**Assigment#01**



**Name** : faiqa bibi

**Reg No**: sp24-bse-019

**Submitted To**:Maam Ambreen

**Course**:Information security

**Date**:12-oct-2025

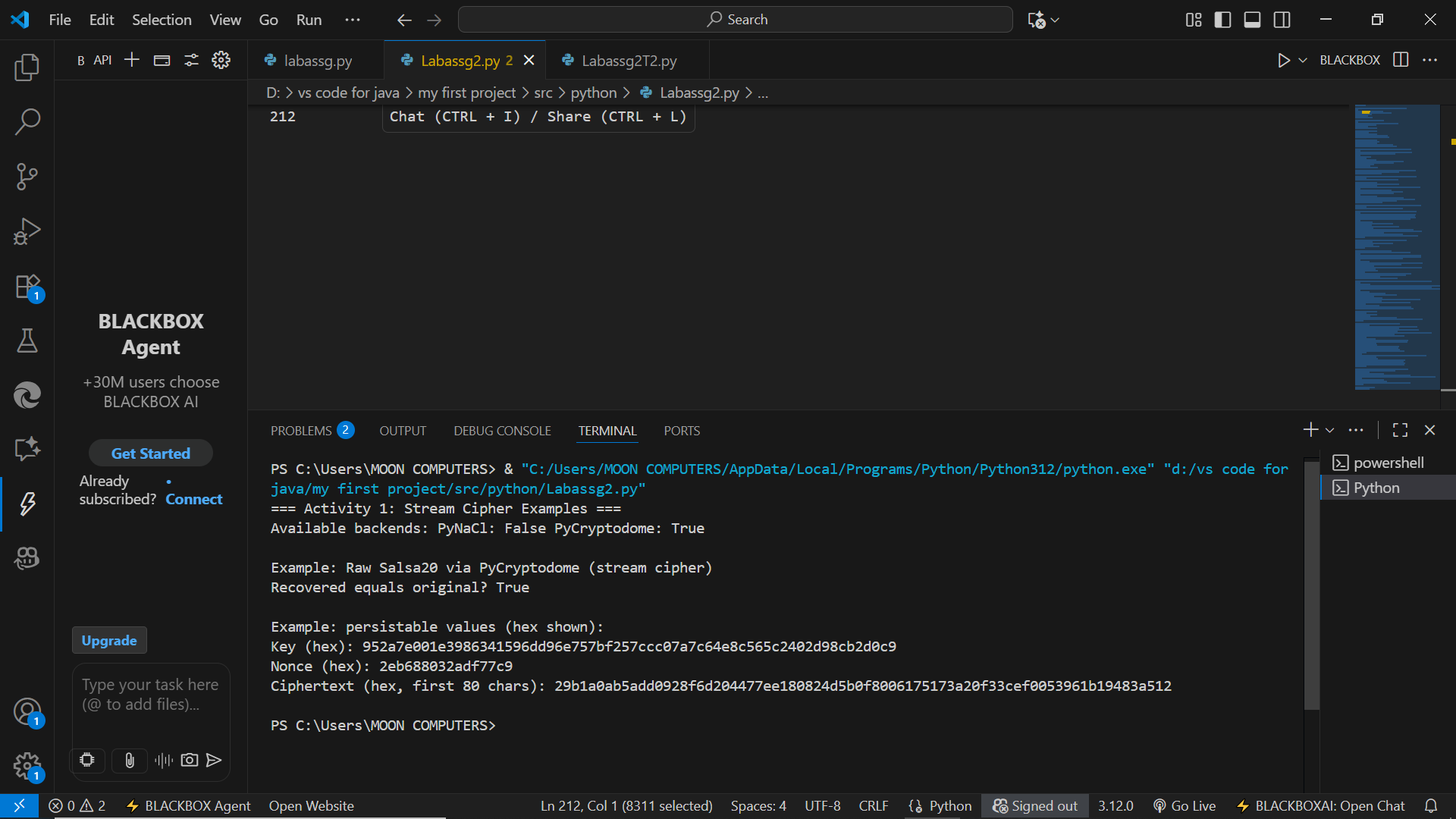
**Study Salsa20 Cipher and do the following activity.**

