# 密码实验2

原创

Juli_Eyre 于 2021-10-25 19:28:20 发布 2279 收藏
分类专栏： 密码学 文章标签： python 安全

密码学 专栏收录该内容
2 篇文章 0 订阅
订阅专栏

## 文章目录

本文参考的博客：
https://github.com/Rajil1213/Cryptopals-Set-2/blob/master/writeup/writeup.md
https://braincoke.fr/write-up/cryptopals/cryptopals-byte-a-time-ecb-decryption-simple/

# Ex1

## Many-Time-Pad 攻击（Cryptography-Stanford实验）

思路：

- 利用空格，空格异或k异或空格得到密钥，所以要得到空格异或k，即空格加密后的密文；
- 一个大写字母与空格异或，结果为其对应的小写字母；一个小写字母与空格异或，结果为其对应的大写字母；对给出的密文两两异或后得到的数据串，判断其中[A-Z]或者[a-z]或者[0]的位置,这些位置都有可能是明文中空格的位置

假设空格来自 `ciphertext_a`，其他密文对应 `ciphertext_b:`

```
for ciphertext_a in ciphertexts:
    space_suspect_counts = [0] * len(ciphertext_a)
    for ciphertext_b in ciphertexts:
        #如果遇到自己，跳过，不和自己做异或
        if ciphertext_a == ciphertext_b:
            continue
```

