登录

Symmetric block ciphers Summary - DES & AES

一叶飘零 / 2018-10-31 09:53:00 / 浏览数 3747 技术文章 技术文章 顶(0) 踩(0)

前言

最近要进行密码学宣讲，所以就稍微总结了一下对称分组密码，毕竟公钥密码（RSA）前面总结过一些常见的了，这里给上链接

```
skysec.top/2018/08/24/RSA■■■■■(1)
skysec.top/2018/08/25/RSA■■■■■(2)
skysec.top/2018/09/13/Crypto-RSA■■■■■■■
skysec.top/2018/09/15/■■RSA-Padding-Attack
skysec.top/2018/09/17/Crypto-RSA-■■■■■■
```

有兴趣的可以自己看看，无非是从公钥、多同余式、p/q或是公式推导入手。
而这里介绍的对称分组密码（DES、AES），主要从其本身和分组模式的问题介绍

DES

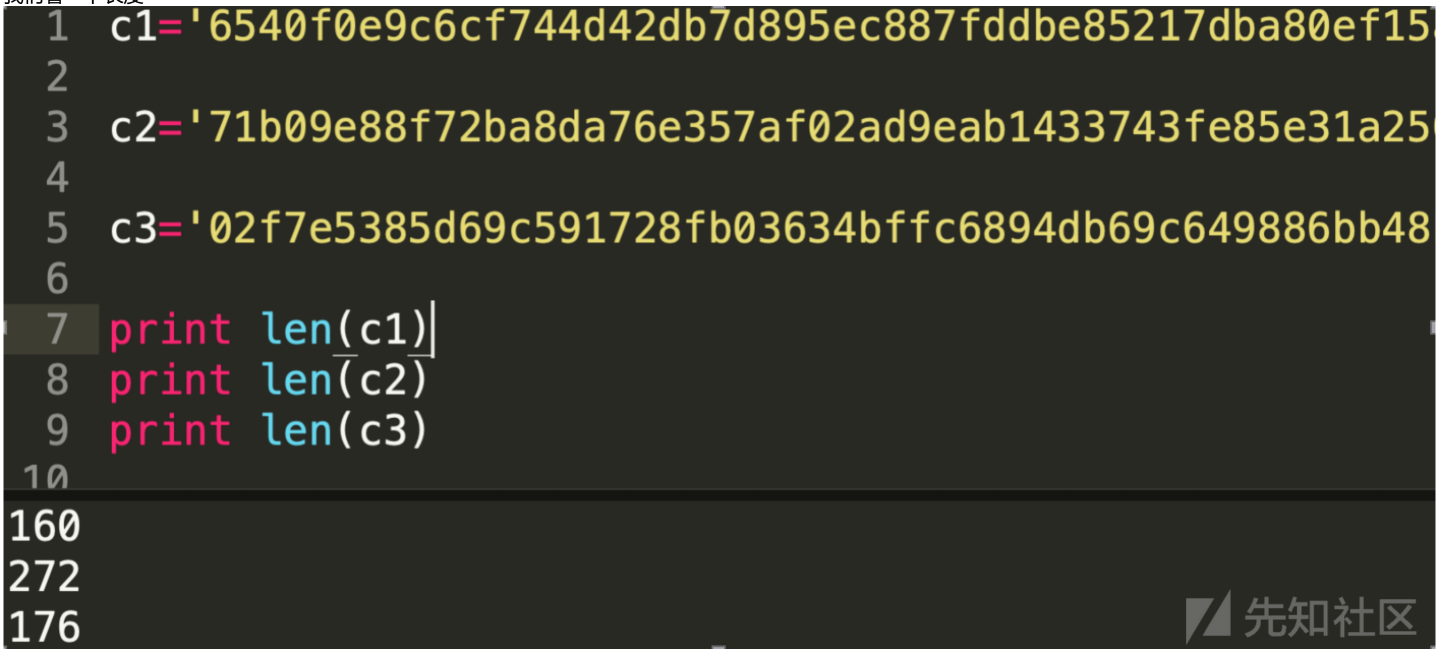
DES由于密钥只有64bit，并且有效位只有仅仅56bit，剩余8bit为校验位，所以存在比较显著的爆破攻击风险

爆破攻击

以最新的hitcon2018为例：oh-my-raddit
这道题就可以作为一个爆破攻击的典例：
1.题目给了大量的密文
2.每个密文对应一篇文章的link
虽然本题乍一看是一道唯密文攻击的题目，但是不难提取出如下特点：

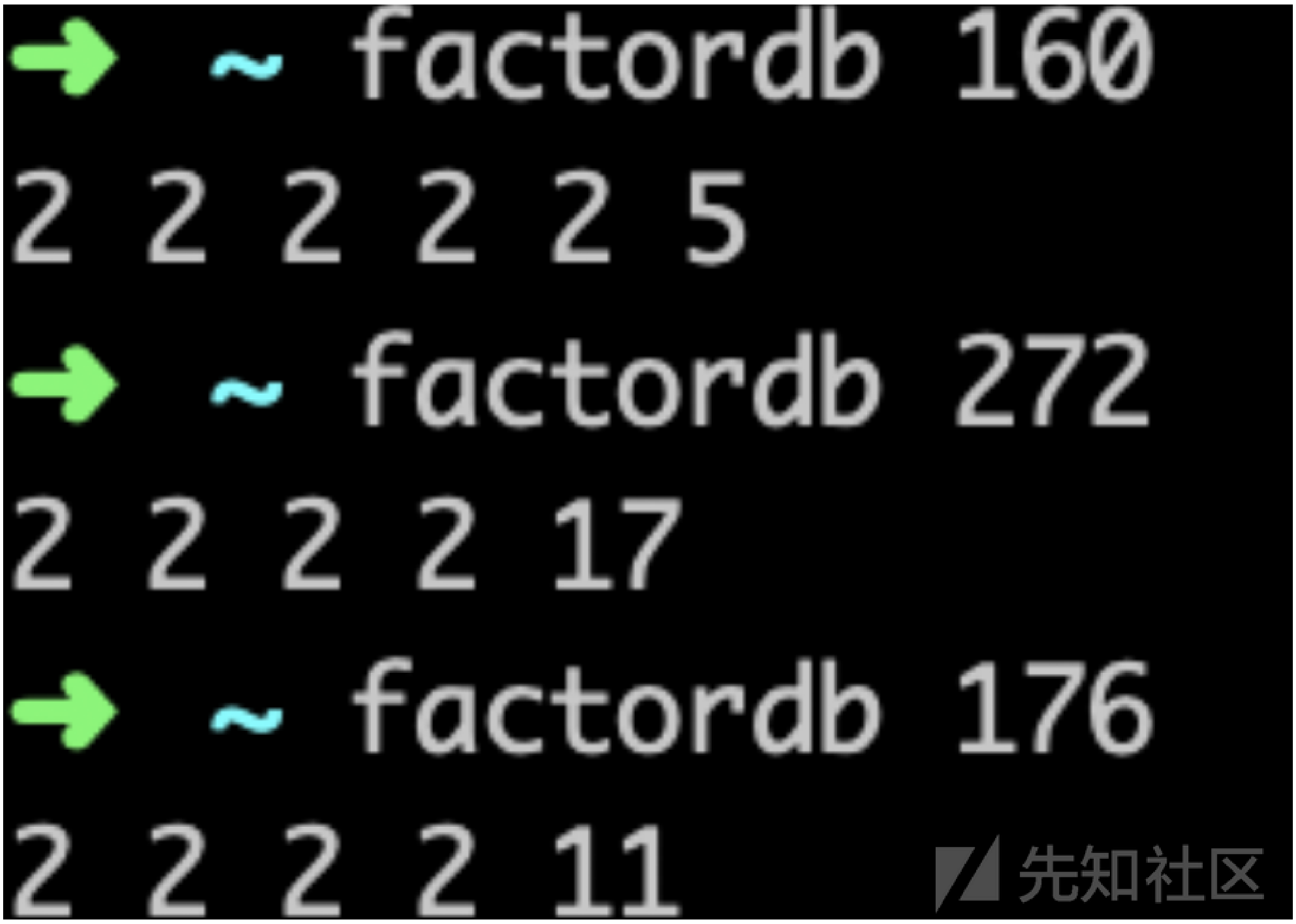
```
c1=6540f0e9c6cf744d42db7d895ec887fddbe85217dba80ef15a370279d62741526126bef1904e34a9754531903e10c8cce11b58e11007a26a55ba2433615

c2=71b09e88f72ba8da76e357af02ad9eab1433743fe85e31a2501049e465bca5e92faefdcb3f7dee61ca73fdc9bc3e4913ab5c2badf4c89831efa48ec2c7f

c3=02f7e5385d69c591728fb03634bffc6894db69c649886bb48b2a80152681b5b7ead5bb13c0a0aed1a415ab01344a59c039790c9d9e7f8e303e88184ca46
```

