K.R. MANGALAM UNIVERSITY
THE COMPLETE WORLD OF EDUCATION

Lab Assignment - 2
CRYPTOGRAPHY
(ENSP207)

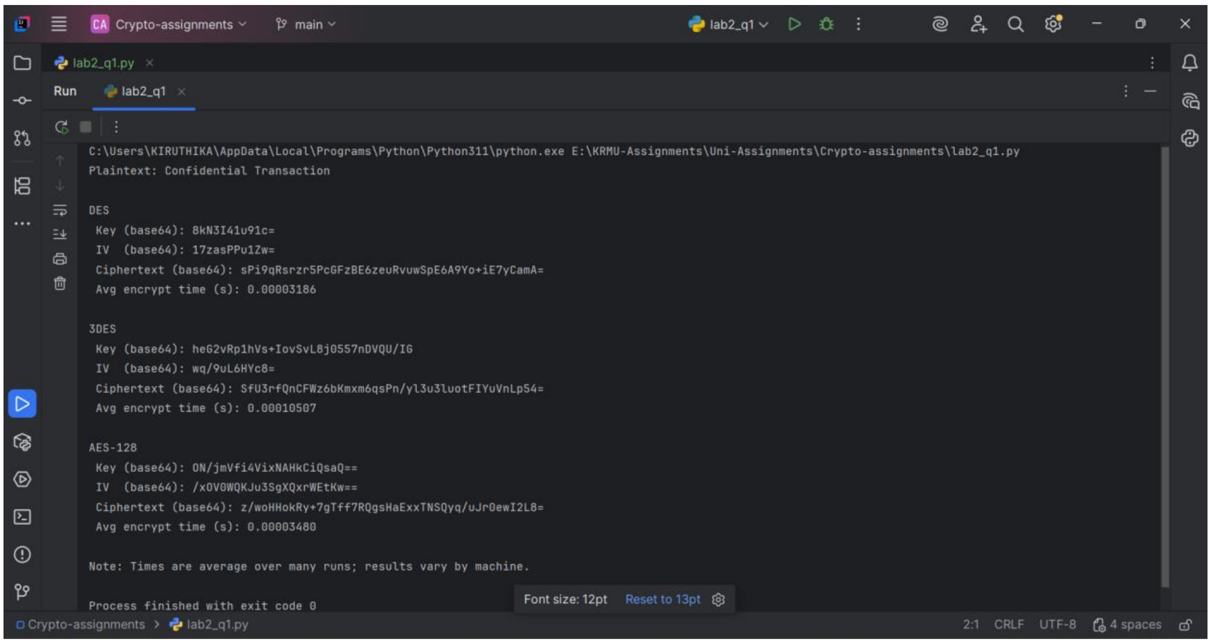
Submitted By :
KIRUTHIKA
2401830004
Bsc.(H) Cybersecurity

