Cyber Attacks & Defense

Return-oriented Programming

Yeongjin Jang

Oregon State University

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 in 2014

- Stack-cookie + DEP + ASLR
  - All enabled in Windows, Linux, MacOS, Android, iOS, etc..
  - But, ASLR was not enabled to program code
    - 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
- Let's bypass them even if they are somewhat combined

DEP + ASLR

- DEP-1, DEP-2, DEP-3
- Your exploit was returning to library functions such as
  - 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?

Two types of ASLR

- Partial-ASLR (mostly, program's code is not randomized)
  - We call this non-PIE (Position Independent Executable)
  - Your program's code 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.

- Full-ASLR with PIE (Position Independent Executable)
  - Your program's code will also be randomized in each time of execution

```
PIE:        PIE enabled
```

Breaking DEP + ASLR

- We will learn how we can break DEP+ASLR step by step
- We will first tackle challenges that does not randomize the program's code section
  - Yes, you can use those functions!
- Let's learn how to get a shell in such a case

Getting a Privileged Shell

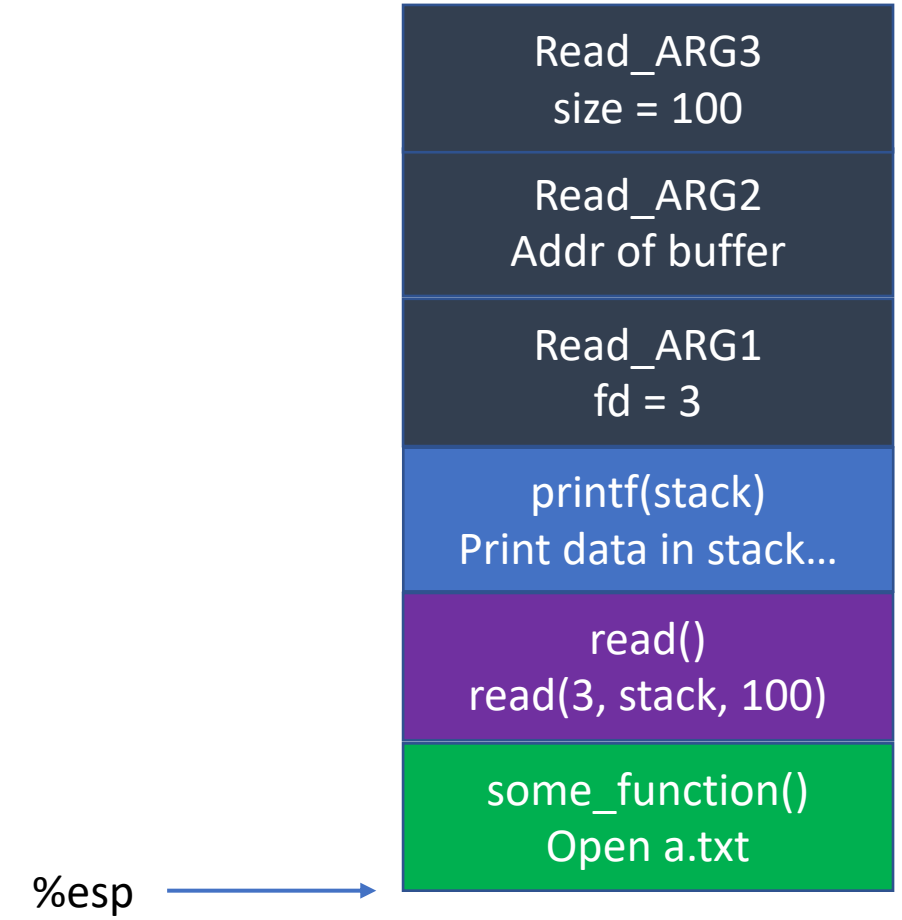
- rop-1-32 and rop-1-64
  - setregid(50000, 50000);
  - execve("/bin/sh", 0, 0);

Really?

  - If the program contains those two functions, we can easily call them.

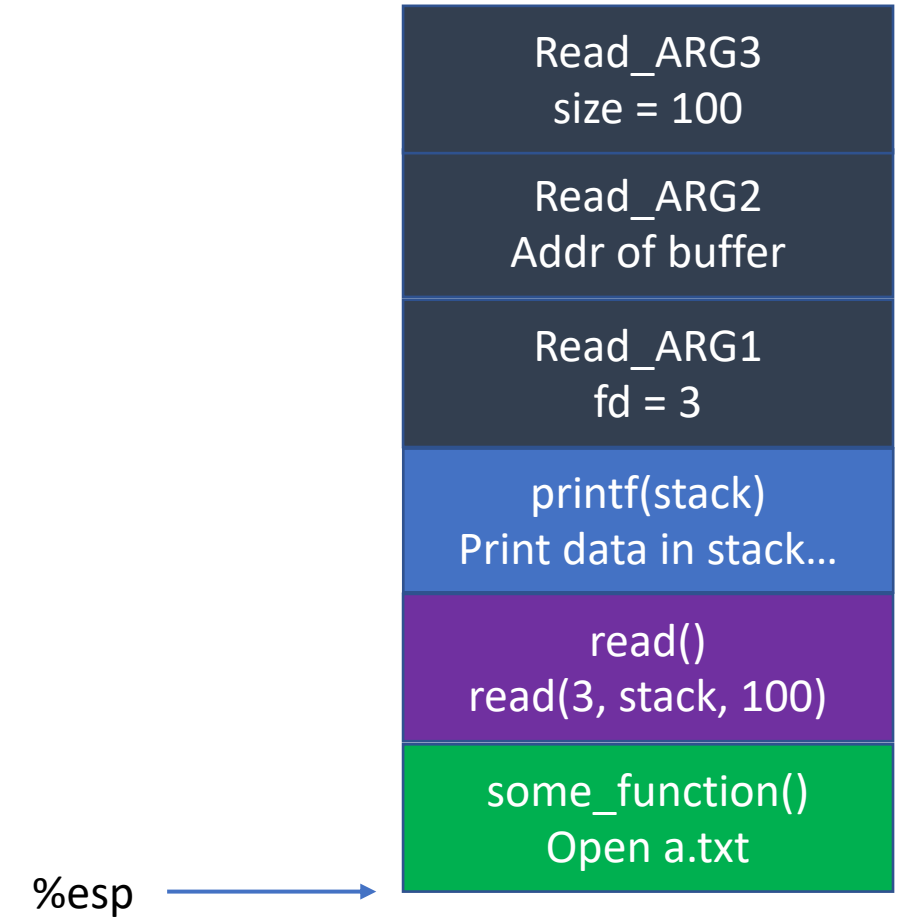
```
$ gdb ./rop-1-32
pwndbg: loaded 177 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./rop-1-32...(no debugging symbols found)...done.
pwndbg> info functions
All defined functions:

0x080483c0  execve@plt
0x080483d0  prctl@plt
0x080483e0  setregid@plt
```

Chaining Function Calls in DEP-3



Chaining Function Calls in DEP-3

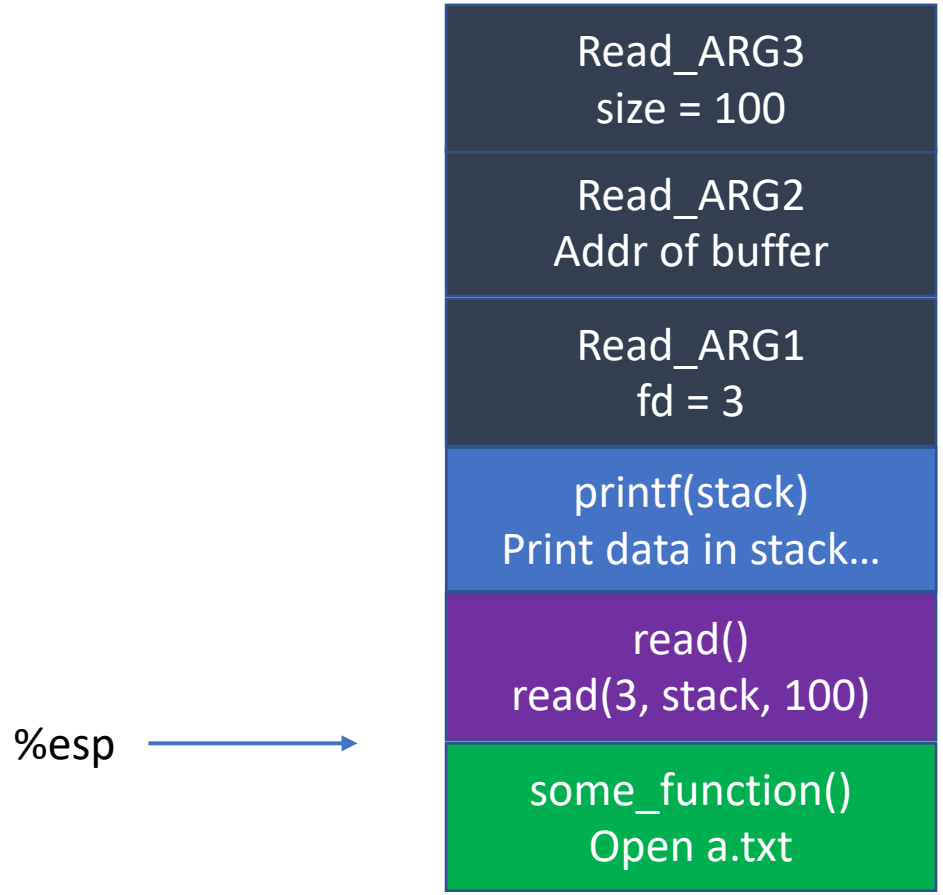
Return!



Chaining Function Calls in DEP-3

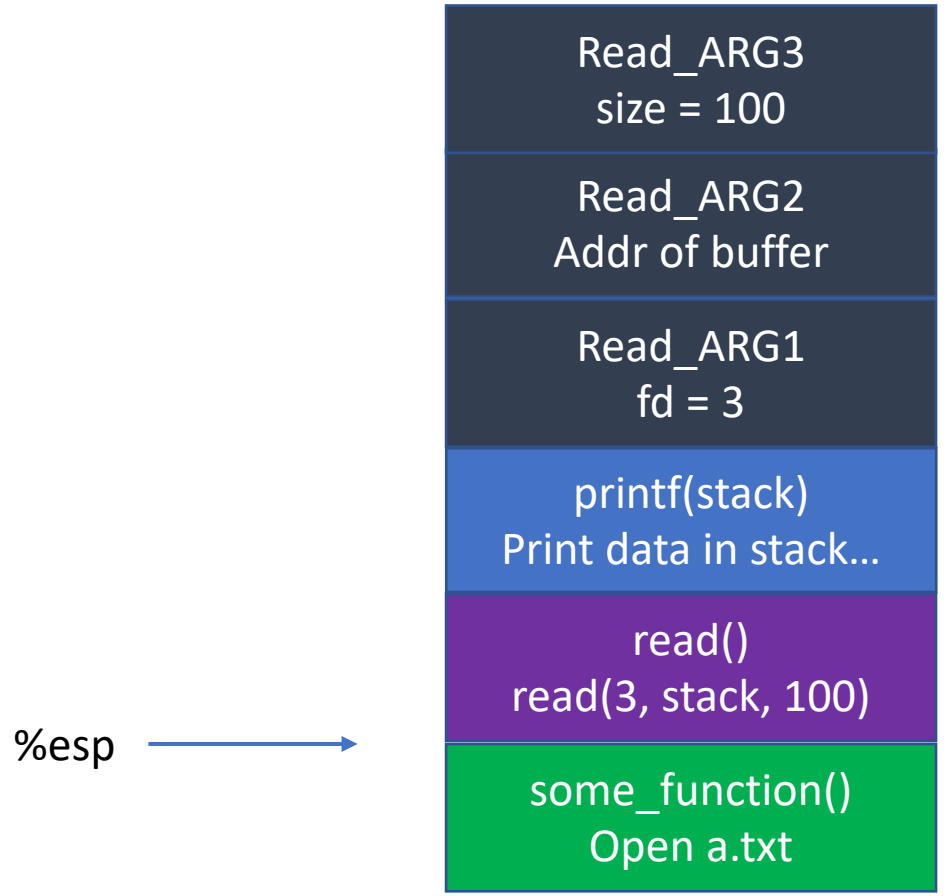
Return!