我们看一下长度



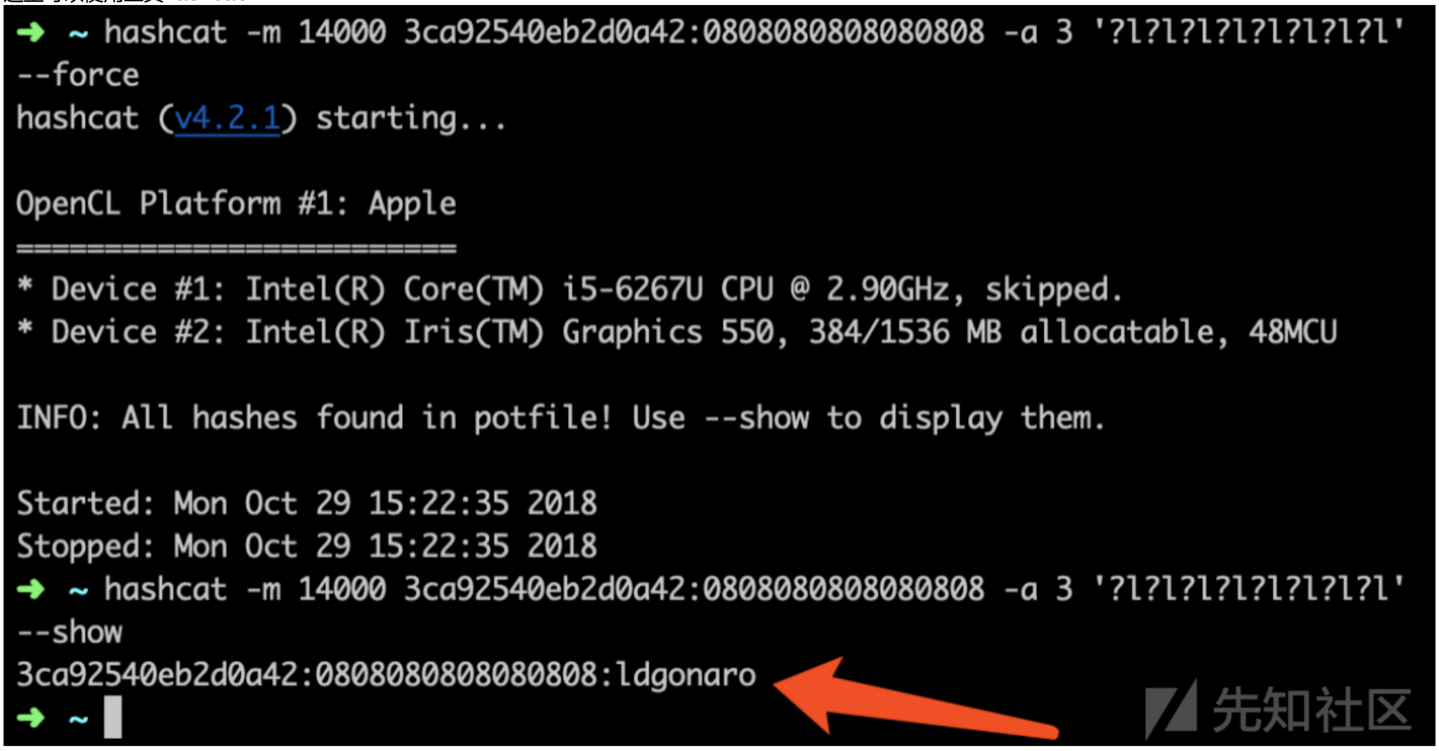
```
c1='6540f0e9c6cf744d42db7d895ec887fddbe85217dba80ef15

c2='71b09e88f72ba8da76e357af02ad9eab1433743fe85e31a25

c3='02f7e5385d69c591728fb03634bffc6894db69c649886bb48

print len(c1)
print len(c2)
print len(c3)
```

```
160
272
176
```



```
➜  ~ factordb 160
2 2 2 2 2 5
➜  ~ factordb 272
2 2 2 2 17
➜  ~ factordb 176
2 2 2 2 11
```

不难发现：
1.每组密文的长度都是16的倍数，可以据此猜出大概为8bytes一组
2.每组密文结尾都是3ca92540eb2d0a42，不难猜出这应该是Padding
3.得到一组明密文对：0808080808080808:3ca92540eb2d0a42
那么爆破即可，如果强行写脚本爆破，肯定是非常慢的。
这里可以使用工具hashcat

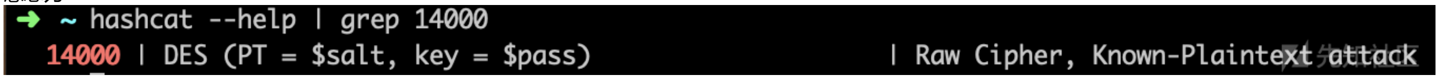


```
➜  ~ hashcat -m 14000 3ca92540eb2d0a42:0808080808080808 -a 3 '?l?l?l?l?l?l?l?l'
--force
hashcat (v4.2.1) starting...

OpenCL Platform #1: Apple
=========================
* Device #1: Intel(R) Core(TM) i5-6267U CPU @ 2.90GHz, skipped.
* Device #2: Intel(R) Iris(TM) Graphics 550, 384/1536 MB allocatable, 48MCU

INFO: All hashes found in potfile! Use --show to display them.

Started: Mon Oct 29 15:22:35 2018
Stopped: Mon Oct 29 15:22:35 2018
➜  ~ hashcat -m 14000 3ca92540eb2d0a42:0808080808080808 -a 3 '?l?l?l?l?l?l?l?l'
--show
3ca92540eb2d0a42:0808080808080808:ldgonaro
➜  ~ 
```