两两异或（这样可以进行多次统计，体现在 对空格可能在的位置进行计数，max=10,若大于8，就确定他是空格），得到空格可能的位置。
如何得到key: (这里其实是假设了key都是可见字符)空格再与ciphertext_a对应位置异或，得到初步确定的key
初步确定的key再与其他密文异或，如果满足全部是在字母范围内，再添加进key值
这样其实得到11组key

```
import binascii

ciphertexts = [
    "315c4eeaa8b5f8aaf9174145bf43e1784b8fa00dc71d885a804e5ee9fa40b16349c146fb778cdf2d3aff021dfff5b403b510d0d0455468aeb98
622b137dae857553ccd8883a7bc37520e06e515d22c954eba5025b8cc57ee59418ce7dc6bc41556bdb36bbca3e8774301fbcaa3b83b220809
560987815f65286764703de0f3d524400a19b159610b11ef3e",
    "234c02ecbbfbafa3ed18510abd11fa724fcda2018a1a8342cf064bbde548b12b07df44ba7191d9606ef4081ffde5ad46a5069d9f7f543bedb9c
861bf29c7e205132eda9382b0bc2c5c4b45f919cf3a9f1cb74151f6d551f4480c82b2cb24cc5b028aa76eb7b4ab24171ab3cdadb8356f",
    "32510ba9a7b2bba9b8005d43a304b5714cc0bb0c8a34884dd91304b8ad40b62b07df44ba6e9d8a2368e51d04e0e7b207b70b9b8261112
bacb6c866a232dfe257527dc29398f5f3251a0d47e503c66e935de81230b59b7afb5f41afa8d661cb",
    "32510ba9aab2a8a4fd06414fb517b5605cc0aa0dc91a8908c2064ba8ad5ea06a029056f47a8ad3306ef5021eafe1ac01a81197847a5c68a1
b78769a37bc8f4575432c198ccb4ef63590256e305cd3a9544ee4160ead45aef520489e7da7d835402bca670bda8eb775200b8dabbba246b13
0f040d8ec6447e2c767f3d30ed81ea2e4c1404e1315a1010e7229be6636aaa",
    "3f561ba9adb4b6ebec54424ba317b564418fac0dd35f8c08d31a1fe9e24fe56808c213f17c81d9607cee021dafe1e001b21ade877a5e68bea
88d61b93ac5ee0d562e8e9582f5ef375f0a4ae20ed86e935de81230b59b73fb4302cd95d770c65b40aaa065f2a5e33a5a0bb5dcaba43722130f
042f8ec85b7c2070",
    "32510bfbacfbb9befd54415da243e1695ecabd58c519cd4bd2061bbde24eb76a19d84aba34d8de287be84d07e7e9a30ee714979c7e1123a
8bd9822a33ecaf512472e8e8f8db3f9635c1949e640c621854eba0d79eccf52ff111284b4cc61d11902aebc66f2b2e436434eacc0aba938220b0
84800c2ca4e693522643573b2c4ce35050b0cf774201f0fe52ac9f26d71b6cf61a711cc229f77ace7aa88a2f19983122b11be87a59c355d25f8e4
",
    "32510bfbacfbb9befd54415da243e1695ecabd58c519cd4bd90f1fa6ea5ba47b01c909ba7696cf606ef40c04afe1ac0aa8148dd066592ded9f
8774b529c7ea125d298e8883f5e9305f4b44f915cb2bd05af51373fd9b4af511039fa2d96f83414aaaf261bda2e97b170fb5cce2a53e675c154c0
d9681596934777e2275b381ce2e40582afe67650b13e72287ff2270abcf73bb028932836fbdecfecee0a3b894473c1bbeb6b4913a536ce4f9b13
f1efff71ea313c8661dd9a4ce",
    "315c4eeaa8b5f8bffd11155ea506b56041c6a00c8a08854dd21a4bbde54ce56801d943ba708b8a3574f40c00fff9e00fa1439fd0654327a3bf
c860b92f89ee04132ecb9298f5fd2d5e4b45e40ecc3b9d59e9417df7c95bba410e9aa2ca24c5474da2f276baa3ac325918b2daada43d6712150
441c2e04f6565517f317da9d3",
    "271946f9bbb2aeadec111841a81abc300ecaa01bd8069d5cc91005e9fe4aad6e04d513e96d99de2569bc5e50eeeca709b50a8a987f4264e
db6896fb537d0a716132ddc938fb0f836480e06ed0fcd6e9759f40462f9cf57f4564186a2c1778f1543efa270bda5e933421cbe88a4a52222190f4
71e9bd15f652b653b7071aec59a2705081ffe72651d08f822c9ed6d76e48b63ab15d0208573a7eef027",
    "466d06ece998b7a2fb1d464fed2ced7641ddaa3cc31c9941cf110abbf409ed39598005b3399ccfafb61d0315fca0a314be138a9f32503bedac
8067f03adbf3575c3b8edc9ba7f537530541ab0f9f3cd04ff50d66f1d559ba520e89a2cb2a83",
    "32510ba9babebbbefd001547a810e67149caee11d945cd7fc81a05e9f85aac650e9052ba6a8cd8257bf14d13e6f0a803b54fde9e77472dbff
89d71b57bddef121336cb85ccb8f3315f4b52e301d16e9f52f904",
 ]
#转换为16进制数字
ciphertexts = [binascii.unhexlify(x) for x in ciphertexts]
#print(ciphertexts)
target_ciphertext = ciphertexts[-1]
suspect_num=[]

#进行异或，zip函数将长的字符串截取到和短的字符串相同的长度
#res直接和最短的同，不会循环加密
def strxor(a, b):
    return "".join([chr(x ^ y) for x, y in zip(a, b)])
def repeat_key_xor(m,key):
   output_bytes=b''
   index=0
   for byte in m:#直接就是对应的ASCII
       #print(byte)
       #print(key[index])
       output_bytes+=bytes([byte^key[index]])
       #bytes[116,112]转为b'tp'
       if(index+1)==len(key):
           index=0
       else:
           index+=1
   return output_bytes
key = [0] * 1000
max_space_count=[0]*1000
#开始做异或,有c1 ⊕c2=m1 ⊕k ⊕m2 ⊕k=m1 ⊕m2
for ciphertext_a in ciphertexts:
    space_suspect_counts = [0] * len(ciphertext_a)
    for ciphertext_b in ciphertexts:
        #如果遇到自己，跳过，不和自己做异或
        if ciphertext_a == ciphertext_b:
            continue
       #开始两两异或
        a_xor_b = strxor(ciphertext_a, ciphertext_b)
        for char_idx, xor_resulting_char in enumerate(a_xor_b):
            #一个大写字母与空格异或，结果为其对应的小写字母；一个小写字母与空格异或，结果为其对应的大写字母。
            ascii_null = "\x00"
            #isaplha()函数：判断字符串中是否是字母
            if xor_resulting_char.isalpha() or xor_resulting_char == ascii_null:
                #获得可能的空格位置，统计这个位置和其他字符串异或出现字母的次数
                space_suspect_counts[char_idx] += 1

   #开始假设空格位置来自于ciphertext_a，如果统计的某个位置上满足要求的次数大于等于我所期望的次数，
   #那么我们假设这个位置上就是空格且来自与ciphertext_a中对应的位置
    #space_tolerance_threshold = 0.8
    for char_idx, suspect_count in enumerate(space_suspect_counts):
        #判断出现的次数是否满足要求
        if suspect_count >= max_space_count[char_idx]:
            #通过判断得知密文这个位置上大概率是space，所以，此时 m=space，所以 c ⊕space=m ⊕k ⊕space=k，从而可以求的key值
           whitespace = ord(" ")
           suspect_key=ciphertext_a[char_idx] ^ whitespace
            #增加判断，将 key再与其他密文对应位置异或，如果满足全部是在字母范围内，再添加进key值。
            #等于说又判断了一次？？？
            for ciphertext_b in ciphertexts:
                if char_idx<len(ciphertext_b):
                    if(0x41<=ciphertext_b[char_idx]^suspect_key<=0x7A):
                        max_space_count[char_idx] = suspect_count
                        key[char_idx]=suspect_key
#print("".join(chr(i)for i in key))
print(len(key))
print(len(ciphertexts[0]))
i=0
target_plaintext=[0]*11
for ciphertext_c in ciphertexts:
    # print(len(key)==len(ciphertext_c)) false
    target_plaintext[i]= strxor(ciphertext_c,key)
    #print("".join(chr(i)for i in key))
    print(f"明文{i}:{target_plaintext[i]}")
    print(len(target_plaintext[i])==len(ciphertexts[i]))
    i+=1
m10=b'The secret message is: When using a stream cipher, never use the key more than once'
key=repeat_key_xor(m10,ciphertexts[10])
#print(len(key)==len(m10))
print(key)
print(bytes([b1^b2 for b1,b2 in zip(m10,ciphertexts[10])]).hex())##key的十六进制表示
print(repeat_key_xor(ciphertexts[7],key))#第八组明文
# key1=strxor(ciphertexts[10],m10)
# print(f"key为:{key1}")
# print(bytes(key1),encoding='utf8')
# print(len(bytes(key1)))
# print(strxor(bytes(key1),ciphertexts[7]))
# print(len(strxor(bytes(key1),ciphertexts[7])))
```

