**COMSATS UNIVERSITY ISLAMABAD**

**(ATTOCK CAMPUS)**

**SUBMITTED BY**

**MISBAH UR REHMAN**

**REGISTRATION NO**

**SP24-BSE-036**

**Abstract**

This assignment studies the Salsa20-style stream cipher concept and demonstrates a designed stream-cipher implementation and an adversarial attack. The implemented cipher uses a hybrid keystream construction combining SHA256-CTR and HMAC-SHA256. The adversarial section demonstrates the dangers of keystream reuse (nonce reuse), including known-plaintext recovery and crib-dragging. Code, explanations, and sample outputs are included.

**Table of Contents**

1. Introduction
2. Objectives
3. Background: Salsa20 (brief)
4. Design & Rationale
5. Implementation (Python code)
6. Adversarial Attack Design & Implementation
7. Results (outputs & screenshots)
8. Discussion & Analysis
9. Mitigation & Recommendations
10. Conclusion

**1. Introduction**

Stream ciphers produce a pseudorandom keystream that is XORed with plaintext to create ciphertext. Salsa20 is a widely-known, fast ARX-based stream cipher; its core idea is generating a keystream from key + nonce + counter and XORing it with plaintext. This assignment implements a Salsa20-style stream cipher using existing cryptographic building blocks and demonstrates common attacks resulting from misuse.

**2. Objectives**

* Implement a designed stream cipher approach in Python for encryption and decryption.
* Use ideas from more than one existing approach (here: SHA256-CTR and HMAC-SHA256).
* Design and implement adversarial attacks that show practical weaknesses (nonce reuse).
* Prepare a clear report with code explanation and output screenshots.

**3. Background: Salsa20 (brief)**

Salsa20 (Daniel Bernstein, 2005) is a stream cipher that maps a 128- or 256-bit key, a 64-bit nonce, and a 64-bit counter to a 512-bit keystream block using ARX (Add-Rotate-XOR) operations. Important security principle: never reuse a nonce with the same key; reuse leads to keystream reuse and trivial plaintext recovery attacks.

**4. Design & Rationale**

**Design choice (one line):**  
Keystream block = SHA256(key || nonce || counter) XOR HMAC\_SHA256(key, nonce || counter).

**Rationale:**

* Uses two independent, well-known PRF constructions (SHA256 in CTR mode and HMAC-SHA256) and XORs their outputs to create a blended keystream block.
* Reflects the assignment requirement to use approaches from more than one developed cipher/technique.
* Keeps implementation simple and portable for demonstration and lab exam viva.

**5. Implementation (Python code)**

# stream\_cipher\_submission.py

# Keystream = SHA256-CTR(key||nonce||counter) XOR HMAC-SHA256(key, nonce||counter)

# Encryption = plaintext XOR keystream

import os

import struct

import hashlib

import hmac

def xor\_bytes(a: bytes, b: bytes) -> bytes:

n = min(len(a), len(b))

return bytes(x ^ y for x, y in zip(a[:n], b[:n]))

def sha256\_ctr\_block(key: bytes, nonce: bytes, counter: int) -> bytes:

ctr = struct.pack(">Q", counter)

return hashlib.sha256(key + nonce + ctr).digest()

def hmac\_sha256\_block(key: bytes, nonce: bytes, counter: int) -> bytes:

ctr = struct.pack(">Q", counter)

return hmac.new(key, nonce + ctr, hashlib.sha256).digest()

def keystream\_bytes(key: bytes, nonce: bytes, length: int) -> bytes:

out = bytearray()

ctr = 0

while len(out) < length:

a = sha256\_ctr\_block(key, nonce, ctr)

b = hmac\_sha256\_block(key, nonce, ctr)

out.extend(xor\_bytes(a, b))

ctr += 1

return bytes(out[:length])

def encrypt(key: bytes, nonce: bytes, plaintext: bytes) -> bytes:

ks = keystream\_bytes(key, nonce, len(plaintext))

return xor\_bytes(plaintext, ks)

def decrypt(key: bytes, nonce: bytes, ciphertext: bytes) -> bytes:

return encrypt(key, nonce, ciphertext) # XOR symmetric

# Demo: correctness

def demo\_encrypt\_decrypt():

key = b"lab\_key\_32\_bytes\_long\_example\_\_\_\_\_1234"[:32] # 32 bytes

nonce = os.urandom(12)

msg = b"Hello - Salsa20 style stream cipher lab test."