其中

```
-m 14000
```

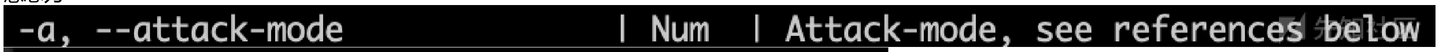
意思为



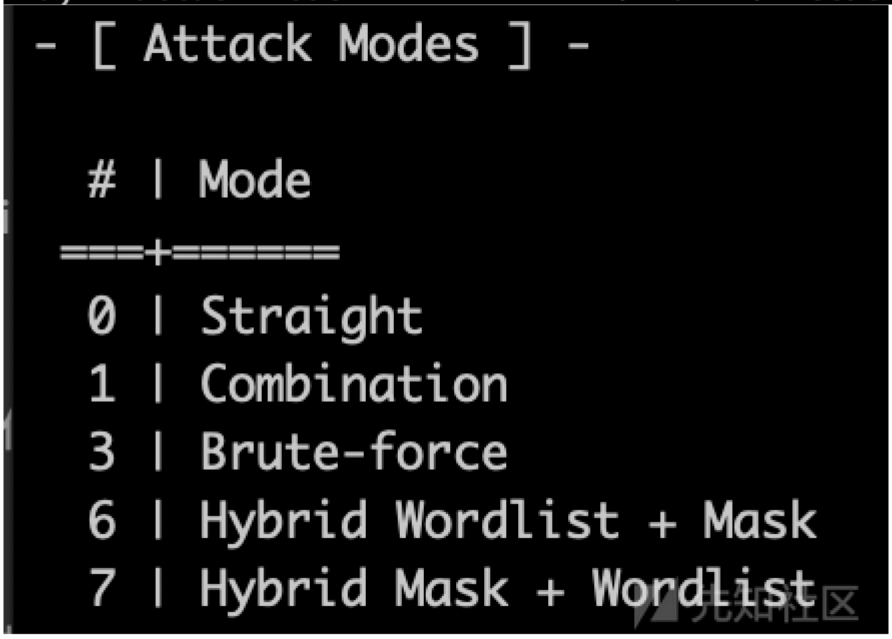
```
➜ ~ hashcat --help | grep 14000
  14000 | DES (PT = $salt, key = $pass)                    | Raw Cipher, Known-Plaintext attack
```

```
-a 3
```

意思为



```
-a, --attack-mode              | Num  | Attack-mode, see references below
```



```
- [ Attack Modes ] -

  # | Mode
 ===+======
  0 | Straight
  1 | Combination
  3 | Brute-force
  6 | Hybrid Wordlist + Mask
  7 | Hybrid Mask + Wordlist
```

```
?l?l?l?l?l?l?l?l
```

意思为

■■?d
■■■■?l
■■■■?u
■■■■?s
■■■■■+■■■■?a

这里意思为纯小写字母的key爆破

弱密钥

之所以叫弱密钥，是因为使用这样的初始密钥会生成16个相同的子密钥，这肯定不是我们期望发生的
这样的弱密钥有

- 0x0101010101010101
- 0xFEFEFEFEFEFEFEFE
- 0xE0E0E0E0F1F1F1F1
- 0x1F1F1F1F0E0E0E0E

同时还有半弱密钥，即存在情况

$$E_{K_1}(E_{K_2}(M)) = M$$

即用K2加密明文，可以用K1解密，这种半弱密钥有：

- 0x011F011F010E010E:0x1F011F010E010E01
- 0x01E001E001F101F1:0xE001E001F101F101
- 0x01FE01FE01FE01FE:0xFE01FE01FE01FE01
- 0x1FE01FE00EF10EF1:0xE01FE01FF10EF10E
- 0x1FFE1FFE0EFE0EFE:0xFE1FFE1FFE0EFE0E
- 0xE0FEE0FEF1FEF1FE:0xFEE0FEE0FEF1FEF1

参考链接：
https://en.wikipedia.org/wiki/Weak_key#Weak_keys_in_DES

子密钥逆推

如果子密钥泄露，可以几乎成功逆推初始密钥，但由于有效位仅56bit，所以子密钥只能恢复56bit的子密钥，还有8bit需要爆破，但2^8并不是很大，所以可以容易破解
下面从一道某春秋的例题去看，考察点主要还是在密钥编排和DES流程分析
注：代码非常冗余，因为是1年前做的题，把脚本直接扒出来了

```
.......(■■■■)
deskey="imnotkey"
DES = des(deskey)
DES.Kn =[
    ........(■■■■■)
]

DES.setMode(ECB)
correct=[
    ......(■■■■)
]
// DES.encrypt(code)==correct
from Crypto.Cipher import Blowfish
import base64
key= deskey+code
cipher = Blowfish.new(key, Blowfish.MODE_ECB)
print cipher.decrypt(base64.b64decode("fxd+VFDXF6lksUAwcB1CMco6fnKqrQcO5nxS/hv3FtN7ngETu95BkjDn/ar+KD+RbmTHximw03g="))
```

我们首先来看一下代码主流程看了什么：

- 设置了一个未知的deskey
- 然后用这个未知的deskey加密了code
- 然后用deskey+code作为key，调用blowfish密码，加密了flag

然后我们有

- deskey的密钥编排后的子密钥
- code加密后的密文correct
- blowfish加密后的密文

所以思路还算清晰：

- 用deskey的子密钥反推deskey
- 用deskey的子密钥解密correct得到code
- 用得到的deskey和code作为密钥解密blowfish密文得到flag

然后我们容易知道DES的密钥编排过程为:

- 首先输入64bit密钥
- 将64bit密钥经过PC-1盒变成56bit
- 将56bit分为C0和D0，分别28bit
- 将C0，D0分别循环左移位，得到C1，D1
- 将C1，D1拼接，经过PC-2盒变成48bit子密钥key1
- 重复步骤4
- 生成16组子密钥

所以这里我有想法:

- 由子密钥key1经过逆PC-2盒推出C1，D1(得到48位已知和8位未知)
- 由C1，D1分别循环右移1位，得到C0，D0
- 由C0，D0经过逆PC-1盒得到deskey(已知48位，未知16位)
- 然后将deskey的16个未知量设置成a,b,c,d......
- 用带有未知参数的deskey生成16个子密钥
- 用16个带未知参数的子密钥和16个已知子密钥建立方程组
- 可以解出其中8个bit的未知量，剩余8个bit不重要，因为deskey实际加密只用了56位密钥
- 随机给剩下8bit赋值，作为一个deskey，解密correct
- 爆破剩余8bit的deskey变量，根据题目特性，应该会有一个可以是明文的字符串，即deskey
- 用deskey+code作为key解密blowfish密文，得到flag

那么我们有子密钥

```
DES.Kn =[
   ........([redacted])
]
```

由子密钥key1开始逆推：
首先是逆PC-2盒：

```
key1 = [1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1
__pc2 = [
       13, 16, 10, 23,  0,  4,
        2, 27, 14,  5, 20,  9,
       22, 18, 11,  3, 25,  7,
       15,  6, 26, 19, 12,  1,
       40, 51, 30, 36, 46, 54,
       29, 39, 50, 44, 32, 47,
       43, 48, 38, 55, 33, 52,
       45, 41, 49, 35, 28, 31
   ]
C1D1 = ['*']*56
for i in range(0,len(key1)):
   C1D1[__pc2[i]] = key1[i]
print C1D1
```

可以得到C1D1的值为:

```
[1, 1, 0, 1, 0, 1, 1, 0, '*', 1, 1, 0, 0, 1, 1, 0, 0, '*', 1, 0, 1, '*', 0, 1, '*', 0, 1, 1, 1, 1, 1, 1, 1, 1, '*', 1, 1, '*',
```

