HGAME2020 Week3 Writeup

签到成功
只求week4不要交白卷

Crypto - Exchange

题目：

```
Our admin hijacked a secret channel and it looks like there are two teams doing
some unspeakable transactions.
nc 47.98.192.231 25258
```

一开始先是考了个week2的Crypto签到题，直接用上次的解题脚本即可
之后的题目是“我”截获了Alice和Bob的通信，并且“我”可以篡改当中的部分内容
然而双方使用了非对称加密，使得“我”无法直接获得解密用的密钥

一段正常的、消息未被篡改的对话记录如下：

```
root@tesla:~# nc 47.98.192.231 25258
sha256(XXXX+xAkaaFN1enC5UQm1) ==
a196d23933aaa95a960505f51b0bbb80e4d5e171c650ee77e5f690db0da49f9d
Give me XXXX: hhA9
Bob: Hi Alice, I got the second half of the flag.
Alice: Really? I happen to have the first half of the flag.
Bob: So let's exchange flags, :-)
Alice: Ok,
Alice: Ah, wait, maybe this channel is not secure,
Alice: Let's do Key Exchange first.

[INFO] : ********** STEP.1 **********
[INFO] : Alice and Bob publicly agree to use a modulus `p` and base `g`

Alice: p =
153556892283026381077940804008394861131515741989658821679175416849698517587575938981504454432773936568307359774473745489589317350169237690857181940569391633955685290014611082866737102235415668265604660796650340393980508711731525896209747614702453521502602117643570282460093254535065811953844942144919365256521
Alice: g =
125332266302111232674806346981076365176457172060484151514673649592576743798883390018905618044338490509191458491310532639726955834187883361816866315049270230

[INFO] : ********** STEP.2 **********
[INFO] : Alice generates her private key `a` which is a random number from the
set {2, 3, ... p-2}.
[INFO] : Bob does exactly the same: he compute his private key `b` which is
taken randomly from the same set of integers.
[INFO] : Now, Alice computes her public key `A` which is simply `g` raises to
the `a`s power modulo `p`, a.k.a, `A = pow(g, a, p)`.
[INFO] : Bob does the same: Bob computes the `B = pow(g, b, p)`.

[INFO] : ********** STEP.3 **********
```

```
[INFO] : Alice and Bob exchange their public key.
[WARNING] : You intercepted Alice's message, which is
[WARNING] : A =
706475118629741676057181269330195341998995537813607425265729582411293548387589806502690800311713864133831583646134847405125292939942312018679564595619043549978815613010169792507176408618268005009322311977829187637906080929520369355883864520988202126165325429444669481294520670495142434741504236558833592226 25
[WARNING] : Do you want to modify this message? (yes/no)
> no

Alice: A =
706475118629741676057181269330195341998995537813607425265729582411293548387589806502690800311713864133831583646134847405125292939942312018679564595619043549978815613010169792507176408618268005009322311977829187637906080929520369355883864520988202126165325429444669481294520670495142434741504236558833592226 25

[WARNING] : Again, you intercepted Bob's message, which is
[WARNING] : B =
550852498751127238569250711436854048550373572026931252676279478727174791502269736445821453738655545320063546834115711412656128899884769793550447902545540686005960026936050360527639154181219515829328090119252237384519525241558090345387138800597783821696321716630729471104507995629617093247887959541907697152 33
[WARNING] : Do you want to modify this message? (yes/no)
> no

Bob: B =
550852498751127238569250711436854048550373572026931252676279478727174791502269736445821453738655545320063546834115711412656128899884769793550447902545540686005960026936050360527639154181219515829328090119252237384519525241558090345387138800597783821696321716630729471104507995629617093247887959541907697152 33

[INFO] : ********** STEP.4 **********
[INFO] : Alice takes public key of Bob (`B`), raises to, again, to the `a`s power modulo `p`, the she gets the secret value `S_a = pow(B, a, p)`.
[INFO] : Bob also gets the secret value `S_b = pow(A, b, p)`.
[INFO] : Right now, they have done the key exchange.
[WARNING] : hint: does Alice and Bob share the same key?

Alice: Now, we have securely shared the secret value.
Bob: Right, let's exchange the encrypted flag!

[INFO] : For the encryption, Alice compute `C_a = (m * S_a) % p`
[INFO] : Bob does the same, `C_b = (m * S_b) % p`

[WARNING] : Bob is trying to send a message to Alice,
[WARNING] : C_b =
110913341603720164796891815210574224350021153266903320843223068059001121967570733691796324525020947859066394223508548881918150742163684558186913833640618265871415287259294426888410901531102711227891803726165091368041974003571462338237395576033657204061415069107115364368082057002132107630185756527268217377247
[WARNING] : Do you want to modify this message? (yes/no)
> no
Alice: Great, I get the flag.
Alice: C_a =
124986611840107028357373332961427876017685009748795981788067670242988099919086562972017332366455079294211132131812413022628117952556356716858374892804876191509540189241840538243159537688361879032001676734836917946408430037028348731342744428672358594885114018261862941151688600653201876316592113600143479932888
```

```
Happy cooperation. :)
```