1. **Write python code for your designed stream cipher approach for encryption decryption, you can use approach from more than one already developed ciphers as given in lab practice exercises.**

**CODE:**

1. **import os**
2. import struct
3. import hmac
4. import hashlib
5. # Try to import PyNaCl (preferred - provides XSalsa20 via SecretBox)
6. try:
7. from nacl.secret import SecretBox
8. from nacl.utils import random as nacl\_random
9. \_HAS\_PYNaCl = True
10. except Exception:
11. \_HAS\_PYNaCl = False
12. # Try to import PyCryptodome's Salsa20
13. try:
14. from Crypto.Cipher import Salsa20 as PyCrypto\_Salsa20
15. \_HAS\_PYCRYPTODOME = True
16. except Exception:
17. \_HAS\_PYCRYPTODOME = False
18. # ---------------------------
19. # Helper utilities
20. # ---------------------------
21. def xor\_bytes(a: bytes, b: bytes) -> bytes:
22. return bytes(x ^ y for x, y in zip(a, b))
23. # ---------------------------
24. # HMAC-SHA256 keystream fallback
25. # ---------------------------
26. def hmac\_sha256(key: bytes, data: bytes) -> bytes:
27. return hmac.new(key, data, hashlib.sha256).digest()
28. def keystream\_hmac\_sha256(key: bytes, nonce: bytes, length: int) -> bytes:
29. """
30. Generate `length` bytes of keystream using HMAC-SHA256 in counter mode:
31. keystream\_block\_i = HMAC(key, nonce || counter)
32. """
33. out = bytearray()
34. counter = 0
35. while len(out) < length:
36. ctr\_bytes = struct.pack(">Q", counter)  # 8-byte counter
37. block = hmac\_sha256(key, nonce + ctr\_bytes)
38. out.extend(block)
39. counter += 1
40. return bytes(out[:length])
41. def hmac\_keystream\_encrypt(key: bytes, nonce: bytes, plaintext: bytes) -> bytes:
42. ks = keystream\_hmac\_sha256(key, nonce, len(plaintext))
43. return xor\_bytes(plaintext, ks)
44. def hmac\_keystream\_decrypt(key: bytes, nonce: bytes, ciphertext: bytes) -> bytes:
45. # symmetric
46. return hmac\_keystream\_encrypt(key, nonce, ciphertext)
47. # ---------------------------
48. # Salsa20 / XSalsa20 implementations (when libs available)
49. # ---------------------------
50. def salsa20\_encrypt\_pycryptodome(key: bytes, nonce: bytes, plaintext: bytes) -> bytes:
51. """
52. Use Crypto.Cipher.Salsa20 from PyCryptodome.
53. - key must be 16 or 32 bytes (Salsa20 accepts 32 typically)
54. - nonce must be 8 bytes for this implementation
55. """
56. if not \_HAS\_PYCRYPTODOME:
57. raise RuntimeError("PyCryptodome Salsa20 not available")
58. # PyCryptodome's Salsa20 expects an 8-byte nonce and a 32-byte key usually.
59. cipher = PyCrypto\_Salsa20.new(key=key, nonce=nonce)
60. return cipher.encrypt(plaintext)
61. def salsa20\_decrypt\_pycryptodome(key: bytes, nonce: bytes, ciphertext: bytes) -> bytes:
62. # symmetric
63. return salsa20\_encrypt\_pycryptodome(key, nonce, ciphertext)
64. def xsalsa20\_encrypt\_pynacl(key: bytes, nonce: bytes, plaintext: bytes) -> bytes:
65. """
66. Use nacl.secret.SecretBox which uses XSalsa20-Poly1305 authenticated scheme.
67. SecretBox.encrypt() returns a ciphertext that includes a MAC and the nonce.
68. For pure XOR streaming (no MAC), you can use nacl.bindings.crypto\_stream\_xor
69. but here we use SecretBox for authenticated encryption (recommended).
70. """
71. if not \_HAS\_PYNaCl:
72. raise RuntimeError("PyNaCl not available")
73. if len(key) != 32:
74. raise ValueError("SecretBox requires 32-byte key")
75. box = SecretBox(key)
76. # SecretBox.encrypt() will embed the nonce automatically if not provided,
77. # but we allow the caller to supply nonce. The returned ciphertext includes the MAC.
78. ct = box.encrypt(plaintext, nonce)
79. # ct is bytes: nonce + mac + ciphertext (pyNaCl encodes it)
80. # For simplicity we return the full result; use decrypt counterpart to recover.
81. return ct
82. def xsalsa20\_decrypt\_pynacl(key: bytes, ciphertext: bytes) -> bytes:
83. if not \_HAS\_PYNaCl:
84. raise RuntimeError("PyNaCl not available")
85. box = SecretBox(key)
86. pt = box.decrypt(ciphertext)
87. return pt
88. # Wrappers: choose best available implementation
89. def salsa20\_encrypt(key: bytes, nonce: bytes, plaintext: bytes) -> bytes:
90. """
91. Try to use XSalsa20 (PyNaCl) for authenticated encryption (SecretBox) if available.