然后我们循环右移动1位逆推出C0D0

```
C1:11010110*11001100*101*01*011
D1:111111*11*1111*1000000010*01
```

循环右移一位：

```
C0:111010110*11001100*101*01*01
D0:1111111*11*1111*1000000010*0
```

然后可以逆PC-1盒得到deskey

```
C0='111010110*11001100*101*01*01'
D0='1111111*11*1111*1000000010*0'
__pc1 = [56, 48, 40, 32, 24, 16, 8,
        0, 57, 49, 41, 33, 25, 17,
        9, 1, 58, 50, 42, 34, 26,
        18, 10, 2, 59, 51, 43, 35,
        62, 54, 46, 38, 30, 22, 14,
        6, 61, 53, 45, 37, 29, 21,
        13, 5, 60, 52, 44, 36, 28,
        20, 12, 4, 27, 19, 11, 3
        ]
C0D0 = C0+D0
res = ['*']*64
deskey = ""
for i in range(0,len(__pc1)):
   res[__pc1[i]] = C0D0[i]
for i in res:
   deskey += i
print deskey
```

得到deskey

```
11000***11**011*0010011*1001011*0111011*11*00*1*1*0*011*1001111*
```

然后我们给每个未知量替换为变量a,b,c......
得到

```
11000abc11de011f0010011g1001011h0111011i11j00k1L1m0n011o1001111p
```

然后我们用这个带未知量的deskey生成16个子密钥:

```
def zuoyiwei(str,num):
   my = str[num:len(str)]
   my = my+str[0:num]
   return my
def key_change_1(str):
   key1_list = [57,49,41,33,25,17,9,1,58,50,42,34,26,18,10,2,59,51,43,35,27,19,11,3,60,52,44,36,63,55,47,39,31,23,15,7,62,54,4
   res = ""
   for i in key1_list:
       res+=str[i-1]
   return res

def key_change_2(str):
   key2_list = [14,17,11,24,1,5,3,28,15,6,21,10,23,19,12,4,26,8,16,7,27,20,13,2,41,52,31,37,47,55,30,40,51,45,33,48,44,49,39,5
   res = ""
   for i in key2_list:
       res+=str[i-1]
   return res
def key_gen(str):
   key_list = []
   key_change_res = key_change_1(str)
   key_c = key_change_res[0:28]
   key_d = key_change_res[28:]
   for i in range(1,17):
       if (i==1) or (i==2) or (i==9) or (i==16):
           key_c = zuoyiwei(key_c,1)
           key_d = zuoyiwei(key_d,1)
       else:
           key_c = zuoyiwei(key_c,2)
           key_d = zuoyiwei(key_d,2)
       key_yiwei = key_c+key_d
       key_res = key_change_2(key_yiwei)
       key_list.append(key_res)
   return key_list
deskey = "11000abc11de011f0010011g1001011h0111011i11j00k1L1m0n011o1001111p"
print key_gen(deskey)
```

得到结果

```
['101110011111010100011001111100110010101110010111', '1j0n111101d110001m001110101k011110100011be0a0111',
'00111010jm100d11111110001011011ae01001111100011b', '1d0111000101110m00101nj1011111b010k00e1111000111',
'11j00011d01010110101110m01k1e1101110010b11001a11', '0000110m1111111010n001d1011011101e110101a10010k1',
'n1d1001100111101m11j10101b101k10111101010110101a', '111m1j00111001n011100001e11011a0110111110k10b010',
'10n111011011m00j0d1101100k00111e11011b01011110a0', '101001100dm011101110101j1100ba01110111010111k100',
'1111101j01110m1d001n0100110010011b0k11101a111e00', '1m001n0010011111j101100d11011001a1011110e0k11101',
'010j01ndm111001001111111b00110110101ka101011110e', '1010n11111011101d10000m01001a0e1011110b1101k0101',
'01md1010011010111110nj11101b001k0a10101e10110101', '00101111100mdj10n111010110110e110110a0k011010b11']
```

和题目中的Kn比对:

```
['101110011111010100011001111100110010101110010111', '110011110111100010001110101001111010001111000111',
'001110101010011111111000101101101010011111000111', '110111000101110000101011011111101000011111000111',
'111000111010101101011100010111101110010111001011', '000011001111111010000111011011101111010101001001',
'011100110011110101111010111010101111010101101010', '111011001110010011100001111011001101111100101010',
'100111011011000101110110000011111101110101111000', '101001100100111011101011110010011101110101110100',
'111110110111001100100100110010011100111010111100', '100010001001111111011001110110010101111010011101',
'010101010111001001111111100110110101001010111101', '101001111101110111000000100100110111101110100101',
'010110100110101111100111101100100010101110110101', '001011111000111001110101101101110110000011010111']
```

我们容易得到8个变量的值，然后得到带有8个未知数的deskey

```
"1100001"+c+"1111011"+f+"0010011"+g+"1001011"+h+"0111011"+i+"1110001"+l+"1000011"+o+"1001111"+p
```

然而这个deskey大家会发现怎么都不对，我们阅读题目中给的程序
发现他对deskey的处理：

```
key = self.__permutate(des.__pc1, self.__String_to_BitList(self.getKey()))
```

我们跟进这个__String_to_BitList()

```
def __String_to_BitList(self, data):
       """Turn the string data, into a list of bits (1, 0)'s"""
       if _pythonMajorVersion < 3:

           data = [ord(c) for c in data]
       l = len(data) * 8
       result = [0] * l
       pos = 0
       for ch in data:
           for i in range(0,8):
               result[(pos<<3)+i]=(ch>>i)&1
           pos+=1

       return result
```

可以发现这个根本不是原版的pydes库的函数，我们来看看原版函数:

```
def __String_to_BitList(self, data):
       """Turn the string data, into a list of bits (1, 0)'s"""
       if _pythonMajorVersion < 3:
           # Turn the strings into integers. Python 3 uses a bytes
           # class, which already has this behaviour.
           data = [ord(c) for c in data]
       l = len(data) * 8
       result = [0] * l
       pos = 0
       for ch in data:
           i = 7
           while i >= 0:
               if ch & (1 << i) != 0:
                   result[pos] = 1
               else:
                   result[pos] = 0
               pos += 1
               i -= 1
       return result
```

容易发现，我们题目中的处理deskey的函数：

- 会先把deskey转换成64bit的二进制
- 然后将64bit的2进制，8个一组进行分组
- 再对每一组倒叙输出
- 然后再把8组拼接回来

什么意思呢？
比如

```
abcdefjhABCDEFJH
```

他处理后会变成

```
hjfedcbaHJFEDCBA
```

所以我们的deskey要进行处理:

```
deskey_old = '1100001"+c+"1111011"+f+"0010011"+g+"1001011"+h+"0111011"+i+"1110001"+L+"1000011"+o+"1001111"+p'.replace('"+','')
deskey_new = ""
for i in range(0,len(deskey_old),8):
   deskey_new += deskey_old[i:i+8][::-1]
print deskey_new
```

