Question 2

T6

```
import sys
import binascii
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Random.random import randrange
from Crypto.Util.Padding import pad

# Recommand: https://braincoke.fr/write-up/cryptopals/cryptopals-ecb-decryption-harder/

class ECB_Oracle:
    def __init__(self,secret_padding):
        self.__key = get_random_bytes(AES.key_size[0])
        self.__secret_padding = secret_padding
        self.__random_prefix = get_random_bytes(randrange(1,20))
        self.__cipher = AES.new(self.__key, AES.MODE_ECB)
        # Use this means not perfect, however without it the bruteforce of padding results in problems.
        self.secret_padding_len = len(secret_padding)

    def encrypt(self,input):
        # The prefix is relatively, this is the base of decryption.
        return self.__cipher.encrypt(pad(self.__random_prefix + input + self.__secret_padding, AES.block_size))

def byte_at_a_time_ECB_decryption_simple(encrypt_Orcale):
    """1 Feed identical bytes of your-string to the function 1 at a time --- start with 1 byte
     ("A"), then "AA", then "AAA" and so on. Discover the block size of the cipher. You know
     it, but do this step anyway. """
    block_len = find_block_len(encrypt_Orcale)

    """2 Detect that the function is using ECB. You already know, but do this step anyways. """
    ct = encrypt_Orcale.encrypt(bytes([0]*64))
    ct_list = [ct[i:i+block_len] for i in range(0,len(ct),block_len)]
    assert(len(ct_list) > len(set(ct_list)))

    """n Find the length of prefix"""
    prefix_len = find_prefix_len(encrypt_Orcale, block_len)

    """3 Knowing the block size, craft an input block that is exactly 1 byte short (for instance,
    if the block size is 8 bytes, make "AAAAAAA"). Think about what the oracle function is going
    to put in that last byte position.
    4 Make a dictionary of every possible last byte by feeding different strings to the oracle;
```

```
    for instance, "AAAAAAAA", "AAAAAAAB", "AAAAAAAC", remembering the first block
of each invocation.
    5 Match the output of the one-byte-short input to one of the entries in your
dictionary.
    You've now discovered the first byte of unknown-string.
    6 Repeat for the next byte."""
    # secret_padding_len = len(encrypt_Orcale.encrypt(b''))
    # 以secret_padding_len遍历会导致填充问题
    detect_secret_padding = b''
    for pos in range(encrypt_Orcale.secret_padding_len):
        detect_secret_padding += get_next_byte(encrypt_Orcale, block_len,
prefix_len, detect_secret_padding)

    # return unpad(detect_secret_padding, AES.block_size)
    return detect_secret_padding

def find_block_len(encrypt_Orcale):
    work = b''
    init_len = len(encrypt_Orcale.encrypt(work))
    new_len = init_len
    while new_len == init_len:
        work += b'A'
        new_len = len(encrypt_Orcale.encrypt(work))
    return new_len - init_len

def find_prefix_len(encrypt_Orcale, block_len):
    # Prefix consists of two parts: prefix_chunks + prefix_bits_left(maybe be
just some chunks)
    # Step 1: find the length of prefix_chunks
    ct1 = encrypt_Orcale.encrypt(b'')
    ct2 = encrypt_Orcale.encrypt(b'\x00')
    # We assert that the first byte of secret_padding is not '\x00' !!!!!!!!!!
    # This makes it that there is 1/256 chance it will run wrong !!!!!!!!!!!!!
    for chunks_len in range(0, len(ct2), AES.block_size):
        if ct1[chunks_len : chunks_len + AES.block_size] != ct2[chunks_len :
chunks_len + AES.block_size]:
            break

    # Step 2:find the length of prefix_bits_left
    for i in range(block_len):
        ct_test = encrypt_Orcale.encrypt(bytes([0] * (i + 2 * AES.block_size)))
        if i == 0:
            if ct_test[chunks_len : chunks_len + AES.block_size] ==
ct_test[chunks_len + AES.block_size : chunks_len + 2 * AES.block_size]:
                bits_left_len = i
                break
        else:
            if ct_test[chunks_len + AES.block_size : chunks_len + 2 *
AES.block_size] == ct_test[chunks_len + 2 * AES.block_size : chunks_len + 3 *
AES.block_size]:
                bits_left_len = AES.block_size - i
                break
    return chunks_len + bits_left_len


def get_next_byte(encrypt_Oracle, block_len, sys_prefix_len,
detect_secret_padding):
```

```
    # Working block consists of 4 parts: sys_prefix_len, prefix, detected_secret,
bruteforce_bit
    prefix_len = (block_len - sys_prefix_len - (len(detect_secret_padding) + 1 ))
% block_len
    prefix = prefix_len * b'A'
    work_len = sys_prefix_len + prefix_len + len(detect_secret_padding) + 1

    real_ct = encrypt_Oracle.encrypt(prefix)
    for bruteforce_bit in range(0x00,0x100):
        bruteforce_byte = bytes([bruteforce_bit])
        bruteforce_ct = encrypt_Oracle.encrypt(prefix + detect_secret_padding +
bruteforce_byte)
        if real_ct[:work_len] == bruteforce_ct[:work_len]:
            return bruteforce_byte
    return b''

if __name__ == "__main__":
    """Approach:
    1) Find the block_length and the encryption mode (as in S2C12)
    2) Find the prefix length
    3) Decrypt byte-by-byte the mysterious message (similar to S2C12)
    """
    secret_padding =
binascii.a2b_base64("Um9sbGluJyBpbiBteSA1LjAKV2l0aCBteSByYWctdG9wIGRvd24gc28gbXkg
aGFpciBjYW4gYmxvdwpUaGUgZ2lybGllcyBvbiBzdGFuZGJ5IHdhdmluZyBqdXN0IHRvIHNheSBoaQpEa
WQgeW91IHN0b3A/IE5vLCBJIGp1c3QgZHJvdmUgYnkK")
    oracle = ECB_Oracle(secret_padding)
    detect_secret_padding = byte_at_a_time_ECB_decryption_simple(oracle)
    print(detect_secret_padding == secret_padding)
    print(detect_secret_padding)
```