Execute some_function()



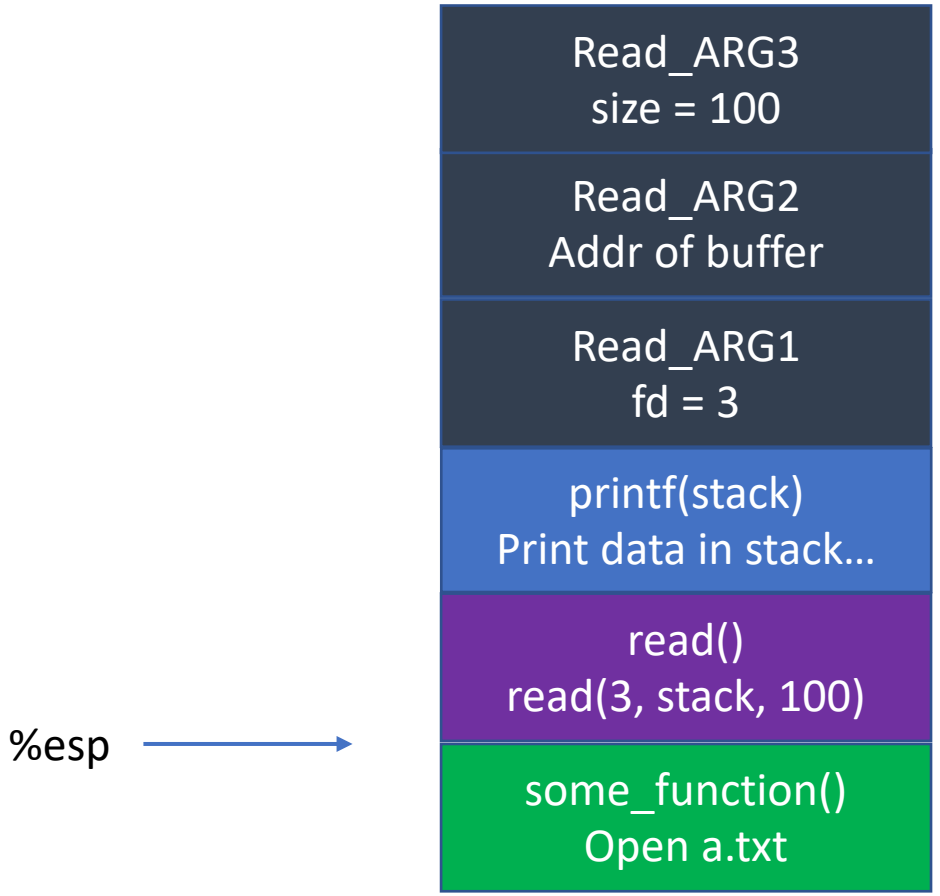
Chaining Function Calls in DEP-3

```
Head of some_function

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



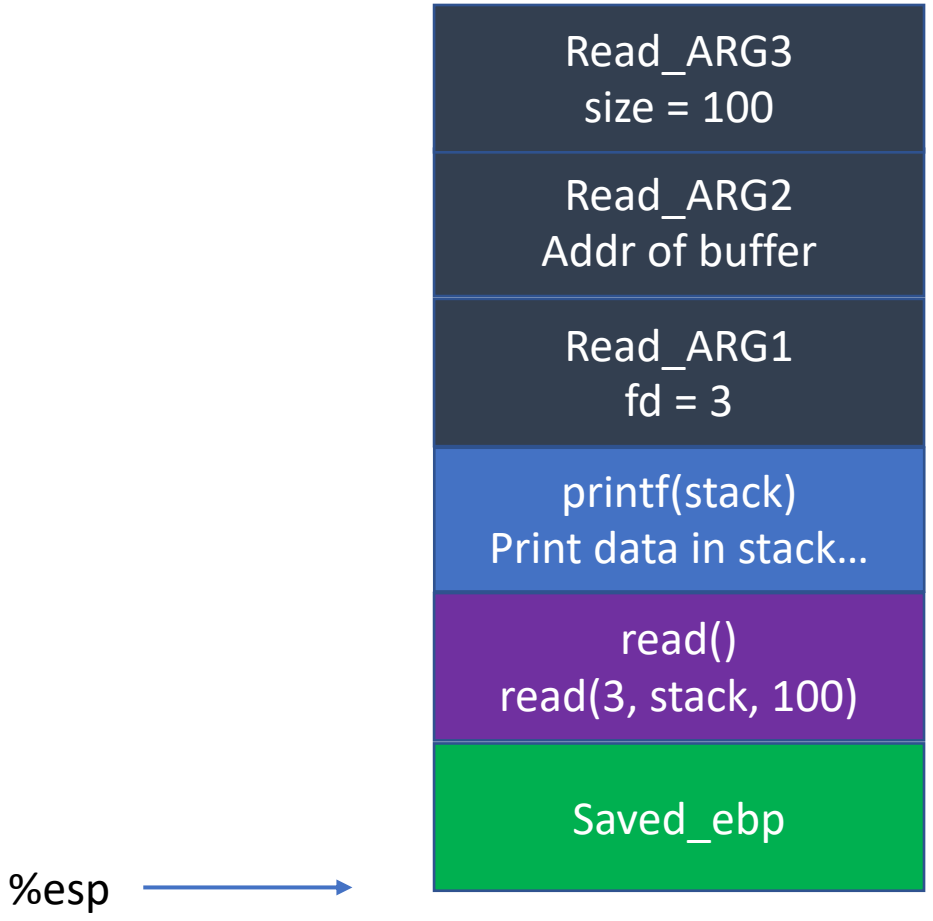
Chaining Function Calls in DEP-3

```
Head of some_function

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



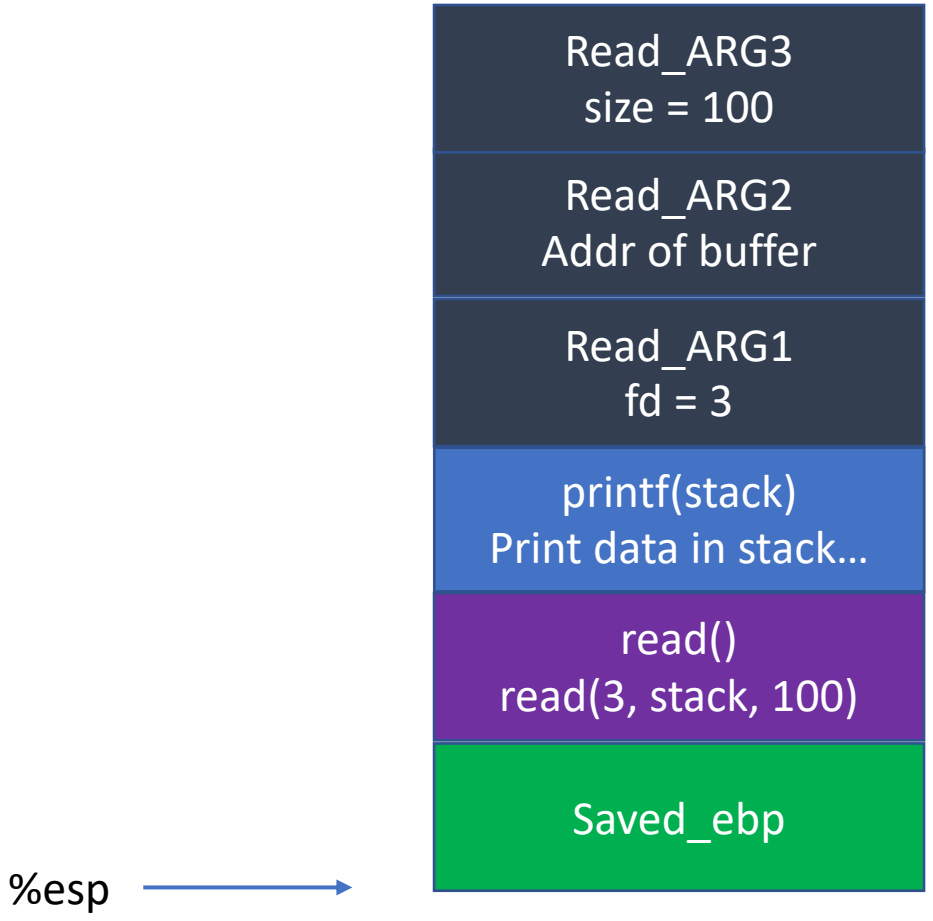
Chaining Function Calls in DEP-3

```
Head of some_function

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



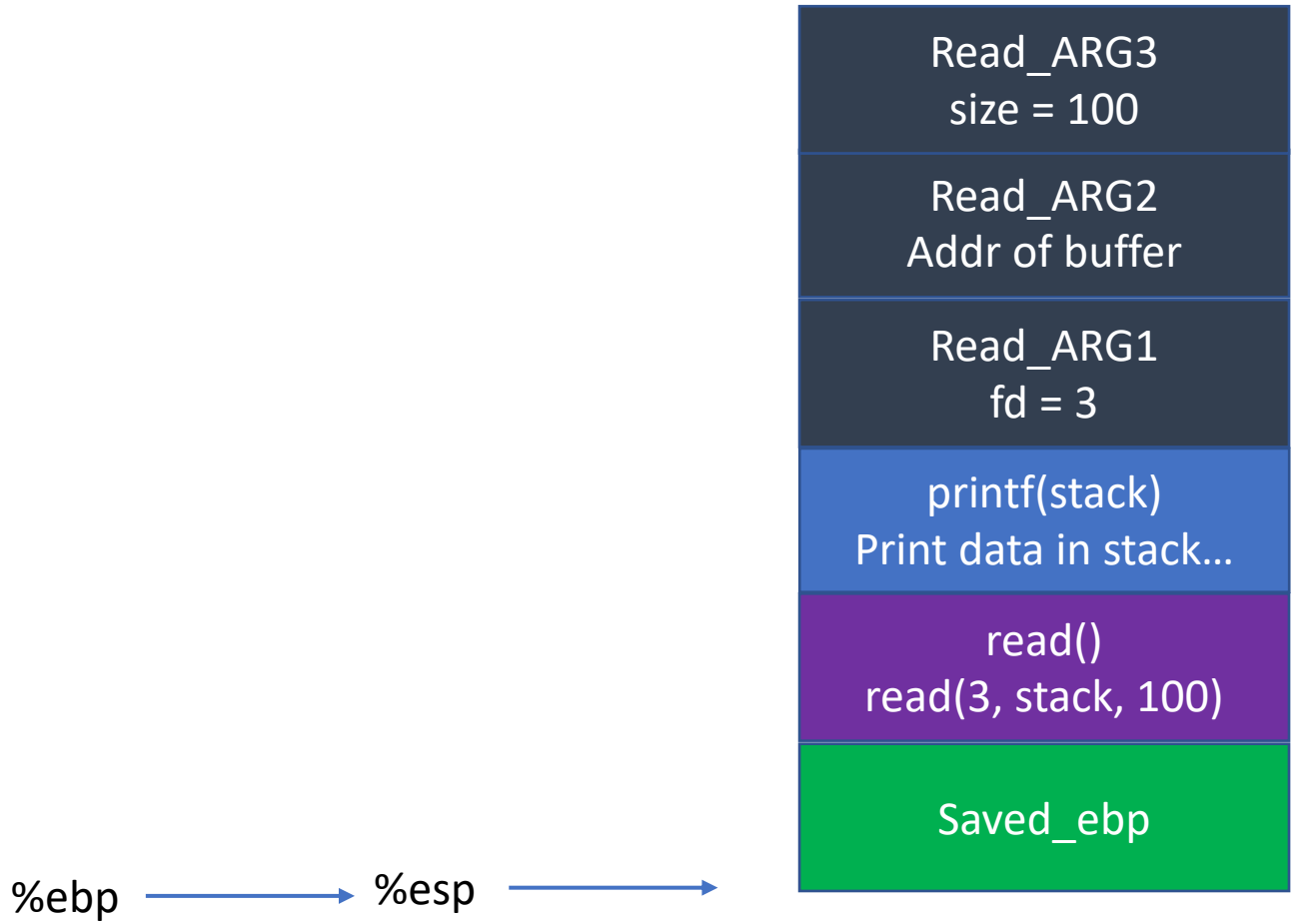
Chaining Function Calls in DEP-3