## PA1 option

**题目：**
Write a program that allows you 'crack" ciphertexts generated using a Vigenere-like cipher, where byte-wise XOR is used instead of addition modulo 26.
**代码：**

```python
from string import ascii_letters, digits

# find_index_key: 确定key中index部分ascii码
# sub_arr: 当key_len和index确定时,使用key中同一位加密的子串
# 返回该子串可能使用的所有可见字符
def find_index_key(sub_arr):  # sub_arr是同一个ki的分组
    # all_ch = printable
    all_key = ascii_letters + digits + ',' + '.' + ' '
    test_key = []
    possible_key = []
    # 遍历整个ascii码(0-127)
    for x in range(0x00, 0xFF):
        test_key.append(x)
        possible_key.append(x)
    # 如果子串中所有位置都可以被ch异或为可见字符,则该ch可能为key的一部分
    for i in test_key:
        for j in sub_arr:
            if chr(i ^ j) not in all_key:
                possible_key.remove(i)
                break
    return possible_key

s = "F96DE8C227A259C87EE1DA2AED57C93FE5DA36ED4EC87EF2C63AAE5B9A7EFFD673BE4ACF7BE8923CAB1ECE7AF2DA3DA44FC
F7AE29235A24C963FF0DF3CA3599A70E5DA36BF1ECE77F8DC34BE129A6CF4D126BF5B9A7CFEDF3EB850D37CF0C63AA2509A76FF9
227A55B9A6FE3D720A850D97AB1DD35ED5FCE6BF0D138A84CC931B1F121B44ECE70F6C032BD56C33FF9D320ED5CDF7AFF9226BE5
BDE3FF7DD21ED56CF71F5C036A94D963FF8D473A351CE3FE5DA3CB84DDB71F5C17FED51DC3FE8D732BF4D963FF3C727ED4AC87
EF5DB27A451D47EFD9230BF47CA6BFEC12ABE4ADF72E29224A84CDF3FF5D720A459D47AF59232A35A9A7AE7D33FB85FCE7AF5923
AA31EDB3FF7D33ABF52C33FF0D673A551D93FFCD33DA35BC831B1F43CBF1EDF67F0DF23A15B963FE5DA36ED68D378F4DC36BF5B
9A7AFFD121B44ECE76FEDC73BE5DD27AFCD773BA5FC93FE5DA3CB859D26BB1C63CED5CDF3FE2D730B84CDF3FF7DD21ED5ADF7
CF0D636BE1EDB79E5D721ED57CE3FE6D320ED57D469F4DC27A85A963FF3C727ED49DF3FFFDD24ED55D470E69E73AC50DE3FE5DA
3ABE1EDF67F4C030A44DDF3FF5D73EA250C96BE3D327A84D963FE5DA32B91ED36BB1D132A31ED87AB1D021A255DF71B1C436BF47
9A7AF0C13AA14794"
ct = bytes.fromhex(s)#有很多不可打印的

# 遍历keylen和index的所有情况
for key_len in range(1, 30):
    for index in range(key_len):
        sub_arr = ct[index::key_len]  # 分组
        possible_ch = find_index_key(sub_arr)
        print('key_len = ', key_len, 'index = ', index, 'possible_ch = ', possible_ch)
        # 遍历所有可能的字符,解密ct中第一个长度为key_len的部分
        if possible_ch:
            k = []
            for j in possible_ch:
                k.append(chr(j ^ sub_arr[0]))
            print(k)

    # decryption
key = [186, 31, 145, 178, 83, 205, 62]
pt = ""
for i in range(len(ct)):
    pt += chr(ct[i] ^ key[i % 7])
print(pt)
```

## Breaking Repeat key XOR

Decrypt a file, which had been base64’d after being encrypted with repeating-key XOR.
思路:
Hamming distance(use XOR for each pair of bytes, count the number of 1 bits)
1.先求key的长度: 从2-42进行遍历，将密文(bytes形式)进行顺序分组，计算分别计算Normalized Hamming distance，选最小的结果就是key的长度
2.按照同一个密钥字符进行分组,每组就转化成了 single_key_XOR: 根据字母出现的频率得到分数最高的, 得到key
3.密文和key进行循环异或

代码：

```
import base64
## So we can use XOR for each pair of bytes, count the 1 bits and the result is the Hamming Distance.
def hamm (s1, s2):
    tot = 0
    for a,b in zip(s1,s2):
        tot += (bin(a^b).count('1'))
    #print('test Hamming distance is: ', tot)
    return(tot)
#print(hamm(b'this is a test',b'wokka wokka!!!'))
def find_key_len(c):
    aver_hamm=[]
    for keylen in range(2,41):
        #将密文分组
        tmp_aver_hamm=[]
        test1 = c
        while len(test1) >= (2 * keylen):
            x = test1[:keylen]
            y = test1[keylen:(2 * keylen)]
            # take the hamming distance and normalize by keysize
            score = hamm(x, y) / keylen
            test1 = test1[ keylen:]
            tmp_aver_hamm.append(score)
            res={
            'keylength': keylen,
            'avg distance': sum(tmp_aver_hamm)/len(tmp_aver_hamm)
            }
            aver_hamm.append(res)
            possible_key_length = sorted(aver_hamm, key=lambda x: x['avg distance'])[0]
    return possible_key_length['keylength']

##按照同一密钥进行分组c[::keylen]
##每个就转化成了 single_key_XOR: 根据字母出现的频率得到分数最高的
def get_english_score(input_bytes):

    # From https://en.wikipedia.org/wiki/Letter_frequency
    # with the exception of ' ', which I estimated.
    character_frequencies = {
    'a': .08167, 'b': .01492, 'c': .02782, 'd': .04253,
    'e': .12702, 'f': .02228, 'g': .02015, 'h': .06094,
    'i': .06094, 'j': .00153, 'k': .00772, 'l': .04025,
    'm': .02406, 'n': .06749, 'o': .07507, 'p': .01929,
    'q': .00095, 'r': .05987, 's': .06327, 't': .09056,
    'u': .02758, 'v': .00978, 'w': .02360, 'x': .00150,
    'y': .01974, 'z': .00074, ' ': .13000
    }
    return sum([character_frequencies.get(chr(byte), 0) for byte in input_bytes.lower()])

def repeat_key_xor(m,key):
    output_bytes=b''
    output_bytes=b''
    index=0
    for byte in m:#直接就是对应的ASCII
        #print(byte)
        #print(key[index])
        output_bytes+=bytes([byte^key[index]])
        #bytes[116,112]转为b'tp'
        if(index+1)==len(key):
            index=0
        else:
            index+=1
    return output_bytes

def single_char_xor(input_bytes, char_value):

    output_bytes = b''
    for byte in input_bytes:
        output_bytes += bytes([byte ^ char_value])
    return output_bytes

def bruteforce_single_char_xor(ciphertext):

    potential_messages = []
    for key_value in range(256):
        message = single_char_xor(ciphertext, key_value)
        score = get_english_score(message)
        data = {
        'message': message,
        'score': score,
        'key': key_value
        }
        potential_messages.append(data)
    return sorted(potential_messages, key=lambda x: x['score'], reverse=True)[0]['key']#最大的

cipher=open('ex1.4.txt').read().replace('\n','')
# convert to bytes
cipher=base64.b64decode(cipher)
keylen=find_key_len(cipher)
print(keylen)
#分组
key=[]
for index in range(keylen):
    sub_cipher=cipher[index::keylen]
    ## single_key_XOR:
    key.append(bruteforce_single_char_xor(sub_cipher))
print(''.join(chr(i) for i in key))
print(repeat_key_xor(cipher,bytes(''.join(chr(i) for i in key),encoding='utf8')))
```

