PRACTICAL NO 1

Aim: Program to implement password salting and hashing to create secure passwords.

code: -

import hashlib

import os

# Function to create a salted hash of the password

def hash\_password(password):

# Generate a random 16-byte salt

salt = os.urandom(16)

# Combine salt and password, then hash it using SHA-256

password\_bytes = password.encode('utf-8') # Convert password to bytes

salted\_password = salt + password\_bytes # Combine salt and password

hashed = hashlib.sha256(salted\_password).hexdigest() # Hash it

# Return both the salt and the hash

return salt.hex(), hashed

# Function to verify password

def verify\_password(stored\_salt, stored\_hash, password\_to\_check):

# Convert stored salt back to bytes

salt = bytes.fromhex(stored\_salt)

# Hash the password to check using the stored salt

password\_bytes = password\_to\_check.encode('utf-8')

hashed = hashlib.sha256(salt + password\_bytes).hexdigest()

# Compare with stored hash

return hashed == stored\_hash

# -------------------------------

# 🔐 Demo for students

# -------------------------------

# 1. User creates password

original\_password = input("Create a password: ")

salt, hashed\_password = hash\_password(original\_password)

print("\nPassword has been securely hashed and salted.")

print("Stored Salt:", salt)

print("Stored Hash:", hashed\_password)

# 2. Later, user enters password to log in

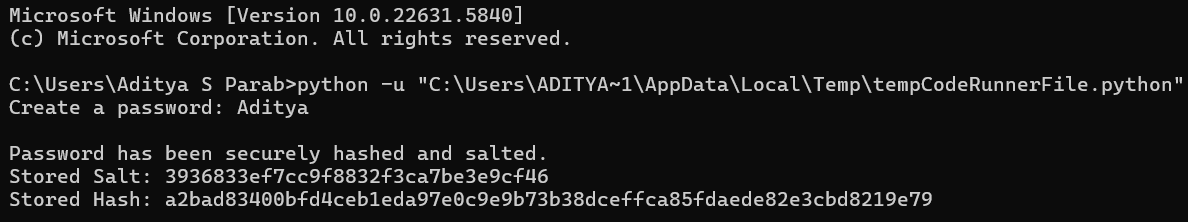
login\_password = input("\nEnter your password to login: ")

if verify\_password(salt, hashed\_password, login\_password):

print("✅ Password is correct! Access granted.")

else:

print("❌ Incorrect password! Access denied.")

Output :-

PRACTICAL NO 2

Aim: Program to implement various classical ciphers-Substitution Cipher, Vigenère Cipher, and Affine cipher

def substitution\_encrypt(text, shift):

result = ""

for char in text:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

result += chr((ord(char) - base + shift) % 26 + base)

else:

result += char

return result

def substitution\_decrypt(text, shift):

return substitution\_encrypt(text, -shift)

def vigenere\_encrypt(text, key):

result = ""

key = key.lower()

key\_index = 0

for char in text:

if char.isalpha():

shift = ord(key[key\_index % len(key)]) - ord('a')

base = ord('A') if char.isupper() else ord('a')

result += chr((ord(char) - base + shift) % 26 + base)

key\_index += 1

else:

result += char

return result

def vigenere\_decrypt(text, key):

result = ""

key = key.lower()

key\_index = 0

for char in text:

if char.isalpha():

shift = ord(key[key\_index % len(key)]) - ord('a')

base = ord('A') if char.isupper() else ord('a')

result += chr((ord(char) - base - shift) % 26 + base)

key\_index += 1

else:

result += char

return result

def mod\_inverse(a, m):

for i in range(1, m):

if (a \* i) % m == 1:

return i

return None

def affine\_encrypt(text, a, b):

result = ""

for char in text:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

x = ord(char) - base

result += chr((a \* x + b) % 26 + base)

else:

result += char

return result

def affine\_decrypt(text, a, b):

result = ""

a\_inv = mod\_inverse(a, 26)

if a\_inv is None:

return "Invalid 'a'. No modular inverse exists."

for char in text:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

y = ord(char) - base

result += chr((a\_inv \* (y - b)) % 26 + base)

else:

result += char

return result

def main():

print("Classical Cipher Program")

print("1. Substitution Cipher")

print("2. Vigenère Cipher")

print("3. Affine Cipher")

choice = input("Enter your choice (1/2/3): ")

message = input("Enter your message: ")

if choice == '1':

shift = int(input("Enter shift value (e.g. 3): "))

enc = substitution\_encrypt(message, shift)

dec = substitution\_decrypt(enc, shift)

print("Encrypted:", enc)

print("Decrypted:", dec)

elif choice == '2':

key = input("Enter keyword (e.g. KEY): ")

enc = vigenere\_encrypt(message, key)

dec = vigenere\_decrypt(enc, key)

print("Encrypted:", enc)

print("Decrypted:", dec)

elif choice == '3':

a = int(input("Enter 'a' (must be coprime with 26, e.g. 5): "))

b = int(input("Enter 'b' (e.g. 8): "))

enc = affine\_encrypt(message, a, b)

dec = affine\_decrypt(enc, a, b)

print("Encrypted:", enc)

print("Decrypted:", dec)

