = os.urandom(AES.block\_size)

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

    ct = cipher.encrypt(pad(msg, AES.block\_size))

    ivct = iv + ct

    print("\n=== AES demo ===")

    print("AES key (hex):", binascii.hexlify(key).decode())

    print("IV+ciphertext (base64):", base64.b64encode(ivct).decode())

    # verify decryption

    pt = unpad(AES.new(key, AES.MODE\_CBC, iv).decrypt(ct), AES.block\_size)

    print("Decrypted plaintext:", pt)

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

    mitm\_demo()

    aes\_demo()

### MITM part (double-DES demo)

PREF = b'\x00'\*6  
→ Builds 6 zero bytes so each DES key = PREF + 2 bytes (makes keys easy to iterate).

P = b'EXAMPLE!'  
→ Known plaintext block (8 bytes = DES block size). Attacker is assumed to know this.

true1, true2 = 0x1A2B, 0x3C4D  
→ Secret demo short keys (the script uses these; attacker would not know them).

K1 = PREF + true1.to\_bytes(2,'big') and K2 = ...  
→ Convert short integers into full 8-byte DES keys.

C = E\_{K2}( E\_{K1}(P) )  
→ Compute ciphertext by encrypting P with K1, then encrypting result with K2.

table = {}; for k1 in range(SEARCH): table[E\_k1(P)].append(k1)  
→ Attacker: try many K1 candidates and store E\_k1(P) → a lookup table mapping intermediate→k1s.

for k2 in range(SEARCH): mid2 = D\_k2(C); if mid2 in table: verify pair  
→ Attacker: try many K2 candidates, compute D\_k2(C) and check if it matches any stored intermediate; verify by full re-encryption to remove false positives.

print results (P, C hex, number of candidates, true keys)  
→ Shows what was intercepted and how many candidate pairs remained after verification.

**Quick takeaway:** Instead of trying all (K1,K2) pairs, MITM computes from both sides and matches intermediate values — much faster.

### AES part (compact in-memory demo)

key = os.urandom(16)  
→ Secure random AES-128 key (16 bytes). Use os.urandom() always.

iv = os.urandom(AES.block\_size)  
→ Fresh random IV (16 bytes) for CBC mode; IV is public and should be different each message.

ct = AES.new(key, CBC, iv).encrypt(pad(msg)) and ivct = iv + ct  
→ Pad plaintext, encrypt in CBC, then prepend IV to ciphertext for transport.

pt = unpad(AES.new(key, CBC, iv).decrypt(ct))  
→ Decrypt and remove padding to recover original message (used here to verify).

**Quick takeaway:** For AES-CBC: use secure key & IV, pad correctly, and send IV with ciphertext.

Output:

