**COMSAT UNIVERSITY ISLAMABAD ATTOCK CAMPUS**

****

**INFORMATION SECURITY**

**NAME:** M USMAN SIDDIQUE

**REGISTRATION NO:** SP24-BSE-024

**LAB ASSIGNEMENT:** 02

**SUBMITTED TO:** Ms. AMBAREEN GUL

**DEPARTMENT:** SOFTWARE ENGINEERING

**DATE:** 12th October 2025

**Stream Cipher**

This report documents the implementation, demonstrations, and adversarial attacks for a Salsa20-style stream cipher. Files generated are saved in /mnt/data.

**1. Implementation Summary**

We implemented a Salsa20 block function (20 rounds), a keystream generator, XOR-based encrypt/decrypt, key derivation via PBKDF2-HMAC-SHA256, and two attack demos: nonce reuse (known-plaintext recovery) and brute-force passphrase attack.

**2. Full Code (stream\_cipher\_demo\_full.py)**

"""  
stream\_cipher\_demo\_full.py  
Complete Salsa20-style stream cipher implementation and adversarial attack demos.  
Save and run with: python stream\_cipher\_demo\_full.py  
Generates outputs and a report in the local directory.  
"""  
  
import struct, hashlib  
from pathlib import Path  
  
def \_rotl32(v, c):  
 return ((v << c) & 0xffffffff) | (v >> (32 - c))  
  
def \_quarterround(y0, y1, y2, y3):  
 z1 = y1 ^ \_rotl32((y0 + y3) & 0xffffffff, 7)  
 z2 = y2 ^ \_rotl32((z1 + y0) & 0xffffffff, 9)  
 z3 = y3 ^ \_rotl32((z2 + z1) & 0xffffffff, 13)  
 z0 = y0 ^ \_rotl32((z3 + z2) & 0xffffffff, 18)  
 return z0 & 0xffffffff, z1 & 0xffffffff, z2 & 0xffffffff, z3 & 0xffffffff  
  
def salsa20\_block(key32, nonce8, counter=0):  
 if len(key32) != 32:  
 raise ValueError("key32 must be 32 bytes")  
 if len(nonce8) != 8:  
 raise ValueError("nonce8 must be 8 bytes")  
 const = b"expand 32-byte k"  
 def le32(b): return struct.unpack("<I", b)[0]  
 k\_words = [le32(key32[i:i+4]) for i in range(0,32,4)]  
 n\_words = [le32(nonce8[i:i+4]) for i in range(0,8,4)]  
 c\_words = [le32(const[i:i+4]) for i in range(0,16,4)]  
 ctr\_words = [counter & 0xffffffff, (counter >> 32) & 0xffffffff]  
 state = [  
 c\_words[0], k\_words[0], k\_words[1], k\_words[2],  
 k\_words[3], c\_words[1], n\_words[0], n\_words[1],  
 ctr\_words[0], ctr\_words[1], c\_words[2], k\_words[4%8],  
 k\_words[5%8], k\_words[6%8], k\_words[7%8], c\_words[3]  
 ]  
 working = state.copy()  
 for \_ in range(10): # 20 rounds (10 double-rounds)  
 # column rounds  
 working[0],working[4],working[8],working[12] = \_quarterround(working[0],working[4],working[8],working[12])  
 working[5],working[9],working[13],working[1] = \_quarterround(working[5],working[9],working[13],working[1])  
 working[10],working[14],working[2],working[6] = \_quarterround(working[10],working[14],working[2],working[6])  
 working[15],working[3],working[7],working[11] = \_quarterround(working[15],working[3],working[7],working[11])  
 # row rounds  
 working[0],working[1],working[2],working[3] = \_quarterround(working[0],working[1],working[2],working[3])  
 working[5],working[6],working[7],working[4] = \_quarterround(working[5],working[6],working[7],working[4])  
 working[10],working[11],working[8],working[9] = \_quarterround(working[10],working[11],working[8],working[9])  
 working[15],working[12],working[13],working[14] = \_quarterround(working[15],working[12],working[13],working[14])  
 out = []  
 for i in range(16):  
 out\_word = (working[i] + state[i]) & 0xffffffff  
 out.append(struct.pack("<I", out\_word))  
 return b"".join(out) # 64 bytes  
  
def salsa20\_keystream(key32, nonce8, length, initial\_counter=0):  
 ks = b""  
 counter = initial\_counter  
 while len(ks) < length:  
 block = salsa20\_block(key32, nonce8, counter)  
 ks += block  
 counter = (counter + 1) & ((1<<64)-1)  
 return ks[:length]  
  
def derive\_key\_from\_passphrase(passphrase, salt=b"salty", iterations=100000):  
 return hashlib.pbkdf2\_hmac("sha256", passphrase if isinstance(passphrase, bytes) else passphrase.encode(), salt, iterations, dklen=32)  
  
def encrypt\_salsa20(plaintext\_bytes, key32, nonce8, counter=0):  
 ks = salsa20\_keystream(key32, nonce8, len(plaintext\_bytes), initial\_counter=counter)  
 return bytes(a ^ b for a,b in zip(plaintext\_bytes, ks))  
  
def decrypt\_salsa20(ciphertext\_bytes, key32, nonce8, counter=0):  
 return encrypt\_salsa20(ciphertext\_bytes, key32, nonce8, counter)  
  
def nonce\_reuse\_recover(ct1, ct2, known\_p1=None):  
 x = bytes(a ^ b for a,b in zip(ct1, ct2))  
 recovered\_p2 = None  
 if known\_p1 is not None:  
 recovered\_p2 = bytes(a ^ b for a,b in zip(x, known\_p1))  
 return x, recovered\_p2  
  
def brute\_force\_passphrase(ciphertext, nonce8, wordlist, salt=b"salty", iterations=100000):  
 for pw in wordlist:  
 key = derive\_key\_from\_passphrase(pw if isinstance(pw, bytes) else pw.encode(), salt=salt, iterations=iterations)

pt = decrypt\_salsa20(ciphertext, key, nonce8)  
 if b"TOP-SECRET" in pt or all(32 <= c < 127 for c in pt[:30]):  
 return pw, pt  
 return None, None  
  
def demo\_nonce\_reuse():  
 passphrase = b"123456"  
 key = derive\_key\_from\_passphrase(passphrase, salt=b"pepper", iterations=20000)  
 nonce = b'\x00\x00\x00\x00\x00\x00\x00\x01'  
 p1 = b"ORDER=PAY $100 TO ACCOUNT 12345"  
 p2 = b"ORDER=PAY $900 TO ACCOUNT 67890"  
 ct1 = encrypt\_salsa20(p1, key, nonce)  
 ct2 = encrypt\_salsa20(p2, key, nonce)  
 x, recovered\_p2\_prefix = nonce\_reuse\_recover(ct1, ct2, known\_p1=b"ORDER=PAY ")  
 x\_full, recovered\_p2\_full = nonce\_reuse\_recover(ct1, ct2, known\_p1=p1)  
 return {  
 "passphrase": passphrase,  
 "key\_hex": key.hex(),  
 "nonce\_hex": nonce.hex(),  
 "p1": p1, "p2": p2,  
 "ct1\_hex": ct1.hex(), "ct2\_hex": ct2.hex(),  
 "xor\_hex": x.hex(),  
 "recovered\_prefix\_p2": recovered\_p2\_prefix,  
 "recovered\_full\_p2": recovered\_p2\_full  
 }  
  
def demo\_bruteforce():  
 secret\_pw = b"123456"  
 key = derive\_key\_from\_passphrase(secret\_pw, salt=b"pepper", iterations=20000)  
 nonce = b'\x11\x22\x33\x44\x55\x66\x77\x88'  
 plaintext = b"TOP-SECRET: Project Falcon"  
 ct = encrypt\_salsa20(plaintext, key, nonce)  
 small\_list = [b"password", b"123456", b"letmein", b"qwerty", b"admin", b"passw0rd"]  
 found\_pw, found\_pt = brute\_force\_passphrase(ct, nonce, small\_list, salt=b"pepper", iterations=20000)  
 return {  
 "secret\_pw": secret\_pw,  
 "ct\_hex": ct.hex(),  
 "found\_pw": found\_pw,  
 "found\_pt": found\_pt  
 }  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 demo1 = demo\_nonce\_reuse()  
 demo2 = demo\_bruteforce()  
 out\_lines = []  
 out\_lines.append("=== Nonce Reuse Demo ===")  
 out\_lines.append(f"Passphrase (secret): {demo1['passphrase']!r}")  
 out\_lines.append(f"Derived key (hex): {demo1['key\_hex']}")  
 out\_lines.append(f"Nonce (hex): {demo1['nonce\_hex']}")  
 out\_lines.append(f"Plaintext 1: {demo1['p1']}")  
 out\_lines.append(f"Plaintext 2: {demo1['p2']}")  
 out\_lines.append(f"Ciphertext1 (hex): {demo1['ct1\_hex']}")  
 out\_lines.append(f"Ciphertext2 (hex): {demo1['ct2\_hex']}")  
 out\_lines.append(f"CT1 xor CT2 (hex): {demo1['xor\_hex']}")  
 out\_lines.append(f"Recovered p2 using known prefix 'ORDER=PAY ': {demo1['recovered\_prefix\_p2']}")  
 out\_lines.append(f"Recovered p2 using full known p1: {demo1['recovered\_full\_p2']}")  
 out\_lines.append("")  
 out\_lines.append("=== Brute-force passphrase Demo ===")  
 out\_lines.append(f"Ciphertext (hex): {demo2['ct\_hex']}")  
 if demo2['found\_pw'] is not None:  
 out\_lines.append(f"Found passphrase: {demo2['found\_pw']!r}")  
 out\_lines.append(f"Recovered plaintext: {demo2['found\_pt']}")  
 else:  
 out\_lines.append("Passphrase not found in the small list.")  
 out\_text = "\\n".join(out\_lines)  
 print(out\_text)  
 Path("stream\_cipher\_output.txt").write\_text(out\_text)

**3. Demo Outputs**  
**Passphrase** (secret): b'123456'  
Derived key (hex): 88b03994ce6819dd0ece67b23450ccebdc8a6932861583714c75aaa01aeceb48  
Nonce (hex): 0000000000000001  
Plaintext 1: b'ORDER=PAY $100 TO ACCOUNT 12345'  
Plaintext 2: b'ORDER=PAY $900 TO ACCOUNT 67890'  
Ciphertext1 (hex): 3a80aa5faea43001fac31a72f427e0328692fefbc396ae1691407015ea8bcd  
Ciphertext2 (hex): 3a80aa5faea43001fac31a7af427e0328692fefbc396ae1691407710e186c8  
CT1 xor CT2 (hex): 000000000000000000000008000000000000000000000000000007050b0d05  
Recovered p2 using known prefix 'ORDER=PAY ': b'ORDER=PAY \x00\x08\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x07\x05\x0b\r\x05'  
Recovered p2 using full known p1: b'ORDER=PAY $900 TO ACCOUNT 67890'  
  
**Brute-force passphrase** Demo   
Ciphertext (hex): 6ea951747f3e0185f3298d2fdb02bc1ccc20f31044db5d7a2463  
Attempting brute-force over small list...  
Found passphrase: b'123456'  
Recovered plaintext: b'TOP-SECRET: Project Falcon'

**4. Output Screenshot**

A close-up of a computer code

AI-generated content may be incorrect.

**5. Notes on Vulnerabilities and Mitigations**

Key points:  
- Never reuse a nonce with the same key for stream ciphers: it leaks p1 xor p2.  
- Use long, high-entropy passphrases or random keys; protect keys well.  
- Use authenticated encryption (AEAD) to avoid tampering and misuse.