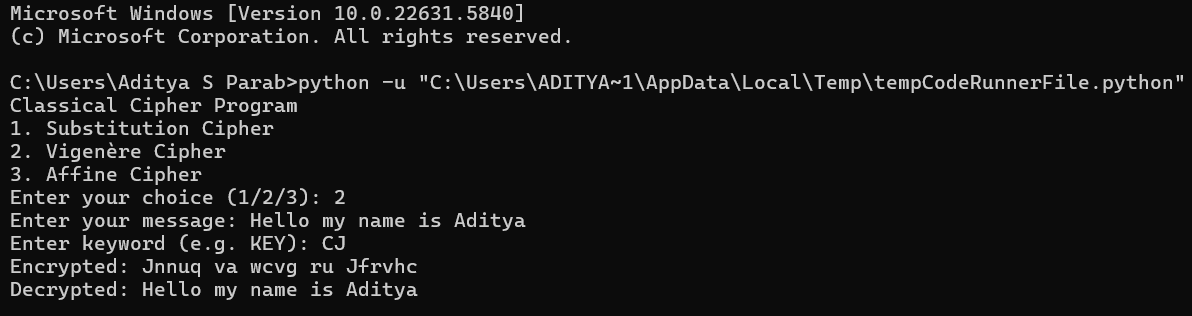
else:

print("Invalid choice.")

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

main()

Output :-



PRACTICAL NO 3

Aim: Program to demonstrate cryptanalysis (e.g., breaking Caesar or Vigener Cipher)

Code: -

*def* caesar\_decrypt(*ciphertext*, *shift*):

"""Decrypt a Caesar cipher text with a given shift."""

result = ""

for char in *ciphertext*:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

# Shift letter backwards and wrap around using modulo 26

result += chr((ord(char) - base - *shift*) % 26 + base)

else:

# Non-alphabetic characters remain unchanged

result += char

return result

# Example Caesar cipher text

caesar\_cipher\_text = "Khoor Zruog" # "Hello World" encrypted with shift 3

print("=== Breaking Caesar Cipher (Brute Force) ===\n")

for key in range(26):

decrypted = caesar\_decrypt(caesar\_cipher\_text, key)

print(*f*"Key {key*:2d*}: {decrypted}")

# -----------------------

# 2. Vigenère Cipher Decryption (with known key)

# -----------------------

*def* vigenere\_decrypt(*ciphertext*, *key*):

"""Decrypt a Vigenère cipher text with a given key."""

result = ""

*key* = *key*.lower()

key\_index = 0

for char in *ciphertext*:

if char.isalpha():

base = ord('A') if char.isupper() else ord('a')

# Determine shift from the key letter

shift = ord(*key*[key\_index % len(*key*)]) - ord('a')

result += chr((ord(char) - base - shift) % 26 + base)

key\_index += 1

else:

# Keep spaces/punctuation as they are

result += char

return result

# Example Vigenère cipher text

vigenere\_cipher\_text = "Rijvs Uyvjn" # "Hello World" encrypted with key "key"

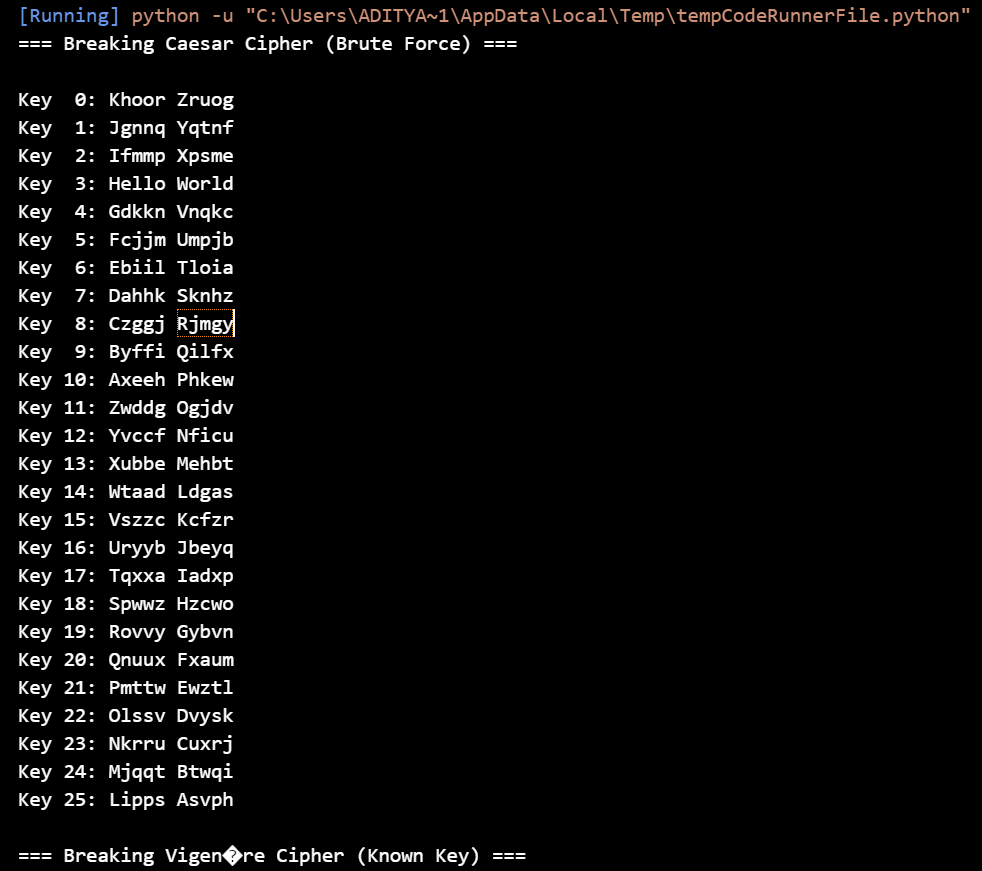
key\_guess = "key" # Suppose we guessed the key