Submitted To :
Dr. ANSHU

Q1) CODE :

```
# Purpose: Encrypt a plaintext using DES, 3DES and AES, show ciphertexts
and timing.


import time

import base64

from Crypto.Cipher import DES, DES3, AES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get_random_bytes


PLAINTEXT = b"Confidential Transaction"


def b64(x: bytes) -> str:

    return base64.b64encode(x).decode()


def time_encrypt(cipher_obj, data, block_size, runs=2000):

    # Run encryption many times and return average time.

    start = time.perf_counter()

    for _ in range(runs):

        _ = cipher_obj.encrypt(pad(data, block_size))

    end = time.perf_counter()

    return (end - start) / runs


def run_des(plaintext):

    key = get_random_bytes(8)  # DES key size: 8 bytes

    iv = get_random_bytes(8)   # block size 8

    cipher = DES.new(key, DES.MODE_CBC, iv)

    ct = cipher.encrypt(pad(plaintext, DES.block_size))

    # measure (recreate cipher each run to simulate real usage)

    avg_time = time.perf_counter()

    runs = 2000

    t0 = time.perf_counter()
    for _ in range(runs):
        c = DES.new(key, DES.MODE_CBC, iv).encrypt(pad(plaintext,
DES.block_size))
    t1 = time.perf_counter()
    avg_time = (t1 - t0) / runs
    return {
        'name': 'DES',
        'key': b64(key),
        'iv': b64(iv),
        'ciphertext': b64(ct),
        'time': avg_time
    }


def run_3des(plaintext):
    # keep trying until key is accepted
    while True:
        key = get_random_bytes(24)  # 3DES 24 bytes (3 x 8)
        try:
            iv = get_random_bytes(8)
            cipher = DES3.new(key, DES3.MODE_CBC, iv)
            ct = cipher.encrypt(pad(plaintext, DES3.block_size))
            break
        except ValueError:
            continue
    runs = 2000
    t0 = time.perf_counter()
    for _ in range(runs):
        _ = DES3.new(key, DES3.MODE_CBC, iv).encrypt(pad(plaintext,
DES3.block_size))
    t1 = time.perf_counter()
    return {
        'name': '3DES',
        'key': b64(key),
        'iv': b64(iv),
        'ciphertext': b64(ct),
        'time': (t1 - t0) / runs
    }

def run_aes(plaintext):
    key = get_random_bytes(16)  # AES-128
    iv = get_random_bytes(16)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ct = cipher.encrypt(pad(plaintext, AES.block_size))
    runs = 2000
    t0 = time.perf_counter()
    for _ in range(runs):
        _ = AES.new(key, AES.MODE_CBC, iv).encrypt(pad(plaintext,
AES.block_size))
    t1 = time.perf_counter()
    return {
        'name': 'AES-128',
        'key': b64(key),
        'iv': b64(iv),
        'ciphertext': b64(ct),
        'time': (t1 - t0) / runs
    }

def main():
    print("Plaintext:", PLAINTEXT.decode())
    for fn in (run_des, run_3des, run_aes):
        r = fn(PLAINTEXT)
        print("\n" + r['name'])
        print(" Key (base64):", r['key'])
        print(" IV  (base64):", r['iv'])
        print(" Ciphertext (base64):", r['ciphertext'])
        print(" Avg encrypt time (s):", "{:.8f}".format(r['time']))
    print("\nNote: Times are average over many runs; results vary by machine.")


if __name__ == "__main__":
    main()
```

OUTPUT :



```
C:\Users\KIRUTHIKA\AppData\Local\Programs\Python\Python311\python.exe E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\lab2_q1.py
Plaintext: Confidential Transaction

DES
 Key (base64): 8kN3I41u91c=
 IV  (base64): 17zasPPu1Zw=
 Ciphertext (base64): sPi9qRsrzr5PcGFzBE6zeuRvuwSpE6A9Yo+iE7yCamA=
 Avg encrypt time (s): 0.00003186

3DES
 Key (base64): heG2vRp1hVs+IovSvL8j0557nDVQU/IG
 IV  (base64): wq/9uL6HYc8=
 Ciphertext (base64): SfU3rfQnCFWz6bKmxm6qsPn/yl3u3luotFIYuVnLp54=
 Avg encrypt time (s): 0.00010507

AES-128
 Key (base64): ON/jmVfi4VixNAHkCiQsaQ==
 IV  (base64): /xOVOWQKJu3SgXQxrWEtKw==
 Ciphertext (base64): z/woHHokRy+7gTff7RQgsHaExxTNSQyq/uJr0ewI2L8=
 Avg encrypt time (s): 0.00003480

Note: Times are average over many runs; results vary by machine.

Process finished with exit code 0
```

POINTS :

- **DES**: 56-bit effective key, insecure (brute-force). 64-bit block size which is small.

- **3DES**: More secure than DES (2-key or 3-key) but slow (three DES ops), block size still 64-bit—deprecated for most modern use.

- **AES**: Modern standard, larger block size (128-bit), supports 128/192/256-bit keys, fast (hardware acceleration on many CPUs).

- **Performance**: AES typically fastest, 3DES slowest, DES in middle. For short messages differences are small; average many runs to get meaningful numbers.

