**INFORMATION SECURITY**

**LAB ASSIGNMENT 2**

**Submitted by:-**

MIAN SAIFULLAH ALAM

**Submitted to:-**

MS AMBREEN GUL

**Registration**

SP24-BSE-005

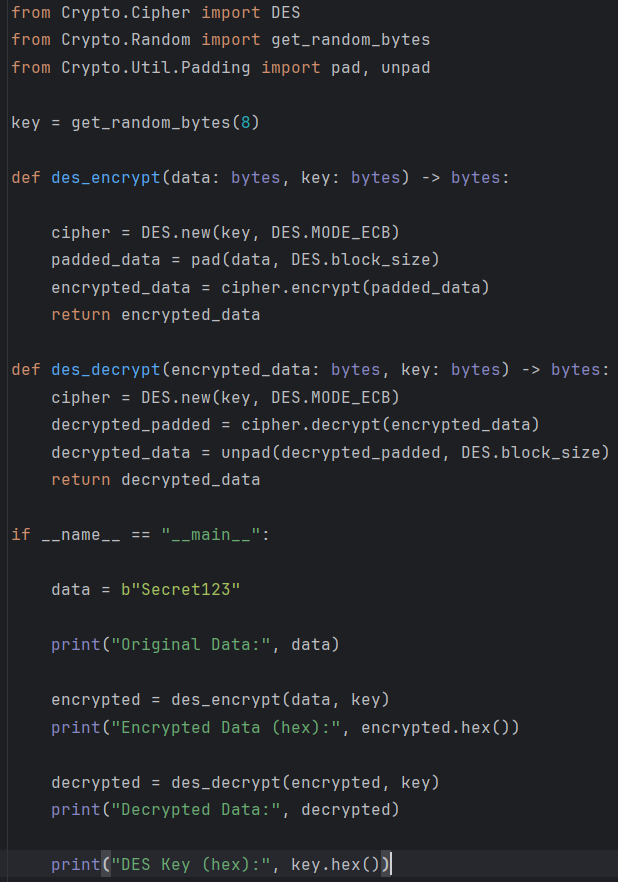
**Date:-**

12 - OCTOBER - 2025

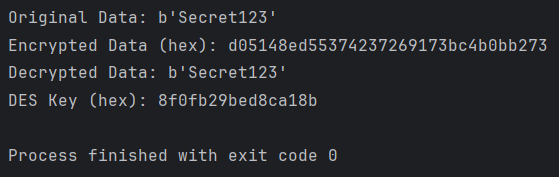
# **COMSATS UNIVERSITY ATTOCK CAMPUS**

**Graded Task 01**

**ANSWER**



**Output**



The code shows how to encrypt and decrypt data using DES (Data Encryption Standard) in ECB mode, and how to pad/unpad plaintext so it fits DES’s 8-byte block size. The same library functions are used in the lab manual example.

* **from Crypto.Cipher import DES**  
  — This imports the DES cipher class from the crypto library so we can create DES objects to encrypt and decrypt.
* **from Crypto.Random import get\_random\_bytes**  
  — This gives us a simple way to generate random bytes for the DES key.
* **from Crypto.Util.Padding import pad, unpad**  
  — DES works on 8-byte blocks. pad adds bytes so the plaintext length becomes a multiple of 8; unpad removes that added padding after decryption.
* **key = get\_random\_bytes(8)**  
  — DES requires an **8-byte key**. This line creates 8 random bytes and saves them in key. In the example we print the hex of this key so you can see which key was used.

**des\_encrypt(data: bytes, key: bytes) -> bytes**

Purpose: encrypt arbitrary-length plaintext with DES and return ciphertext.

**Step-by-step:**

1. **cipher = DES.new(key, DES.MODE\_ECB)**  
   — Create a DES cipher object in **ECB** mode using the provided 8-byte key. ECB is the mode shown in the manual.
2. **padded\_data = pad(data, DES.block\_size)**  
   — Make the plaintext length a multiple of DES’s block size (8 bytes) by adding padding bytes (PKCS#7 style). This prevents errors if the plaintext isn't exactly 8, 16, 24... bytes.
3. **encrypted\_data = cipher.encrypt(padded\_data)**  
   — Encrypt the padded data block-by-block. DES encrypts each 8-byte block and returns the concatenated ciphertext bytes.
4. **return encrypted\_data**  
   — Return the ciphertext bytes to the caller.

Simple takeaway: des\_encrypt prepares the plaintext (pad) and then encrypts it using DES ECB with the given key.

**des\_decrypt(encrypted\_data: bytes, key: bytes) -> bytes**

Purpose: decrypt ciphertext back to the original plaintext bytes.