## MTC3 Crack sha-1 Password

题目链接

```
import hashlib
import itertools
import datetime
starttime = datetime.datetime.now()
hash1="67ae1a64661ac8b4494666f58c4822408dd0a3e4"
str1="QqWw%58(=0Ii*+nN"
str2=[['Q', 'q'],[ 'W', 'w'],[ '%', '5'], ['8', '('],[ '=', '0'], ['I', 'i'], ['*', '+'], ['n', 'N']]
##str2是用来交换的，比如 Qxxx xxxx xxxx都不对，就换成q xxxxxxxxx 时间复杂度为0()由于需要在10s之内求解，
str4=""
str3=[0]*8
#首先要知道密码有多少位吧。。咋确定就是8位呢
for a in range(0,2):
    str3[0]=str2[0][a]
    for b in range(0,2):
        str3[1]=str2[1][b]
        for c in range(0,2):
            str3[2]=str2[2][c]
            for d in range(0,2):
                str3[3] = str2[3][d]
                for e in range(0,2):
                    str3[4] = str2[4][e]
                    for f in range(0,2):
                        str3[5] = str2[5][f]
                        for g in range(0,2):
                            str3[6] = str2[6][g]
                            for h in range(0,2):
                                str3[7] = str2[7][h]
                                newS="".join(str3)
                                for i in itertools.permutations(newS, 8):#返回可迭代对象的所有数学全排列方式。
                                    str4 =hashlib.sha1("".join(i).encode("utf-8")).hexdigest()
                                    if str4==hash1:
                                        print("".join(i))
                                        endtime = datetime.datetime.now()
                                        print(f"运行时间是：{(endtime - starttime).seconds}s")
```

# Ex2

## Padding Oracle Attack

### 题目

- 要求解密一个 CBC 工作模式，PKCS #7 方式填充，AES 加密的挑战密文。为此，你需要访问一个服务器（该服务器能用挑战密钥解密任何你发送的密文），它将返回你发送的密文是否填充正确。
- 该项目的第一步是将挑战密文发送到服务器，并验证您是否收到 “无错误” 消息。一旦你可以做到这一点，其余的是 “只是” 加密…
- 明文转换为 ASCII 时，是可读的英文文本，所以一旦成功恢复明文，便可得知。

### 题解

学长博客

```python
from pwn import *

c = ['9F0B13944841A832B2421B9EAF6D9836',
    '813EC9D944A5C8347A7CA69AA34D8DC0',
    'DF70E343C4000A2AE35874CE75E64C31']
r = remote('128.8.130.16', 49101)


# 发送并判断返回是否正确
def send_payload(build):
   msg = bytes.fromhex(build)
   lst = list(msg)
   lst.insert(0, 2)
   lst.append(0)
   r.send(bytes(lst))
   return r.recv(numb=2).decode()[0] == '1'


# 从最后一块开始破译 c[-x]
for x in range(1, len(c)):
   print("Detecting Block {}".format(x))
   block = c[-x]  # 当前分组
   prior_block = [int(i) for i in bytes.fromhex(c[-x - 1])]  # 前一分组

   iv = [0] * 16  # 初始向量
   lvalue = []  # 中间值
   m = []  # 明文(hex)

   # 一个分组16B
   for i in range(1, 17):
      print(" round {}".format(i))

      # 修正
      l = len(lvalue)
      iv = iv[:16 - l] + [x ^ i for x in lvalue[::-1]]

      # 爆破vi[-i]
      for j in range(1, pow(2, 8)):
         iv[-i] = j
         build = ''.join([str(hex(i)[2:].zfill(2)) for i in iv]) + block
         if send_payload(build):
            break

      # 更新
      lvalue.append(iv[-i] ^ i)
      m.append(lvalue[-1] ^ prior_block[-i])
      print(" - IV     : {}".format(''.join([str(hex(i)[2:].zfill(2)) for i in iv])))
      print(" - lvalue : {}".format(''.join([str(hex(i)[2:].zfill(2)) for i in lvalue[::-1]])))
      print(" - m      : {}".format(''.join([str(hex(i)[2:].zfill(2)) for i in m[::-1]])))

   # print the message
   print(''.join(chr(i) for i in m[::-1]))
```

## Byte-at-a-time ECB decryption (Simple)

确定块长block_size
逐位增加可管理字符串的len，当可管理字符串的len和unkown encrypted的len发生变化时，即表示block的个数变化了，那么block的大小就是变化后的len减去之前的len