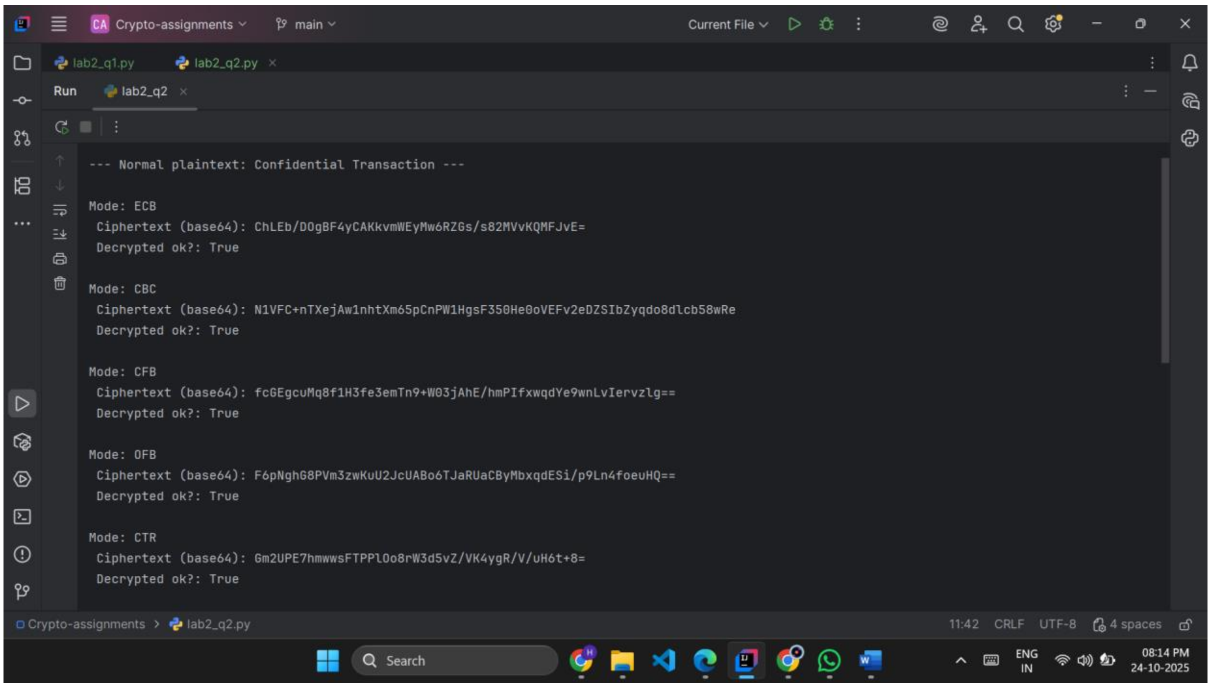
- **Security**: Always prefer AES

Q2) CODE :

