

# Cybrics CTF 2019 Quals Writeup Revrse

转载

[weixin\\_30502965](#) 于 2019-08-13 21:33:00 发布 106 收藏

文章标签: [java](#) [python](#)

原文链接: <http://www.cnblogs.com/helica/p/11348740.html>

版权

目录

- [Cybrics CTF 2019 Quals Writeup Revrse](#)
  - [Oldman Reverse](#)
  - [matreshka](#)
  - [Hidden Flag](#)

## Cybrics CTF 2019 Quals Writeup Revrse

上周末和天枢的大佬们一起打了 Cybrics 这个比赛，金砖国家 CTF 比赛，群里大佬们说是搬金砖 hhhh。场上做出了两个逆向题，这里做一下总结。

### Oldman Reverse

难度: Baby

链接: <https://cybrics.net/tasks/oldman>

题目给出了一段汇编代码，很短，简单分析一下就能确定代码流程。

```
.MCALL .TTYOUT, .EXIT
START:
    mov    #MSG r1
    mov    #0d r2
    mov    #32d r3
loop:
    mov    #MSG r1
    add    r2 r1
    movb   (r1) r0
    .TTYOUT
    sub    #1d r3
    cmp    #0 r3
    beq    DONE
    add    #33d r2
    swab   r2
    clrb  r2
    swab   r2
    br     loop
DONE:
    .EXIT

MSG:
    .ascii "cp33AI9~p78f8h1Ucsp0tKMQbxSKdq~^0yANxbnN)d}k&6eUNr66UK7Hsk_uFSb5#9b&PjV5_8phe7C#CLc#<Qsr0sb6{%N
.end START
```

这段代码加载 MSG 字符串到寄存器 r1 中，之后进一个循环，每轮循环输出 MSG[r2]。在每轮循环中，r2 先增加33，之后三个操作SWAB r2; CLRB r2; SWAB r2;不是太明白。经过一番百度 google，能够确定 SWAB 交换寄存器的字节顺序，CLRB 清除寄存器。可能没办法直接理解到这段代码的含义，不过结合 33 这个自增长度，还有 MSG 的总长是 256，基本就能猜到这是在模 256。写代码确认一下就得到了 flag。

```
msg = "cp33AI9~p78f8h1Ucsp0tKMQbxSKdq~^0yANxbnN)d}k&6eUNr66UK7Hsk_uFSb5#9b&PjV5_8phe7C#CLc#<Qsr0sb6{%NC8G|r

def loop(s):
    r1 = s
    r2 = 0
    r3 = 32
    ans = ''
    while True:
        ans += r1[r2]
        r3 -= 1
        if r3 <= 0:
            break
        r2 += 33
        r2 %= 256

    return ans

print len(msg)
print loop(msg)
```

```
cybrics{pdp_gpg_crc_dtd_bkb_php}
```

ps. CTF time 的 wp 说这是 PDP11 的汇编代码。

## matreshka

难度: Easy

链接: <https://cybrics.net/tasks/matreshka>

终于遇到了一道很典型的逆向题。拿到的是一个 .class 文件，拖进 jd-gui 直接能看到反编译的 java 代码。

```

public static void main(String[] paramArrayOfString)
    throws Exception
{
    String str = "matreha!";
    byte[] arrayOfByte1 = encode(System.getProperty("user.name").getBytes(), str);
    byte[] arrayOfByte2 = { 76, -99, 37, 75, -68, 10, -52, 10, -5, 9, 92, 1, 99, -94, 105, -18 };
    for (int i = 0; i < arrayOfByte2.length; i++) {
        if (arrayOfByte2[i] != arrayOfByte1[i])
        {
            System.out.println("No");
            return;
        }
    }
    File localFile = new File("data.bin");
    FileInputStream localFileInputStream = new FileInputStream(localFile);
    byte[] arrayOfByte3 = new byte[(int)localFile.length()];
    localFileInputStream.read(arrayOfByte3);
    localFileInputStream.close();
    byte[] arrayOfByte4 = decode(arrayOfByte3, System.getProperty("user.name"));
    FileOutputStream localFileOutputStream = new FileOutputStream("stage2.bin");
    localFileOutputStream.write(arrayOfByte4, 0, arrayOfByte4.length);
    localFileOutputStream.flush();
    localFileOutputStream.close();
}

```

分析一下代码的流程，首先加密 `System.getProperty("user.name")`，加密的值和 `arrayOfByte2` 对比，如果相同，就进入下一步，否则程序结束。之后解密 `data.bin`，将结果保存到 `stage2.bin`。`stage2.bin` 显然是下一关的程序。这里通过百度能确定，`System.getProperty("user.name")` 就是系统的用户名。