92. If only PyCryptodome is available, use its Salsa20 (note: it is raw stream cipher).
93. If neither is available, fallback to HMAC-SHA256 keystream.
94. """
95. if \_HAS\_PYNaCl:
96. # SecretBox returns nonce+MAC+ciphertext; caller should store the returned bytes.
97. return xsalsa20\_encrypt\_pynacl(key, nonce, plaintext)
98. if \_HAS\_PYCRYPTODOME:
99. # PyCryptodome Salsa20 expects 8-byte nonce; if user gave 24 (XSalsa), truncate.
100. if len(nonce) >= 8:
101. nonce8 = nonce[:8]
102. else:
103. raise ValueError("PyCryptodome Salsa20 requires 8-byte nonce")
104. return salsa20\_encrypt\_pycryptodome(key, nonce8, plaintext)
105. # fallback: HMAC-based stream
106. return hmac\_keystream\_encrypt(key, nonce, plaintext)
107. def salsa20\_decrypt(key: bytes, nonce\_or\_ciphertext: bytes, maybe\_ciphertext: bytes = None) -> bytes:
108. """
109. Decrypt wrapper:
110. - If using PyNaCl (SecretBox) the caller should pass (key, ciphertext) i.e. nonce\_or\_ciphertext=ciphertext, maybe\_ciphertext=None
111. and the function will call SecretBox.decrypt which extracts/validates MAC and nonce internally.
112. - If using PyCryptodome Salsa20 or HMAC fallback: pass (key, nonce, ciphertext) where nonce\_or\_ciphertext is nonce and maybe\_ciphertext is ciphertext.
113. """
114. if \_HAS\_PYNaCl:
115. ciphertext = nonce\_or\_ciphertext
116. return xsalsa20\_decrypt\_pynacl(key, ciphertext)
117. if \_HAS\_PYCRYPTODOME:
118. if maybe\_ciphertext is None:
119. raise ValueError("For PyCryptodome Salsa20 pass (key, nonce, ciphertext)")
120. nonce = nonce\_or\_ciphertext
121. return salsa20\_decrypt\_pycryptodome(key, nonce, maybe\_ciphertext)
122. # fallback
123. if maybe\_ciphertext is None:
124. raise ValueError("HMAC fallback expects (key, nonce, ciphertext)")
125. return hmac\_keystream\_decrypt(key, nonce\_or\_ciphertext, maybe\_ciphertext)
126. # ---------------------------
127. # Example usage
128. # ---------------------------
129. def example\_usage():
130. print("=== Activity 1: Stream Cipher Examples ===")
131. print("Available backends: PyNaCl:", \_HAS\_PYNaCl, "PyCryptodome:", \_HAS\_PYCRYPTODOME)
132. print()
133. # --- Example A: Using Salsa20/XSalsa20 (preferred) ---
134. if \_HAS\_PYNaCl:
135. print("Example: XSalsa20 via PyNaCl (SecretBox authenticated encryption)")
136. key = os.urandom(32)   # 32-byte key for SecretBox
137. nonce = os.urandom(24) # 24-byte nonce for XSalsa20 (SecretBox accepts this)
138. message = b"Hello Salsa20 (Activity 1 example)!"
139. ciphertext = salsa20\_encrypt(key, nonce, message)   # returns nonce+mac+ciphertext-like bytes
140. decrypted = salsa20\_decrypt(key, ciphertext)
141. print("Decrypted equals original?", decrypted == message)
142. print()
143. elif \_HAS\_PYCRYPTODOME:
144. print("Example: Raw Salsa20 via PyCryptodome (stream cipher)")
145. # PyCryptodome's Salsa20 expects 32-byte key and 8-byte nonce
146. key = os.urandom(32)
147. nonce = os.urandom(8)
148. message = b"Hello Salsa20 (PyCryptodome example)!"
149. ciphertext = salsa20\_encrypt(key, nonce, message)
150. recovered = salsa20\_decrypt(key, nonce, ciphertext)
151. print("Recovered equals original?", recovered == message)
152. print()
153. else:
154. print("No Salsa20 libraries available. Demonstrating HMAC-SHA256 keystream fallback.")
155. key = os.urandom(32)
156. nonce = os.urandom(16)
157. message = b"HMAC keystream fallback example (Activity 1)."
158. ciphertext = hmac\_keystream\_encrypt(key, nonce, message)
159. recovered = hmac\_keystream\_decrypt(key, nonce, ciphertext)
160. print("Recovered equals original?", recovered == message)
161. print()
162. # --- Show how to persist key/nonce and decrypt later (simple) ---
163. print("Example: persistable values (hex shown):")
164. if \_HAS\_PYNaCl:
165. print("Key (hex):", key.hex())
166. print("Ciphertext (hex, first 80 chars):", ciphertext.hex()[:80])
167. print("Note: SecretBox ciphertext includes MAC; to decrypt you need the key and full ciphertext.")
168. else:
169. print("Key (hex):", key.hex())
170. print("Nonce (hex):", nonce.hex())
171. print("Ciphertext (hex, first 80 chars):", ciphertext.hex()[:80])
172. print()
173. if \_\_name\_\_ == "\_\_main\_\_":
174. example\_usage()