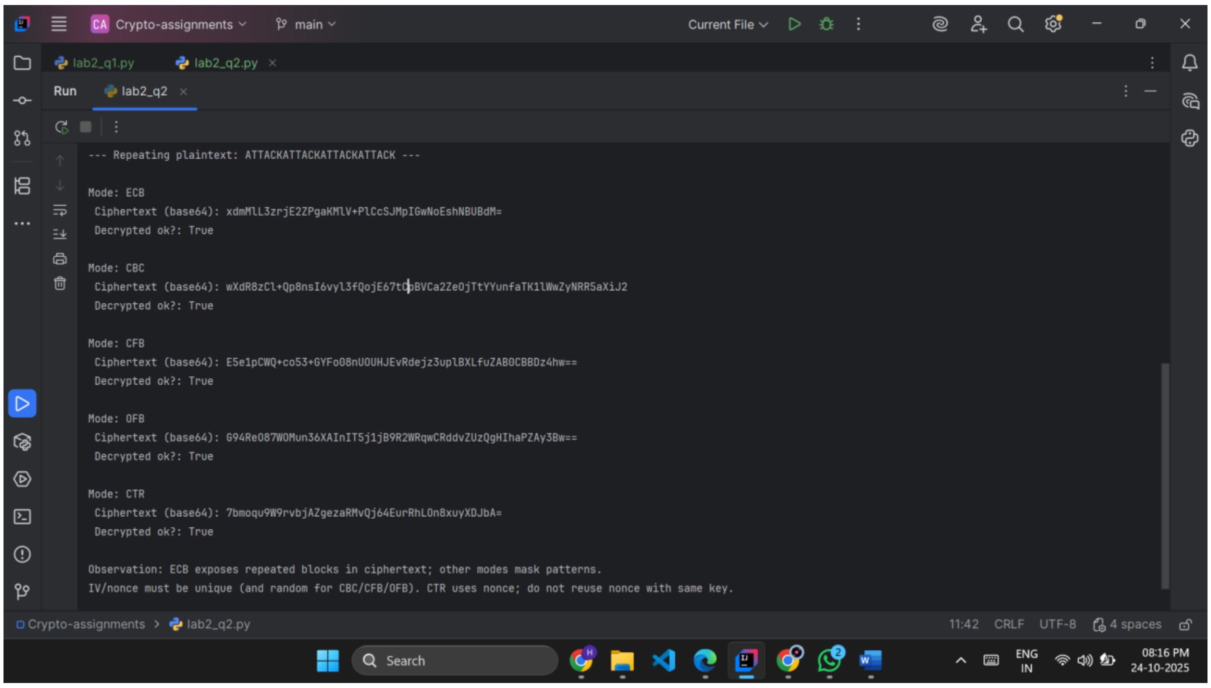
```
# Show AES under different modes and how ciphertexts look.
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes
import base64


def b64(x: bytes) -> str:
    return base64.b64encode(x).decode()


def test_mode(mode_name, key, plaintext):
    if mode_name == 'ECB':
        cipher = AES.new(key, AES.MODE_ECB)
        ct = cipher.encrypt(pad(plaintext, AES.block_size))
        # decrypt
        dec = AES.new(key, AES.MODE_ECB).decrypt(ct)
        pt = unpad(dec, AES.block_size)
        return ct, pt
    elif mode_name == 'CBC':
        iv = get_random_bytes(16)
        cipher = AES.new(key, AES.MODE_CBC, iv)
        ct = cipher.encrypt(pad(plaintext, AES.block_size))
        dec = AES.new(key, AES.MODE_CBC, iv).decrypt(ct)
        pt = unpad(dec, AES.block_size)
        return iv + ct, pt  # prefix iv so we can show it
    elif mode_name == 'CFB':
        iv = get_random_bytes(16)
        cipher = AES.new(key, AES.MODE_CFB, iv)
        ct = cipher.encrypt(plaintext)  # CFB works on bytes directly
        dec = AES.new(key, AES.MODE_CFB, iv).decrypt(ct)
        return iv + ct, dec
    elif mode_name == 'OFB':
        iv = get_random_bytes(16)
        cipher = AES.new(key, AES.MODE_OFB, iv)
        ct = cipher.encrypt(plaintext)
        dec = AES.new(key, AES.MODE_OFB, iv).decrypt(ct)
        return iv + ct, dec
    elif mode_name == 'CTR':
        # use nonce parameter (recommended)
        nonce = get_random_bytes(8)
        cipher = AES.new(key, AES.MODE_CTR, nonce=nonce)
        ct = cipher.encrypt(plaintext)
        dec = AES.new(key, AES.MODE_CTR, nonce=nonce).decrypt(ct)
        return nonce + ct, dec


def main():
    key = get_random_bytes(16)
    # normal plaintext and repeated plaintext for pattern observation
    plain1 = b"Confidential Transaction"
    plain2 = (b"ATTACKATTACKATTACKATTACK")  # repeating pattern to
visualize ECB weakness


    modes = ['ECB','CBC','CFB','OFB','CTR']
    for plain, name in ((plain1, 'Normal'), (plain2, 'Repeating')):
        print(f"\n--- {name} plaintext: {plain.decode(errors='ignore')} --
-")
        for m in modes:
            ct, dec = test_mode(m, key, plain)
            print(f"\nMode: {m}")
            print(" Ciphertext (base64):", b64(ct))
            print(" Decrypted ok?:", dec == plain)


    print("\nObservation: ECB exposes repeated blocks in ciphertext; other
modes mask patterns.")
```

```
    print("IV/nonce must be unique (and random for CBC/CFB/OFB). CTR uses nonce; do not reuse nonce with same key.")


if __name__ == "__main__":
    main()
```