知道block_size后如何获取每一位：
使字符串的 len 从 blocksize-1,0 变化；通过对比 e1: managabled string+plain+unkown[:blocksize] 和 e2:managabled string+plain+ one bit guessed[：blocksize]；当e1==e2，则可管理字符串减一位，plain+=the bit

```
from base64 import b64decode
from Crypto import Random
from Crypto.Cipher import AES

UNKNOWN_STRING = b"""
Um9sbGluJyBpbiBteSA1LjAKV2l0aCBteSByYWctdG9wIGRvd24gc28gbXkg
aGFpciBjYW4gYmxvdwpUaGUgZ2lybGllcyBvbiBzdGFuZGJ5IHdhdmluZyBq
dXN0IHRvIHNheSBoaQpEaWQgeW91IHN0b3A/IE5vLCBJIGp1c3QgZHJvdmUg
YnkK"""

KEY = Random.new().read(16)

def pad(your_string, msg):

    paddedMsg = your_string + msg

    size = 16
    length = len(paddedMsg)
    if length % size == 0:
        return paddedMsg

    padding = size - (length % size)
    padValue = bytes([padding])
    paddedMsg += padValue * padding

    return paddedMsg


def encryption_oracle(your_string):

  # msg = bytes('The unknown string given to you was:\n', 'ascii')
    # append the `UNKNOWN_STRING` given to us to the `msg`
    #plaintext = msg + b64decode(UNKNOWN_STRING)
    plaintext = b64decode(UNKNOWN_STRING)
    # add `your_string` to prepend to `plaintext` and apply `PKCS#7` padding to correct size
    paddedPlaintext = pad(your_string, plaintext)

    cipher = AES.new(KEY, AES.MODE_ECB)
    ciphertext = cipher.encrypt(paddedPlaintext)

    return ciphertext

"""using the managable string before the unkown
increase the len of managable string bit by bit,when the len of managable string and unkown encrypted changes,that means the number of blo
ck changes,then the blocking size is the changed len substract the before len;
"""
def detect_block_size():

    feed = b"A"
    length = 0
```

```python
    length = 0
    while True:
        cipher = encryption_oracle(feed)
        # on every iteration, add one more character
        feed += feed
        # if the length of the ciphertext increases by more than 1,feed += feed
        # PKCS#7 padding must have been added to make the size of plaintext == block_size
        # increase in the size gives the value of block_size
        if not length == 0 and len(cipher) - length > 1:
            return len(cipher) - length
        length = len(cipher)


def detect_mode(cipher):

    chunkSize = 16
    chunks = []
    for i in range(0, len(cipher), chunkSize):
        chunks.append(cipher[i:i+chunkSize])

    uniqueChunks = set(chunks)
    if len(chunks) > len(uniqueChunks):
        return "ECB"
    return "not ECB"


def ecb_decrypt(block_size):

    # common = lower_cases + upper_cases + space + numbers
    # to optimize brute-force approach
    common = list(range(ord('a'), ord('z'))) + list(range(ord('A'),
                                                        ord('Z'))) + [ord(' ')] + list(range(ord('0'), ord('9')))
    rare = [i for i in range(256) if i not in common]
    possibilities = bytes(common + rare)

    plaintext = b''  # holds the entire plaintext = sum of `found_block`'s
    check_length = block_size

    while True:
        # as more characters in the block are found, the number of A's to prepend decreases
        prepend = b'A' * (block_size - 1 - (len(plaintext) % block_size))
        actual = encryption_oracle(prepend)[:check_length]
"""how to get the every bit after knowing the blocking size: the len of managabled string changes from blocksize-1,0;by constrasting the result
of e1:managabled string+plain+unkown[:blocksize] and e2: managabled string+plain+one bit guessed[:blocksize];when e1==e2,then managabl
ed string substrcats,plain+=one bit
"""
        found = False
        for byte in possibilities:
            value = bytes([byte])
            your_string = prepend + plaintext + value
            produced = encryption_oracle(your_string)[:check_length]
            if actual == produced:
                plaintext += value
                found = True
                break

        if not found:
            print(f'Possible end of plaintext: No matches found.')
            print(f"Plaintext: \n{ plaintext.decode('ascii') }")
            return
    if len(plaintext) % block_size == 0:
        check_length += block_size


def main():
    # detect block size
    block_size = detect_block_size()
    print(f"Block Size is { block_size }")

    # detect the mode (should be ECB)
    repeated_plaintext = b"A" * 50
    cipher = encryption_oracle(repeated_plaintext)
    mode = detect_mode(cipher)
    print(f"Mode of encryption is { mode }")

    # decrypt the plaintext inside `encryption_oracle()`
    ecb_decrypt(block_size)


if __name__ == "__main__":
    main()
```

## Byte-at-a-time ECB decryption (Harder)

### 题目要求

在加密前，在明文后面添加一段文本（添加之前需要对该文本进行Base64解码）。文本如下

```
Um9sbGluJyBpbiBteSA1LjAKV2l0aCBteSByYWctdG9wIGRvd24gc28gbXkg
aGFpciBjYW4gYmxvdwpUaGUgZ2lybGllcyBvbiBzdGFuZGJ5IHdhdmluZyBq
dXN0IHRvIHNheSBoaQpEaWQgeW91IHN0b3A/IE5vLCBJIGp1c3QgZHJvdmUg
YnkK
```

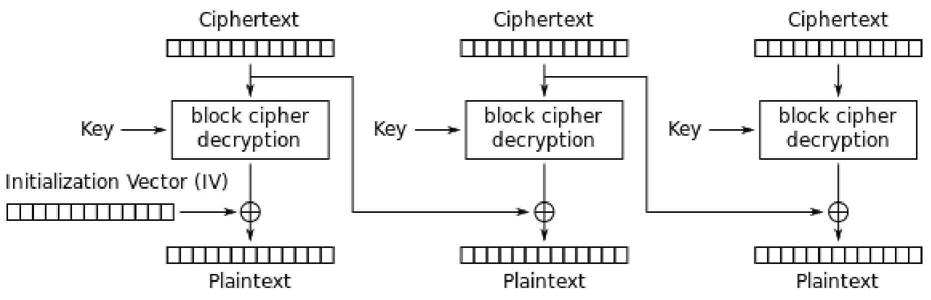
使用一个恒定未知的密钥，通过ECB模式加密。加密函数格式为：

`AES-128-ECB(your-string || unknown-string, random-key)`

现在，已知加密函数的接口encrypt(string)，需要解密"unknown-string"。

### 题解

CBC解密：



Cipher Block Chaining (CBC) mode decryption

确定 random_bytes 占用的完整块数。
我们知道每个块在 AES_ECB 中是独立加密的，并且我们知道 random_bytes 是恒定的。这意味着如果random_bytes 占据了两个完整的块，那么无论我们向 oracle() 发送什么作为输入参数，这些块将始终被加密以生成相同的密文块。
即使更改 oracle() 函数的输入也不会改变的密码块数为我们提供了 random_bytes 占用的完整块数。