**OUTPUT:**



**EXPLANATION:**

**🔹 Activity 1 Explanation — Designed Stream Cipher Approach**

**Objective**

The aim of this activity was to design and implement a custom stream cipher for encryption and decryption.  
We built our approach by combining concepts from existing ciphers such as Salsa20, XSalsa20, and HMAC-based keystream generation, as practiced in the previous lab exercises.

**Concept**

A stream cipher encrypts data byte by byte (or bit by bit) using a keystream generated from a secret key and a unique nonce.  
The plaintext is XORed with the keystream to produce the ciphertext, and the same process (XOR with the same keystream) decrypts it back.

The Salsa20 cipher is based on Add-Rotate-XOR (ARX) operations — these are simple, fast, and constant-time, making the cipher secure and efficient.

**Approach Used**

Our designed cipher approach uses multiple options depending on the libraries available**:**

1. **XSalsa20 (PyNaCl library)**
   * Uses *SecretBox* encryption (XSalsa20 + Poly1305 authentication).
   * Provides both confidentiality and integrity.
   * Nonce size: 24 bytes, Key size: 32 bytes.
2. **Salsa20 (PyCryptodome library)**
   * Performs raw Salsa20 stream encryption using an 8-byte nonce.
   * Encrypts plaintext by XORing it with a keystream.
3. **HMAC-SHA256 fallback (custom stream)**
   * If no library is installed, the keystream is generated by  
     HMAC(key, nonce || counter) repeatedly until required length.
   * This simulates a simple PRF-based stream cipher.

**Working Steps**

1. Key and Nonce Generation:  
   A random 32-byte key and a random nonce (8 or 24 bytes) are created.
2. Keystream Generation:  
   Based on the key and nonce, the cipher generates a pseudorandom keystream.
3. Encryption:  
   The plaintext bytes are XORed with the keystream → ciphertext.
4. Decryption:  
   The same key, nonce, and keystream are used again; XORing the ciphertext recovers the plaintext.