OUTPUT :



```
--- Normal plaintext: Confidential Transaction ---

Mode: ECB
 Ciphertext (base64): ChLEb/DOgBF4yCAKkvmWEyMw6RZGs/s82MVvKQMFJvE=
 Decrypted ok?: True

Mode: CBC
 Ciphertext (base64): N1VFC+nTXejAw1nhtXm65pCnPW1HgsF350He0oVEFv2eDZSIbZyqdo8dlcb58wRe
 Decrypted ok?: True

Mode: CFB
 Ciphertext (base64): fcGEgcuMq8f1H3fe3emTn9+W03jAhE/hmPIfxwqdYe9wnLvIervzlg==
 Decrypted ok?: True

Mode: OFB
 Ciphertext (base64): F6pNghG8PVm3zwKuU2JcUABo6TJaRUaCByMbxqdESi/p9Ln4foeuHQ==
 Decrypted ok?: True

Mode: CTR
 Ciphertext (base64): Gm2UPE7hmwwsFTPPlOo8rW3d5vZ/VK4ygR/V/uH6t+8=
 Decrypted ok?: True
```



```
--- Repeating plaintext: ATTACKATTACKATTACKATTACK ---

Mode: ECB
 Ciphertext (base64): xdmMlL3zrjE2ZPgaKMlV+PlCcSJMpIGwNoEshNBUBdM=
 Decrypted ok?: True

Mode: CBC
 Ciphertext (base64): wXdR8zCl+Qp8nsI6vyl3fQojE67tCpBVCa2ZeOjTtYYunfaTK1lWwZyNRR5aXiJ2
 Decrypted ok?: True

Mode: CFB
 Ciphertext (base64): E5e1pCWQ+co53+GYFo08nUOUHJEvRdejz3uplBXLfuZAB0CBBDz4hw==
 Decrypted ok?: True

Mode: OFB
 Ciphertext (base64): G94ReO87WOMun36XAInIT5j1jB9R2WRqwCRddvZUzQgHIhaPZAy3Bw==
 Decrypted ok?: True

Mode: CTR
 Ciphertext (base64): 7bmoqu9W9rvbjAZgezaRMvQj64EurRhLOn8xuyXDJbA=
 Decrypted ok?: True

Observation: ECB exposes repeated blocks in ciphertext; other modes mask patterns.
IV/nonce must be unique (and random for CBC/CFB/OFB). CTR uses nonce; do not reuse nonce with same key.
```

POINTS :

- **ECB**: identical plaintext blocks → identical ciphertext blocks (pattern leakage). Show example with repeating plaintext.

- **CBC**: uses IV + chaining; hides repeating patterns.

- **CFB/OFB**: stream-like; no block alignment leakage like ECB; still need IV.

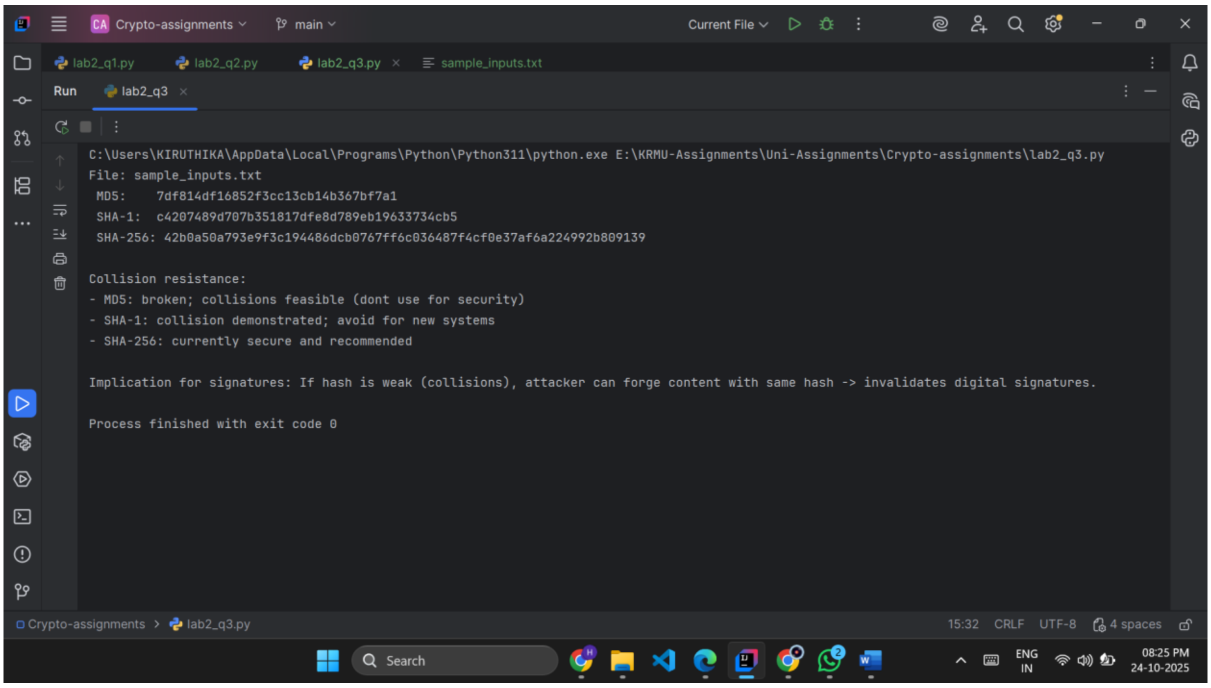
- **CTR**: turns block cipher into stream using nonce; also requires unique nonce. Fast and parallelizable.