一开始先是根据p、g两个名字去找相关的加密算法，在StackOverflow上面找到一个问答，似乎与SSH2有关。之后搜索发现找不到相关加密算法。。。后来又去CTF Wiki上面翻RSA相关内容（目前就只知道这个了），发现还是没有思路
之后由于整个week3都没啥会做的，而这道题在Crypto当中完成人数最多，决定还是来啃一下

因为找不到类似的加密算法，而且最后求密文的时候用的是乘法（似乎在现代加密算法当中不太常见），因此猜测可能是要从加密过程本身来进行分析
分析两人在对话过程中所获得的信息如下：

```
A:p,g
B:p,g
A:p,g,a,A
B:p,g,b,B
A:p,g,a,A,B
B:p,g,b,B,A
A:p,g,a,A,B,S_a
B:p,g,b,B,A,S_b
A:p,g,a,A,B,S_a,C_b
B:p,g,b,B,A,S_b,C_a
```

整个过程中a,b是私钥，A,B是传给对方的公钥，A,B应该是用于解密的（题目中没给具体解密算法所以未知）

尝试分析一下密文，一个非常不严谨的分析过程如下（还没开始学数论）：

```
Ca=m%p*Sa%p
    =m%p*Sa
    =m%p*B^a%p
    =m*B^a%p
    =m*g^b%p^a%p
    =m*g^b^a%p
    =m*g^a^b%p
    =m*g^a%p^b%p
    =m*A^b%p
    =m%p*A^b%p
    =m%p*Sb
```

意外发现S_a与S_b好像是相同的
怀疑自己推导过程中出了问题，但看到聊天记录中有"hint: does Alice and Bob share the same key?"，考虑到两人公钥私钥不同，所以题目应该是在说Sa和Sb相同
之后又卡壳了几天

后来冷静分析一波，这个加密的安全性就体现在a与b不需要发送给对方（也就是无法被截获），而目前似乎没有方法用其它信息推出a与b。加密密钥S受a与b共同影响，因此解密所需的S ^ -1也变成了未知状态。一种思路是让S变为定值或者用其它信息推出S，那么密文就可以解出来了
因为S与a,b两个量有关，结合这几周Crypto题当中多次出现的逆元，很自然就想到了构造公钥让S当中的a、b互相抵消。经过各种尝试（因为不会证明），存在以下破解流程：

```
A = g^a%p（不变）
B = A^-1（篡改）
S_a = B^a%p = g
```

这样S就变成了g，就可以解出m了

然而具体实施的过程并没有这么顺利。由于两人特定的通信顺序，出现了两种情况：
1.篡改B是为了使S_a=g，进而通过C_a解出m。而如果B被篡改，则Alice无法用错误的B解出正确的flag，因此不会提供C_a
2.截获公钥的顺序永远是先A后B，因此无法篡改A使得A=B ^ -1
破解失败

强烈怀疑两人之间ABBA的通信顺序就是用来堵这种方法的。不过这也提供了破解的方向：只能通过篡改A密钥来破解C_b中的内容

之后想到了一个更简单的方法，对于指数运算，1也是一个很好的工具。令A=1，则S_b=1，那么C_b就可以破解了。
兴高采烈地把A改为1，结果如下：

```
[WARNING] : Select a decimal number
> 1
Invalid A
Rua!!!
```

如此简单的方法被ban也是情理之中
接着尝试令A=2，些许能从尾数当中看出点东西。然而经过m * S_b的运算之后也看不出什么了

因为之前构造逆元的时候是从g出发，于是这次准备换个角度，从p出发找找思路。对p进行简单拼凑之后突然回想起二项式定理，(p+1) ^ a展开后的a+1项中前a项都含有因式p，而最后一项为1 ^ a=1，也就是有(p+1) ^ a = kp+1，因此((p+1) ^ a)%p = 1。通过令A=p+1也可以使得S_b变为1。
兴高采烈地根据p把A改为p+1，结果如下：

```
Invalid A
Rua!!!
```

接着又尝试把A改为2p,(p ^ 2)+1和2p+1，都失败了
因为摸不透出题人对于合法A的验证方式，于是又卡壳了几天

后来想到计算A的时候也涉及到模p的操作，因此一个正常的A一定小于p，而前几次用二项式定理构造A的时候都不满足这一条件。为了使得A < p，可以构造A=p-1使得S_b变为1或-1（取决于b是否为偶数），这样也满足让S变为定值的攻击条件。

令A=p-1，直接得到以下C_b

```
[WARNING] : Bob is trying to send a message to Alice,
[WARNING] : C_b = 43286382716192805950852726972475444929294853598910542O5
```