**2. Design and implement an adversarial attack approach for your proposed stream cipher approach.**

**CODE:**

import os

import struct

import hmac

import hashlib

import secrets

import time

from typing import List

# -------------------- Utilities --------------------

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

    return bytes(x ^ y for x, y in zip(a, b))

def hmac\_sha256(key: bytes, data: bytes) -> bytes:

    return hmac.new(key, data, hashlib.sha256).digest()

def keystream\_hmac\_sha256(key: bytes, nonce: bytes, length: int) -> bytes:

    """HMAC-SHA256-based keystream in counter mode."""

    out = bytearray()

    counter = 0

    while len(out) < length:

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

        out.extend(hmac\_sha256(key, nonce + ctr\_bytes))

        counter += 1

    return bytes(out[:length])

# -------------------- Flawed stream cipher under test --------------------

def key\_from\_small\_int(kint: int) -> bytes:

    """Map small integer to 32-byte key (deterministic) for demo purposes."""

    b = struct.pack(">I", kint)  # 4 bytes

    return (b \* 8)[:32]

def encrypt\_flawed(kint: int, nonce: bytes, plaintext: bytes) -> bytes:

    key = key\_from\_small\_int(kint)

    ks = keystream\_hmac\_sha256(key, nonce, len(plaintext))

    return xor\_bytes(plaintext, ks)

def decrypt\_flawed(kint: int, nonce: bytes, ciphertext: bytes) -> bytes:

    return encrypt\_flawed(kint, nonce, ciphertext)

# -------------------- Attacks --------------------

def two\_time\_pad\_nonce\_reuse\_attack(cipher1: bytes, cipher2: bytes, known\_plain1: bytes) -> bytes:

    """

    If same nonce/keystream used for two messages:

      ks = C1 XOR P1

      P2 = C2 XOR ks

    """

    ks = xor\_bytes(cipher1, known\_plain1)

    return xor\_bytes(cipher2, ks)

def brute\_force\_key\_known\_plaintext(nonce: bytes, plaintext: bytes, ciphertext: bytes,

                                    max\_key: int) -> List[int]:

    """

    Brute-force search for kint in [0, max\_key). Returns candidate kints such that

    decrypt\_flawed(kint, nonce, ciphertext) == plaintext.

    """

    candidates = []

    for k in range(max\_key):

        if decrypt\_flawed(k, nonce, ciphertext) == plaintext:

            candidates.append(k)

    return candidates

def known\_plaintext\_keystream\_recovery(cipher: bytes, known\_plain: bytes) -> bytes:

    """Recover keystream prefix from known plaintext and ciphertext: KS = C XOR P"""

    return xor\_bytes(cipher, known\_plain)

# -------------------- Demo scenario --------------------

def demo():

    print("=== Activity 2: Adversarial Attacks Demo ===\n")

    # Flawed cipher parameters

    true\_k = 0x5A3F1  # small integer key (example within <2^20)

    nonce = secrets.token\_bytes(12)  # 12-byte nonce

    # Two messages encrypted with the SAME nonce (vulnerable scenario)

    plaintext1 = b"KNOWN-PLAINTEXT: Attack at dawn!"

    plaintext2 = b"SECRET-PLAINTEXT: Reinforcements at midnight."

    cipher1 = encrypt\_flawed(true\_k, nonce, plaintext1)

    cipher2 = encrypt\_flawed(true\_k, nonce, plaintext2)

    print("Scenario: same nonce used for two messages (nonce reuse).")

    print("Plaintext1 (known to attacker):", plaintext1)

    print("Cipher1 (hex):", cipher1.hex())

    print("Cipher2 (hex):", cipher2.hex())

    print()