再看一下 `decode` 和 `encode` 函数，可以看到单纯地做了 DES 加解密。

```

public static byte[] decode(byte[] paramArrayOfByte, String paramString)
    throws Exception
{
    SecretKeyFactory localSecretKeyFactory = SecretKeyFactory.getInstance("DES");
    byte[] arrayOfByte1 = paramString.getBytes();
    DESKeySpec localDESKeySpec = new DESKeySpec(arrayOfByte1);
    SecretKey localSecretKey = localSecretKeyFactory.generateSecret(localDESKeySpec);
    Cipher localCipher = Cipher.getInstance("DES");
    localCipher.init(2, localSecretKey);
    byte[] arrayOfByte2 = localCipher.doFinal(paramArrayOfByte);
    return arrayOfByte2;
}

public static byte[] encode(byte[] paramArrayOfByte, String paramString)
    throws Exception
{
    SecretKeyFactory localSecretKeyFactory = SecretKeyFactory.getInstance("DES");
    byte[] arrayOfByte1 = paramString.getBytes();
    DESKeySpec localDESKeySpec = new DESKeySpec(arrayOfByte1);
    SecretKey localSecretKey = localSecretKeyFactory.generateSecret(localDESKeySpec);
    Cipher localCipher = Cipher.getInstance("DES");
    localCipher.init(1, localSecretKey);
    byte[] arrayOfByte2 = localCipher.doFinal(paramArrayOfByte);
    return arrayOfByte2;
}

```

拿 python 模拟一下，就拿到了 `stage2.bin`。注意这里的 DES 是 ECB 模式。

```

from Crypto.Cipher import DES
import binascii
import base64

key_1 = 'matreha!'
cipher = [76, -99, 37, 75, -68, 10, -52, 10, -5, 9, 92, 1, 99, -94, 105, -18]
cipher_s = ''
for x in range(len(cipher)):
    cipher_s += '%02x' % (cipher[x] % 256)

cipher_s = binascii.unhexlify(cipher_s)
cipherX = DES.new(key_1, DES.MODE_ECB)
y = cipherX.decrypt(cipher_s)

fp = open("data.bin", "rb")
fpp = open("steg2.bin", "wb")

key_2 = "lettreha"
print len(key_2)

cipherXX = DES.new(key_2, DES.MODE_ECB)
cipher = fp.read()
print len(cipher)

for cnt in range(len(cipher) / 16):
    res = cipherXX.decrypt(cipher[cnt*16:cnt*16+16])
    fpp.write(res)

fp.close()
fpp.close()

```

stage2.bin 是一个 elf 文件，拖进 IDA 会发现特别混乱。做过这种题就能知道这是 go 语言的二进制程序，第一次的同学百度一下反编译出的函数名也就明白了。

go 的二进制程序主函数在 main\_main 中，反编译的结果虽然不是很清晰，但是能确定做了 RC4 加密，之后对比加密字符串，如果正确就进入下一步。下一步对另一段做 RC4 解密，解密结果保存在文件当中。

```

60 v8 = __readfsqword(0xFFFFFFFF8);
61 if ( (unsigned __int64)&v57 + 9 <= *(_QWORD *)(v8 + 16) )
62     runtime_morestack_noctxt(a1, a2, a3, v8, a5);
63 v56 = main_statictmp_0;
64 *(_QWORD *)&v57 = main_statictmp_1;
65 *(__int128 *)((char *)&v57 + 1) = *(_OWORD *)((char *)&main_statictmp_1 + 1);
66 *(_QWORD *)&key_v9 = &v56;
67 *(_QWORD *)&key_v9 + 1 = 8LL;
68 crypto_rc4_NewCipher(a1, a2, a3, v8, a5, a8, key_v9);
69 v62 = (_BYTE *)v50;
70 path_filepath_Dir(a1, a2, qword_52DE48, os_executablePath, v10, v11, os_executablePath, qword_52DE48);
71 path_filepath_Base(a1, a2, v12, 8LL, v13, v14, (char *)8, v50);
72 runtime_stringtoslicebyte(a1, a2, v15, 8LL, v16, v17);
73 v58 = v50;
74 runtime_makeslice(a1, a2, v54, v52);
75 v18 = v52;
76 v60 = v50;
77 crypto_rc4___Cipher___XORKeyStream(
78     a1,
79     a2,
80     v50,
81     v54,
82     v19,
83     v20,