可以发现这次得到的C_b长度明显小于正常C_b的长度。而且经过多次尝试，C_b较短时永远是43286382716192805950852726972475444929294853598910542O5（b为偶数），而当C_b较长时（b为奇数），将得到的C_b乘上-1关于p的逆元（用gmpy2.invert()求）并模p，也可以得到43286382716192805950852726972475444929294853598910542O5，说明刚才的思路应该是正确的

直接在python中运行

```
print((43286382716192805950852726972475444929294853598910542O5).to_bytes(50,'big
'))
```

得到结果

```
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00-1n~ThE+miDd!3_4TtAck~}'
```

那么flag的后半段就是

```
-1n~ThE+miDd!3_4TtAck~}
```

~~由于flag里面是有意义的内容，接下来暴力猜测flag前半段即可~~

要得到flag的前半段就要对C_a进行处理，而要处理C_a就要首先获得C_a。经过测试，Alice只有在成功拿到flag后半段之后才会把C_a发出来，这就要求Alice能通过手上的B解出C_b
如果使用刚才的方法，修改B使得S_a已知进而解出C_a，那么Alice通过错误的B和正确的C_b无法解出flag后半段，就不会提供C_a了
如果修改A，那么Alice通过正确的B和错误的C_b也无法解出flag后半段，并且这种构造方法无法影响S_a、C_a，对解题没有帮助
又一次陷入了僵局。。。。

继续分析，发现直接修改A/B无法成功的原因在于C_b和B无法对应起来。题目刚好允许篡改C_b，并且结合之前的思路，应该令B=p-1，同时修改C_b使得Alice可以通过两者解出flag后半段，这样她就会提供C_a，即可用先前的方式解出flag前半段
因为已知flag后半段，而Alice解密的时候会用C_b去乘S_a关于p的逆元，而S_a = (B ^ a)%p=±1，因此当a为偶数的时候直接将C_b改为
43286382716192805950852726972475444929294853598910542O5（flag后半段对应数字）即可。
因为每一次对话过程中a的奇偶性不确定，所以多试几次，当a刚好为偶数的时候就能获得C_a，而且此时C_a就是flag前半段对应的数字

测试结果如下：

```
[WARNING] : Bob is trying to send a message to Alice,
[WARNING] : C_b =
40484057514218572415601529748988352447573488701051111978715974421109661614203811
62855341790057073468939451451896012635245924826239866348903892862629943901509961
66838199062375756720794071153656484930332844221913915218493934018900027690188041
86035186954501411955032326315075837266787650114023412030957491696992
[WARNING] : Do you want to modify this message? (yes/no)
> yes
[WARNING] : Select a decimal number
> 43286382716192805950852726972475444929294853598910542O5
Alice: Great, I get the flag.
Alice: C_a = 999990028100335377380800951730725431764016517334573O638

[INFO] : Alice is offline

Bob: Damn it! She lied on me...
```

直接在python中运行

```
print((999990028100335377380800951730725431764016517334573O638).to_bytes(50,'big'))
```

得到结果

```
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00hgame{Wow!+U_d0_tH3_m@N'
```

那么flag的前半段就是

```
hgame{Wow!+U_d0_tH3_m@N
```

最终flag：hgame{Wow!+U_d0_tH3_m@N-1n~ThE+miDd!3_4TtAck~}

后记：
似乎只需让A=B=p-1，C_b=432863827161928059508527269724754449292948535989105420 5，那么当a、b均为偶数时就可以让Bob和我都拿到flag前半段了
不过人太懒没有去试
心疼Bob一秒
因为上周写的脚本是半自动的，所以每次验证思路都很麻烦。通过求助出题人知道了telnetlib和socket结合使用的方法，写了个脚本，代码如下：

```
import string, sys, re
from socket import socket
from telnetlib import Telnet
from hashlib import sha256
from multiprocessing import Pool, Pipe
from time import sleep

table = (string.ascii_letters + string.digits).encode()
prefix = [string.ascii_lowercase.encode()[0:13],string.ascii_lowercase.encode()
[13:26],string.ascii_uppercase.encode()[0:13],   string.ascii_uppercase.encode()
[13:26],string.digits.encode()]

def task(pipe,index, c, part):
    print('task',index,"is running")
    for i in prefix[index]:
        for j in table:
            for k in table:
                for l in table:
                    raw = i.to_bytes(1, 'big')
                    raw += j.to_bytes(1, 'big')
                    raw += k.to_bytes(1, 'big')
                    raw += l.to_bytes(1, 'big')
                    raw += part
                    if sha256(raw).hexdigest().encode() == c:
                        pipe.send(raw.decode()[:4].encode())

if __name__ == '__main__':
    sock = socket()
    sock.connect(("47.98.192.231", 25258))
    text = sock.recv(1024).decode()
    sleep(0.02)
    sock.recv(1024)
    print(text)
    part = re.search(r'(?<=\+).{16,}(?=\))', text,).group(0).encode()
    c = re.search(r'(?<=\=\= ).+', text,).group(0).encode()
    (root,sub)=Pipe()
    pool = Pool(processes=5)
```

```
    for i in range(5):
        pool.apply_async(func=task, args=(sub, i, c, part))
    print("all process started")
    pool.close()
    pool.join()
    print("all process stopped")
    res = root.recv()
    print(res)
    sock.send(res)
    sleep(0.02)
    tel = Telnet()
    tel.sock = sock
    tel.interact()
```