```
Head of some_function

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



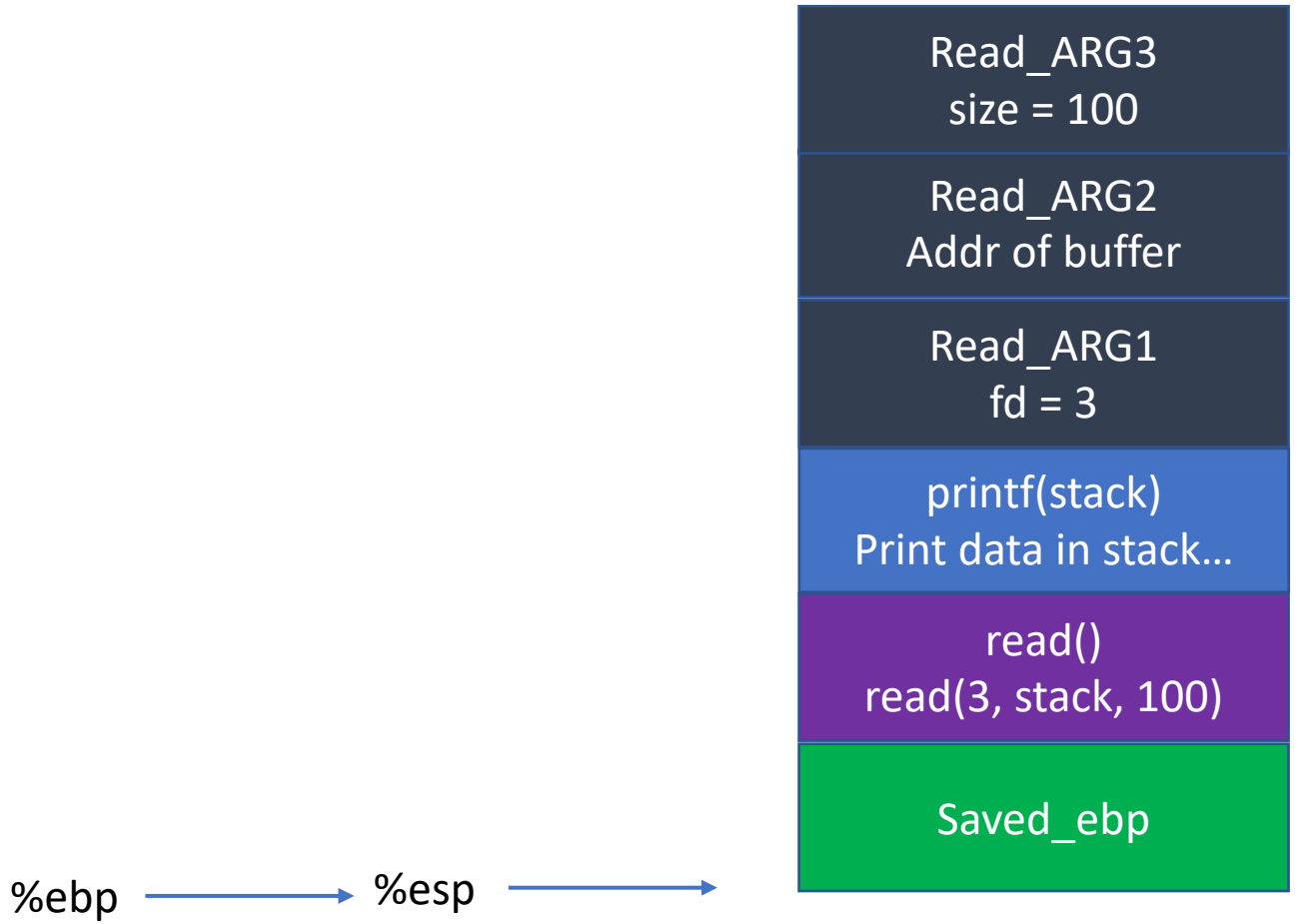
Chaining Function Calls in DEP-3

```
Head of some_function

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



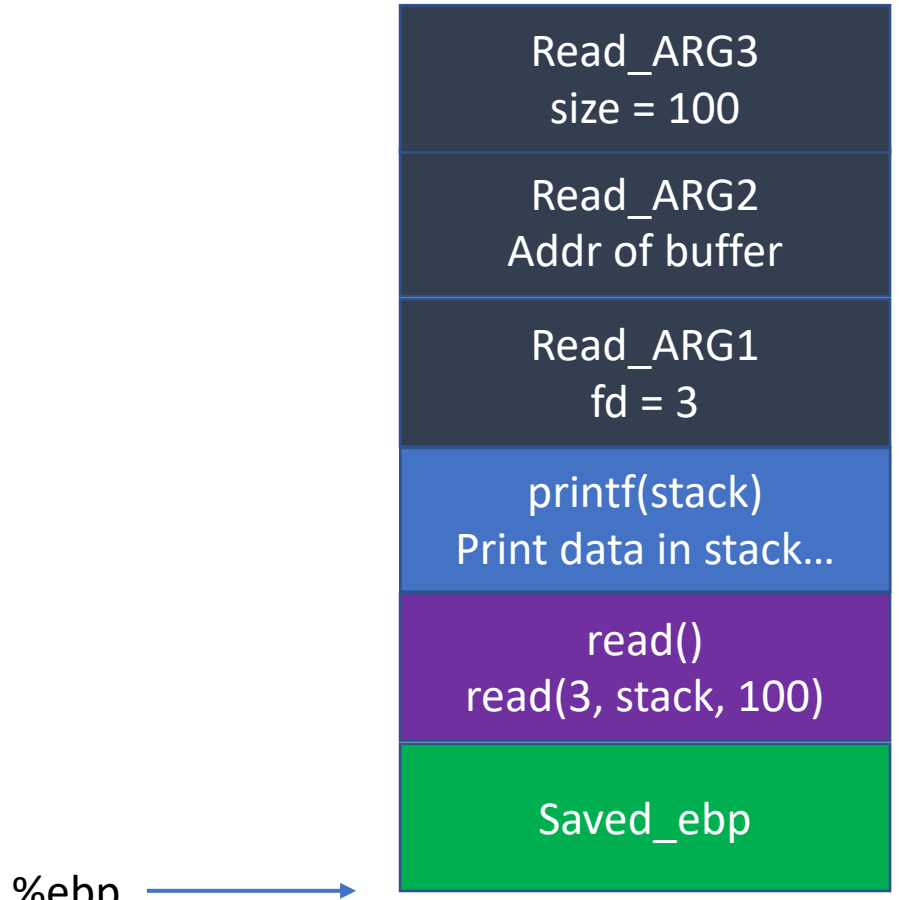
Chaining Function Calls in DEP-3

```
Head of some_function

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



Chaining Function Calls in DEP-3

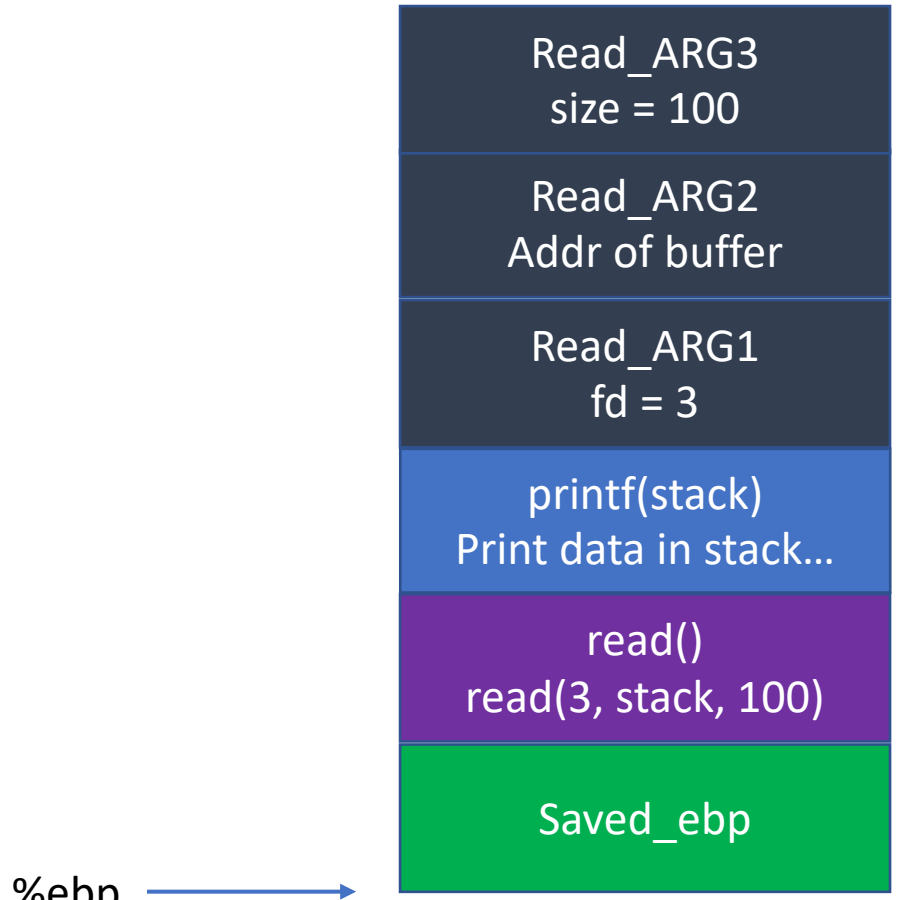
```
Head of some_function

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



Oregon State University

Chaining Function Calls in DEP-3

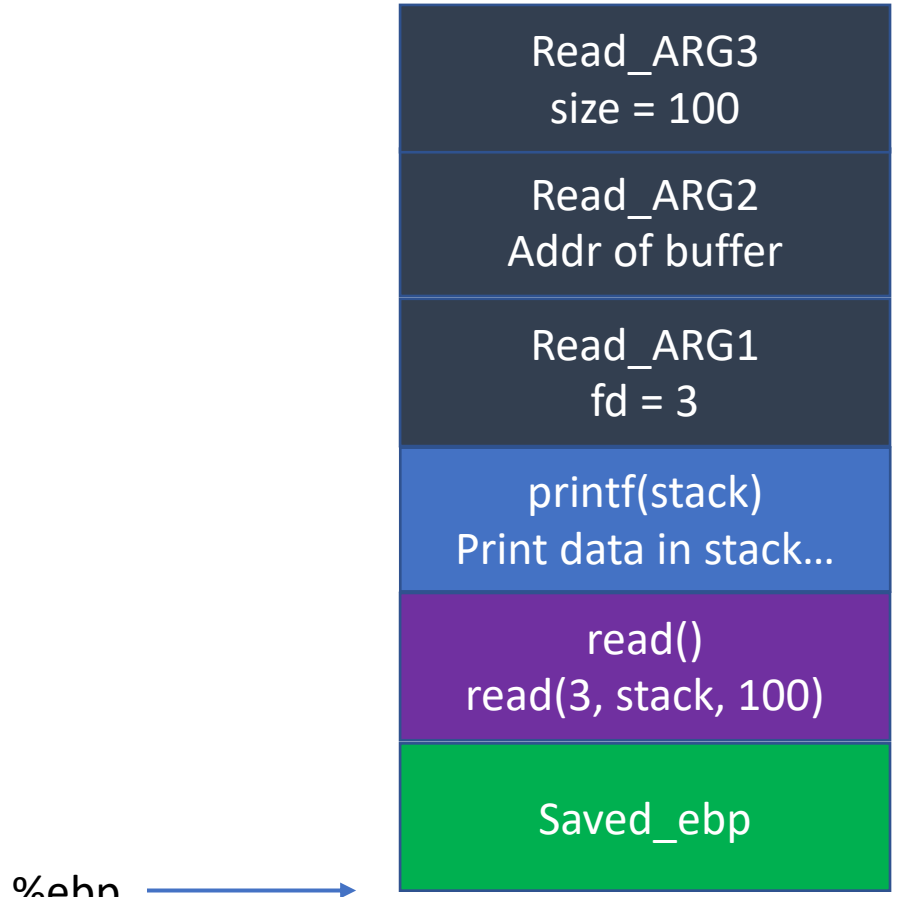
```
End of some_function