c = encrypt(key, nonce, msg)

p = decrypt(key, nonce, c)

print("=== Demo: Encrypt/Decrypt ===")

print("Original:", msg)

print("Nonce (hex):", nonce.hex())

print("Cipher (hex):", c.hex())

print("Decrypted:", p)

assert p == msg

# Attack helpers

def xor\_ct(c1: bytes, c2: bytes) -> bytes:

return xor\_bytes(c1, c2)

def recover\_known\_plain(c\_known: bytes, p\_known: bytes, c\_target: bytes) -> bytes:

ks = xor\_bytes(c\_known, p\_known)

return xor\_bytes(c\_target, ks)

def crib\_drag(pxor: bytes, crib: bytes):

hits = []

for pos in range(len(pxor) - len(crib) + 1):

cand = xor\_bytes(pxor[pos:pos+len(crib)], crib)

try:

s = cand.decode('utf-8')

if any(ch.isalpha() for ch in s):

hits.append((pos, s))

except:

pass

return hits

# Demo: nonce reuse attack

def demo\_nonce\_reuse\_attack():

key = b"fixed\_key\_for\_reuse\_demo\_32\_bytes\_\_"[:32]

nonce = b"fixednonce123" # intentionally reused

p1 = b"The secret meeting is at midnight. Bring blue folder."

p2 = b"Hello friend, please review the attached notes about project X."

c1 = encrypt(key, nonce, p1)

c2 = encrypt(key, nonce, p2)

pxor = xor\_ct(c1, c2)

recovered\_p2 = recover\_known\_plain(c1, p1, c2)

print("\n=== Demo: Nonce Reuse Attack ===")

print("P1:", p1)

print("P2:", p2)

print("C1 (hex):", c1.hex())

print("C2 (hex):", c2.hex())

print("C1 xor C2 (hex):", pxor.hex())

print("\n-- Known-plaintext recovery --")

print("Recovered P2:", recovered\_p2)

print("\n-- Crib-dragging examples --")

for crib in [b"the", b"meeting", b"attached", b"project", b"midnight"]:

hits = crib\_drag(pxor, crib)

if hits:

print(f"Crib '{crib.decode()}' hits:", hits[:5])

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

demo\_encrypt\_decrypt()

demo\_nonce\_reuse\_attack()

**6. Adversarial Attack Design & Implementation**

Attack principle: If the same keystream (same key + nonce) is used to encrypt two messages, then C1 XOR C2 = P1 XOR P2. From P1 XOR P2 the attacker can:

* Recover P2 if they know P1 (Known-plaintext recovery).
* Use crib-dragging (guess common words) on P1 XOR P2 to find readable fragments and reconstruct messages.

Code provided demonstrates both attacks.

**7. Results**

=== Demo: Encrypt/Decrypt ===

Original: b'Hello - Salsa20 style stream cipher lab test.'

Nonce (hex): dfe3f0e09126fd779c37e22f

Cipher (hex): 7229d2b68425f4b1c83bdaac0b5be6661ced35c34f057eb369524f17d728b1c2049cd8842461d2a9035a2ecd01

Decrypted: b'Hello - Salsa20 style stream cipher lab test.'

=== Demo: Nonce Reuse Attack ===

P1: b'The secret meeting is at midnight. Bring blue folder.'

P2: b'Hello friend, please review the attached notes about project X.'