Crypto - Feedback

题目：

```
听说上周Classic_CrackMe的CBC很简单? 来要个CFB试试 XD
nc 47.98.192.231 25147
```

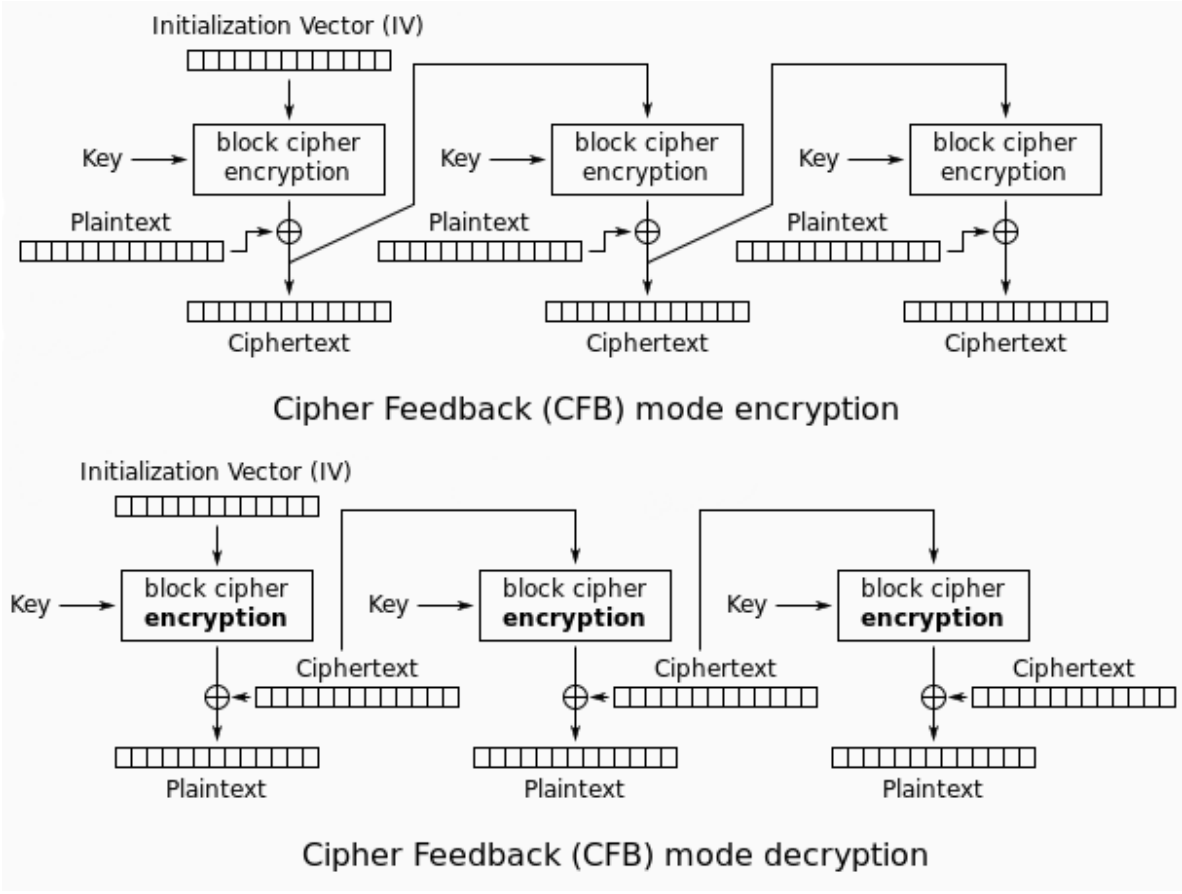
然后是服务端的脚本：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import os, random
import string, binascii
import signal
import socketserver
from hashlib import sha256
from Crypto.Cipher import AES

from secret import MESSAGE
assert len(MESSAGE) == 48

class Task(socketserver.BaseRequestHandler):
    def __init__(self, *args, **kargs):
        self.KEY = b""
        self.IV = b""
        super().__init__(*args, **kargs)

    def _recvall(self):
        BUFF_SIZE = 2048
        data = b''
        while True:
            part = self.request.recv(BUFF_SIZE)
            data += part
            if len(part) < BUFF_SIZE:
                break
        return data.strip()

    def send(self, msg, newline=True):
        try:
            if newline: msg += b'\n'
            self.request.sendall(msg)
        except:
```

