CTF-Crypto 2021-10-4 记录

原创

Ch1lkat 于 2021-10-04 16:58:30 发布 129 收藏

分类专栏： BUUCTF Crypto 文章标签： 算法 线性代数 概率论 密码学

文章目录

基础知识

CBC翻转攻击

简介：

> 当我们的一个值C是由A和B异或得到
> C = A XOR B
> 那么
> A XOR B XOR C很明显是=0的
> 当我们知道B和C之后，想要得到A的值也很容易
> A = B XOR C
> 因此，A XOR B XOR C等于0。有了这个公式，我们可以在XOR运算的末尾处设置我们自己的值，即可改变。

CBC-AES:**Encrypt(明文^IV)**

例如，已知CBC第一块（16位）

1. 明文
2. 密文

想直接修改第11位字符 `0`->`1`

> 已知第11位字符为`1`，我们可以直接将加密后的
> 密文[10]=密文[10] xor ‘0’ xor ‘1’
> 即可修改

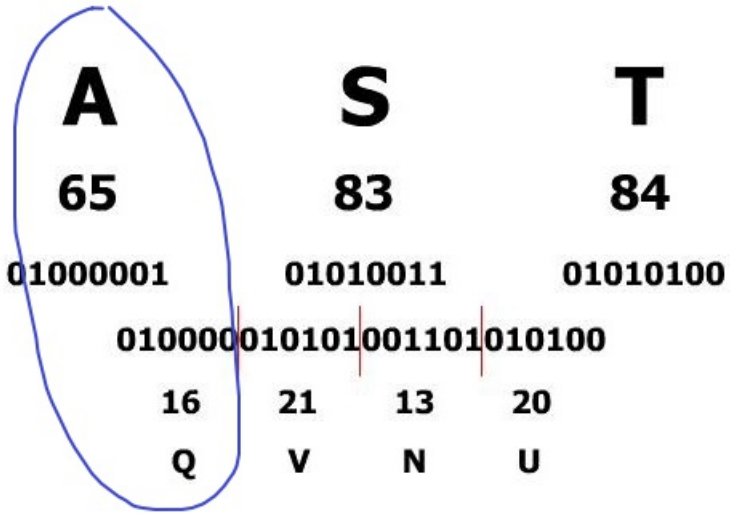
BASE64基本加密原理

A=>QQ==
a=>YQ==

> 97=110 0001=>右移2位 成6位=>1 1000==>查表(24)Y
> 上一个剩余的01二进制右移四位 010000=>查表(16)Q
> 填充（= =）
> 结果YQ==

1. 在BASE64中，通常是以3个字符(ASCII)开始编码为一组
2. 在转换为2进制中，又会以6位二进制分组(即一组:4*6=24)
3. 在空数据，以`=`作为填充字符

a			
1 1000	010000		
Y	Q	=	=




```
28
29      for(i = 0; i < part; i++)
30      {
31          in[0] = s[i*3], in[1] = s[i*3+1], in[2] = s[i*3+2];
32          add = len - i*3;
33
34          newc[i*4] = charset[in[0] >> 2];
35          newc[i*4+1] = charset[((in[0] & 0x3) << 4) | ((in[1] & 0xf0) >> 4)];
36          newc[i*4+2] = (unsigned char)(add > 1 ? charset[((in[1] & 0xf) << 2)
37          | ((in[2] & 0xc0) >> 6) ] : '=');
38          newc[i*4+3] = (unsigned char)(add > 2 ? charset[in[2] & 0x3f] : '=');
39      }
40      newc[i*4] = '\0';
```

A	0	Q	16	g	32	w	48
B	1	R	17	h	33	x	49
C	2	S	18	i	34	y	50
D	3	T	19	j	35	z	51
E	4	U	20	k	36	0	52
F	5	V	21	l	37	1	53
G	6	W	22	m	38	2	54
H	7	X	23	n	39	3	55
I	8	Y	24	o	40	4	56
J	9	Z	25	p	41	5	57
K	10	a	26	q	42	6	58
L	11	b	27	r	43	7	59
M	12	c	28	s	44	8	60
N	13	d	29	t	45	9	61
O	14	e	30	u	46	+	62
P	15	f	31	v	47	/	63

SpecialLCG

分析

MSSCTF2021-Crypto WriteUp

3. SpecialLCG:

相同的我们去分析题目，一开始assert了flag的长度是24位，并且用这个flag的前8位作为LCG的a,中间8位作为LCG的b,后8位作为LCG的c，并且我们是需要生成一个64位的素数n比a,b,c都大的，同时n也是已知的。seed1 和seed2都是随机生成的64位数 LCG内置中的初始化 state的初始状态是seed1和seed2。