确定 random_bytes 中剩余的字节数。
剩余的字节，是random_bytes 末尾的那些字节，它们不占用一个完整的块。例如，如果 AES_ECB 使用的块大小为 4，random_bytes 的大小为 10 $(2\times 4+2)$，则 random_bytes 占用 2 个块，剩余的 2 个字节为余数。

为了确定余数，我们首先找到需要添加到密文中的字节数，直到一个新的块（块比完全被 random_bytes 占用的块数大1；在这种情况下，它是第三个块）不改变。当从块大小中减去这个数字时，我们就得到了残差的值。如下所示：

```
let random_bytes = b'abcdefghij' // 10 bytes
let block_size = 4
// now let's separate random_bytes into blocks of block_size
abcd efgh ij
// this is prepended to a plaintext
let plaintext = b'yellow'
// then the following blocks are formed
**abcd efgh ijye llow**
// now we can inject attacker bytes between random_bytes and plaintext
// let's start with an 'a'
**abcd efgh ijay ello w...**
// the cipher is the same for the first two blocks, and different for the next 3
// one more 'a'
**abcd efgh ijaa yell ow..**
// the cipher is again the same for the first two blocks, and different for the next 3
// one more 'a'
**abcd efgh ijaa ayel low.**
// this time even the THIRD BLOCK is the same as before (in the cipher)
// so we needed to add 3 a's so that a new block does not change
// this means two a's were necessary to form a complete block
// this means that there were (4-2)=2 characters of `random_bytes` in the third block
```

```python
def detect_prefix_length():

    block_size = detect_block_size()

    # first find number of integer blocks occupied
    test_case_1 = encryption_oracle(b'a')#prefix+your string + unknown
    test_case_2 = encryption_oracle(b'b')

    length1 = len(test_case_1)
    length2 = len(test_case_2)

    blocks = 0
    min_length = min(length1, length2)#我咋觉得他两个是一样的？？
    # if the any of the blocks (starting from the left) are the same,
    # these blocks are occupied by the `PREFIX`
    for i in range(0, min_length, block_size):
        if test_case_1[i:i+block_size] != test_case_2[i:i+block_size]:
            break
        blocks += 1 # the number of the same blocks of cipher

    # now calculate the residual number of bytes and add to total size
    test_input = b''
    length = blocks * block_size
    # if adding an extra `?` does not change the current block of cipher-text
    # we've reached the end of that block, and so,
    # we've found the number of extra characters needed to complete the block with some prefix characters
    for extra in range(block_size):
        test_input += b'?'
        curr = encryption_oracle(test_input)[length: length+block_size]
        next = encryption_oracle(test_input + b'?')[length: length+block_size]
        if curr == next:
            break

    residue = block_size - len(test_input)
    length += residue
    return length
```

```
def ecb_decrypt(block_size):

    # common = lower_cases + upper_cases + space + numbers
    # to optimize brute-force approach
    common = list(range(ord('a'), ord('z'))) + list(range(ord('A'),
                                                ord('Z'))) + [ord(' ')] + list(range(ord('0'), ord('9')))
    rare = [i for i in range(256) if i not in common]
    possibilities = bytes(common + rare)

    plaintext = b''  # holds the entire plaintext = sum of `found_block`'s
    check_length = block_size

    prefix_len = detect_prefix_length()
    print(f"Calculated Length of Prefix = { prefix_len }")
    check_begin = (prefix_len // block_size) * block_size
    #check_begin = (prefix_len // block_size+1) * block_size
    residue = prefix_len % block_size

    while True:
        # as more characters in the block are found, the number of A's to prepend decreases
        #padding1 = b'A'*(block_size-residue)
        # prepend = b'A' * (block_size - 1 - (len(plaintext) % block_size))
        # prepend = padding1+prepend
        prepend = b'A' * (block_size - 1 -
                          ((len(plaintext)+residue) % block_size))
        actual = encryption_oracle(
            prepend)[check_begin: check_begin+check_length]

        found = False
        for byte in possibilities:
            value = bytes([byte])
            your_string = prepend + plaintext + value
            produced = encryption_oracle(your_string)[
                check_begin: check_begin+check_length]
            if actual == produced:
                plaintext += value
                found = True
                break

        if not found:
            print(f'Possible end of plaintext: No matches found.')
            print(f"Plaintext: \n{ plaintext.decode('ascii') }")
            return

        if (len(plaintext) + residue) % block_size == 0:
            check_length += block_size
```

## 判断pkcs#7是否正确

### 题目要求

检查一段文本，是否为有效的PKCS#7填充，如果是，则去掉填充。

示例：

```
string1="ICE ICE BABY\x04\x04\x04\x04" 为有效填充，结果为"ICE ICE BABY"。
string2="ICE ICE BABY\x05\x05\x05\x05" 不是有效填充。
string3="ICE ICE BABY\x01\x02\x03\x04" 不是有效填充。
```