C1 (hex): 3c7c69a0e5e35eaade593d8d101dfeb9498379a1c238433a4aeb9645a35405d2610db637caa6c12e66289e8f78b091664459d1decf

C2 (hex): 207160ecf9a65baad24873845958fabc42852aad916a473803e38801b955079a7457e214dba7ca2d66249d8e78e3d7684a52c1d8c115f164780463b6135e66

C1 xor C2 (hex): 1c0d094c1c4505000c114e09494504050b06530c5352040249081e441a010248155a542311010b03000c03010053460e0e0b10060e

-- Known-plaintext recovery --

Recovered P2: b'Hello friend, please review the attached notes about '

-- Crib-dragging examples --

Crib 'the' hits: [(0, 'hel'), (1, 'ya)'), (2, '}$y'), (3, '8t '), (4, 'h-`')]

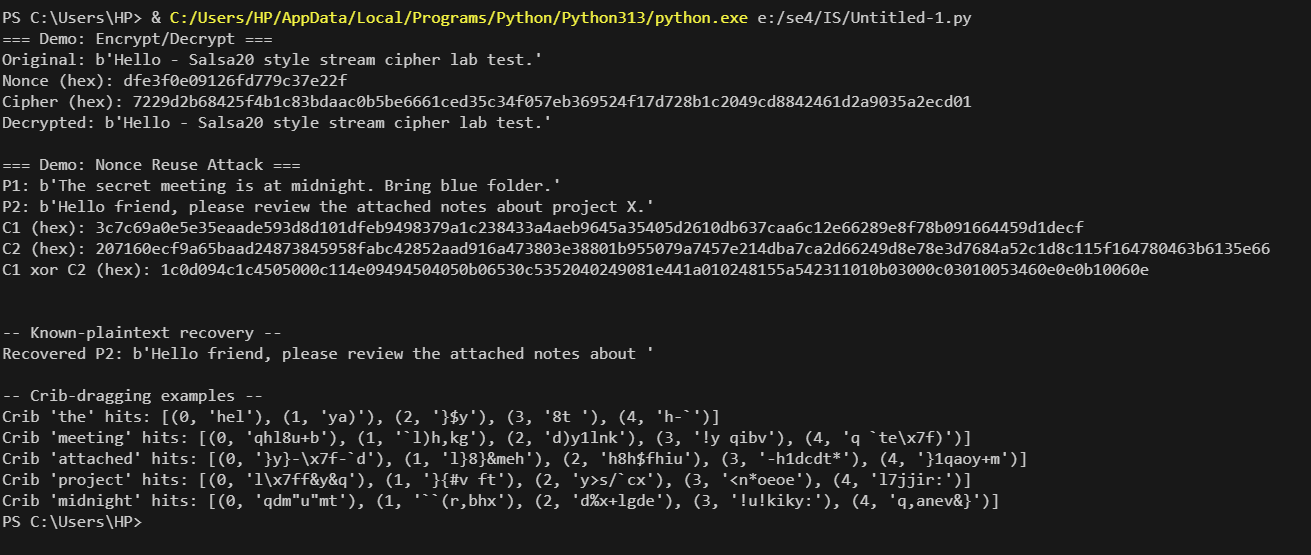
Crib 'meeting' hits: [(0, 'qhl8u+b'), (1, '`l)h,kg'), (2, 'd)y1lnk'), (3, '!y qibv'), (4, 'q `te\x7f)')]

Crib 'attached' hits: [(0, '}y}-\x7f-`d'), (1, 'l}8}&meh'), (2, 'h8h$fhiu'), (3, '-h1dcdt\*'), (4, '}1qaoy+m')]

Crib 'project' hits: [(0, 'l\x7ff&y&q'), (1, '}{#v ft'), (2, 'y>s/`cx'), (3, '<n\*oeoe'), (4, 'l7jjir:')]

Crib 'midnight' hits: [(0, 'qdm"u"mt'), (1, '``(r,bhx'), (2, 'd%x+lgde'), (3, '!u!kiky:'), (4, 'q,anev&}')]

PS C:\Users\HP>



**8. Discussion & Analysis**

The encryption/decryption demo shows correct operation: decrypt(encrypt(m)) = m. The nonce reuse demo shows that reusing the same nonce and key leaks P1 XOR P2. With a known plaintext or intelligent word guesses (cribs), an attacker can recover the other message. This demonstrates why unique nonces per message or AEAD schemes are required.

**9. Mitigation & Recommendations**

* Never reuse a nonce with the same key in a stream cipher.
* Use authenticated encryption (AEAD): ChaCha20-Poly1305 or XSalsa20-Poly1305 (libsodium) to get confidentiality and integrity.
* Use standard, well-tested libraries (libsodium, PyCryptodome) instead of custom constructions for production.