最后给出的已知量是5个data和一个n。

那么我们可以列出以下式子：

$$y3 = (a * y2 + b * y1 + c)\%n$$

$$y4 = (a * y3 + b * y2 + c)\%n$$

$$y5 = (a * y4 + b * y3 + c)\%n$$

```python
from Crypto.Util.number import*

flag = xxxxxxxxxxxxxxxxxxxxxxxx
assert(len(flag) == 24)

class LCG:
    def __init__(self, seed1 , seed2):
        self.seed1 = seed1
        self.seed2 = seed2
        self.state = [seed1,seed2]
        self.n = getPrime(64)
        while 1:
            self.a = bytes_to_long(flag[:8])
            self.b = bytes_to_long(flag[8:16])
            self.c = bytes_to_long(flag[16:])
            if self.a < self.n and self.b < self.n and self.c < self.n:
                break
        print("n = " + str(self.n))

    def next(self):
        new = (self.a * self.state[-1] + self.b * self.state[-2] + self.c) % self.n
        self.state.append( new )
        return new

def main():
    seed1 = getRandomInteger(64)
    seed2 = getRandomInteger(64)
    lcg = LCG(seed1 , seed2)
    data = []
    for i in range(5):
        data.append(lcg.next())
    print("data = " + str(data))

if __name__ == "__main__":
    main()

'''
output:

n = 18253588106473969889
data = [8331802587873314500, 16970700310063771377, 16378474859328460142,
13073117282614811463, 74743330141643633]
'''
```

MSSCTF2021-Crypto WriteUp

3. SpecialLCG:

由于他是一个线性的我们就可以这样进行分析，之后得到的就是对应所得的a b c。

最后再把他们都拼接起来 就得到了最终的flag

flag: mssctf{w0w!y0u_knoW_lcg}

设

$$x4 = y5 - y4, x3 = y4 - y3, x2 = y3 - y2, x1 = y2 - y1 \tag{1}$$

于是

$$x4 = (a * x3 + b * x2)\%n, x3 = (a * x2 + b * x1)\%n \tag{2}$$

所以我们就可以得到

$$\frac{x4}{x2} = (a\frac{x3}{x2} + b)\%n, \frac{x3}{x1} = (a\frac{x2}{x1} + b)\%n \tag{3}$$

接着只需要用常规的乘法逆元就能得到a了

$$a = \frac{\frac{x4}{x2} - \frac{x3}{x1}}{\frac{x3}{x2} - \frac{x2}{x1}} \tag{4}$$

拿到了a之后我们就只需要代入（3）中任意一条我们就能拿到b了，

$$b = (\frac{x3}{x1} - a\frac{x2}{x1})\%n \tag{5}$$

c就相对更加简单了直接代入原来的表达式当中

$$c = (y5 - a * y4 - b * y3)\%n \tag{6}$$

笔记

`from Crypto.Util.number.inverse = gmpy2.invert 求逆元`

`bin(数值)，可求出数值的二进制`

- 不要想着现学现用，长期来看，慢慢去理解 不执著

EXP

```
from Crypto.Util.number import *

n=18253588106473969889
data=[8331802587873314500,16970700310063771377,16378474859328460142,13073117282614811463,74743330141643643 3]

t=[]

for i in range(4):
    t.append(data[i+1]-data[i])

a1=(t[2]*inverse(t[0],n)-t[3]*inverse(t[1],n))*inverse((t[1]*inverse(t[0],n)-t[2]*inverse(t[1],n)),n)%n
b1=(t[3]-a1*t[2])*inverse(t[1],n)%n
c1=(data[2]-data[1]*a1-data[0]*b1)%n

print(long_to_bytes(a1)+long_to_bytes(b1)+long_to_bytes(c1))
```

babyLCG

分析

a、b、m已知，求seed就可解出LCG

线性递归表达式：

`self._state = (self._key['a'] * self._state + self._key['b']) % self._key['m']`

```
14389363062759901320772309404495957196 8=
(10776326268249480919180302621301510180 2*x1+153582801876235638173762045261195852087)%22664963412624814184138871 2969771891297
```

跟题目去理解（LCG算法==>欧几里得拓展算法）

https://blog.csdn.net/superprintf/article/details/108964563

根据公式，还原seed（求出a,n的逆元，再套公式算seed）

还原seed的公式：`seed = (ani*(seed-b))%n`

```
a=107763262682494809191803026213015101802

n=226649634126248141841388712969771891297

b = 153582801876235638173762045261195852087

c =11267068470666042741<<64#old str

ani=invert(a,n)#
seed=c

print(ani)
print("seed:\n")

seed = (ani*(seed-b))%n#公式
print(seed)# 种子：22243527821180557867557087731905566211 9
```