```
            pass

    def recv(self, prompt=b'> '):
        self.send(prompt, newline=False)
        return self._recvall()

    def encrypt(self, data):
        assert len(data) % 16 == 0
        aes = AES.new(self.KEY, AES.MODE_CFB, self.IV, segment_size=128)
        return aes.encrypt(data)

    def decrypt(self, data):
        assert len(data) % 16 == 0
        aes = AES.new(self.KEY, AES.MODE_CFB, self.IV, segment_size=128)
        return aes.decrypt(data)

    def handle(self):
        signal.alarm(60)
        self.KEY = os.urandom(32)
        self.IV = os.urandom(16)

        self.send(b"You have only 3 times to decrypt sth, then I'll give u the
FLAG.")
        try:
            for _ in range(3):
                self.send(b"Give me sth(hex) to decrypt")
                hex_input = self.recv()
                if not hex_input:
                    break
                ciphertext = binascii.unhexlify(hex_input)
                plaintext = self.decrypt(ciphertext)
                self.send( binascii.hexlify(plaintext) )
        except:
            self.send(b"Rua!!!")
            self.request.close()

        enc_msg = self.encrypt(MESSAGE)
        self.send(b"Here is your encrypted FLAG(hex): ", newline=False)
        self.send( binascii.hexlify(enc_msg) )
        self.request.close()