### 实现方法

1. 提取字符串最后一位last_byte，转化为10进制数，即为填充长度paddingCount。
2. 比较字符串的倒数paddingCount的位置上的字符到最后一位是否等于last_byte，即 `paddedMsg[-last_byte:] == bytes([last_byte]) * last_byte`，如果相等，则为有效填充。

```python
def valid_padding(paddedMsg, block_size):

    # if the length of the `paddedMsg` is not a multiple of `block_size`
    if len(paddedMsg) % block_size != 0:
        return False

    last_byte = paddedMsg[-1]

    # if the value of the last_byte is greater than or equal to block_size
    if last_byte >= block_size:
        return False

    padValue = bytes([last_byte]) * last_byte
    # if all the padding bytes are not the same
    if paddedMsg[-last_byte:] != padValue:
        return False

    # if, after removing the padding, the remaining characters are not all printable
    if not paddedMsg[:-last_byte].decode('ascii').isprintable():
        return False

    return True


def PKCS7_restore(m):
    return m[:-m[-1]]


def test(m, size):
    try:
        if not valid_padding(m, size):
            raise ValueError
    except ValueError:
        print(f"{m} has invaild PKCS#7 padding.")
        return

    print(f"Padding successfully...")
    print(f"Before padding removal: { m }")
    print(f"After padding removal: { PKCS7_restore(m) }")
```

## CBC bitflipping attacks

### 题目要求

首先生成一个随机AES密钥，然后实现两个功能。

功能1：对于userdata，在前面添加`“comment1=cooking%20MCs;userdata=”`，在字符串后面添加`”;comment2=%20like%20a%20pound%20of%20bacon”`，然后对该文本进行填充和加密，返回加密后的结果。对于userdata的内容，不允许存在”;“和”=“。

功能2：解密字符串，语法分析查找是否存在`”admin=true;“`，返回True或False。

如果函数1实现正确，那么不会出现函数2中查找的字符串。

## 题解

现在，需要做的是修改密文，基于CBC模式的比特翻转攻击，来使其可以找到

**一个例子：**

> Given ciphertext = ax5A dg74 Actual plaintext = hell oeen Target plaintext = hell oben
> We need ‘b’ in position 2 of block 2 where there was previously ‘e’.
> Then, we would simply change the 2nd position of block 1 from ‘x’ to:
>
> > target xor original_plain xor original_cipher
> > => b’b’ xor b’e’ xor b’x’

```
from Crypto.Cipher import AES
from Crypto import Random
import re
prepend = b"comment1=cooking%20MCs;userdata="
append = b";comment2=%20like%20a%20pound%20of%20bacon"  # len==42


def pad(value, size):
    if len(value) % size == 0:
        return value
    padding = size - len(value) % size
    padValue = bytes([padding]) * padding
    return value + padValue


class InvalidPaddingError(Exception):

    def __init__(self, paddedMsg, message="has invalid PKCS#7 padding."):
        self.paddedMsg = paddedMsg
        self.message = message
        super().__init__(self.message)

    def __repr__(self):
        return f"{ self.paddedMsg } { self.message }"


def valid_padding(paddedMsg, block_size):
    # if the length of the `paddedMsg` is not a multiple of `block_size`
    if len(paddedMsg) % block_size != 0:
        return False

    last_byte = paddedMsg[-1]

    # if the value of the last_byte is greater than or equal to block_size
```

```python
    if last_byte >= block_size:
        return False

    padValue = bytes([last_byte]) * last_byte
    # if all the padding bytes are not the same
    if paddedMsg[-last_byte:] != padValue:
        return False

    # if, after removing the padding, the remaining characters are not all printable
    if not paddedMsg[:-last_byte].decode('ascii').isprintable():
        return False

    return True


def remove_padding(paddedMsg, block_size):
    if not valid_padding(paddedMsg, block_size):
        raise InvalidPaddingError

    last_byte = paddedMsg[-1]
    unpadded = paddedMsg[:-last_byte]
    return unpadded


# this is the dictionary for replacements
QUOTE = {b';': b'%3B', b'=': b'%3D'}

KEY = Random.new().read(AES.block_size)
IV = bytes(AES.block_size)  # for simplicity just a bunch of 0's


def cbc_encrypt(input_text):
    for key in QUOTE:
        input_text = re.sub(key, QUOTE[key], input_text)

    plaintext = prepend + input_text + append
    plaintext = pad(plaintext, AES.block_size)

    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    ciphertext = cipher.encrypt(plaintext)

    return ciphertext


def check(ciphertext):

    cipher = AES.new(KEY, AES.MODE_CBC, IV)
    plaintext = cipher.decrypt(ciphertext)
    print(f"Plaintext: { plaintext }")

    if b";admin=true;" in plaintext:
        return True

    return False
```