得到

```
c1000011f1101111g1100100h1101001i1101110L1000111o1100001p1111001
```

然后我们就可以爆破8bit寻找可读明文字符串了

```
def bintostr(str):
   res = ""
   for i in range(0,len(str),8):
       res += chr(int(str[i:i+8],2))
   return res
for c in "01":
   for f in "01":
       for g in "01":
           for h in "01":
               for i in "01":
                   for L in "01":
                       for o in "01":
                           for p in "01":
                               str = c+"1000011"+f+"1101111"+g+"1100100"+h+"1101001"+i+"1101110"+L+"1000111"+o+"1100001"+p+"11
                               str = bintostr(str)
                               print str
```

运行程序容易发现,只有一个可见字符串:

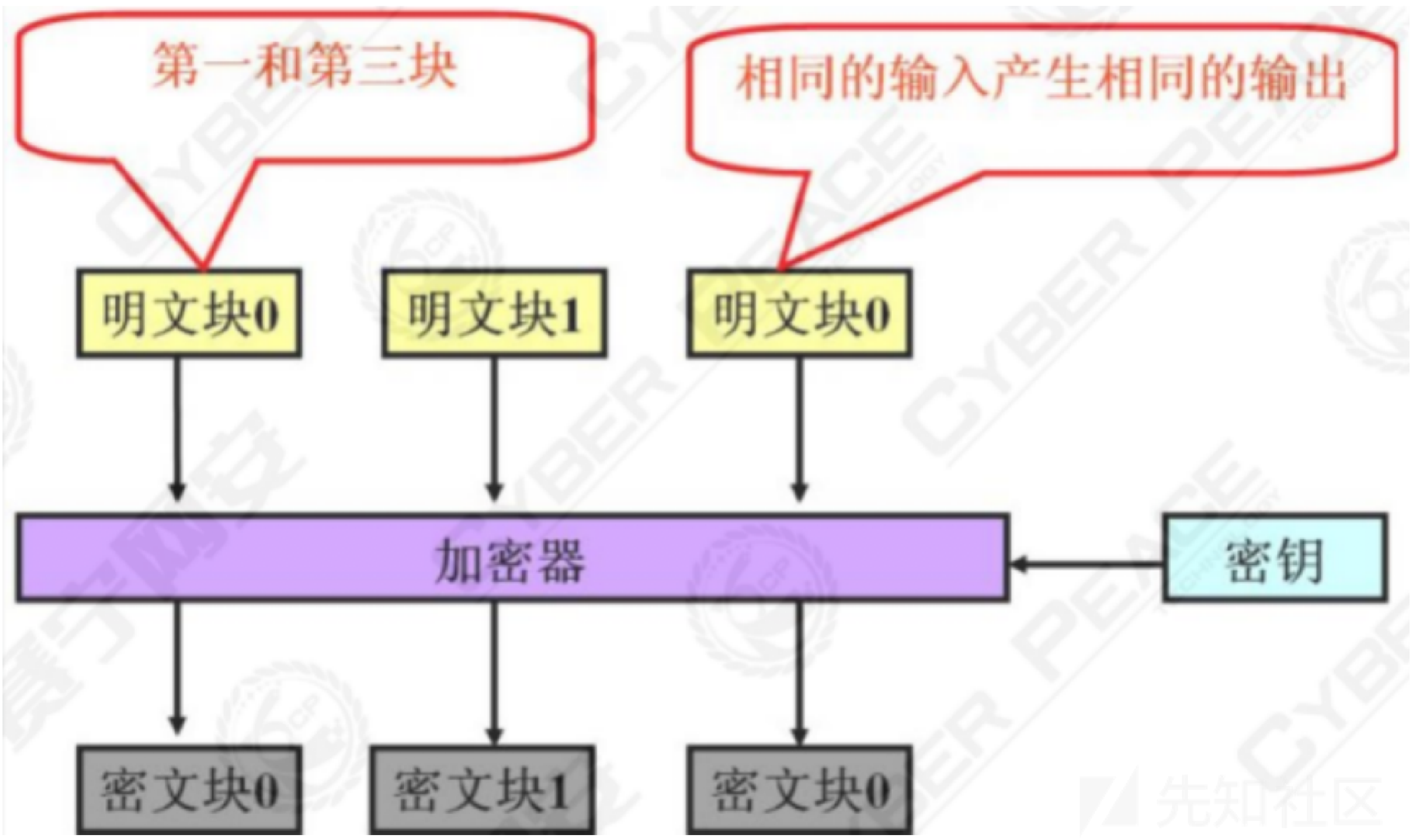
```
CodinGay
```

所以这一定是我们的deskey，后续的步骤就不再写出了，和这篇文章标题不符，也比较容易了，毕竟有Key，直接解就行了

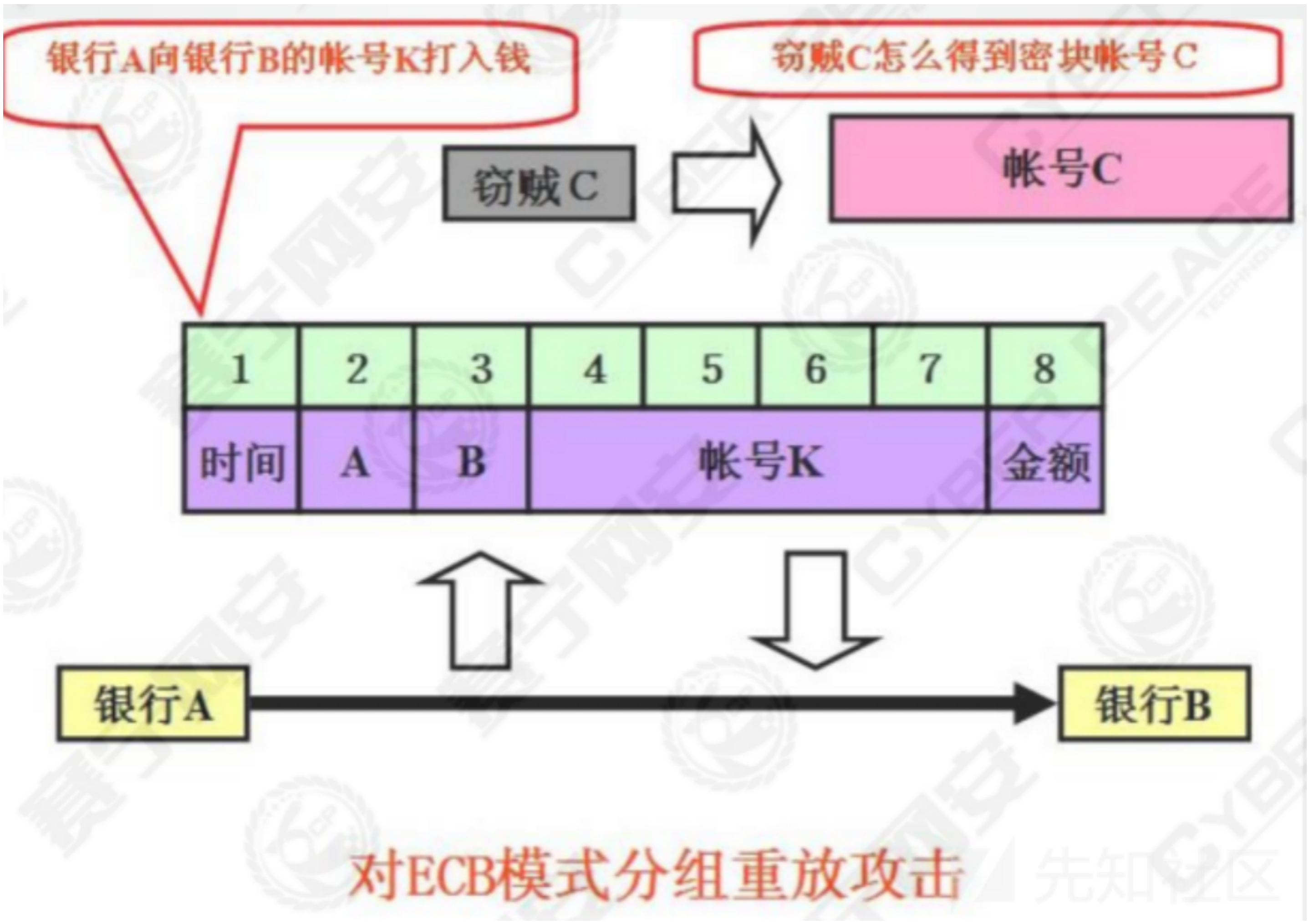
分组问题

ECB-Replay Attack

ECB模式相对来说，是最简单的一个模式，如图



其就是将明文分成多块，然后加密，那么这样就一定会存在重放攻击的危险，例如

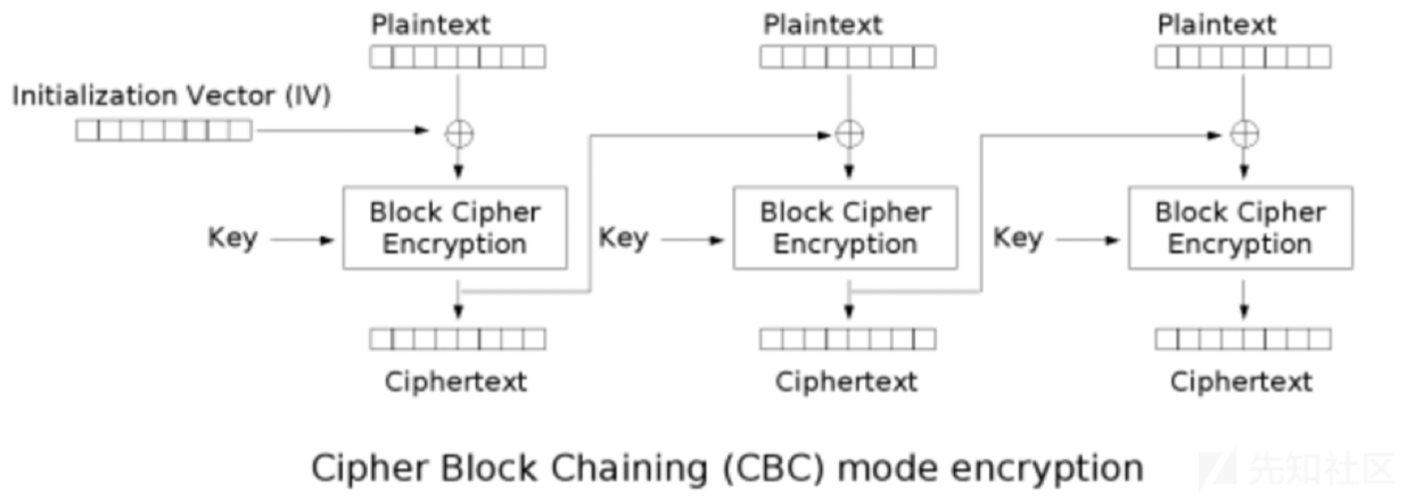


- 银行A给银行B传输信息使用ECB模式
- 窃贼C中间拦截下了该消息
- 窃贼C将密文中的一组替换成了自己的信息
- 导致银行B得到的消息被篡改

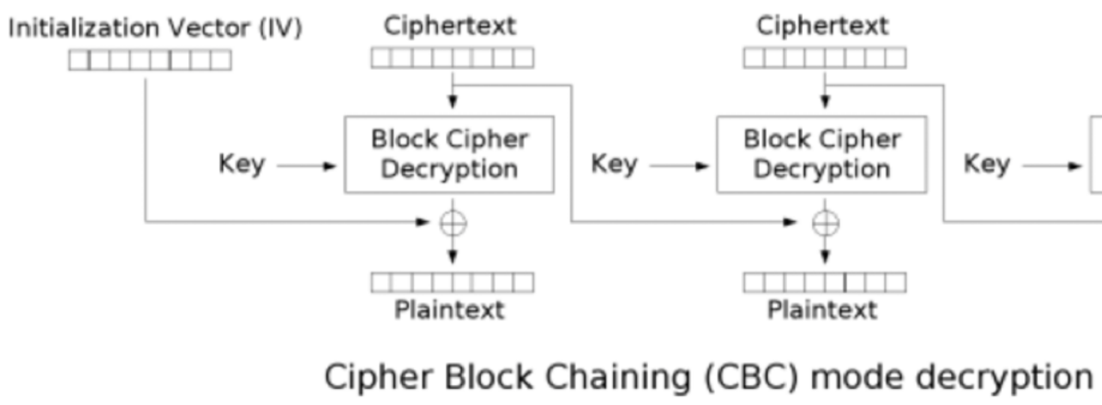
我们可以看到，该过程中，C对该密码算法的密钥是完全未知的，他可以向银行A或者银行B打钱，这样就能拥有密文，可以将其中的账号的密文组抠出来，用来替换
注：当然如果有签名校验就是另一回事了，这里我们单看这个模式的安全性