能解出第一个seed，相关seed有什么联系？（下一个seed与上一个Seed无关）

[ACTF新生赛2020]crypto-aes

题目描述

```
from Cryptodome.Cipher import AES
import os
import gmpy2
from flag import FLAG
from Cryptodome.Util.number import *

def main():
    key=os.urandom(2)*16
    iv=os.urandom(16)
    print(bytes_to_long(key)^bytes_to_long(iv))
    aes=AES.new(key,AES.MODE_CBC,iv)
    enc_flag = aes.encrypt(FLAG)
    print(enc_flag)
if __name__=="__main__":
    main()

114419658666294256389576961430023234302669102942774706570738172862284907975 7
b'\x8c-\xcd\xde\xa7\xe9\x7f.b\x8aKs\xf1\xba\xc75\xc4d\x13\x07\xac\xa4&\xd6\x91\xfe\xf3\x14\x10|\xf8p'
```

分析

要大胆去猜，大胆去尝试

xor是一种可还原的算法

```
a^b=c
c^a=b
```

CBC特性: Ⅳ=前一块密文，明文需要先异或Ⅳ再加密

EXP

```
from Crypto.Cipher import AES
import os
import gmpy2
#from flag import FLAG
from Crypto.Util.number import *
XOR = lambda s1 , s2 : bytes([x1^x2 for x1,x2 in zip(s1,s2)])
def main():
    flag="flag{xxx}"

    oldxorcipher=b'\xc9\x81\xc9\x81\xc9\x81\xc9\x81\xc9\x81\xc9\x81\xc9\x81\xc9\x81N\xed\x98\xe3\x80\xb15gc\x84\
x990\xc8P\xb9\xcd'
    #泄漏16位明文key,后16被xor 是可还原的算法 xor

    cipher=b'\x8c-\xcd\xde\xa7\xe9\x7f.b\x8aKs\xf1\xba\xc75\xc4d\x13\x07\xac\xa4&\xd6\x91\xfe\xf3\x14\x10|\xf8p'
    iv=b'\x87lQbI0\xfc\xe6\xaa\x05P\xb1\x01\xd1pL'#long_to_bytes(bytes_to_long(key[:16])^bytes_to_long(oldxorcip
her[16:]))
    key=b"\xc9\x81"*16

    aes=AES.new(key,AES.MODE_CBC,iv)
    ans=aes.decrypt(cipher[:16])




    print(ans)#pre 16:actf{W0W_y0u_can

    aes=AES.new(key,AES.MODE_CBC,cipher[:16])#CBC特性 IV=前一块密文
    ans=aes.decrypt(cipher[16:])
    print(ans)#last 16:_so1v3_AES_now!}

    #actf{W0W_y0u_can_so1v3_AES_now!}


if __name__=="__main__":
    main()

# 911441965866629425638957696143002323430266910294277470657073817286228490797 57
# b'\x8c-\xcd\xde\xa7\xe9\x7f.b\x8aKs\xf1\xba\xc75\xc4d\x13\x07\xac\xa4&\xd6\x91\xfe\xf3\x14\x10|\xf8p'
```

StandardCBC

题目描述

```
from gmssl import sm4 #https://github.com/duanhongyi/gmssl
import socketserver
import signal
from flag import flag
import os
from base64 import *
import random
menu = '''1.enc;
2.dec;
3.getflag;
'''

XOR = lambda s1 , s2 : bytes([x1^x2 for x1,x2 in zip(s1,s2)])
def pad(m):
```

```
    padlen = 16 - len(m) % 16
    return m + padlen * bytes([padlen])
def unpad(m):
    return m[:-m[-1]]

def enc(iv , m , key):
    enc = sm4.CryptSM4(mode=sm4.SM4_ENCRYPT)
    enc.set_key(key = key , mode = sm4.SM4_ENCRYPT)
    c = enc.crypt_cbc(iv, m)
    return iv + c

def dec(iv , c , key):
    dec = sm4.CryptSM4(mode=sm4.SM4_DECRYPT)
    dec.set_key(key = key , mode = sm4.SM4_DECRYPT)
    m = dec.crypt_cbc(iv, c)
    return m

class server(socketserver.BaseRequestHandler):
    def _recv(self):
        data = self.request.recv(1024)
        return data.strip()

    def _send(self, msg, newline=True):
        if isinstance(msg , bytes):
            msg += b'\n'
        else:
            msg += '\n'
            msg = msg.encode()
        self.request.sendall(msg)

    def handle(self):
        signal.alarm(600)
        key = os.urandom(16)
        secret = os.urandom(random.randint(16 , 31))
        while 1:
            try:
                iv = os.urandom(16)
                self._send(menu)
                choice = self._recv()
                if choice == b'1':
                    self._send(b'your message:')
                    msg = b64decode(self._recv())
                    self._send(b64encode(enc(iv , msg + secret , key)))
                elif choice == b'2':
                    self._send('your ciphertext:')

                    c = b64decode(self._recv())
                    self._send('your iv:')
                    iv = b64decode(self._recv())

                    self._send(b64encode(dec(iv , c , key))[-1:])
                elif choice == b'3':
                    self._send('do you know my secret?')
                    guess = b64decode(self._recv())
                    if guess == secret:
                        self._send('congratulations')
                        self._send(flag)
                    else:
                        self._send('I know you can\'t know it')
                        break
```

```
                else:
                    self._send('wrong!')
                    break
            except:
                pass



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

if __name__ == "__main__":
    HOST, PORT = '0.0.0.0', 10001
    server = ForkedServer((HOST, PORT), server)
    server.allow_reuse_address = True
    server.serve_forever()
```