leave
ret
```



Oregon State University

Chaining Function Calls in DEP-3

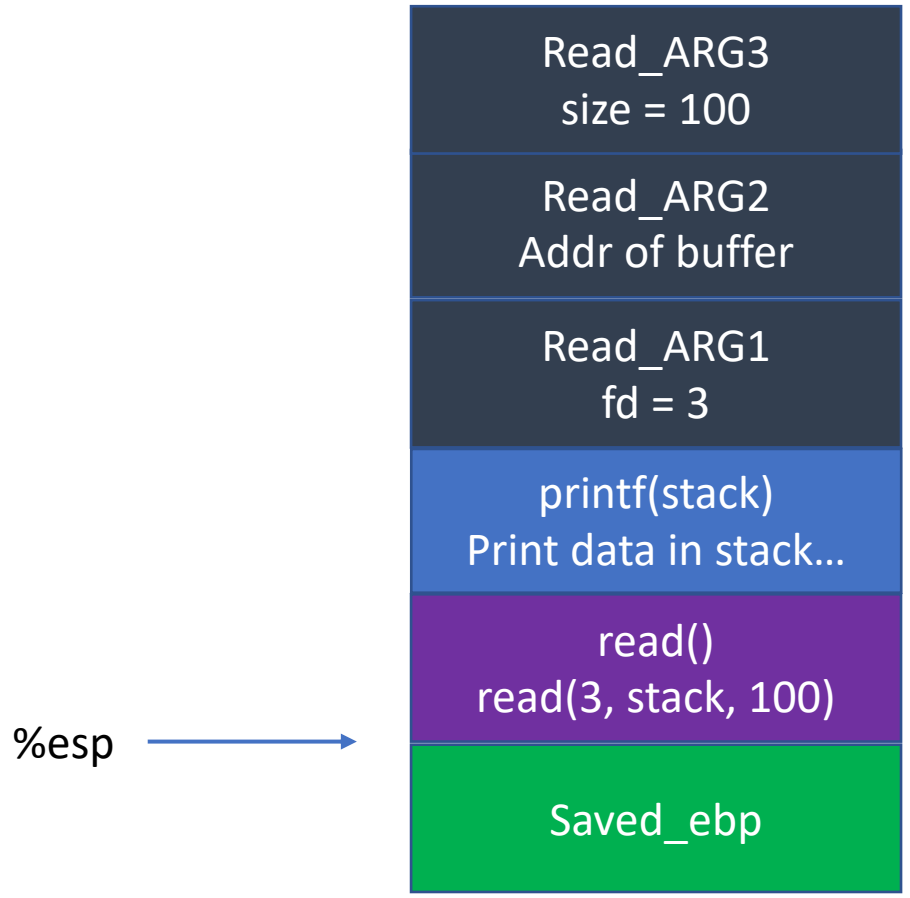
```
End of some_function

leave
ret
```



Oregon State University

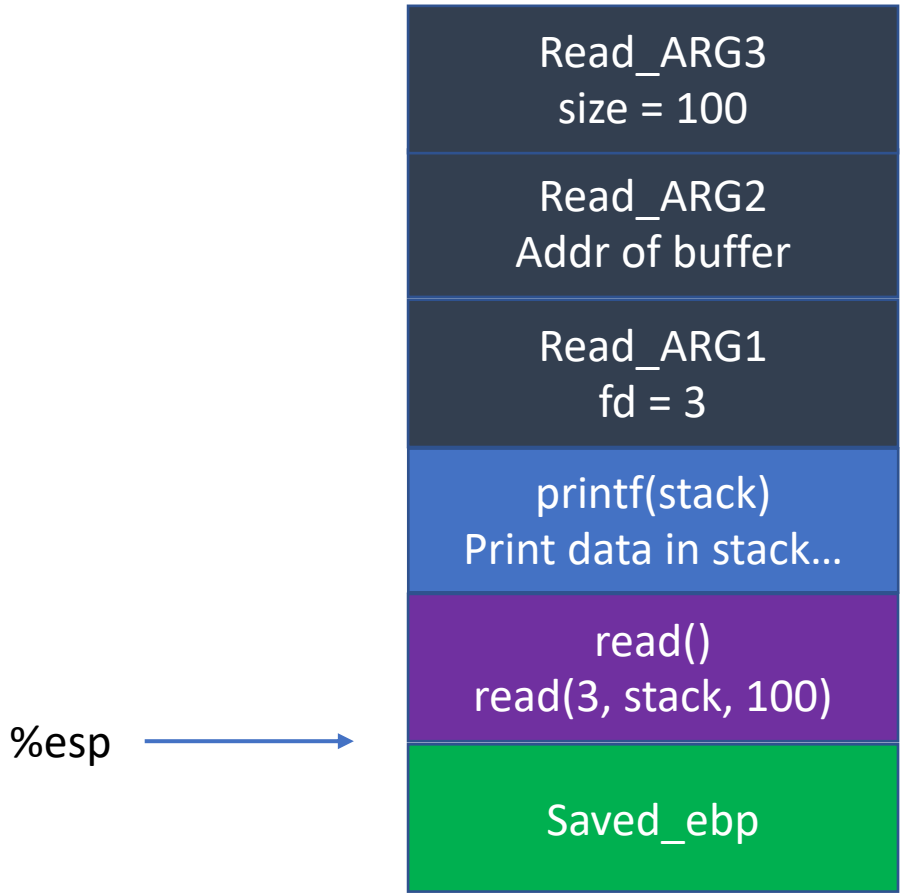
Chaining Function Calls in DEP-3

```
End of some_function

leave
ret
```



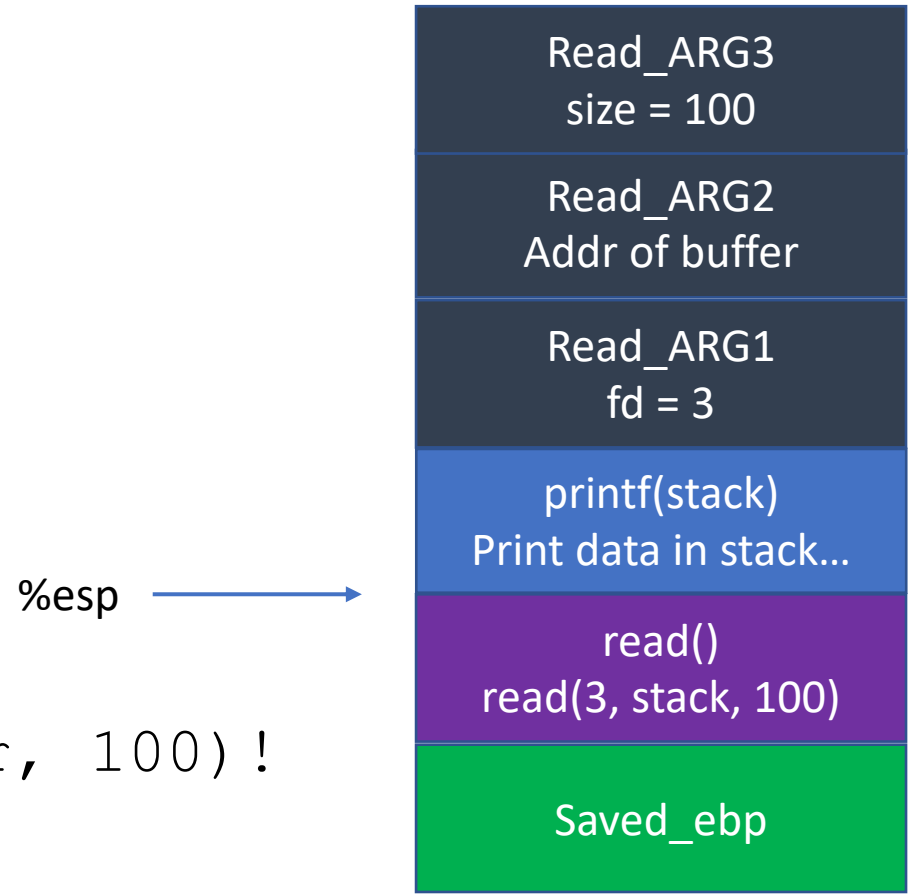
Chaining Function Calls in DEP-3