```

逆向到这里，函数参数的分析就很头疼了。很难一眼就确定 RC4 的密钥、明文是什么，这个时候动态调试往往能事半功倍。

打开我无敌的 GDB，先在函数入口处下断点，断点下在 main.main 上，注意 IDA 反编译会把函数名的 "." 转成 "\_"。main.main 函数一开始有很多跳转，直接下断点到 RC4 异或加密的地方。这下能够很明显的看出，第一步 RC4 加密的明文是当前目录的名字。同样的方法也能确定 RC4 的密钥。

```
[-----code-----]
0x476109 <main.main+345>:  mov    QWORD PTR [rsp+0x28],rcx
0x47610e <main.main+350>:  mov    rcx,QWORD PTR [rsp+0x48]
0x476113 <main.main+355>:  mov    QWORD PTR [rsp+0x30],rcx
=> 0x476118 <main.main+360>:  call   0x45e340 <crypto/rc4.(*Cipher).XORKeyStream>
0x47611d <main.main+365>:  mov    rax,QWORD PTR [rsp+0x60]
0x476122 <main.main+370>:  cmp    rax,0x11
0x476126 <main.main+374>:  jne    0x4762ea <main.main+826>
0x47612c <main.main+380>:  mov    rax,QWORD PTR [rsp+0x98]
No argument
[-----stack-----]
0000| 0xc0000366d0 --> 0xc0000a0000 --> 0xe200000041
0008| 0xc0000366d8 --> 0xc0000a2020 --> 0x0
0016| 0xc0000366e0 --> 0x11
0024| 0xc0000366e8 --> 0x11
0032| 0xc0000366f0 --> 0xc0000a2000 ("kroshka_matreshka")
0040| 0xc0000366f8 --> 0x11
0048| 0xc000036700 --> 0x20 (' ')
0056| 0xc000036708 --> 0x1
Legend: code, data, rodata, value
Thread 1 "steg2.bin" hit Breakpoint 2, 0x000000000476118 in main.main () at /home/awengar/Distr/Hacks/ORG/SPBCTF/CyBrics/rev100_matreshka/2.go:22
22 in /home/awengar/Distr/Hacks/ORG/SPBCTF/CyBrics/rev100_matreshka/2.go
gdb-peda$
```

写脚本解密 RC4 的密文，得到正确的目录名，之后运行程序就自动解密出了下一关的代码 result.pyc。

```
import random, base64
from hashlib import sha1

def rc4_xor(data, key):
    """RC4 algorithm"""
    x = 0
    box = range(256)
    for i in range(256):
        x = (x + box[i] + ord(key[i % len(key)])) % 256
        box[i], box[x] = box[x], box[i]
    x = y = 0
    out = []
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))

    return ''.join(out)

key = '\x01\x03\x03\x07\x03\x03\x08\x01'
data = '\x53\xdd\xc5\x87\xe4\x63\x99\x14\x4f\xa4\x14\x2d\xc4\x24\x04\xc0\xb0'

print rc4_xor(data, key)
```

拿到 result.pyc 后，反编译的结果却很诡异。

```

def decode(data, key):
    idx = 0
    res = []
# WARNING: Decompile incomplete

flag = [
    40,
    11,
    82,
    58,
    93,
    82,
    64,
    76,
    6,
    70,
    100,
    26,
    7,
    4,
    123,
    124,
    127,
    45,
    1,
    125,
    107,
    115,
    0,
    2,
    31,
    15]
print('Enter key to get flag:')
key = input()
if len(key) != 8:
    print('Invalid len')
    quit()
res = decode(flag, key)
# WARNING: Decompile incomplete

```

代码用输入的8位 **key** 和 **flag** 做某些操作，之后的结果应该就是 **flag**。最关键的 **decode** 函数反编译失败，是工具出了问题吗？用 **010Editor** 再去看 **result.pyc**，发现它本来就是不完整的。

这个时候只能先猜了，最简单的就是逐位异或了，而且我们知道**flag**的前几位是 "cybrics{"，正好是8位。拿 "cybrics{" 和 **flag** 的前8位异或，结果是 "Kr0H4137"，显然我们猜对了。最后异或整个**flag**，得到答案。

```

import binascii

flag = [
    40,
    11,
    82,
    58,
    93,
    82,
    64,
    76,
    6,
    70,
    100,
    26,
    7,
    4,
    123,
    124,
    127,
    45,
    1,
    125,
    107,
    115,
    0,
    2,
    31,
    15]

fake_flag = "cybrics{"
print len(flag)

for x in range(len(flag)):
    print "%02x" % (flag[x]),
print ""

print binascii.hexlify(fake_flag)

for i in range(8):
    print chr(flag[i] ^ ord(fake_flag[i]) ),

def decode(key, data):
    ans = ''
    for x in range(len(data)):
        data[x] ^= ord(key[x % len(key)])
        ans += chr(data[x])
    return ans

print decode("Kr0H4137", flag)

```

```
cybrics{M4TR35HK4_15_B35T}
```

## Hidden Flag

难度: hard

链接: <https://cybrics.net/tasks/hiddenflag>

TODO

转载于:<https://www.cnblogs.com/helica/p/11348740.html>