SCC.363 - Security and Risk - Coding Coursework 1

Task 1

==========

Implement a function that encrypts a finite string by XORing it against a repeating key and returns the hex value as a string. The length of the key is less or equal to the length of the plaintext.

The function's name has to be RepeatingXOREncrypt and take two parameters per the code below.

```
# YOUR IMPORTS

def RepeatingXOREncrypt(key, string):
      # YOUR IMPEMENTATION

      return # The result of XORing the string with the repeating key
```

You can test your code in your system (NOT IN YOUR CODERUNNER SUBMISSION) as follows:

```
if __name__ == "__main__":

    # TASK 1
    result = RepeatingXOREncrypt("01", "0123")
    print(result)
```

Test case

Input:

`key = "01"` (this is a string)

`string = "0123"` (this is a string)

Output:

`00000202` (this is a hex value returned as a string)

Marking scheme: This task's weight is 30% for returning the correct output. Your code will be checked against a set of test cases.

Task 2

==========

Users A and B use the Diffie-Hellman (DH) key exchange protocol to share a secret key and start encrypting data. You can assume that users A and B agreed on some DH parameters and calculated their private keys. You are given the following private keys for users A and B, respectively.

User's A private key (in PEM, no password protected):

```
b'-----BEGIN PRIVATE KEY-----
\nMIGcAgEAMFMGCSqGSIb3DQEDATBGAkEAlry2DwPC+pK/0QiOicVAtt6ANsfjmD9P\nQrDC6Zk
YcrRf0q0RVzMDTnHWk1mRLVvb6av4HOSkIsk1mMogBcqV0wIBAgRCAkBm\nZK4qUqvU6WaPy4fN
G9oWIXchxzztxmA7p9BFXbMzn3rHcW84SDwTWXAjkRd35XPV\n/9RAl06sv191BNFFPyg0\n---
--END PRIVATE KEY-----\n'
```

User's B private key (in PEM, no password protected):

```
b'-----BEGIN PRIVATE KEY-----
\nMIGcAgEAMFMGCSqGSIb3DQEDATBGAkEAlry2DwPC+pK/0QiOicVAtt6ANsfjmD9P\nQrDC6Zk
YcrRf0q0RVzMDTnHWk1mRLVvb6av4HOSkIsk1mMogBcqV0wIBAgRCAkBn\n9zn/q8GMs7SJjZ+V
LlPG89bB83Cn1kDRmGEdUQF3OSZWIdMAVJb1/xaR4NAhlRya\n7jZHBW5DlUF5rrmecN4A\n---
--END PRIVATE KEY-----\n'
```

Consider the following key derivation configuration:

```
HKDF(
  algorithm=hashes.SHA256(),
  length=32,
  salt=None,
  info=b'handshake data')
```

Complete the key exchange and derive the shared key between users A and B. Use the derived key to encrypt using XOR a finite plaintext, e.g., `b"Encrypt me with the derived key!"`. In your implementation, you should use modules from cryptography.io.

```
# YOUR IMPORTS

def DHandEncrypt(A_Private_Key, B_Private_Key, PlainText):

      # TODO

      return # You should return 2 variables, i.e., the derived key from
Diffie-Hellman and ciphertext in this order.
```

You can test your code in your system (NOT IN YOUR CODERUNNER SUBMISSION) as follows:

```
# Main
if __name__ == "__main__":

      A_PRIVATE_KEY = b'-----BEGIN PRIVATE KEY-----
\nMIGcAgEAMFMGCSqGSIb3DQEDATBGAkEAlry2DwPC+pK/0QiOicVAtt6ANsfjmD9P\nQrDC6Zk
YcrRf0q0RVzMDTnHWk1mRLVvb6av4HOSkIsk1mMogBcqV0wIBAgRCAkBm\nZK4qUqvU6WaPy4fN
G9oWIXchxzztxmA7p9BFXbMzn3rHcW84SDwTWXAjkRd35XPV\n/9RAl06sv191BNFFPyg0\n---
--END PRIVATE KEY-----\n'
```

```
    B_PRIVATE_KEY = b'-----BEGIN PRIVATE KEY-----
\nMIGcAgEAMFMGCSqGSIb3DQEDATBGAkEAlry2DwPC+pK/0QiOicVAtt6ANsfjmD9P\nQrDC6Zk
YcrRf0q0RVzMDTnHWk1mRLVvb6av4HOSkIsk1mMogBcqV0wIBAgRCAkBn\n9zn/q8GMs7SJjZ+V
LlPG89bB83Cn1kDRmGEdUQF3OSZWIdMAVJb1/xaR4NAhlRya\n7jZHBW5DlUF5rrmecN4A\n---
--END PRIVATE KEY-----\n'

      PlainText = b"Encrypt me with the derived key!"

      STD_KEY, STD_CIPHER = DHandEncrypt(A_PRIVATE_KEY, B_PRIVATE_KEY,
PlainText)
```

Information on the type of variables:
* A_Private_Key and B_Private_Key are in PEM format
* Plaintext as bytes, e.g., b"this is a message"
* Both the returned shared key and cipher have to be in bytes

Test case:
Using the above private keys and `PlainText = b"Encrypt me with the derived key!"` the output should be the following:

Output:
You have to find the key by implementing DH, hence it can't be provided since it is part of the task's solution.
XORing the key you have found with `PlainText = b"Encrypt me with the derived key!"` will result in:
```
b'\xd8W\xd1\xfe\xb2\xb9_\x89\x90?O\tF\xde\xeb\xe1\xa1Gx\xb18\x1cY\x1e\xaf\x
e0QmL\xf6\xeb\x0e'
```

Marking scheme: This task's weight is 40%, i.e., 10% for returning the correct shared key and 30% for returning the correct ciphertext. Your code will be checked against a set of test cases.

Task 3

==========

You are provided with the following implementation of AES in CTR mode using the cryptography.io modules.

```
# START OF CODE
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes

def AES_CTR_Encrypt(key, nonce_counter, data):

    key = bytes.fromhex(key)
    nonce_counter = bytes.fromhex(nonce_counter)

    aesCipher = Cipher(algorithms.AES(key), modes.CTR(nonce_counter))
    aesEncryptor = aesCipher.encryptor()

    cipherText = aesEncryptor.update(data)
    cipherText += aesEncryptor.finalize()

    return cipherText

# You can test your code in you system (NOT IN YOUR CODERUNNER SUBMISSION)
as follows:
# Main
if __name__ == "__main__":
      key =
'0000000000000000000000000000000000000000000000000000000000000001'
      nonce_counter = '00000000000000000000000000000001'
      data = b"12345678901234567890123456789012"
      result = AES_CTR_Encrypt(key, nonce_counter, data)
      print(result)

# END OF CODE
```

Re-implement the function above using ONLY the ECB mode of AES, i.e., implement the encrypt operation of AES-CTR using an AES-ECB cipher and encryptor.

Test case:

Using the key, nonce_counter and data in the code above will produce the following output:

```
b'\xf8\xe0\x86\xef
\xcb\xbe:\x9f\xdb\xc5J\xf0\x83\xb5"n\xafT\x9f\xf9Px\xfbe\x17v\xbb\xfb-
\xbc\xc3'
```

Marking scheme: This task's weight is 30% for returning the correct output. Your code will be checked against a set of test cases.