T7

```
def is_pkcs7_padding(bytes_data):
    # The type of index value of bytes is int.
    padding = bytes_data[- bytes_data[-1]:]
    # Get the use of all!
    try:
        return all(padding[i] == len(padding) for i in range(len(padding)))
    except IndexError as err:
        # This is not a good example of throwing an exception.
        print("The padding is not pkcs#7",err)
        return False

if __name__ == "__main__":
    print(is_pkcs7_padding(b"ICE ICE BABY\x04\x04\x04\x04"))
    print(is_pkcs7_padding(b"ICE ICE BABY\x05\x05\x05\x05"))
    print(is_pkcs7_padding(b"ICE ICE BABY\x01\x02\x03\x04"))
    # print(is_pkcs7_padding(b"ICE ICE BABY\x01\x02\x03\xff"))
```

T8

```
from Crypto.Cipher import AES
```

```
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad,unpad
from s2c14_byte_at_a_time_ECB_decryption_harder import
find_block_len,find_prefix_len

class CBC_Oracle:
    def __init__(self) -> None:
        self.__key = get_random_bytes(AES.key_size[0])
        self.__iv = get_random_bytes(AES.block_size)
        self.__prefix = "comment1=cooking%20MCs;userdata="
        self.__suffix = ";comment2=%20like%20a%20pound%20of%20bacon"
        # self.__cipher = AES.new(self.__key, AES.MODE_CBC, self.__key)

    def encrypt(self, message):
        # Without adding iv at the head of cipher text.
        message = message.replace(';','').replace('=','')
        total_pt = self.__prefix + message + self.__suffix
        cipher = AES.new(self.__key, AES.MODE_CBC, self.__iv)
        return cipher.encrypt(pad(total_pt.encode(), AES.block_size))

    def decrypt_and_verify(self, ct):
        # Verify whether the cipher text contains ";admin=true;"
        cipher = AES.new(self.__key, AES.MODE_CBC, self.__iv)
        pt = unpad(cipher.decrypt(ct), AES.block_size)
        print("PT =",pt)
        # return pt.contains(";admin=true;")
        return b";admin=true;" in pt

def find_block_len(oracle_encrypt):
    work = ''
    init_len = len(oracle_encrypt(work))
    new_len = init_len
    while new_len == init_len:
        work += 'A'
        new_len = len(oracle_encrypt(work))
    return new_len - init_len

def find_prefix_len(oracle_encrypt, block_len):
    encrypt_A = oracle_encrypt('A')
    encrypt_B = oracle_encrypt('B')
    for i in range(0, len(encrypt_A), block_len):
        if encrypt_A[i:i+block_len] != encrypt_B[i:i+block_len]:
            chunk_len = i
            break
    for i in range(1, block_len + 1):
        encrypt_A = oracle_encrypt(i*'x' + 'A')
        encrypt_B = oracle_encrypt(i*'x' + 'B')
        if encrypt_A[chunk_len:chunk_len+block_len] ==
encrypt_B[chunk_len:chunk_len+block_len]:
            bits_left_len = block_len - i
    return chunk_len + bits_left_len

def bitflippling_attacks(oracle):
    # block_len = find_block_len(oracle.encrypt())
    block_len = find_block_len(oracle.encrypt)
    prefix_len = find_prefix_len(oracle.encrypt, block_len)
    prefix_add_len = (block_len - prefix_len) % block_len
```

```
    pt = "?admin?true?"
    pt_add_len = (block_len - len(pt)) % block_len

    true_ct = oracle.encrypt(prefix_add_len*'?' + pt_add_len*'?' + pt)
    # total_len = prefix_len + prefix_add_len +  pt_add_len + len(pt)
    # print(total_len)
    # fake_ct =  true_ct[:total_len-12] + bytes([true_ct[total_len-12] ^
ord('?')^ord(';')]) + \
    #            true_ct[total_len-11:total_len-6] + bytes([true_ct[total_len-
6]^ord('?')^ord('=')]) + \
    #            true_ct[total_len-5:total_len-1] + bytes([true_ct[total_len-
1]^ord('?')^ord(';')]) +\
    #            true_ct[total_len:]
    total_len = prefix_len + prefix_add_len
    fake_ct =  true_ct[:total_len-12] + bytes([true_ct[total_len-12] ^
ord('?')^ord(';')]) + \
               true_ct[total_len-11:total_len-6] + bytes([true_ct[total_len-
6]^ord('?')^ord('=')]) + \
               true_ct[total_len-5:total_len-1] + bytes([true_ct[total_len-
1]^ord('?')^ord(';')]) +\
               true_ct[total_len:]
    # print("len(true)={} len(fake)={}".format(len(true_ct),len(fake_ct)))
    return oracle.decrypt_and_verify(fake_ct)

if __name__ == "__main__":
    oracle = CBC_Oracle()
    print(bitflippling_attacks(oracle))
```

运行结果