**Step-by-step:**

1. **cipher = DES.new(key, DES.MODE\_ECB)**  
   — Create a DES cipher object with the same key and same ECB mode (must match encryption).
2. **decrypted\_padded = cipher.decrypt(encrypted\_data)**  
   — Decrypt the ciphertext to get the padded plaintext (still has padding bytes).
3. **decrypted\_data = unpad(decrypted\_padded, DES.block\_size)**— Remove the padding bytes added during encryption to get the original plaintext length.
4. **return decrypted\_data**  
   — Return the original plaintext bytes.

Simple takeaway: des\_decrypt reverses des\_encrypt — decrypt then unpad.

**if \_\_name\_\_ == "\_\_main\_\_": — Demo run**

This part shows how to use the two functions. It’s what runs when you execute the file.

**Step-by-step:**

1. **data = b"Secret123"**  
   — Choose a plaintext to encrypt. It must be bytes, so we prefix with b"...". In the example the plaintext is 9 bytes long.
2. **print("Original Data:", data)**  
   — Print the plaintext so you can check it.
3. **encrypted = des\_encrypt(data, key)**  
   — Call the encryption function defined above. This returns ciphertext (bytes).
4. **print("Encrypted Data (hex):", encrypted.hex())**  
   — Print the ciphertext in hexadecimal so it’s readable (binary bytes shown as hex string).
5. **decrypted = des\_decrypt(encrypted, key)**  
   — Decrypt the ciphertext to recover the original plaintext bytes.
6. **print("Decrypted Data:", decrypted)**  
   — Print the decrypted result (should equal the original data).
7. **print("DES Key (hex):", key.hex())**  
   — Print the random 8-byte key used (in hex) so you can reproduce or verify results.

Simple takeaway: This block shows encrypt → decrypt and prints everything so you can verify the round-trip works.

**security notes**

* DES uses 8-byte keys and is **outdated** for real security — it’s used here only because the manual teaches it. Real systems use AES or other modern ciphers.
* ECB mode (used here) is insecure for many uses because identical plaintext blocks produce identical ciphertext blocks. The manual uses ECB for teaching; in real systems choose CBC, GCM, or other secure modes and manage IVs correctly.
* Padding/unpadding is necessary because DES operates on fixed-size (8-byte) blocks.

This program demonstrates DES encryption and decryption in ECB mode: it pads the plaintext to fit 8-byte blocks, encrypts with a randomly generated 8-byte DES key, and then decrypts and unpads to recover the original message; the same key and mode must be used to correctly reverse the operation.

**Graded Task 02**

**Answer**

**int\_to\_des\_key(n: int) -> bytes**

* Purpose: turn a small integer (like 5) into an 8-byte DES key (DES needs 8 bytes).
* What it does: n.to\_bytes(8,'big') makes bytes like b'\x00\x00\x00\x00\x00\x00\x00\x05'.
* Why: this makes simple, unique keys for small integers so we can test many keys quickly in class.

**des\_encrypt\_block(block, key) and des\_decrypt\_block(block, key)**

* Purpose: encrypt or decrypt exactly one DES block (8 bytes) using ECB mode.
* What they do: create a DES cipher object from the key and call encrypt or decrypt.
* Important note: we do not pad here — the MITM explanation works on single 8-byte blocks.

**double\_des\_encrypt\_block(plain\_block, k1\_int, k2\_int)**

* Purpose: simulate the victim encrypting with double-DES (first K1 then K2).
* **Steps:**
  1. Convert k1\_int and k2\_int into 8-byte keys (with int\_to\_des\_key).
  2. Encrypt plaintext with K1 to get an intermediate block.
  3. Encrypt that intermediate block with K2 to get the final ciphertext.
* This simulates the real encryption the attacker observes (P and C).