print("\n=== Breaking Vigenère Cipher (Known Key) ===\n")

vigenere\_decrypted = vigenere\_decrypt(vigenere\_cipher\_text, key\_guess)

print(*f*"Key = '{key\_guess}' → {vigenere\_decrypted}")

Output: -



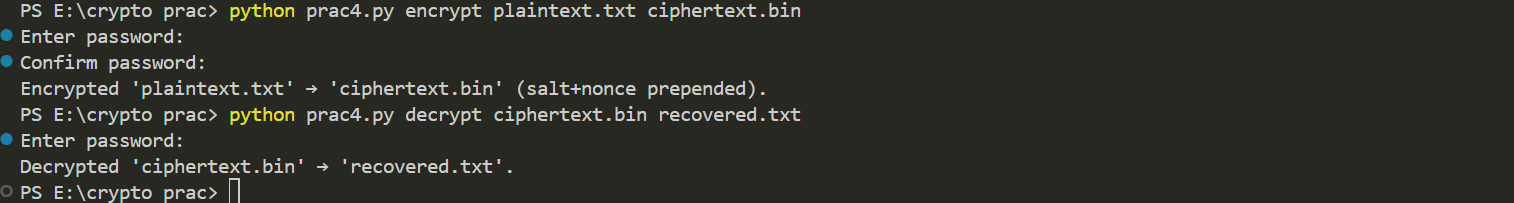
PRACTICAL NO 4

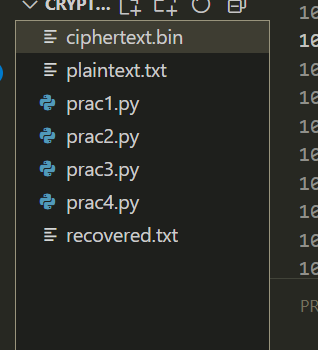
Aim: Program to implement AES algorithm for file encryption and decryption

Code: -

import os  
import argparse  
import getpass  
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC  
from cryptography.hazmat.primitives import hashes  
from cryptography.hazmat.primitives.ciphers.aead import AESGCM  
from cryptography.exceptions import InvalidTag  
  
  
SALT\_SIZE = 16 # 128-bit salt  
NONCE\_SIZE = 12 # nonce size recommended for GCM (96 bits)  
KEY\_SIZE = 32 # 256-bit AES key  
PBKDF2\_ITERATIONS = 600\_000 # iteration count for PBKDF2  
  
  
def derive\_key\_from\_password(password: str, salt: bytes, iterations: int = PBKDF2\_ITERATIONS) -> bytes:  
"""  
Derive a symmetric key from a password using PBKDF2-HMAC-SHA256.  
Returns KEY\_SIZE bytes suitable for AES-256.  
"""  
password\_bytes = password.encode('utf-8')  
kdf = PBKDF2HMAC(  
algorithm=hashes.SHA256(),  
length=KEY\_SIZE,  
salt=salt,  
iterations=iterations,  
)  
return kdf.derive(password\_bytes)  
  
  
def encrypt\_file(in\_path: str, out\_path: str, password: str) -> None:  
"""  
Encrypt the entire input file and write to out\_path.  
Format: [ salt (16) | nonce (12) | ciphertext+tag (rest) ]  
"""  
with open(in\_path, 'rb') as f:  
plaintext = f.read()  
  
salt = os.urandom(SALT\_SIZE)  
key = derive\_key\_from\_password(password, salt)  
aesgcm = AESGCM(key)  
nonce = os.urandom(NONCE\_SIZE)  
  
ciphertext\_and\_tag = aesgcm.encrypt(nonce, plaintext, associated\_data=None)  
  
with open(out\_path, 'wb') as f:  
f.write(salt + nonce + ciphertext\_and\_tag)  
  
print(f"Encrypted '{in\_path}' → '{out\_path}' (salt+nonce prepended).")  
  