```
End of some_function

leave
ret
```



Chaining Function Calls in DEP-3

```
End of some_function

leave
ret
```

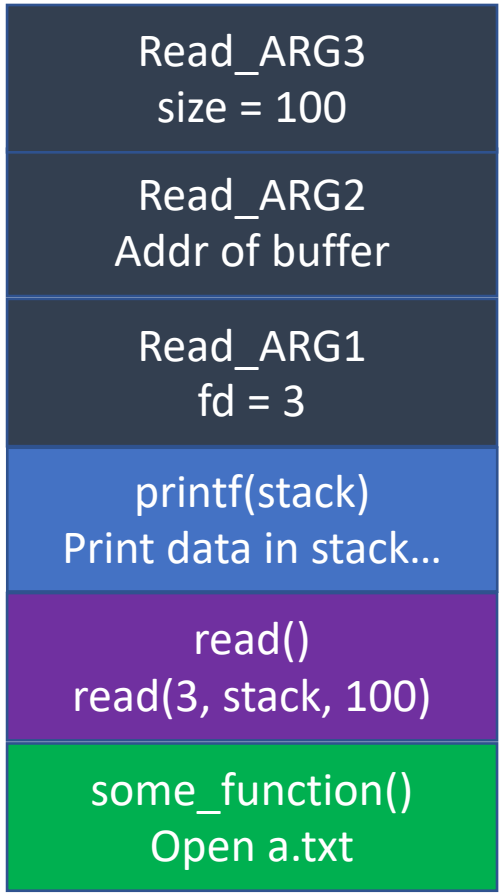


Execute read(3, buffer, 100)!

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…

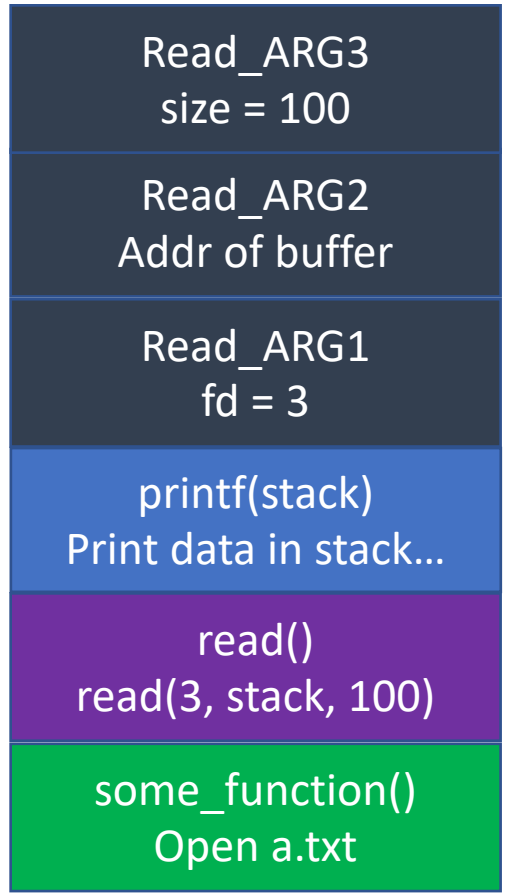
- some_function()
- read(3, stack_addr, 100)
- printf(stack_addr)



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

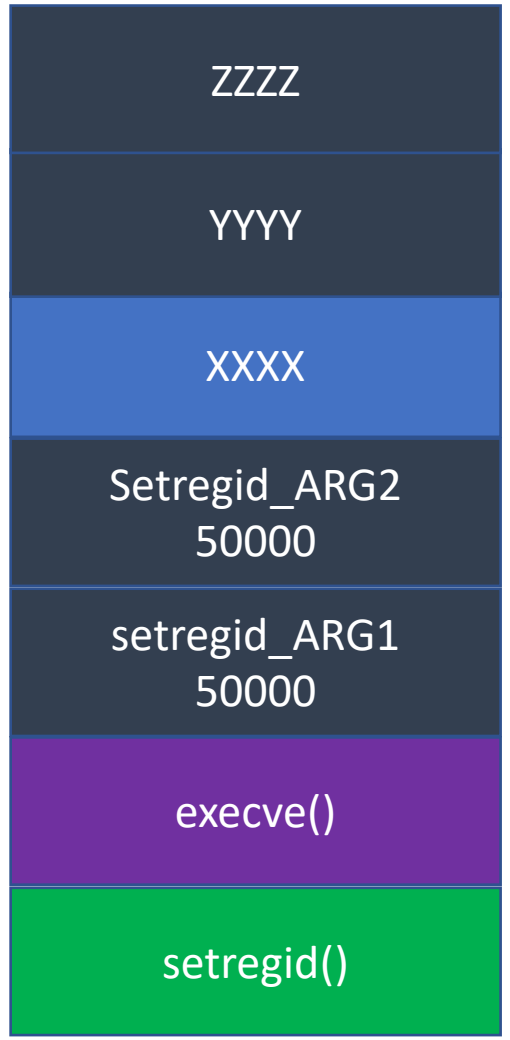
- And then, we put execve at +4, to chain the call

ROP-1: Function Arguments..

- setregid(50000, 50000);
- execve("/bin/sh", 0, 0);
- And then, we put execve at +4, to chain the call
- Seems that we are calling
  - 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

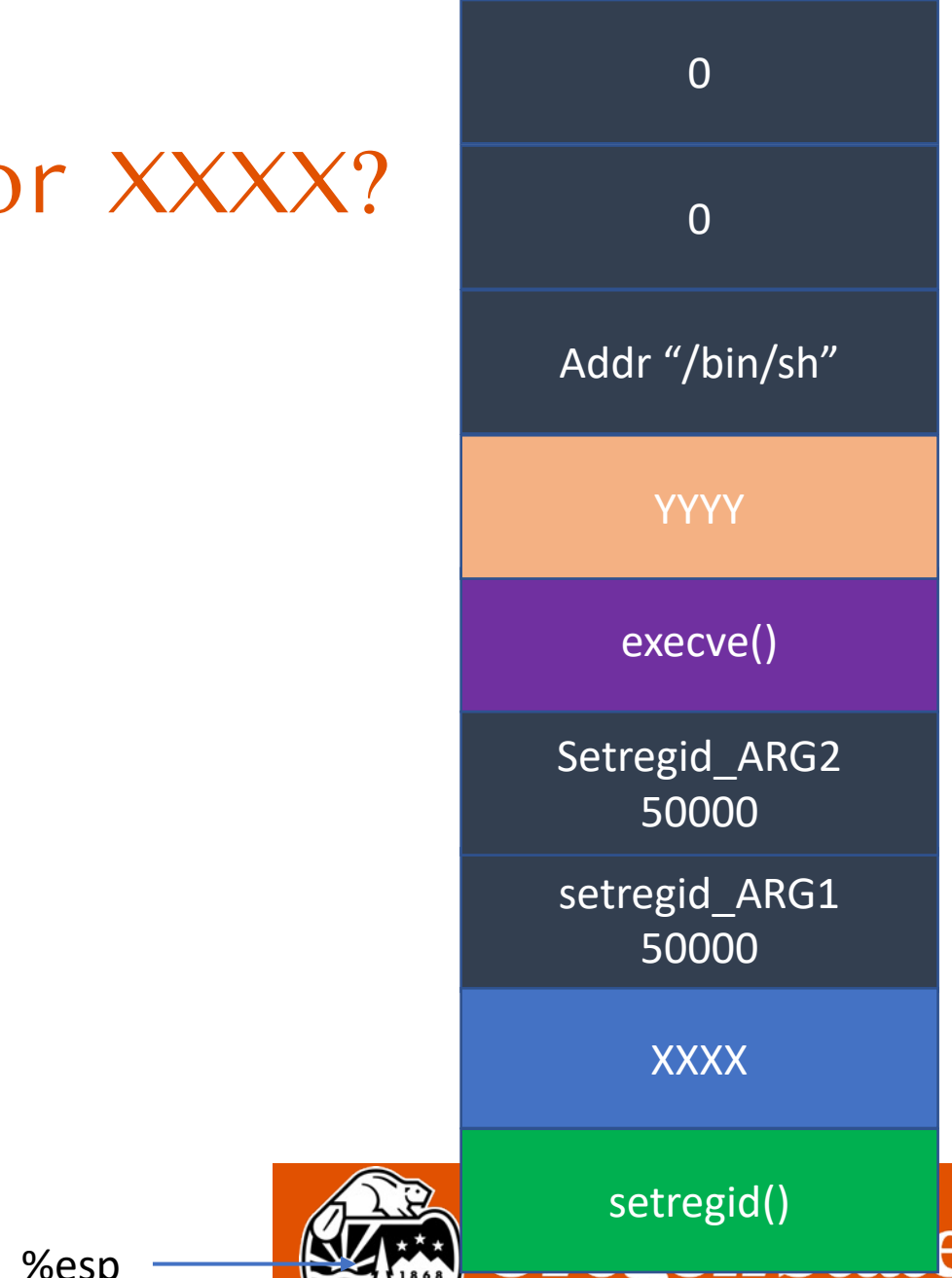