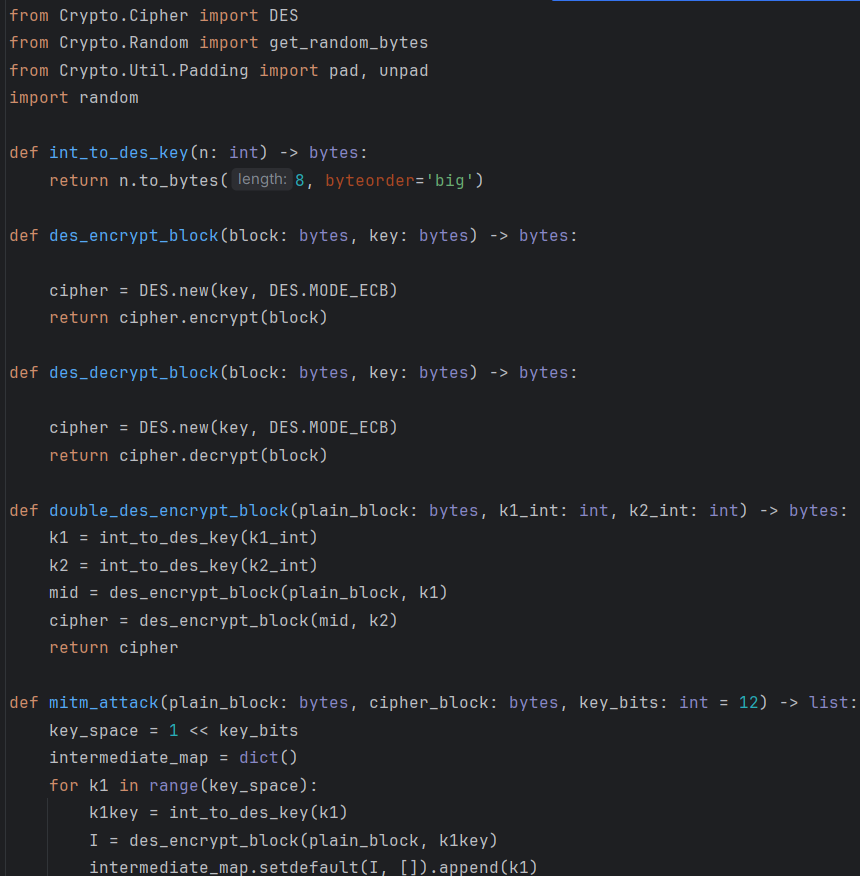
**mitm\_attack(plain\_block, cipher\_block, key\_bits=12)**

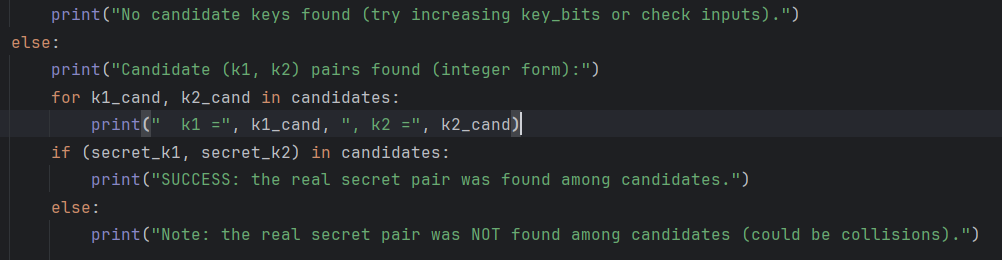
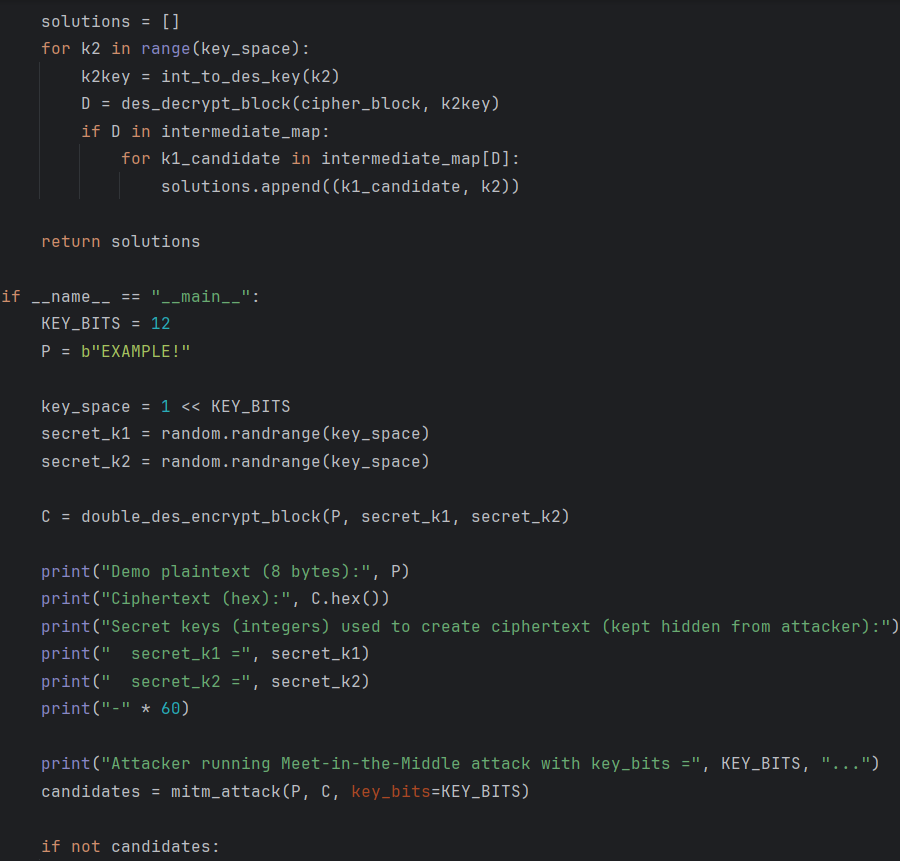
This is the heart of the MITM method — exactly the flow shown in your lab manual.