  
def decrypt\_file(in\_path: str, out\_path: str, password: str) -> None:  
"""  
Decrypt file written by encrypt\_file.  
Expects format: [ salt (16) | nonce (12) | ciphertext+tag (rest) ]  
"""  
with open(in\_path, 'rb') as f:  
file\_bytes = f.read()  
  
if len(file\_bytes) < (SALT\_SIZE + NONCE\_SIZE + 16):  
raise ValueError("Input file is too short or not a valid encrypted file.")  
  
salt = file\_bytes[:SALT\_SIZE]  
nonce = file\_bytes[SALT\_SIZE:SALT\_SIZE + NONCE\_SIZE]  
ciphertext\_and\_tag = file\_bytes[SALT\_SIZE + NONCE\_SIZE:]  
  
key = derive\_key\_from\_password(password, salt)  
aesgcm = AESGCM(key)  
  
try:  
plaintext = aesgcm.decrypt(nonce, ciphertext\_and\_tag, associated\_data=None)  
except InvalidTag:  
raise InvalidTag("Decryption failed: authentication tag is invalid (wrong password or corrupted file).")  
  
with open(out\_path, 'wb') as f:  
f.write(plaintext)  
  
print(f"Decrypted '{in\_path}' → '{out\_path}'.")  
  
  
def main():  
parser = argparse.ArgumentParser(description="AES-GCM file encrypt/decrypt (password-based).")  
parser.add\_argument('mode', choices=['encrypt', 'decrypt'], help="encrypt or decrypt")  
parser.add\_argument('infile', help="input file path")  
parser.add\_argument('outfile', help="output file path")  
args = parser.parse\_args()  
  
if not os.path.isfile(args.infile):  
print("Error: input file does not exist.")  
return  
  
password = getpass.getpass("Enter password: ")  
if not password:  
print("Error: empty password not allowed.")  
return  
  
if args.mode == 'encrypt':  
password\_confirm = getpass.getpass("Confirm password: ")  
if password != password\_confirm:  
print("Error: passwords do not match.")  
return  
  
try:  
if args.mode == 'encrypt':  
encrypt\_file(args.infile, args.outfile, password)  
else:  
decrypt\_file(args.infile, args.outfile, password)  
except InvalidTag as e:  
print("Decryption/Authentication failed:", e)  
except Exception as e:  
print("Error:", e)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
main()

Output:-





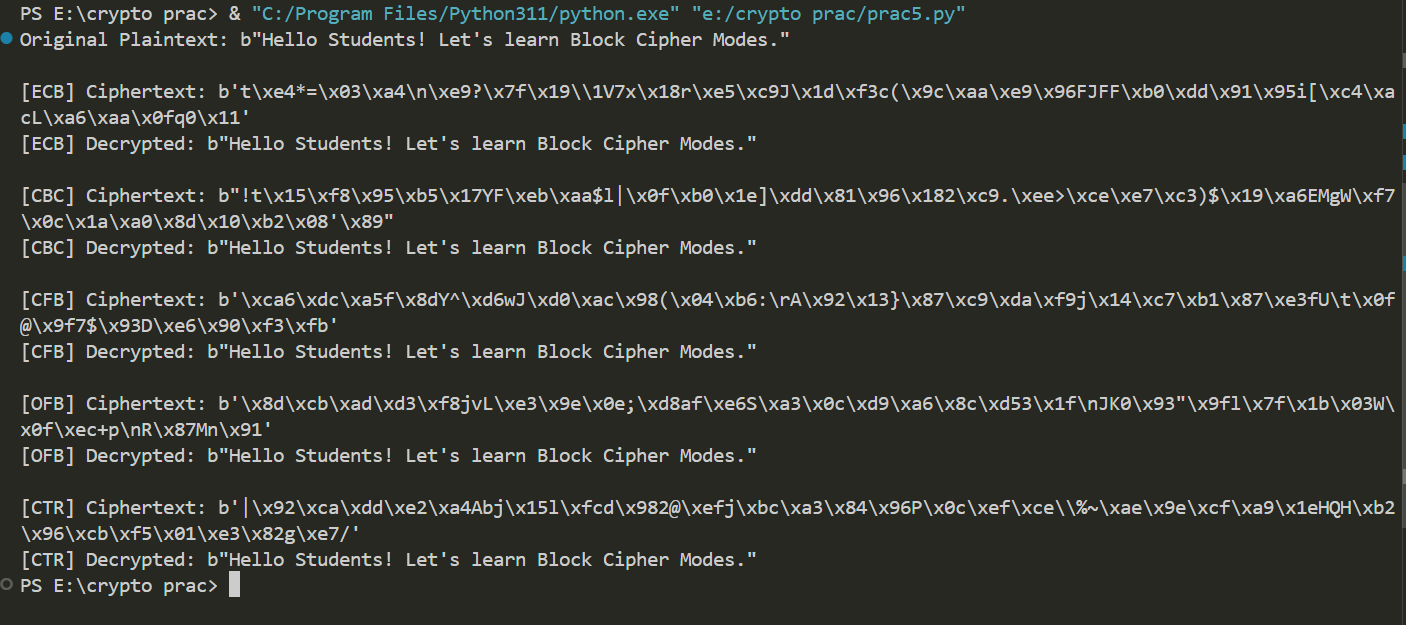
PRACTICAL NO 5

Aim: Program to implement various block cipher modes

Code:

# Install required library:  
# pip install pycryptodomex  
  
from Crypto.Cipher import AES  
from Crypto.Random import get\_random\_bytes  
from Crypto.Util.Padding import pad, unpad  
from Crypto.Util import Counter  
  
  
  
# 🔑 AES requires a key of length 16, 24, or 32 bytes  
key = get\_random\_bytes(16) # 128-bit key  
plaintext = b"Hello Students! Let's learn Block Cipher Modes."  
  
print("Original Plaintext:", plaintext)  
  
# ------------------------ ECB MODE ------------------------  
cipher\_ecb = AES.new(key, AES.MODE\_ECB)  
ciphertext\_ecb = cipher\_ecb.encrypt(pad(plaintext, AES.block\_size))  
print("\n[ECB] Ciphertext:", ciphertext\_ecb)  
  
decipher\_ecb = AES.new(key, AES.MODE\_ECB)  
decrypted\_ecb = unpad(decipher\_ecb.decrypt(ciphertext\_ecb), AES.block\_size)  
print("[ECB] Decrypted:", decrypted\_ecb)  
  
# ------------------------ CBC MODE ------------------------  
iv\_cbc = get\_random\_bytes(16)  
cipher\_cbc = AES.new(key, AES.MODE\_CBC, iv\_cbc)  
ciphertext\_cbc = cipher\_cbc.encrypt(pad(plaintext, AES.block\_size))  
print("\n[CBC] Ciphertext:", ciphertext\_cbc)  
  
decipher\_cbc = AES.new(key, AES.MODE\_CBC, iv\_cbc)  
decrypted\_cbc = unpad(decipher\_cbc.decrypt(ciphertext\_cbc), AES.block\_size)  
print("[CBC] Decrypted:", decrypted\_cbc)  
  
# ------------------------ CFB MODE ------------------------  
iv\_cfb = get\_random\_bytes(16)  
cipher\_cfb = AES.new(key, AES.MODE\_CFB, iv\_cfb)  
ciphertext\_cfb = cipher\_cfb.encrypt(plaintext)  
print("\n[CFB] Ciphertext:", ciphertext\_cfb)  
  
decipher\_cfb = AES.new(key, AES.MODE\_CFB, iv\_cfb)  
decrypted\_cfb = decipher\_cfb.decrypt(ciphertext\_cfb)  
print("[CFB] Decrypted:", decrypted\_cfb)  
  
# ------------------------ OFB MODE ------------------------  
iv\_ofb = get\_random\_bytes(16)  
cipher\_ofb = AES.new(key, AES.MODE\_OFB, iv\_ofb)  
ciphertext\_ofb = cipher\_ofb.encrypt(plaintext)  
print("\n[OFB] Ciphertext:", ciphertext\_ofb)  
  
decipher\_ofb = AES.new(key, AES.MODE\_OFB, iv\_ofb)  
decrypted\_ofb = decipher\_ofb.decrypt(ciphertext\_ofb)  
print("[OFB] Decrypted:", decrypted\_ofb)  
  
# ------------------------ CTR MODE ------------------------  
ctr\_enc = Counter.new(128)  
cipher\_ctr = AES.new(key, AES.MODE\_CTR, counter=ctr\_enc)  
ciphertext\_ctr = cipher\_ctr.encrypt(plaintext)  
print("\n[CTR] Ciphertext:", ciphertext\_ctr)  
  
# Reset counter for decryption  
ctr\_dec = Counter.new(128)  
decipher\_ctr = AES.new(key, AES.MODE\_CTR, counter=ctr\_dec)  
decrypted\_ctr = decipher\_ctr.decrypt(ciphertext\_ctr)  
print("[CTR] Decrypted:", decrypted\_ctr)

Output: -



PRACTICAL NO 6

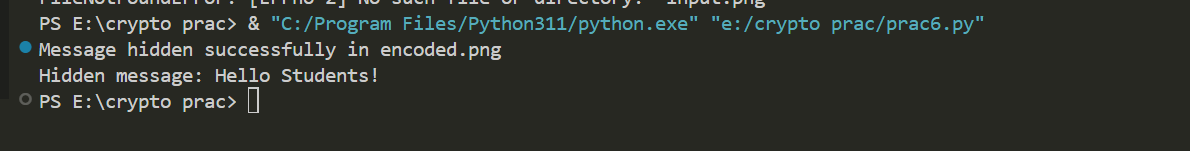
Aim: Program to implement Steganography for hiding messages inside the image file.