NO!



Oregon State University

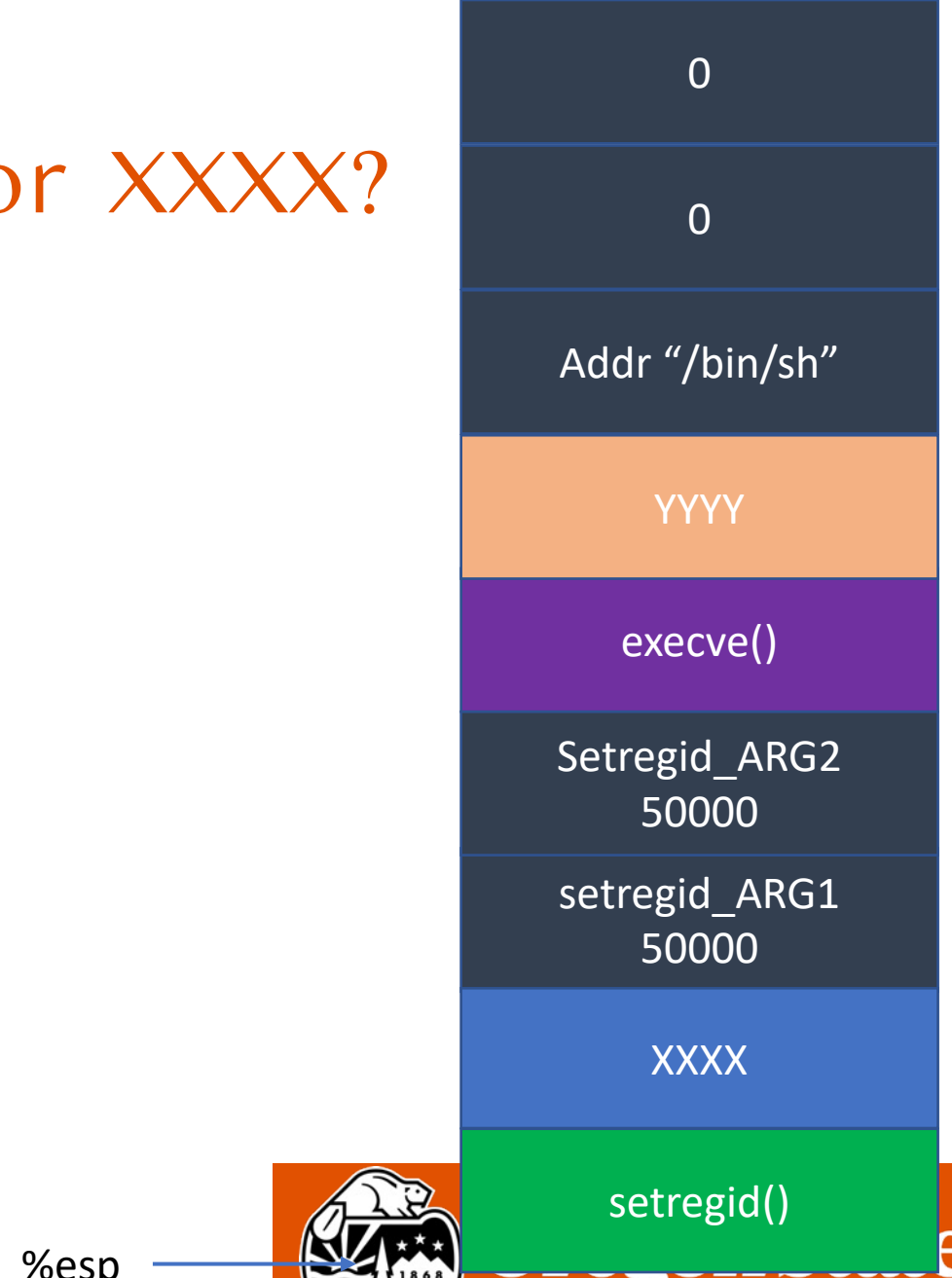
What Do You Want to Put for XXXX?

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



What Do You Want to Put for XXXX?

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

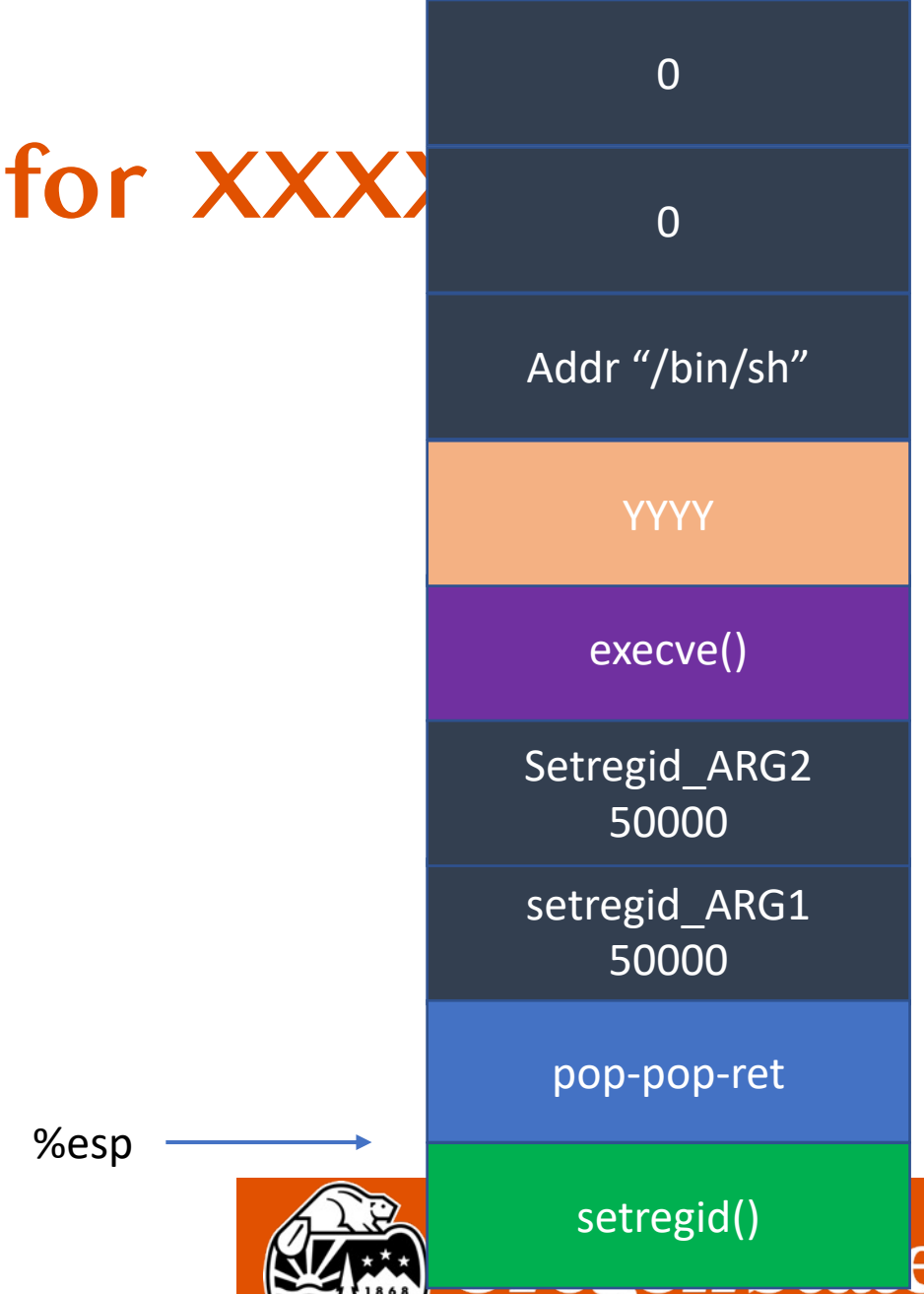


What Do You Want to Put for XXXX

- Let's take a look at the stack on the right
- At return
  - setregid(50000, 50000)

What Do You Want to Put for XXXX

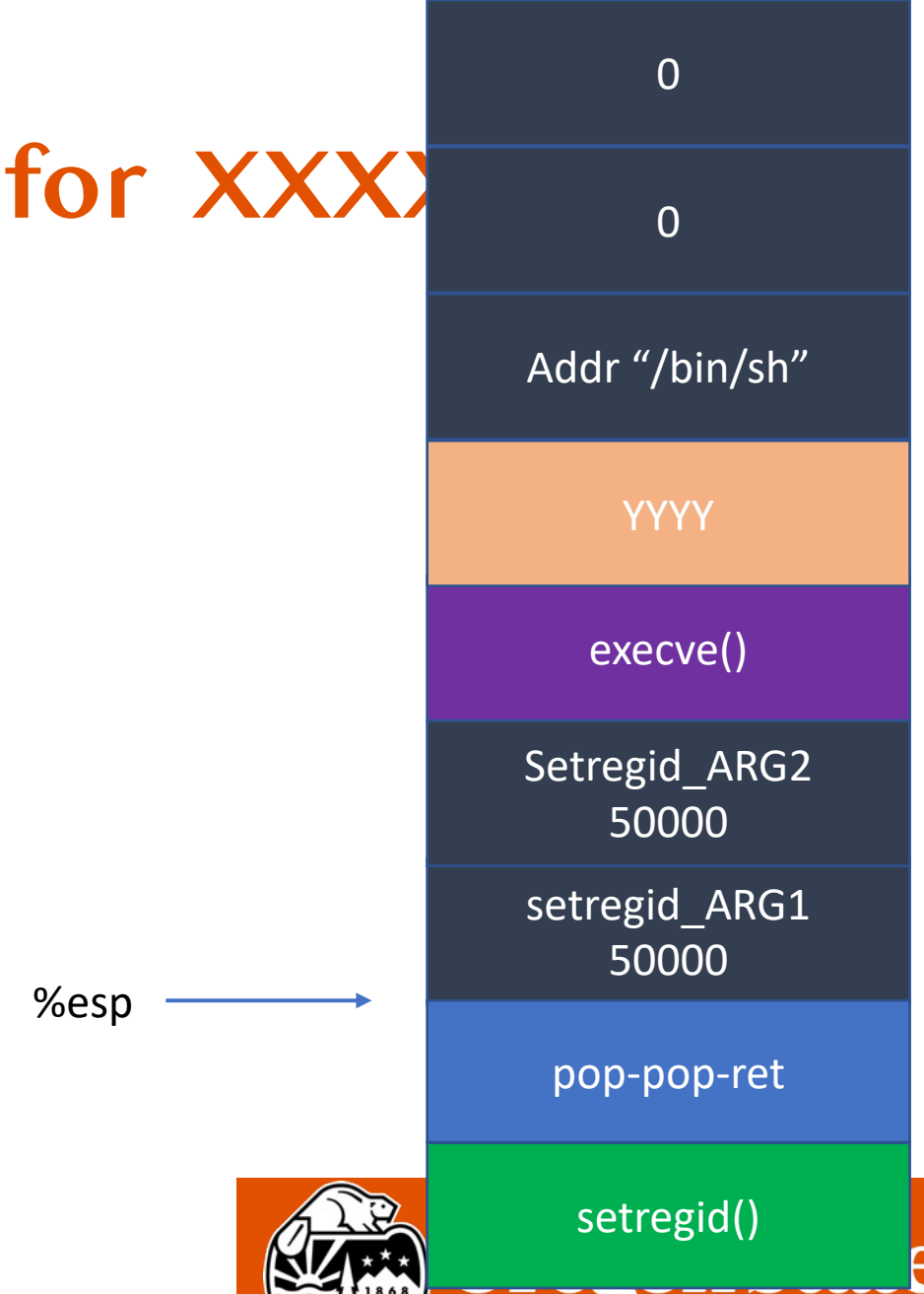
- Let's take a look at the stack on the right
- At return
  - setregid(50000, 50000)
- What if we run for XXXX?
  - pop %edi
  - pop %ebp
  - ret



What Do You Want to Put for XXXX

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