class ForkedServer(socketserver.ForkingMixIn, socketserver.TCPServer):
    pass

if __name__ == "__main__":
    HOST, PORT = '0.0.0.0', 1234
    server = ForkedServer((HOST, PORT), Task)
    server.allow_reuse_address = True
    server.serve_forever()
```

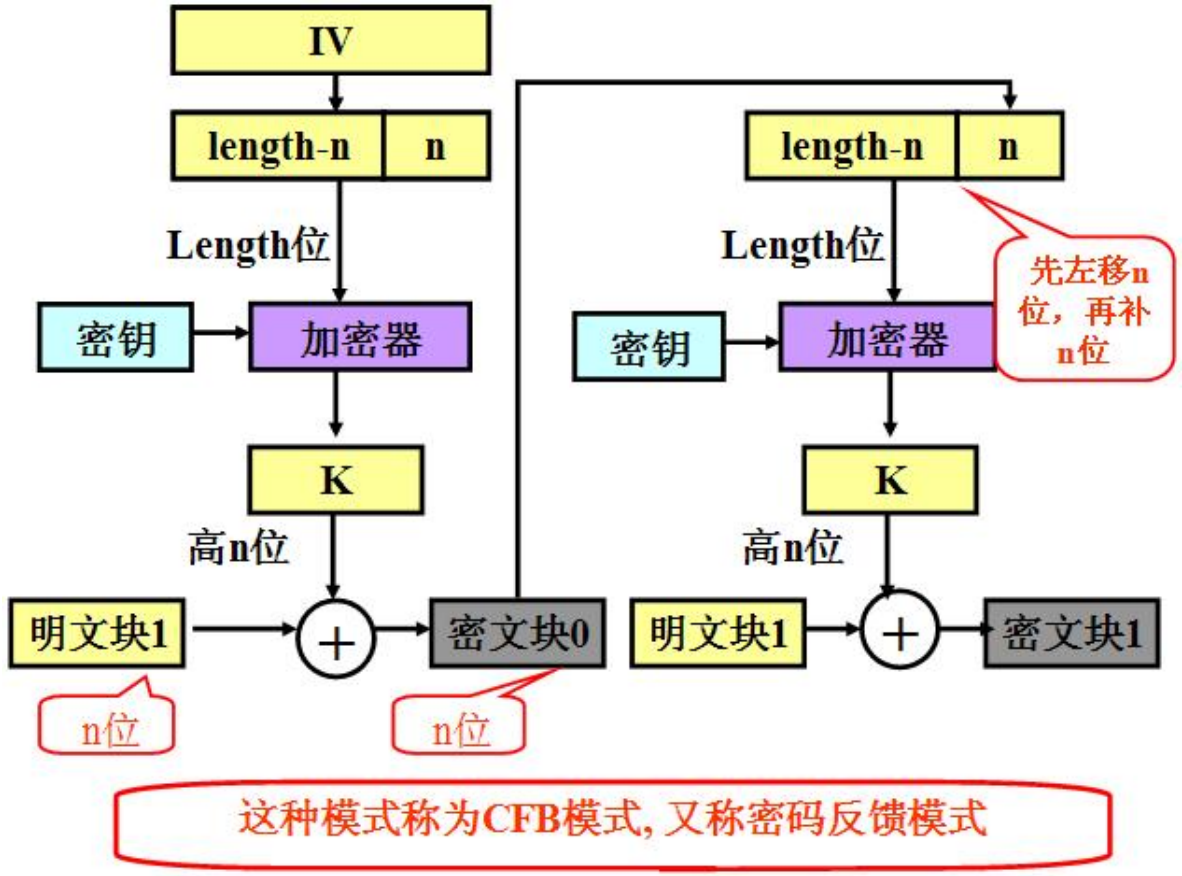
连接指定地址后，服务器会提供3次机会解密任意文本，并将解密结果以及加密后的flag显示出来
脚本中有os.urandom()，因此每一次连接中服务器使用的加解密参数都是不一样的

题目里面明确提到了CFB，经过百度，发现这是AES的一种加密模式（坑）。原理图如下：

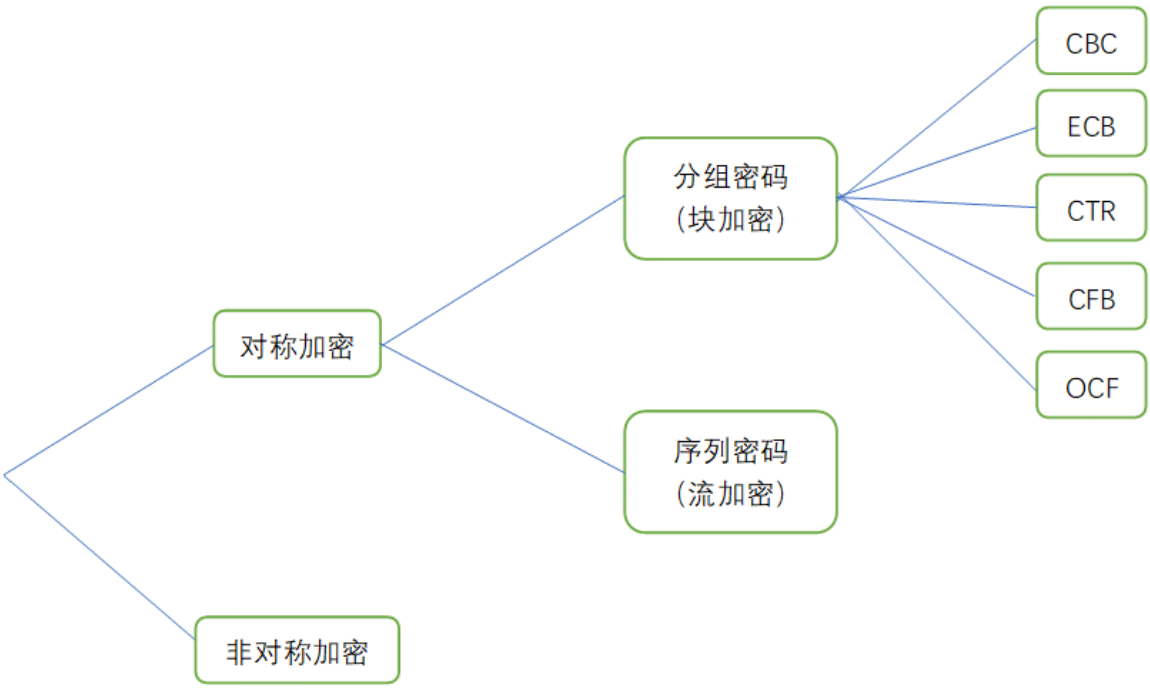


对图上的各个部分进行搜索，得知图中的⊕表示异或，IV是一个用于首轮加密的初始参数（初始化向量），然而中间的block cipher encryption就不知道是什么意思了

之后找到了用中文标注的图片，但还是没弄清楚加密器是怎么生成K的



之后直接百度AES加密，得知这是一种对称加密的方式，而且其原理很复杂（对我来说），并且之前查到的几种加密模式（CBC、ECB、CTR、CFB和OCF）并不只是针对于AES，而是所有分组加密（块加密）都可以使用的
整个关系图应该是这个样子



而之前图片中的block cipher encryption以及“加密器”就是所选用的具体加密算法，在这里是AES
后来查到服务端脚本中的segment_size=128表明每处理128位调用一次加密器，而因为len(MESSAGE) == 48，所以整个加密器只被调用了一次（坑）。结合之前的CFB原理图，我只需要提供一串长度为48的文本C，将其提供给服务器解密得到m，之后计算K=C^m，然后将加密后的flag与K异或即可获得明文。题目给的3次机会明显多余（坑）
然而题目很明显不会这么简单。实际操作结果如下：