- **IV importance**: must be random/unique (CBC/CFB/OFB) and not reused with same key; CTR requires unique nonce. Wrong IV/nonce reuse leads to catastrophic leakage.

Q3) CODE :

```
# Hash a file using MD5, SHA1, SHA256 and print results.

import hashlib

def hash_file(path):
    # read file in chunks to support large files
    h_md5 = hashlib.md5()
    h_sha1 = hashlib.sha1()
    h_sha256 = hashlib.sha256()
    with open(path, 'rb') as f:
        while True:
            chunk = f.read(8192)
            if not chunk:
                break
            h_md5.update(chunk)
            h_sha1.update(chunk)
            h_sha256.update(chunk)
    return h_md5.hexdigest(), h_sha1.hexdigest(), h_sha256.hexdigest()

def main():
    path = "sample_inputs.txt"
    md5, sha1, sha256 = hash_file(path)
    print("File:", path)
    print(" MD5:    ", md5)
    print(" SHA-1: ", sha1)
    print(" SHA-256:", sha256)
    print("\nCollision resistance:")
    print("- MD5: broken; collisions feasible (dont use for security)")
    print("- SHA-1: collision demonstrated; avoid for new systems")
    print("- SHA-256: currently secure and recommended")
    print("\nImplication for signatures: If hash is weak (collisions), 
attacker can forge content with same hash -> invalidates digital 
signatures.")

if __name__ == "__main__":
    main()
```

OUTPUT :



```
C:\Users\KIRUTHIKA\AppData\Local\Programs\Python\Python311\python.exe E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\lab2_q3.py
File: sample_inputs.txt
 MD5:    7df814df16852f3cc13cb14b367bf7a1
 SHA-1:  c4207489d707b351817dfe8d789eb19633734cb5
 SHA-256: 42b0a50a793e9f3c194486dcb0767ff6c036487f4cf0e37af6a224992b809139

Collision resistance:
- MD5: broken; collisions feasible (dont use for security)
- SHA-1: collision demonstrated; avoid for new systems
- SHA-256: currently secure and recommended

Implication for signatures: If hash is weak (collisions), attacker can forge content with same hash -> invalidates digital signatures.

Process finished with exit code 0
```

POINTS :

- Provide the three hex digests (they will differ per sample_inputs.txt).

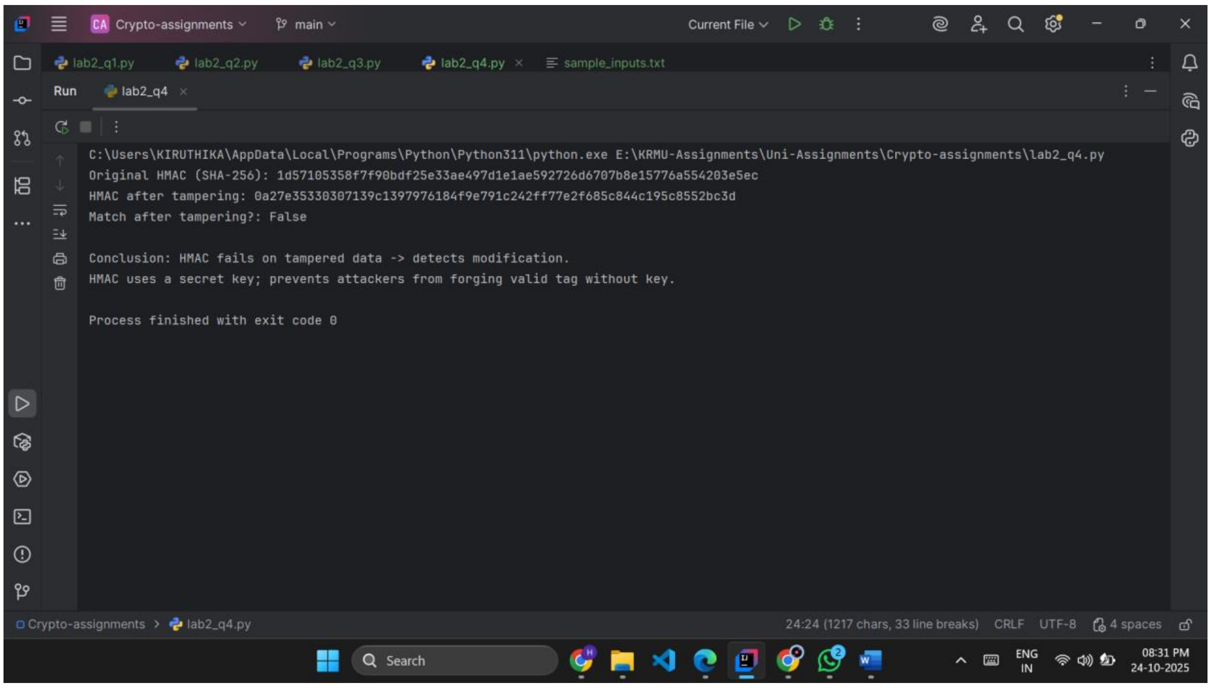
- Explain collision resistance: MD5 broken (collisions), SHA-1 deprecated, SHA-256 secure enough for digital signatures.