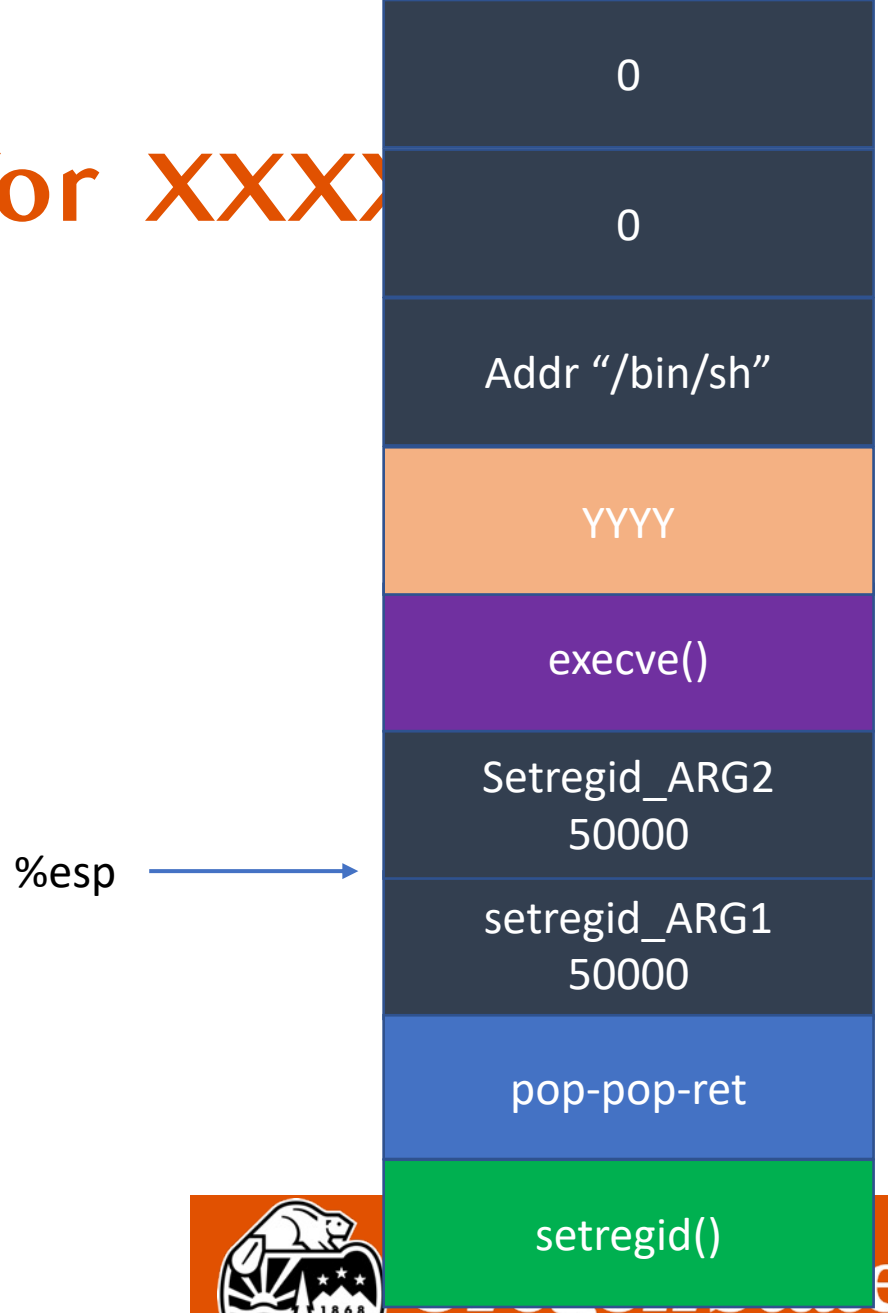
- `pop %edi`
- `pop %ebp`
- `ret`



What Do You Want to Put for XXXX

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

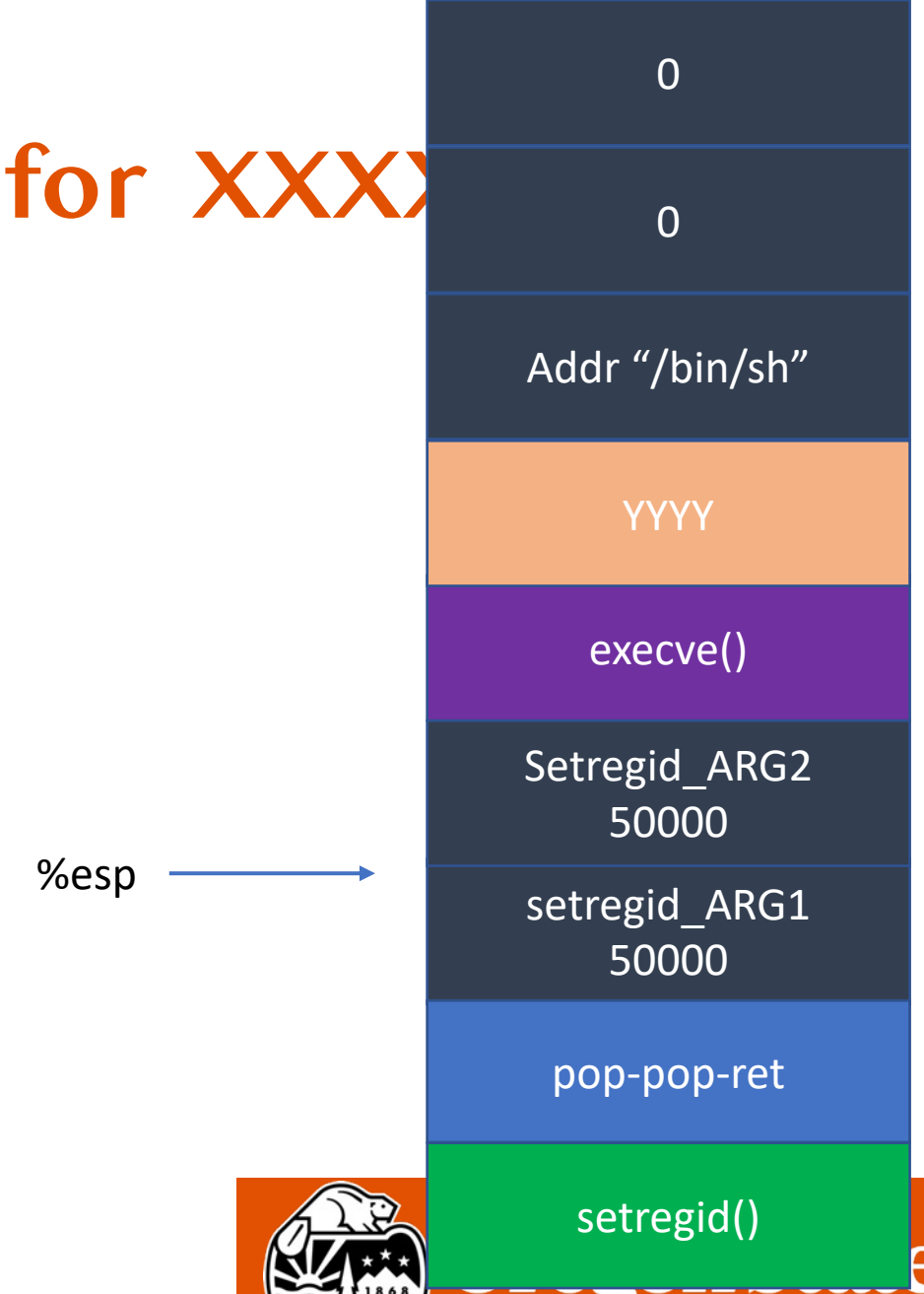
- **`pop %edi`**
- `pop %ebp`
- `ret`



What Do You Want to Put for XXXX

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

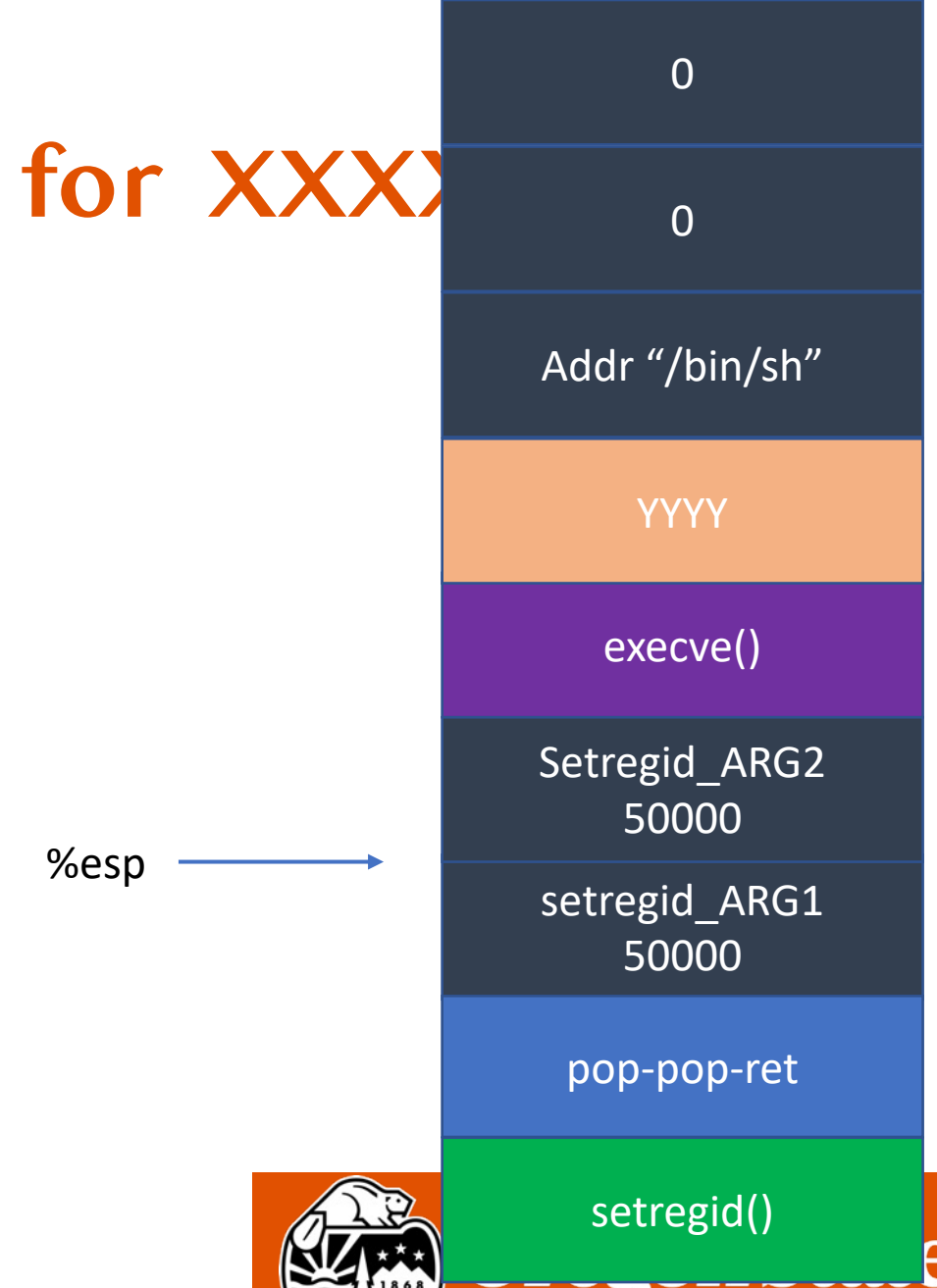
- **`pop %edi = 50000`**
- `pop %ebp`
- `ret`



What Do You Want to Put for XXXX

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

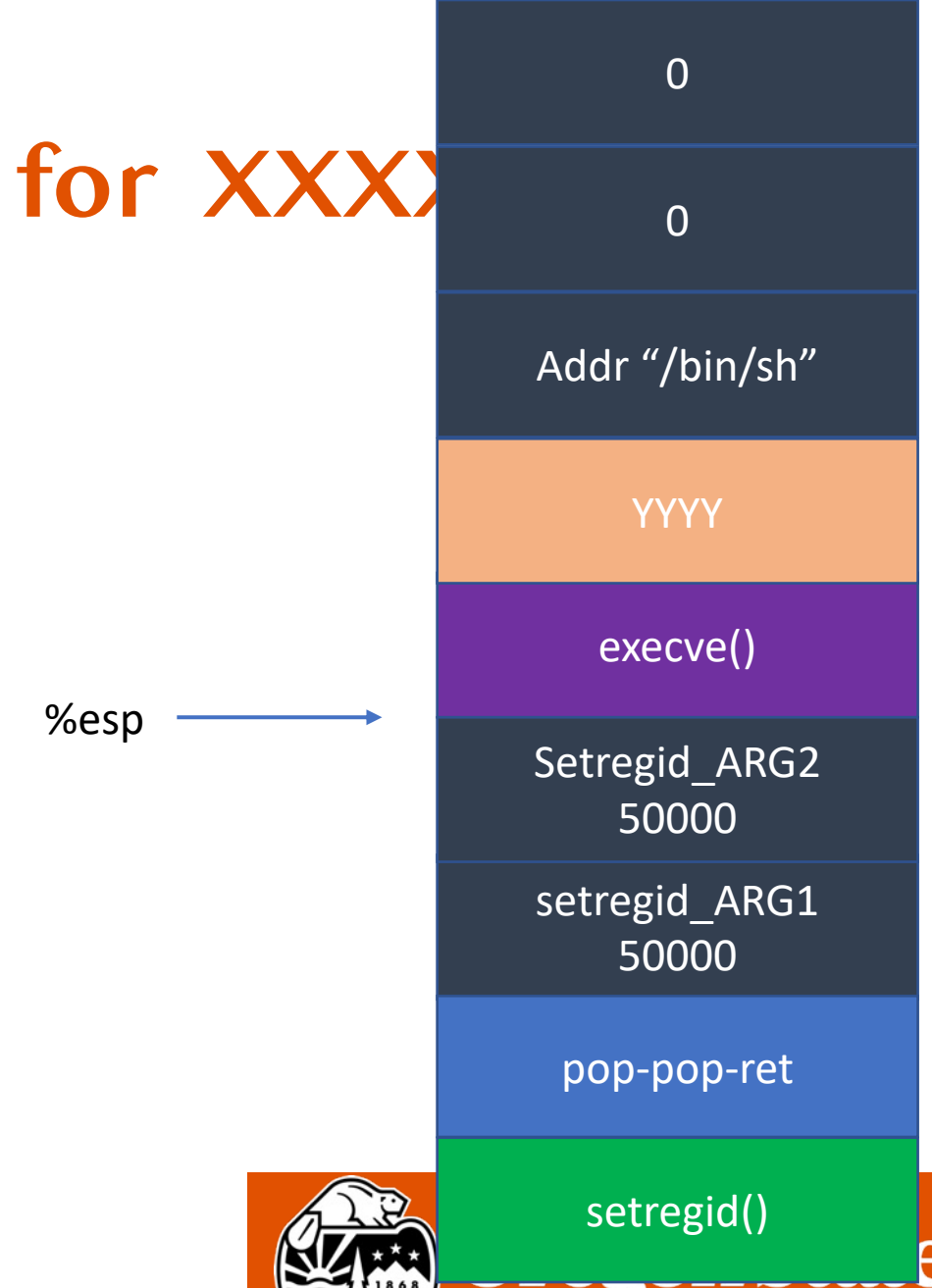
- `pop %edi = 50000`
- **`pop %ebp`**
- `ret`



What Do You Want to Put for XXXX

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

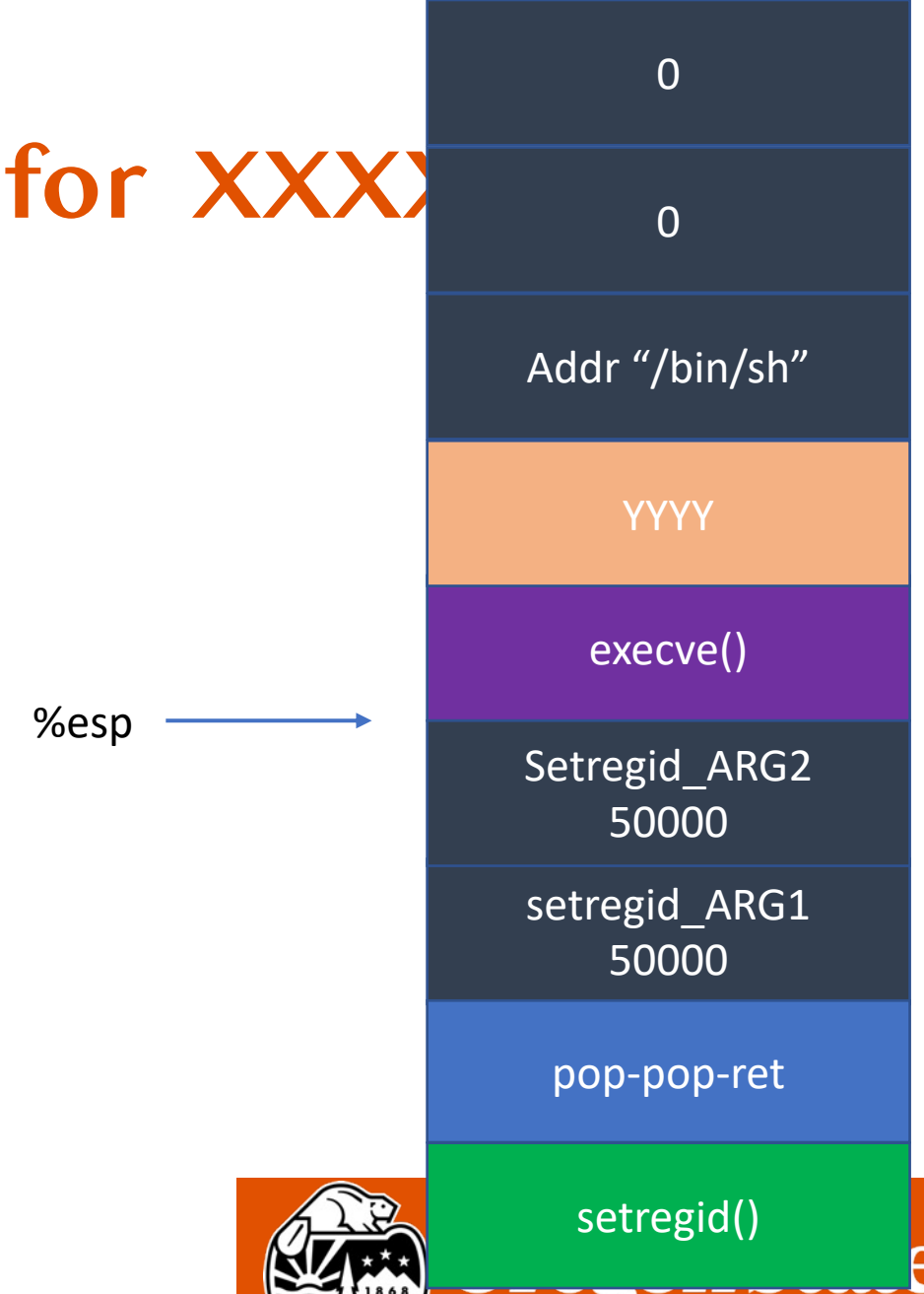
- `pop %edi = 50000`
- **`pop %ebp`**