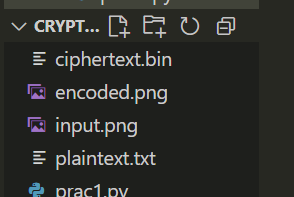
Code:

from PIL import Image

# Function to hide a secret message in an image  
def hide\_message(image\_path, message, output\_path):  
 # Open the image  
 img = Image.open(image\_path)  
 encoded = img.copy()  
 width, height = img.size  
 index = 0  
  
 # Append a special character at the end of the message to mark the stop point  
 message += "###"  
  
 for row in range(height):  
 for col in range(width):  
 if index < len(message) \* 8: # Each character needs 8 bits  
 pixel = list(img.getpixel((col, row)))  
  
 # Modify the LSB of the pixel's red channel  
 ascii\_val = ord(message[index // 8]) # character to ASCII  
 bit\_val = (ascii\_val >> (index % 8)) & 1 # extract bit  
 pixel[0] = pixel[0] & ~1 | bit\_val # set LSB  
  
 encoded.putpixel((col, row), tuple(pixel))  
 index += 1  
 encoded.save(output\_path)  
 print(f"Message hidden successfully in {output\_path}")  
  
  
# Function to extract hidden message from an image  
def reveal\_message(image\_path):  
 img = Image.open(image\_path)  
 width, height = img.size  
 bits = []  
 message = ""  
  
 for row in range(height):  
 for col in range(width):  
 pixel = list(img.getpixel((col, row)))  
 bits.append(pixel[0] & 1) # extract LSB  
  
 if len(bits) == 8: # One character formed  
 char = chr(int("".join(str(bit) for bit in bits[::-1]), 2))  
 message += char  
 bits = []  
 if message.endswith("###"): # Stop when delimiter is found  
 return message[:-3]  
 return message  
  
  
# ----------------- DEMO -----------------  
# Hide message  
hide\_message("input.png", "Hello Students!", "encoded.png")  
  
# Reveal message  
secret = reveal\_message("encoded.png")  
print("Hidden message:", secret)

Output:-





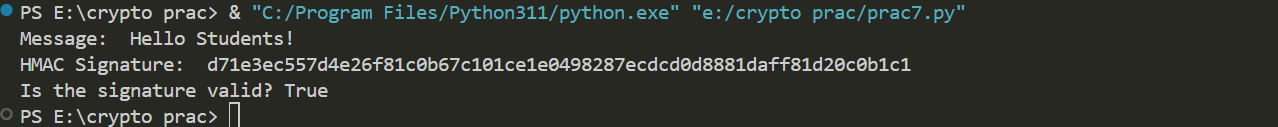
PRACTICAL NO 7

Aim: Program to implement HMAC for signing messages.

Code:

import hmac  
import hashlib  
  
# Function to generate HMAC signature  
def generate\_hmac(key, message):  
 # Create HMAC object using SHA256  
 signature = hmac.new(key.encode(), message.encode(), hashlib.sha256)  
 return signature.hexdigest()  
  
# Function to verify HMAC signature  
def verify\_hmac(key, message, signature):  
 # Generate a new signature for the message  
 new\_signature = generate\_hmac(key, message)  
 # Compare securely  
 return hmac.compare\_digest(new\_signature, signature)  
  
# Example usage  
if \_\_name\_\_ == "\_\_main\_\_":  
 secret\_key = "my\_secret\_key" # Shared secret key  
 message = "Hello Students!" # Message to be signed  
  
 # Generate signature  
 signature = generate\_hmac(secret\_key, message)  
 print("Message: ", message)  
 print("HMAC Signature: ", signature)  
  
 # Verify signature  
 is\_valid = verify\_hmac(secret\_key, message, signature)  
 print("Is the signature valid?", is\_valid)

Output:-



PRACTICAL NO 8A

Aim: Program to implement Sending Secure Messages Over IP Networks.

Code:

# secure\_server.py

import socket

from cryptography.hazmat.primitives.ciphers.aead import AESGCM

HOST = '127.0.0.1'

PORT = 65432

# KEY = AESGCM.generate\_key(bit\_length=128) # In practice, share this securely

KEY = b"0123456789abcdef0123456789abcdef" # 32 bytes → AES-256

print("Key length:", len(KEY)) # Debugging

def start\_server():

print("Server running...")

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

s.bind((HOST, PORT))

s.listen()

conn, addr = s.accept()

with conn:

print(f"Connected by {addr}")

nonce = conn.recv(12) # 96-bit nonce for AES-GCM

ciphertext = conn.recv(1024)

aesgcm = AESGCM(KEY)

try:

plaintext = aesgcm.decrypt(nonce, ciphertext, None)

print("Decrypted message:", plaintext.decode())

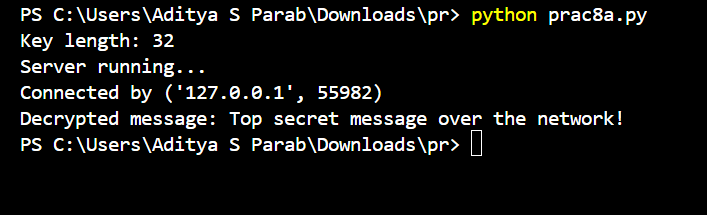
except Exception as e:

print("Decryption failed:", e)

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

start\_server()

Output:-



PRACTICAL NO 8B

Aim: Program to implement Sending Secure Messages Over IP Networks.