- For digital signatures, use SHA-256 or stronger; weak hashes allow signature forgery.

Q4) CODE :

```
# Compute HMAC-SHA256 for a file, then show that modifying the file
changes HMAC (integrity check).

import hmac
import hashlib

def hmac_of_file(path, key):
    h = hmac.new(key, digestmod=hashlib.sha256)
    with open(path, 'rb') as f:
        while True:
            chunk = f.read(8192)
            if not chunk:
                break
            h.update(chunk)
    return h.hexdigest()

def main():
    path = "sample_inputs.txt"
    key = b'secret_shared_key_123'  # in real world use a strong random
key stored securely
    original_hmac = hmac_of_file(path, key)
    print("Original HMAC (SHA-256):", original_hmac)

    # Simulate file tampering: read, modify, and compute hmac of modified
bytes (without changing disk)
    with open(path, 'rb') as f:
        data = f.read()
    tampered = data + b"\n# tampered"
    h_tampered = hmac.new(key, tampered, hashlib.sha256).hexdigest()
    print("HMAC after tampering:", h_tampered)
    print("Match after tampering?:", h_tampered == original_hmac)
    print("\nConclusion: HMAC fails on tampered data -> detects
modification.")
    print("HMAC uses a secret key; prevents attackers from forging valid
tag without key.")

if __name__ == "__main__":
    main()
```

OUTPUT :



```
C:\Users\KIRUTHIKA\AppData\Local\Programs\Python\Python311\python.exe E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\lab2_q4.py
Original HMAC (SHA-256): 1d57105358f7f90bdf25e33ae497d1e1ae592726d6707b8e15776a554203e5ec
HMAC after tampering: 0a27e35330307139c1397976184f9e791c242ff77e2f685c844c195c8552bc3d
Match after tampering?: False

Conclusion: HMAC fails on tampered data -> detects modification.
HMAC uses a secret key; prevents attackers from forging valid tag without key.

Process finished with exit code 0
```