```
C(编码前，自定义)=b'123456789012345678901234567890123456789012345678'
C(编码
后)=313233343536373839303132333435363738393031323334353637383930313233343536373
39303132333435363738
m(16进制编
码)=f8a940949cbce4fa99c87070055517fd926d8107c69aa64675d188fc9461400352cac50ad9af
1660e9def81c1bbccd70
m(解码
后)=b'\xf8\xa9@\x94\x9c\xbc\xe4\xfa\x99\xc8pp\x05U\x17\xfd\x92m\x81\x07\xc6\x9a\
xa6Fu\xd1\x88\xfc\x94a@\x03R\xca\xc5\n\xd9\xaf\x16`\xe9\xde\xf8\x1c\x1b\xbc\xcdp
'
K=b'\xc9\x9bs\xa0\xa9\x8a\xd3\xc2\xa0\xf8AB6a"\xcb\xa5U\xb87\xf7\xa8\x95r@\xe7\x
bf\xc4\xadQq1a\xfe\xf0<\xee\x97/P\xd8\xec\xcb(.\x8a\xfaH'
flag(16进制编码，解密
前)=8fd732e789e3a0e2c89f202f531a17facd049273a075ec70f393376473f833a7486269c7076d
3341cad2c40412009150
flag(解密前)=b'\x8f\xd72\xe7\x89\xe3\xa0\xe2\xc8\x9f
/S\x1a\x17\xfa\xcd\x04\x92s\xa0u\xecp\xf3\x937ds\xf83\xa7Hbi\xc7\x07m3A\xca\xd2\
xc4\x04\x12\x00\x91P'
flag(解密后)=b'FLAG is
hgame{51hQ*DW\xddy\x02\xb3t\x88\xa0\xde\xa9B\x96)\x9c\x99\xfb\xe9\xfa\x1c\x11\x1
2>\x0f,<\x8ak\x18'
```

发现flag的后半段是无法打印的字符，并且最后也没有出现}
怀疑要对flag后半段进行异或。因为0x7d('}' 对应编码) ^ 0x18=0x65=101，所以将刚才的flag整体与101异或，得到以下结果

```
b'#)$"E\x0c\x16E\r\x02\x04\x08\x00\x1ePT\r4O!2\xb8\x1cg\xd6\x11\xed\xc5\xbb\xcc\
'\xf3L\xf9\xfc\x9e\x8c\x9fytw[jIY\xef\x0e}'
```

叒叒叒陷入了僵局。。。
后来才发现segment_size=128指的是128位（16个字节），而len(MESSAGE) == 48是指48个字节，因此整个加密器被使用了3次。上面的操作只能解出MESSAGE的前16个字节
（回想一下也觉得自己挺傻的，既然考点是CFB，那么一次简单的异或肯定不能解决问题）

之后进一步分析CFB，发现在这种模式下，某个块的加密结果和前一个块的加密结果有关。举个例子，在segment_size=8（1个字节）的情况下，使用相同的key和初始向量IV加密b'12345'和b'02345'，即使明文只有第1个字节不同，之后相同的'2345'加密之后的结果也不一样。通过这种方式加密，即使明文中出现了多个完全相同的块，攻击者也无法从密文当中直接看出来
（与其形成鲜明对比的是ECB模式，块与块的加密过程完全独立（类似于week1的仿射密码，在指定A、B的情况下单个明文字符和密文字符的对应关系是唯一的），虽然方便并行计算，但是在攻击者拥有大量明文和对应密文的情况下更容易受攻击）

而从本题的角度来分析，如果我提交的待解密密文是任意的，那么刚才的方法只能确保加密器第一次生成的K（K用于与明文/密文异或，图中有说明）与加密flag的第一个K相同（因为都是用IV和Key生成的），而之后加密器生成的K就与加密flag的K不一样了（因为参数中只有Key相同，而上一个块的密文不同），因此前者的K不能用于解密后者。这也就是刚才的解密过程只能获得MESSAGE前16个字节（128位）的原因

不过目前已知第一个块的明文m1，在获取了第一个块的K1之后可以算出第一个块的密文c1=m1 ^ K1，将第一个块的密文与任意字符串S2+X拼接得到密文C'=c1+S2+X(仍然需要确保len(C')==48，且S2=16)，将其提交给服务器解密得到M'=m1+s2+X，此时计算K2=S2 ^ s2即可得到正确的K2，并且用K2可以解出flag的第二部分m2
同理，K3,m3也可以求出来，最后flag=m1+m2+m3

当前思路用符号表述如下（括号内为字符长度）：
其中S是提交的密文，s是服务器返回的明文，K用于与明文/密文异或（CFB中的K）,C是flag的密文，m是flag的明文，X为任意字符串
整个过程中m全程不变，其余参数在单次连接内不变

第一次连接：
任意构造S=S1(16)+X(32)，提交一次得到s=s1(16)+X(16)，计算K1(16)=S1 ^ s1
之后服务器提供C=C1(16)+X(32)，计算m1(16)=C1 ^ K1

第二次连接：
任意构造S=S1(16)+X(32)，提交一次得到s=s1(16)+X(16)，计算K1(16)=S1 ^ s1，C1(16)=m1 ^ K1
之后构造S'=C1(16)+S2(16)+X(16)，再提交一次得到s'=m1+s2(16)+X，计算K2(16)=S2 ^ s2
之后服务器提供C'=C1(16)+C2(16)+X(16)，计算m2(16)=C2 ^ K2

第三次连接：
任意构造S=S1(16)+X(32)，提交一次得到s=s1(16)+X(16)，计算K1(16)=S1 ^ s1，C1(16)=m1 ^ K1
之后构造S'=C1(16)+S2(16)+X(16)，再提交一次得到s'=m1+s2(16)+X，计算K2(16)=S2 ^ s2，C2(16)=m2 ^ K2
之后构造S''=C1(16)+C2(16)+S3(16)，再提交一次得到s''=m1+m2+s2(16)，计算K3(16)=S3 ^ s3
之后服务器提供C''=C1(16)+C2(16)+C3(16)，计算m3(16)=C3 ^ K3

最后得到flag=m1+m2+m3

由于每一次连接服务器时的Key、IV都不同，因此每一次连接服务器时取得的Kx都不一样，所以每次连接的关键在于获得一部分flag（明文块）mx，这样才可以进行下一步

具体实现过程中还有个问题，因为服务端脚本中有signal.alarm(60)，所以每次连接之后的所有操作要在60秒内完成，否则连接会自动断开。只能写脚本了
最后编写的半自动脚本如下：