Code:

# secure\_client.py

import socket

from cryptography.hazmat.primitives.ciphers.aead import AESGCM

import os

HOST = '127.0.0.1'

PORT = 65432

# KEY = AESGCM.generate\_key(bit\_length=128) # Same key as server in real setup

KEY = b"0123456789abcdef0123456789abcdef" # 32 bytes → AES-256

print("Key length:", len(KEY)) # Debugging

def send\_secure\_message(message: str):

aesgcm = AESGCM(KEY)

nonce = os.urandom(12) # GCM standard nonce size

print(message)

ciphertext = aesgcm.encrypt(nonce, message.encode(), None)

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

s.connect((HOST, PORT))

s.sendall(nonce)

s.sendall(ciphertext)

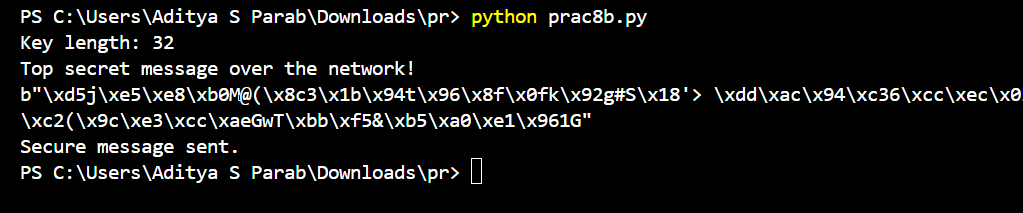
print(ciphertext)

print("Secure message sent.")

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

send\_secure\_message("Top secret message over the network!")

Output:-



PRACTICAL NO 9

Aim: Program to implement RSA encryption/decryption

Code:

Note - pip install pycryptodomex

from Cryptodome.PublicKey import RSA

from Cryptodome.Cipher import PKCS1\_OAEP

# Step 1: Generate RSA key pair (public + private)

key = RSA.generate(2048) # 2048-bit key

private\_key = key.export\_key()

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

print("🔑 Private Key:")

print(private\_key.decode())

print("\n🔑 Public Key:")

print(public\_key.decode())

# Step 2: Import keys for encryption/decryption

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

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

# Step 3: Create cipher objects

encryptor = PKCS1\_OAEP.new(public\_key\_obj)

decryptor = PKCS1\_OAEP.new(private\_key\_obj)

# Step 4: Encrypt a message

message = "Hello Students, RSA is secure!"

print("\n📩 Original Message:", message)

encrypted\_msg = encryptor.encrypt(message.encode())

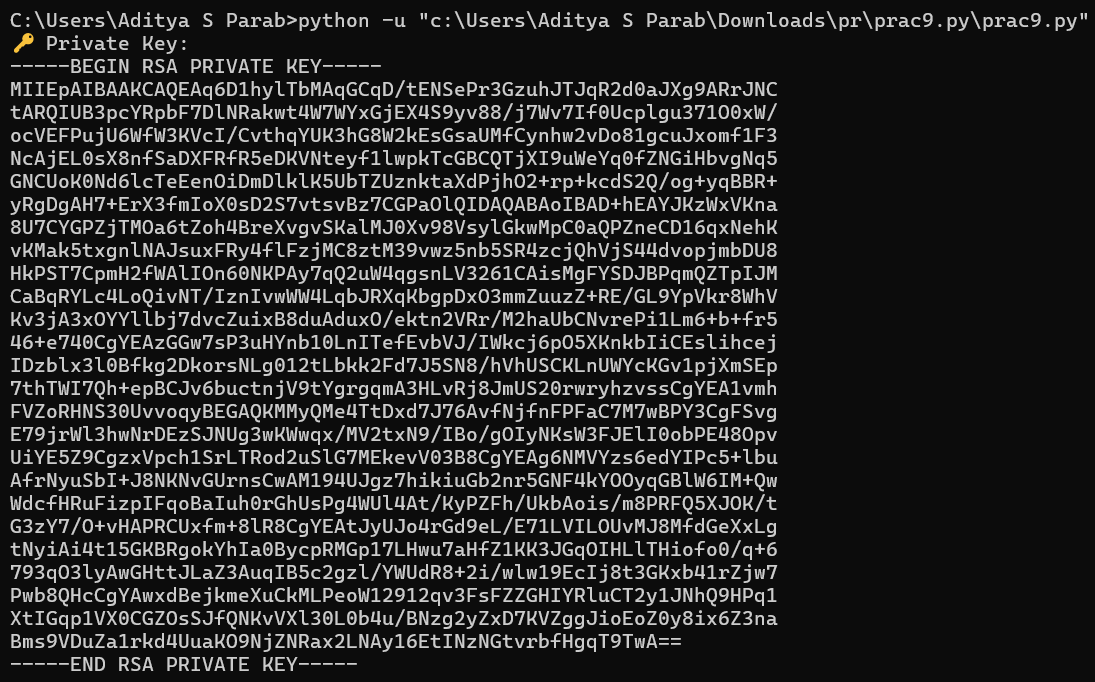
print("\n🔒 Encrypted Message:", encrypted\_msg)