* Purpose: find candidate key pairs (K1, K2) such that E\_{K2}(E\_{K1}(P)) == C.
* Key idea from the manual: split the search into two halves so you don't try all K1×K2 combinations.
* How it works (step-by-step):
  1. **Stage 1 (forward):** For every possible k1 in [0, 2^key\_bits):
     + Compute I = E\_{k1}(P) (encrypt plaintext with k1).
     + Store I → k1 in a dictionary (a map from intermediate value to keys that produced it).
     + After this loop, we have a table of possible intermediate values for every k1.
  2. **Stage 2 (backward):** For every possible k2 in [0, 2^key\_bits):
     + Compute D = D\_{k2}(C) (decrypt ciphertext with k2).
     + See if D exists in the Stage 1 table (i.e., if some k1 produced the same intermediate value).
     + If yes, then (k1, k2) is a candidate — their intermediate values match and therefore  
       E\_{k2}( E\_{k1}(P) ) == C for that pair.
* **Why this is faster:** Instead of testing 2^key\_bits × 2^key\_bits combos, we do 2 × 2^key\_bits operations and a lookup (much faster in practice).

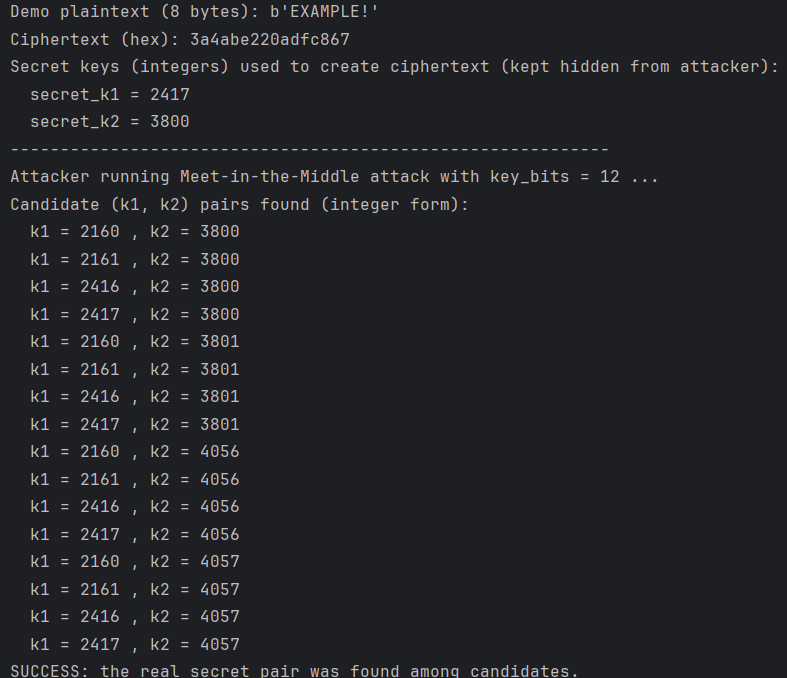
**if \_\_name\_\_ == "\_\_main\_\_": (demo run)**

* KEY\_BITS = 12 — we pick a **small** key space so the code runs quickly on your computer (2^12 = 4096 possibilities per half). In real systems keys are much larger. The manual explains splitting the key into halves; here we pick a small half-size for demonstration.
* P = b"EXAMPLE!" — an 8-byte plaintext block (DES block size).
* We randomly pick secret\_k1 and secret\_k2 (these simulate the real keys the victim used).
* We compute the observed ciphertext C by double-encrypting P with secret\_k1 then secret\_k2.
* The attacker knows P and C and runs mitm\_attack(P, C, key\_bits=KEY\_BITS) to find candidate key pairs.
* The program prints:
  + The ciphertext (so you can see it),
  + The candidate pairs found by the attack,
  + Whether the real secret pair appears among candidates.
* Meet-in-the-Middle splits the key search in two: encrypt the known plaintext with all possible K1 values, decrypt the ciphertext with all possible K2 values, and find matches — this recovers the key pair much faster than naive full search. (This is exactly what the lab manual explains.)





**Output**

****