```
from socket import socket
from telnetlib import Telnet
from time import sleep
from binascii import hexlify,unhexlify
import re

XOR = lambda s1, s2: bytes([x ^ y for x, y in zip(s1, s2)])

base = '12345678901234567890123456789012345678901234567890123456789012345678'.encode()
flag = 'FLAG is hgame{51b72d4cd23b2fe672a874cb44020868}.'.encode()

sock = socket()
sock.connect(("47.98.192.231", 25147))
tel = Telnet()
tel.sock = sock
sleep(0.1)

text = tel.read_until(b'>').decode()
print(text, end='\n--------------------\n')

tel.write(hexlify(base))
sleep(0.1)
text = tel.read_until(b'>').decode()
print(text, end='\n--------------------\n')

c = unhexlify(re.search(r'(?<= ).+(?=\n)', text).group(0))
tmp1 = c[:16]
tmp2 = base[:16]
key = XOR(tmp1, tmp2)
print(key)

base = XOR(key, flag[:16]) + base[16:]
tel.write(hexlify(base))
sleep(0.1)
text = tel.read_until(b'>').decode()
print(text, end='\n--------------------\n')

c = unhexlify(re.search(r'(?<= ).+(?=\n)', text).group(0))
tmp1 = c[16:32]
tmp2 = base[16:32]
key = XOR(tmp1, tmp2)
print(key)

base = base[:16]+ XOR(key, flag[16:32]) + base[32:]
tel.write(hexlify(base))
sleep(0.1)
text = tel.read_until(b'>').decode()
print(text, end='\n--------------------\n')

c = unhexlify(re.search(r'(?<= ).+(?=\n)', text).group(0))
```

```
tmp1 = c[32:]
tmp2 = base[32:]
key = XOR(tmp1, tmp2)
print(key)

tel.interact()
```

最终flag：hgame{51b72d4cd23b2fe672a874cb44020868}

后记：
刚看到初始化向量(IV)的时候不知道其用途，以为解密方不需要IV只通过Key就能解密，于是猜测IV可以增强破解难度
后来看资料和服务端代码发现解密时也需要提供IV
经过后续分析，IV只是CFB加密的递归过程中的一个初始条件

Misc - 美 人 鲸

题目：

```
我们受过严格的训练，无论在什么环境里，都不会乱放东西，
除非忍不住。
```

后面是Docker镜像的地址

```
https://hub.docker.com/r/zhouweitong/hgame2020-misc
```

先是在Kali上安装Docker并下载镜像，然后百度到docker有个export命令可以将容器的文件系统提取为一个压缩包，提取并解压后直接搜索hgame无果，搜flag发现/usr/share/nginx/html/index.html里面有以下内容

```
You want flag? See $FLAG.
```

之后在Docker镜像中以交互模式运行sh，输入env
查看环境变量，发现以下内容

```
FLAG=Find flag.tar.gz!
```

而刚刚的搜索结果当中就有flag.tar.gz!
用压缩包管理器打开，里面含有flag.zip和README文件。flag.zip是被加密的压缩包，而README当中有以下内容

```
See sh history.
```

之后在镜像的sh中输入sh history，出现以下错误

```
sh: can't open 'history': No such file or directory
```

于是百度如何查看sh的历史记录，得知直接运行history命令即可。运行后发现一条记录

```
/ # history
   0 echo -e "Zip password is somewhere else in /etc.\nFind it!"
```

到头来还是要从文件搜索入手。直接搜password，发现在/etc/issue文件中有以下内容

```
Welcome to Alpine Linux 3.10
Kernel \r on an \m (\l)

Zip Password: cfuzQ3Gd6gqKG@$N
```

用该密码解压flag.zip，得到flag.db。Kali系统中可以直接打开这个数据库文件，得到flag
最终flag：hgame{v3RWI3qSpcKZhp^xv$kaBhNjVqxk##3e}

后记：
该题时间分配：5%搜相关命令 5%解题 90%安装Docker（雾
（写了篇安装Docker的博文不到3天访问破千）
（网上有现成的安装方法，因为想修改一点点所以装了很久）

看完week4的题，感觉自己要凉

2020.02.08