# Step 5: Decrypt the message

decrypted\_msg = decryptor.decrypt(encrypted\_msg).decode()

print("\n✅ Decrypted Message:", decrypted\_msg)

Output: -





PRACTICAL NO 10A

Aim: Program to implement (i) El-Gamal Cryptosystem (ii) Elliptic Curve Cryptography

Code:

from Cryptodome.Random import random

from Cryptodome.Util.number import getPrime, inverse, GCD

# Key Generation

def generate\_keys(bits=256):

p = getPrime(bits)

g = random.randint(2, p - 1)

x = random.randint(1, p - 2) # Private key

y = pow(g, x, p) # Public key

return (p, g, y), x

# Encryption: ciphertext = (c1, c2)

def encrypt(p, g, y, message):

k = random.randint(1, p - 2)

c1 = pow(g, k, p)

s = pow(y, k, p)

c2 = (s \* message) % p

return c1, c2

# Decryption: message = c2 \* s^-1 mod p

def decrypt(p, x, c1, c2):

s = pow(c1, x, p)

s\_inv = inverse(s, p)

message = (c2 \* s\_inv) % p

return message

# Example usage

def main():

# Generate keys

public\_key, private\_key = generate\_keys()

p, g, y = public\_key

# Message to encrypt (must be an integer < p)

original\_message = 12345

print("Original Message:", original\_message)

# Encrypt

c1, c2 = encrypt(p, g, y, original\_message)

print("Encrypted:", (c1, c2))

# Decrypt

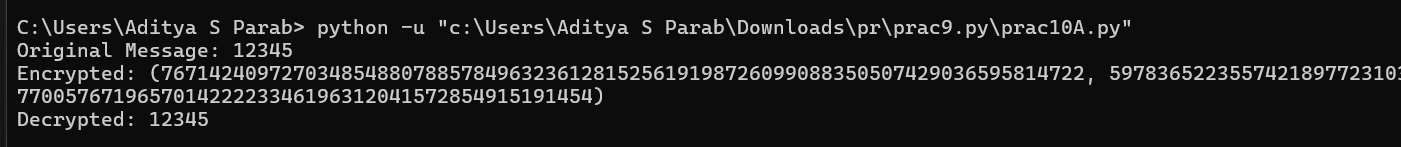
decrypted = decrypt(p, private\_key, c1, c2)

print("Decrypted:", decrypted)

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

main()

Output:-



PRACTICAL NO 10B

Aim: Program to implement (i) El-Gamal Cryptosystem (ii) Elliptic Curve Cryptography

Code:

from cryptography.hazmat.primitives.asymmetric import ec

from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.ciphers.aead import AESGCM

import os

# Generate ECC key pair

def generate\_ecc\_keys():

private\_key = ec.generate\_private\_key(ec.SECP256R1())

public\_key = private\_key.public\_key()

return private\_key, public\_key

# Derive shared secret and encrypt using AES-GCM

def encrypt\_message(sender\_private\_key, receiver\_public\_key, plaintext):

shared\_key = sender\_private\_key.exchange(ec.ECDH(), receiver\_public\_key)

# Derive symmetric key using HKDF

derived\_key = HKDF(

algorithm=hashes.SHA256(),

length=32,

salt=None,

info=b'ecc encryption',

).derive(shared\_key)

aesgcm = AESGCM(derived\_key)

nonce = os.urandom(12)

ciphertext = aesgcm.encrypt(nonce, plaintext.encode(), None)

return nonce, ciphertext

# Decrypt message

def decrypt\_message(receiver\_private\_key, sender\_public\_key, nonce, ciphertext):

shared\_key = receiver\_private\_key.exchange(ec.ECDH(), sender\_public\_key)

derived\_key = HKDF(

algorithm=hashes.SHA256(),

length=32,

salt=None,

info=b'ecc encryption',

).derive(shared\_key)

aesgcm = AESGCM(derived\_key)

plaintext = aesgcm.decrypt(nonce, ciphertext, None)

return plaintext.decode()

# Demo

def main():

# Generate sender and receiver key pairs

sender\_private, sender\_public = generate\_ecc\_keys()

receiver\_private, receiver\_public = generate\_ecc\_keys()

# Encrypt message

message = "Elliptic Curve Crypto is cool!"

print("Original Message: ", message)

nonce, ciphertext = encrypt\_message(sender\_private, receiver\_public, message)

print("Encrypted:", ciphertext.hex())

# Decrypt message

decrypted = decrypt\_message(receiver\_private, sender\_public, nonce, ciphertext)

print("Decrypted:", decrypted)

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

main()

Output:-