分析

MINIL的题目

aes(明文^iv)=cipher

第二块chunk enc(明文^上一块密文)

不用怀疑，是程序逻辑需要爆破，dec只返回倒数第一位

请先看清程序整体逻辑，再写代码~
不要[[过度简化]]，也不要想的太复杂

干就完了！

事实：

1. IV每一次都是不一样的
2. 解密只返回倒数1位的base64编码字符

难度有点高 暂时放弃

EXP

EXP选择了暴力破解思路
From XDSEC GITHUB:https://github.com/XDSEC/miniLCTF_2021/blob/main/WriteUps/H4n53r/H4n53r-TEAM.md

```
from pwn import *
from base64 import b64decode, b64encode
from Crypto.Util.number import long_to_bytes

def get_least_length():
    for i in range(1, 16):
        guess = b'\x00' * i
        c = b64decode(get_recv(guess))
        if i == 1:
            base = len(c)
        if len(c) != base:
            return base - 16 - i
```

```python
def get_recv(x):
    io.send(b'1')
    io.recvuntil(b':')
    io.send(b64encode(x))
    Res = io.recvuntil(b'flag;').decode().split('\n')
    return Res[1]

def get_message_last(c):
    guess = long_to_bytes(66) * 239
    for i in range(256):
        G = guess + long_to_bytes(i)
        io.send(b'2')
        io.recvuntil(b':')
        io.send(b64encode(G + c))
        io.recvuntil(b':')
        io.send(b64encode(IV))
        resp = io.recvuntil('flag;').decode().split('\n')[1]
        if resp == '':
            return i

if __name__ == "__main__":
    IV = b'\x00'*16
    LengTh = 0
    ciphertext = []
    M = [0]*17
    while LengTh == None or LengTh != 17:
        try:
            io = remote('0.0.0.0', 10001)
            io.recv()
            LengTh = get_least_length()
            print(LengTh)
        except:
            io.close()
    print('Get Length!!!')
    for i in range(16):
        pad = b'\x76' * 16 + (15 - i) * b'\x00'
        res = get_recv(pad)
        ciphertext.append(b64decode(res))
    print('Get Ciphertext!!!')
    i = 0
    for c in ciphertext:
        print(i)
        if i == 0:
            c16 = c[48:64]
            M[-1] = long_to_bytes(get_message_last(c16) ^ c[47])
        c16 = c[32:48]
        M[i] = long_to_bytes(get_message_last(c16) ^ c[31])
        i += 1
    m = b''.join(M)
    print('Get Message!!!')
    io.recv()
    io.send(b'3')
    print(io.recv())
    io.send(b64encode(m))
    print(io.recv())
    print(io.recv())
```

额外知识

[[Padding-Oracle]]
故名思义，Padding Oracle Attack背后的关键性概念便是加/解密时的填充（Padding）。明文信息可以是任意长度，但是块状加密算法需要所有的信息都由一定数量的数据块组成。为了满足这样的需求，便需要对明文进行填充，这样便可以将它分割为完整的数据块。

加密时可以使用多种填充规则，但最常见的填充方式之一是在PKCS#5标准中定义的规则。PCKS#5的填充方式为：明文的最后一个数据块包含N个字节的填充数据（N取决于明文最后一块的数据长度）。下图是一些示例，展示了不同长度的单词（FIG、BANANA、AVOCADO、PLANTAIN、PASSIONFRUIT）以及它们使用PKCS#5填充后的结果（每个数据块为8字节长）。

什么是PKCS#5？
PKCS#5，就是一种由RSA信息安全公司设计的填充标准。
对于PKCS#5标准来说，一般缺少几位，就填充几位那个数字。
比方说，在上面的例子里，我们有三位空缺，那么就要在空缺处都填上。这样，第二组的内容就变成了‘bc333’ 。

一个0x01（0x01）
两个0x02（0x02，0x02）
三个0x03（0x03，0x03，0x03）
四个0x04（0x04，0x04，0x04，0x04）
……
如果解密后的最后一个数据块末尾并非这些合法的字节序列，大部分加/解密程序都会抛出一个填充异常。这个异常对于攻击者尤为关键，它是Padding Oracle Attack的基础。