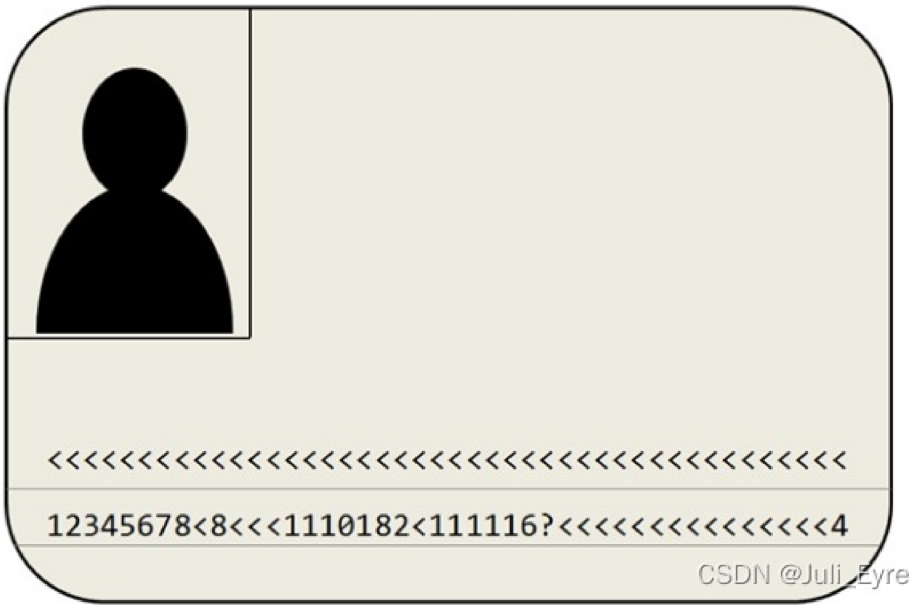
代码的关键部分：

```python
def test():

  # send two blocks of just A's
  input_string = b'A' * AES.block_size * 2
  print(AES.block_size)  # 16
  ciphertext = cbc_encrypt(input_string)
  print(len(ciphertext))  # 112
  # replace first block of A's with the `required` plain-text
  required = pad(b";admin=true;", AES.block_size)
  # xor each byte of the required with each byte of second block i.e, with 'A'
  inject = bytes([r ^ ord('A') for r in required])  # one block of input
  print(len(inject))  # 16
  # extra = length of ciphertext - length of injected text - length of prefix
  # = one block of input + suffix
  extra = len(ciphertext) - len(inject) - len(prepend)
  # print(extra)
  # keep `inject` fill either side with 0's to match length with original ciphertext
  # xor with 0 does not change value
  # this replaces the first block of input with `required` while the rest is unchanged
  # 0*len(prefix)+infect(A^AES(A)^target)+0*len(suffix+padding)
  inject = bytes(2 * AES.block_size) + inject + bytes(extra)

  # to craft cipher-text, xor the `inject` bytes with
  # corresponding byte of the ciphertext
  crafted = bytes([x ^ y for x, y in zip(ciphertext, inject)])

  if check(crafted):
    print("Admin Found")
  else:
    print("Admin Not Found")
if __name__ == "__main__":
  test()
```

## MTC3 AES KEY欧洲护照

### 题目分析

1. AES加密模式为CBC，初始化矢量即IV为零，填充为01-00。此外，相应的密钥在身份证件上的机器可读区域（MRZ）等表格中，它与欧洲的电子护照一起使用时并不十分完整。



2. 目标是找到以下base64编码消息的明文：

```
9MgYwmuPrjiecPMx61O6zIuy3MtIXQQ0E59T3xB6u0Gyf1gYs2i3K9Jxaa0zj4gTMazJuApwd6+jdyeI5iGHvhQyDHGVlAuYTgJrbFDrfB22Fpil2NfNnWFBTXyf7SDI
```

    对于加密，已生成并应用基于基本访问控制（BAC）协议的密钥KENC。对于解密，已经发送了以下字符，从中可以导出KENC（这些字符的编码类型在[1]中描述）：`12345678 <8 <<< 1110182 <111116？<<<<<<<<<<<<<<< 4`

    在传输过程中丢失了并且突出显示了一个"？"。可以在[2]的帮助下恢复它。为了能够在之后计算密钥KENC，可以找到应用编码的概述[3]，[4]中的协议和[5]中的一个例子。

    在解密之前解码base64代码。

总而言之，就是根据它给的那一串数字，然后按照它参考文章里面用到的方法去还原加密的明文。

## 解题思路

1. 根据计算规则得到缺失的一位校验码
2. 根据规则得到机读区的信息
3. 对机读区信息进行sha1加密，在连接'00000001'，对连续返回的字符串进行sha1,把哈希的结果的0-15和16-32位进行奇偶检验调整，把结果连接作为密钥
4. 用AEC-CBC解密，IV为'0'*32

```python
from Crypto.Cipher import AES
import binascii
import hashlib
import base64
import codecs


def get_unkown_digit():
    a = [1, 1, 1, 1, 1, 6]
    b = [7, 3, 1, 7, 3, 1]
    c = 0
    for i in range(6):
        c += a[i] * b[i]
        res = c % 10
    return res


# 奇偶校验位的判断
def jiaoyan(x):
    k = []
    a = bin(int(x, 16))[2:]
    for i in range(0, len(a), 8):
        if (a[i:i + 7].count("1")) % 2 == 0:
            k.append(a[i:i + 7])
            k.append('1')
        else:
            k.append(a[i:i + 7])
            k.append('0')
    a1 = hex(int(''.join(k), 2))
    # print("this is " + x + "---" +a1)
    return a1[2:]


s = '12345678811101821111674'
K_seed = hashlib.new("sha1", s.encode("utf8")).hexdigest()[:32]

c = '00000001'
# 0x00000001
d = K_seed + c
print(d)
H_d = hashlib.sha1(codecs.decode(d, "hex")).hexdigest()
# 十六进制先变为二进制散列再换成十六进制
# print(H_d)
ka = hashlib.sha1(codecs.decode(d, "hex")).hexdigest()[:16]
kb = hashlib.sha1(codecs.decode(d, "hex")).hexdigest()[16:32]

k_1 = jiaoyan(ka)
k_2 = jiaoyan(kb)
key = k_1 + k_2
print(key)
# ea8645d97ff725a898942aa280c43179
IV = bytes(AES.block_size)
print(IV)
cipher = '9MgYwmuPrjiecPMx61O6zluy3MtIXQQ0E59T3xB6u0Gyf1gYs2i3K9Jxaa0zj4gTMazJuApwd6+jdyeI5iGHvhQyDHGVIAuYTgJrbFDrfB22F
piI2NfNnWFBTXyf7SDI'
cipher = base64.b64decode(cipher)
print(cipher)
# returns the binary string that is represented by any hexadecimal string
m = AES.new(binascii.unhexlify(key), AES.MODE_CBC, IV)
# m = AES.new(key.encode(), AES.MODE_CBC, IV.encode())
print(m.decrypt(cipher))
# b'Herzlichen Glueckwunsch. Sie haben die Nuss geknackt. Das Codewort lautet: Kryptographie!\x01\x00\x00\x00\x00\x00\x00'
```

## 实验心得

看懂题目很重要，要善于快速查找资料

知道攻击原理，数学知识很重要

要加强自己的编程能力

直接使用 Python 的内置库会方便很多