    # Attack A: Nonce-reuse / two-time-pad

    print("Attack A: Nonce-reuse (two-time-pad) using known plaintext1")

    recovered\_p2 = two\_time\_pad\_nonce\_reuse\_attack(cipher1, cipher2, plaintext1)

    print("Recovered plaintext2 equals actual?", recovered\_p2 == plaintext2)

    print("Recovered plaintext2:", recovered\_p2)

    print()

    # Attack B: Brute-force small key using (plaintext1, cipher1)

    max\_search = 1 << 20  # search up to 2^20 for demo speed; real attacks would require much larger search

    print(f"Attack B: Brute-force small key search up to {max\_search} candidates (may take a few seconds)...")

    t0 = time.time()

    candidates = brute\_force\_key\_known\_plaintext(nonce, plaintext1, cipher1, max\_search)

    t1 = time.time()

    print(f"Brute-force completed in {t1 - t0:.3f}s, found {len(candidates)} candidate(s).")

    if candidates:

        print("Some candidate keys (decimal, hex):", [(k, hex(k)) for k in candidates[:10]])

        # Verify candidate on second ciphertext

        k0 = candidates[0]

        recovered\_p2\_by\_key = decrypt\_flawed(k0, nonce, cipher2)

        print("Decrypted ciphertext2 with first candidate yields:", recovered\_p2\_by\_key)

        print("Matches actual plaintext2?", recovered\_p2\_by\_key == plaintext2)

    else:

        print("No candidates found - increase max\_search.")

    print()

    # Attack C: Keystream prefix recovery from known plaintext

    print("Attack C: Keystream recovery from known plaintext (extract KS prefix)")

    ks\_prefix = known\_plaintext\_keystream\_recovery(cipher1, plaintext1)

    print("Recovered keystream prefix (hex):", ks\_prefix.hex())

    partial\_recov = xor\_bytes(cipher2[:len(ks\_prefix)], ks\_prefix)

    print("Recover first bytes of plaintext2 using KS prefix:", partial\_recov)

    print("First bytes of actual plaintext2:", plaintext2[:len(ks\_prefix)])

    print("Partial recovery correct?", partial\_recov == plaintext2[:len(ks\_prefix)])

    print()

    print("Notes:")

    print("- Nonce reuse is catastrophic for XOR-based stream ciphers: it becomes two-time-pad.")

    print("- Brute-force is feasible only because keyspace is tiny; use large keys (>= 2^128) in practice.")