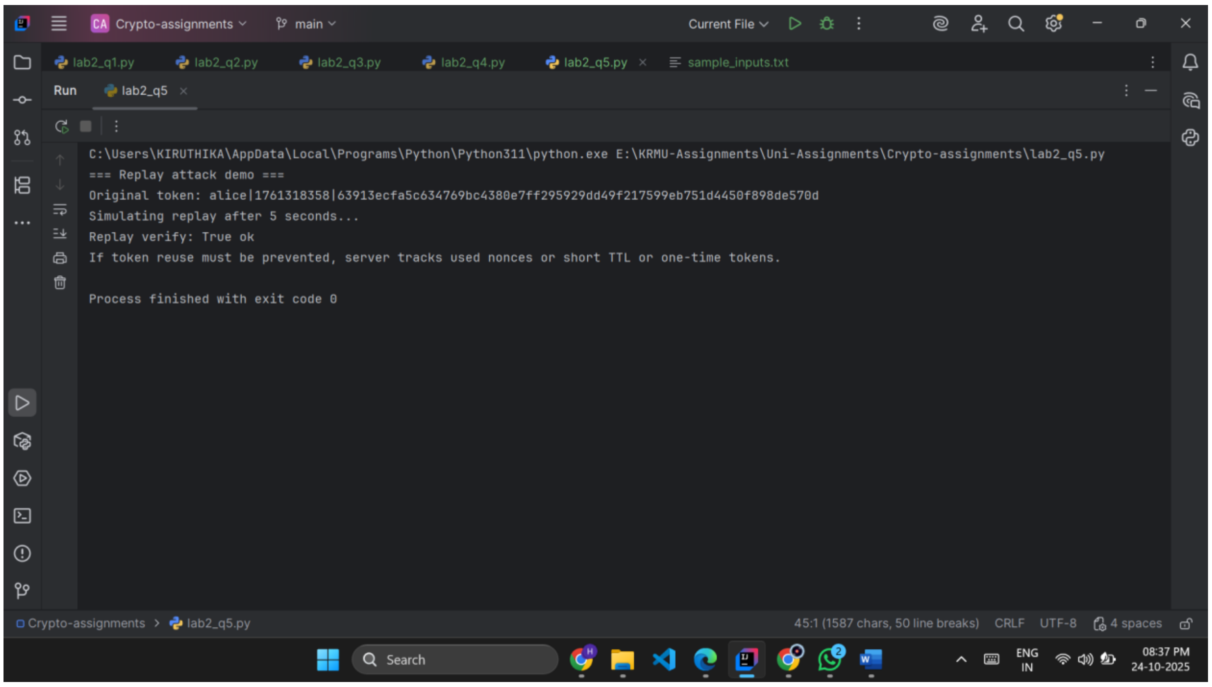
POINTS :

- HMAC ensures both integrity and authenticity: only parties with key can produce valid HMAC.
- If attacker changes data, HMAC will not match — recipient detects tampering.
- Use SHA-256 or better for HMAC; never use non-keyed hashes for integrity.

Q5) CODE :

```
# A replay-attack demo.

import time
import hmac
import hashlib

# --------- Replay attack simulation ---------
# Simple token format: token = HMAC(key, username || timestamp)
def create_token(username, key, timestamp):
    msg = f"{username}|{timestamp}".encode()
    tag = hmac.new(key, msg, hashlib.sha256).hexdigest()
    return f"{username}|{timestamp}|{tag}"

def verify_token(token, key, max_age=30):
    # parse token, check timestamp age and HMAC
    try:
        user, ts_str, tag = token.split("|")
        ts = int(ts_str)
    except Exception:
        return False, "invalid format"
    # check age
    now = int(time.time())
    if abs(now - ts) > max_age:
        return False, "token expired"
    # recompute tag
    msg = f"{user}|{ts}".encode()
    expected = hmac.new(key, msg, hashlib.sha256).hexdigest()
    if not hmac.compare_digest(expected, tag):
        return False, "bad tag"
    return True, "ok"

def replay_demo():
    key = b'shared_key_demo'
    user = "alice"
    ts = int(time.time())
    token = create_token(user, key, ts)
    print("Original token:", token)
    # Attacker can replay token later:
    print("Simulating replay after 5 seconds...")
    time.sleep(5)
    ok, reason = verify_token(token, key, max_age=60)
    print("Replay verify:", ok, reason)
    # Prevention: use nonce or one-time token store, or very short expiry.
    print("If token reuse must be prevented, server tracks used nonces or
short TTL or one-time tokens.")

def main():
    print("=== Replay attack demo ===")
    replay_demo()

if __name__ == "__main__":
    main()
```

OUTPUT :



```
C:\Users\KIRUTHIKA\AppData\Local\Programs\Python\Python311\python.exe E:\KRMU-Assignments\Uni-Assignments\Crypto-assignments\lab2_q5.py
=== Replay attack demo ===
Original token: alice|1761318358|63913ecfa5c634769bc4380e7ff295929dd49f217599eb751d4450f898de570d
Simulating replay after 5 seconds...
Replay verify: True ok
If token reuse must be prevented, server tracks used nonces or short TTL or one-time tokens.

Process finished with exit code 0
```

Crypto-assignments > lab2_q5.py
45:1 (1587 chars, 50 line breaks) CRLF UTF-8 4 spaces
Search
ENG IN
08:37 PM
24-10-2025

POINTS :

Replay: reusing a valid token allows attacker to authenticate. Mitigation: timestamps + short expiry, nonces + server-side nonce storage, one-time tokens, or challenge-response.