CBC-Padding Oracle Attack

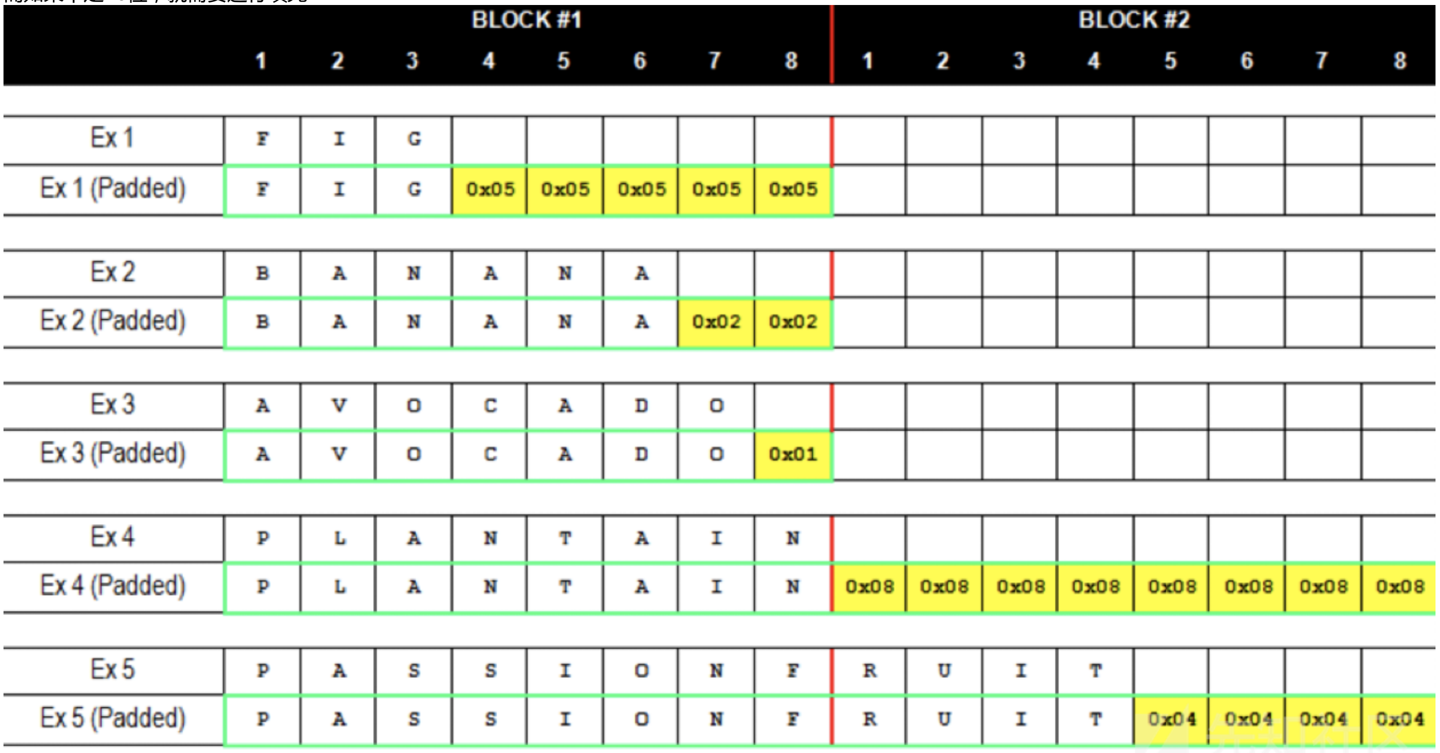
CBC加密模式：



CBC解密模式：



这里主要关注一下解密过程
密文cipher首先进行一系列处理，如图中的Block Cipher Decryption
我们将处理后的值称为middle中间值
然后middle与我们输入的iv进行异或操作
得到的即为明文
但这里有一个规则叫做Padding填充：
因为加密是按照16位一组分组进行的
而如果不足16位，就需要进行填充



	BLOCK #1								BLOCK #2							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Ex 1	F	I	G													
Ex 1 (Padded)	F	I	G	0x05	0x05	0x05	0x05	0x05								
Ex 2	B	A	N	A	N	A										
Ex 2 (Padded)	B	A	N	A	N	A	0x02	0x02								
Ex 3	A	V	O	C	A	D	O									
Ex 3 (Padded)	A	V	O	C	A	D	O	0x01								
Ex 4	P	L	A	N	T	A	I	N								
Ex 4 (Padded)	P	L	A	N	T	A	I	N	0x08	0x08	0x08	0x08	0x08	0x08	0x08	0x08
Ex 5	P	A	S	S	I	O	N	F	R	U	I	T				
Ex 5 (Padded)	P	A	S	S	I	O	N	F	R	U	I	T	0x04	0x04	0x04	0x04

比如我们的明文为admin
则需要被填充为 `admin\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b`
一共11个`\x0b`
如果我们输入一个错误的iv，依旧是可以解密的，但是middle和我们输入的iv经过异或后得到的填充值可能出现错误
比如本来应该是`admin\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b`
而我们错误的得到`admin\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x3b\x2c`
这样解密程序往往会抛出异常`(Padding Error)`
应用在web里的时候，往往是302或是500报错
而正常解密的时候是200
所以这时，我们可以根据服务器的反应来判断我们输入的iv
我们假设middle中间值为(为了方便，这里按8位分组来阐述)

```
0x39 0x73 0x23 0x22 0x07 0x6a 0x26 0x3d
```

正确的解密iv应该为

```
0x6d 0x36 0x70 0x76 0x03 0x6e 0x22 0x39
```

解密后正确的明文为：

```
T E S T 0x04 0x04 0x04 0x04
```

但是关键点在于，我们可以知道iv的值，却不能得到中间值和解密后明文的值
而我们的目标是只根据我们输入的iv值和服务器的状态去判断出解密后明文的值
这里的攻击即叫做`Padding Oracle Attack`
这时候我们选择进行爆破攻击
首先输入iv

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

这时候和中间值middle进行异或得到：

```
0x39 0x73 0x23 0x22 0x07 0x6a 0x26 0x3d
```

而此时程序会校验最后一位padding字节是否正确
我们知道正确的padding的值应该只有`0x01~0x08`，这里是`0x3d`，显然是错误的
所以程序会抛出500
知道这一点后，我们可以通过遍历最后一位iv，从而使这个iv和middle值异或后的最后一位是我们需要`0x01`
这时候有256种可能，不难遍历出
Iv:

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x3c
```

Middle:

```
0x39 0x73 0x23 0x22 0x07 0x6a 0x26 0x3d
```

两者异或后得到的是：

```
0x39 0x73 0x23 0x22 0x07 0x6a 0x26 0x01
```

这时候程序校验最后一位，发现是0x01，即可通过校验，服务器返回200
我们根据这个200就可以判断出，这个iv正确了
然后我们有公式：

```
Middle[8]^■■■iv[8] = plain[8]
Middle[8]^■■■iv[8] = 0x01
```

故此，我们可以算出

```
middle[8] = 0x01^■■■iv[8]
```

然后带入式1：

```
Plain[8] = 0x01^■■■iv[8]^■■■iv
```

即可获取明文`plain[8]= 0x01^0x3c^0x39=0x04`
和我们之前解密成功的明文一致（最后4位为填充）
下面我们需要获取plain[7]
方法还是如出一辙
但是这里需要将iv更新，因为这次我们需要的是2个0x02，而非之前的一个0x01
所以我们需要将■■■`iv[8] = middle[8]^0x02`
注：为什么是■■`iv[8] = middle[8]^0x02`？
因为■■■`iv[8]^middle[8]=`■■■■■■■
而我们遍历倒数第二位，应该是2个0x02，所以服务器希望得到的是0x02，所以

```
■■■iv[8]^middle[8]=0x02
■■iv[8] = middle[8]^0x02
```

然后再继续遍历现在的`iv[7]`
方法还是和上面一样，遍历后可以得到
Iv:

```
0x00 0x00 0x00 0x00 0x00 0x00 0x24 0x3f
```

Middle:

```
0x39 0x73 0x23 0x22 0x07 0x6a 0x26 0x3d
```

两者异或后得到的是：

```
0x39 0x73 0x23 0x22 0x07 0x6a 0x02 0x02
```

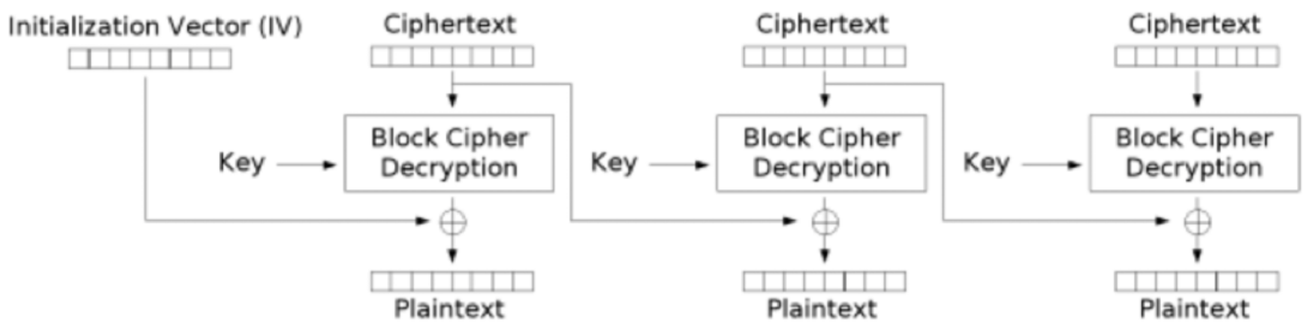
然后此时的明文值：