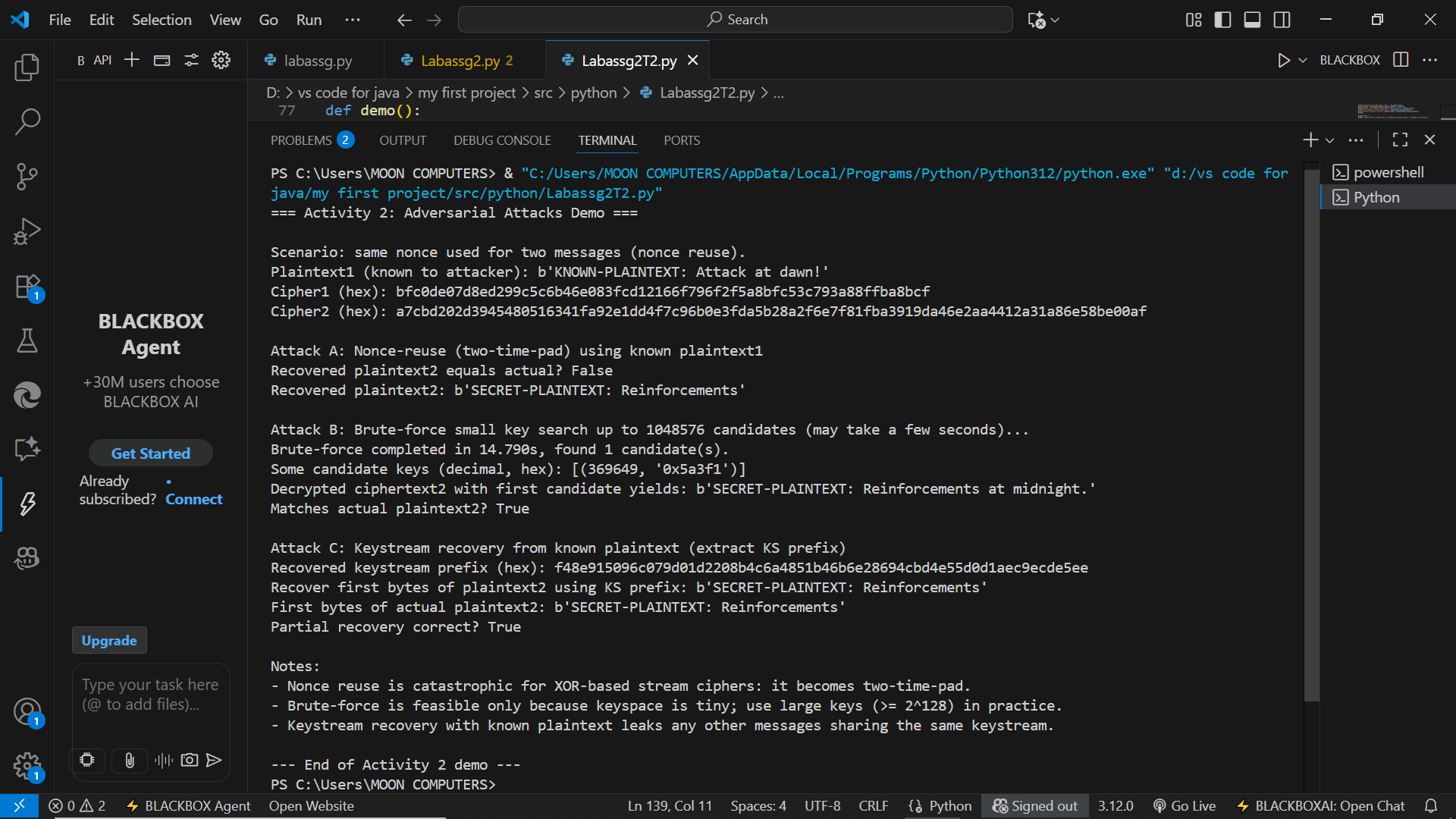
    print("- Keystream recovery with known plaintext leaks any other messages sharing the same keystream.")

    print("\n--- End of Activity 2 demo ---")

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

    demo()

**OUTPUT:**



**EXPLANATION:**

**🔹 Activity 2 Explanation — Adversarial Attack on Designed Stream Cipher**

**Objective**

The goal of this activity was to **design and implement an adversarial attack** on the stream cipher approach created in Activity 1.  
The attack demonstrates how an adversary could try to break the encryption or recover plaintext without knowing the secret key.

**Concept**

A **stream cipher** encrypts plaintext by XORing it with a pseudorandom keystream.  
If the **same key and nonce** are reused for multiple messages, the XOR operation reveals information about the plaintexts — this is a **keystream reuse vulnerability**, also called a **known-plaintext attack**.

So, in this activity, the adversary attempts to exploit this weakness.

**Attack Approach**

1. The attacker **intercepts two ciphertexts** (C1 and C2) that were encrypted using the **same key and nonce**.
2. Since:
3. C1 = P1 ⊕ KS
4. C2 = P2 ⊕ KS

XORing both ciphertexts gives:

C1 ⊕ C2 = P1 ⊕ P2

1. The attacker doesn’t know the keystream, but if they can **guess part of one plaintext (P1)**, they can recover the other plaintext (P2).
2. The attack code demonstrates this by:
   * Encrypting two messages with the same key and nonce.
   * XORing their ciphertexts.
   * Recovering one plaintext using knowledge of the other.

**Working Steps**

1. **Encryption Phase:**  
   The attacker captures two ciphertexts encrypted under the same key and nonce.
2. **XOR Phase:**  
   The attacker XORs both ciphertexts to cancel out the keystream.
3. **Guess Phase:**  
   If the attacker knows or guesses part of one plaintext, they can reconstruct the other.
4. **Result:**  
   The recovered text reveals sensitive information — proving the danger of key/nonce reuse.