```
Plain[7]=■■■iv[7]^■■■iv[7]^0x02
```

所以`Plain[7] = 0x02^0x24^0x22=0x04`
和我们之前解密成功的明文一致（最后4位为填充）
最后遍历循环，即可得到完整的plain

CBC-Byte Flip Attack

这个实际上和padding oracle攻击差不多



Cipher Block Chaining (CBC) mode decryption

还是关注这个解密过程
但这时，我们是已知明文，想利用iv去改变解密后的明文
比如我们知道明文解密后是`1dmin`
我们想构造一个iv，让他解密后变成`admin`
还是原来的思路

```
■■■Iv[1]^middle[1]=plain[1]
```

而此时
我们想要

```
■■■iv[1]^mddle[1]=’a’
```

所以我们可以得到

```
■■■iv[1] = middle[1]^’a’
```

而

```
middle[1]=■■■iv[1]^plain[1]
```

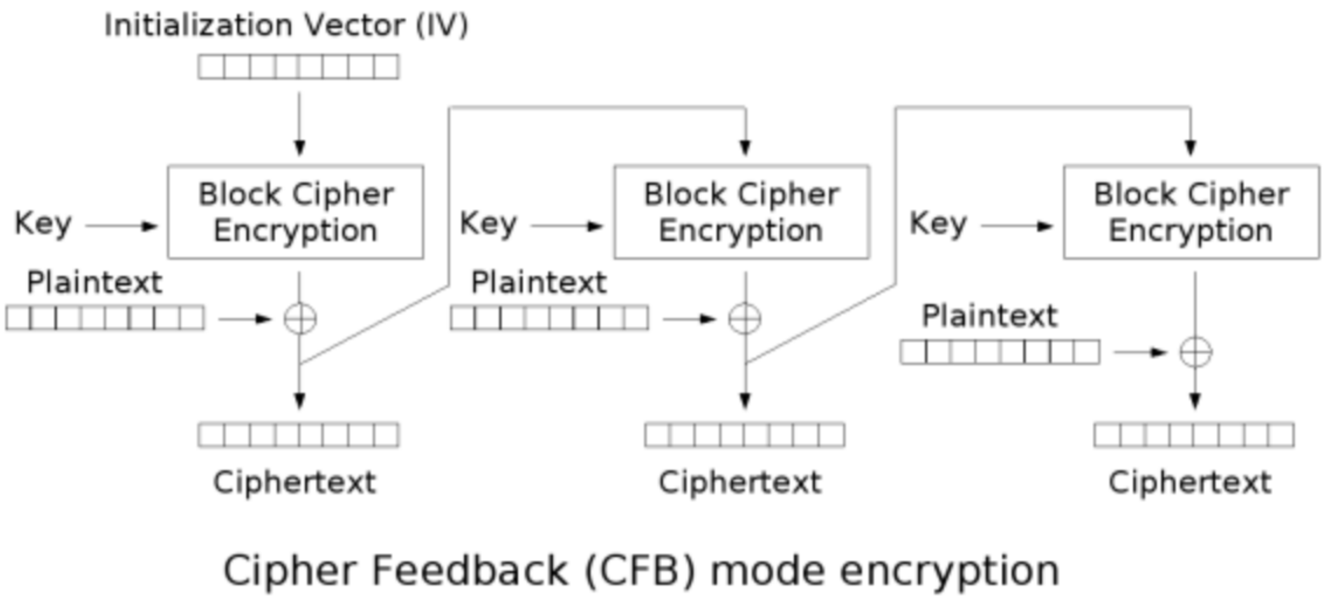
所以最后可以得到公式

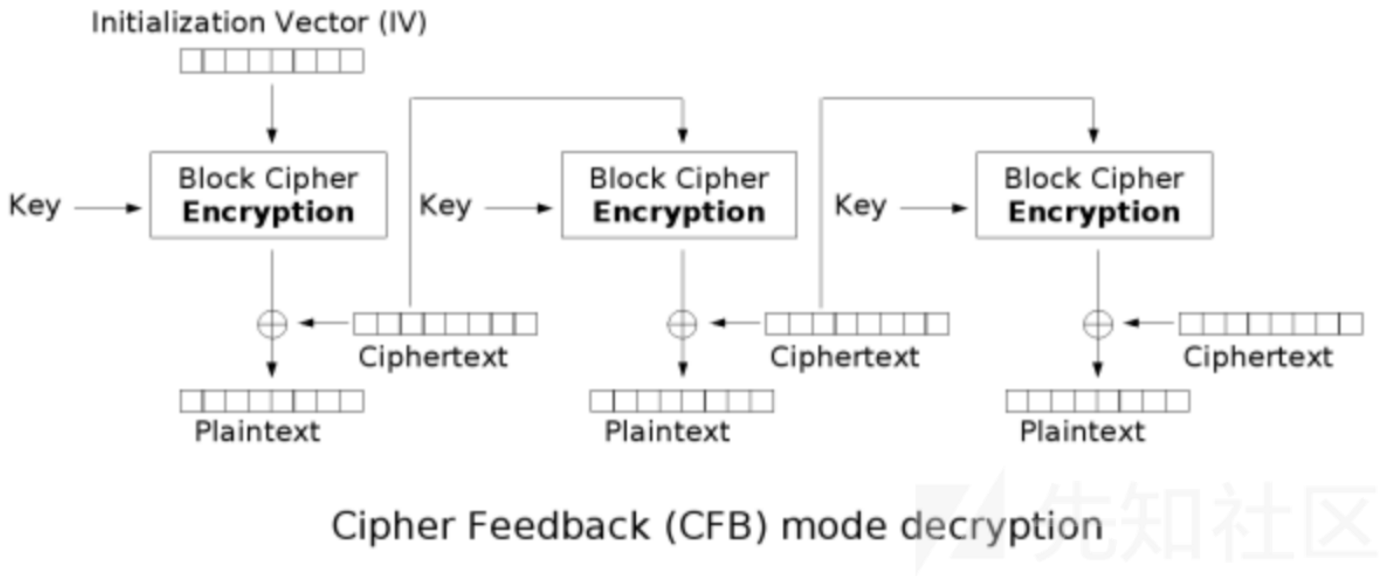
```
■■■iv[1]= ■■■iv[1]^plain[1]^’a’
```

所以即可造成数据的伪造
我们可以用这个式子，遍历明文，构造出iv，让程序解密出我们想要的明文

CFB-Replay Attack

CFB模式的加解密方式：


Initialization Vector (IV)
Key
Block Cipher Encryption
Plaintext
Ciphertext
Cipher Feedback (CFB) mode encryption


Initialization Vector (IV)
Key
Block Cipher Encryption
Ciphertext
Plaintext
Cipher Feedback (CFB) mode decryption

这里有一道例题写的比较详细，我就不再赘述：

`http://www.ifuryst.com/archives/AES_CFB_Attack.html`

后记

后面应该还会继续做一些补充和探索，因为目前写的头疼，所以先写到这里。
因为Crypto纯属兴趣，毕竟我是个web狗，若文章中有错误，还请指出：）

[上一篇：picoCTF2018 Write...]() [下一篇：[红日安全]代码审计Day16 -...]()