- `ret`



What Do You Want to Put for XXXX

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

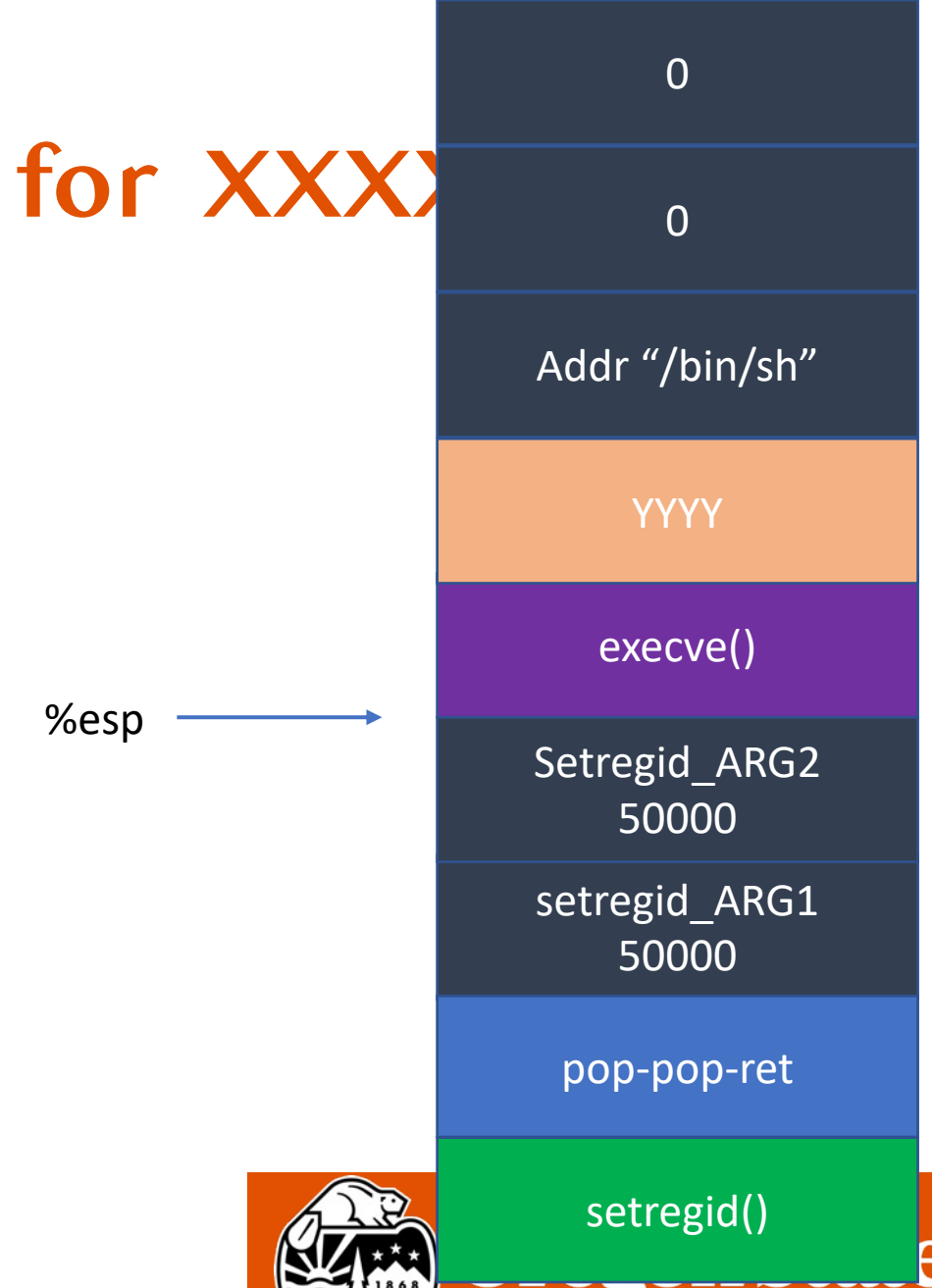
- `pop %edi = 50000`
- **`pop %ebp = 50000`**
- `ret`



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`**

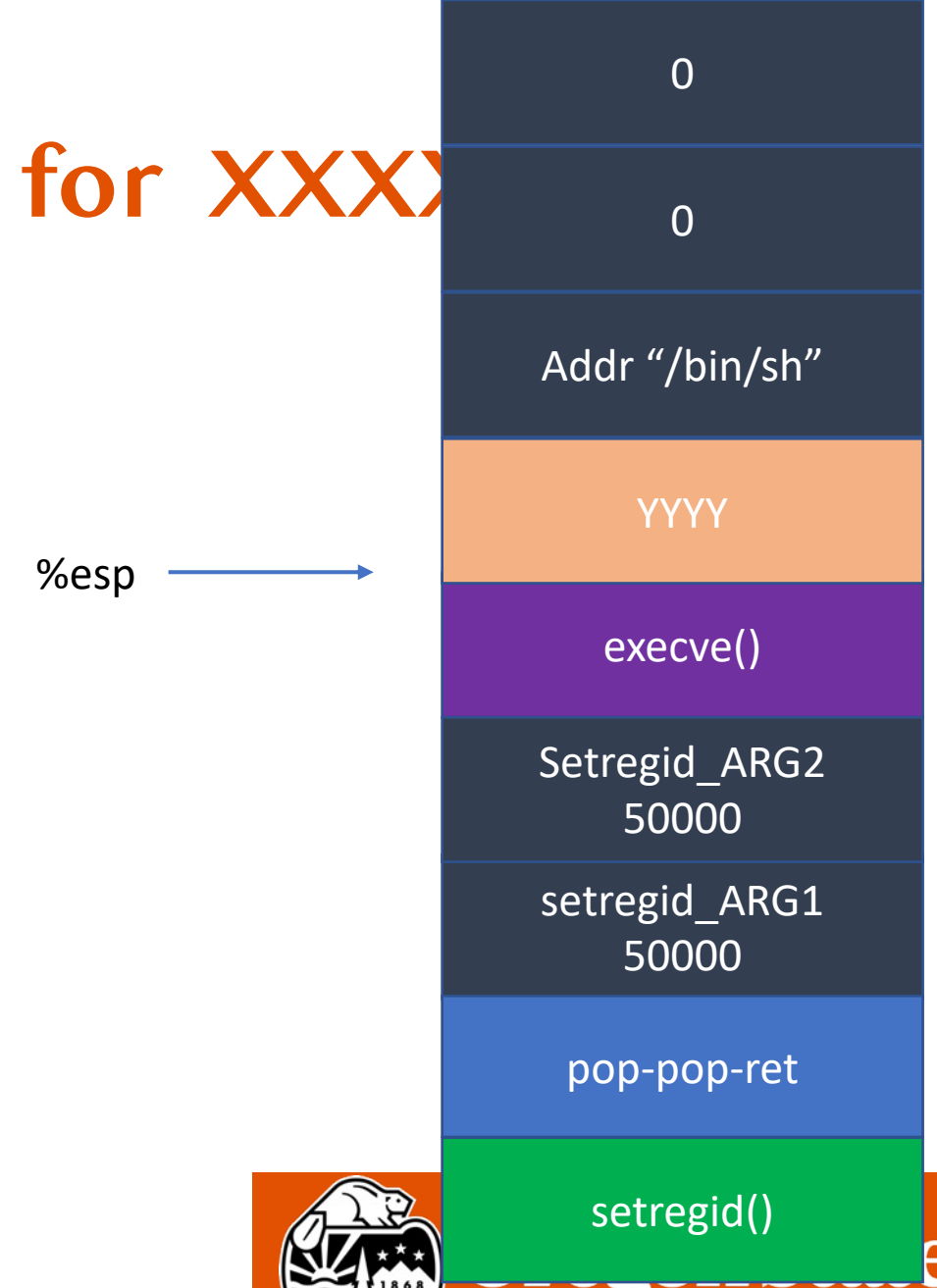


What Do You Want to Put for XXXX

- 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 arguments
  - [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 pops?
- From disassembly
  - Or from 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);

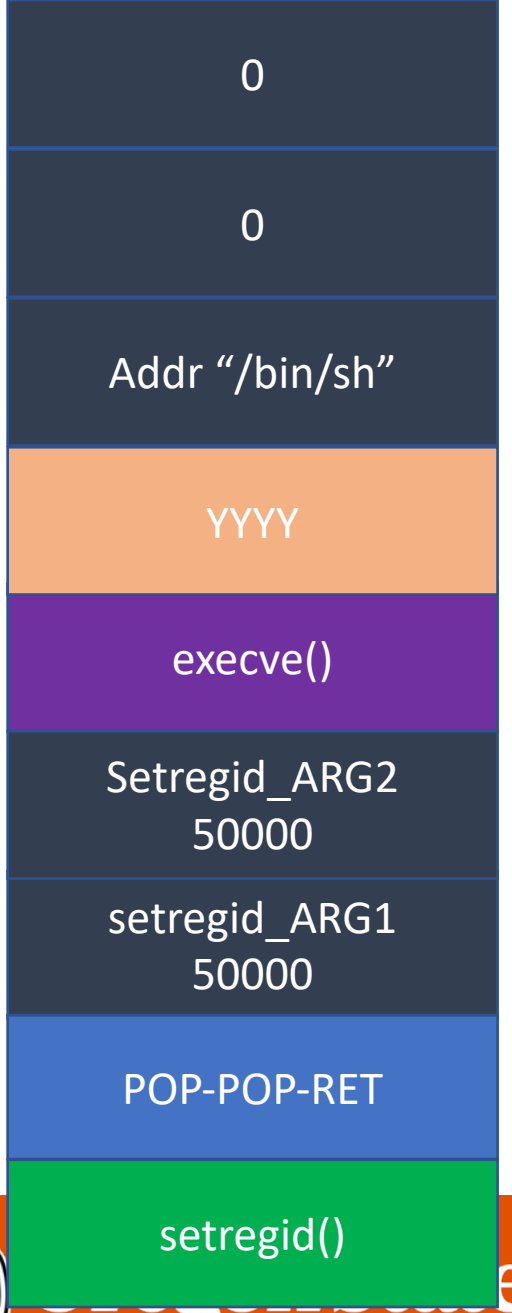
```
0x0804865a : pop edi ; pop ebp ; ret
```

ROP-1-64: 64bit

- ROP in 32 bit is easier than 64bit because 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!



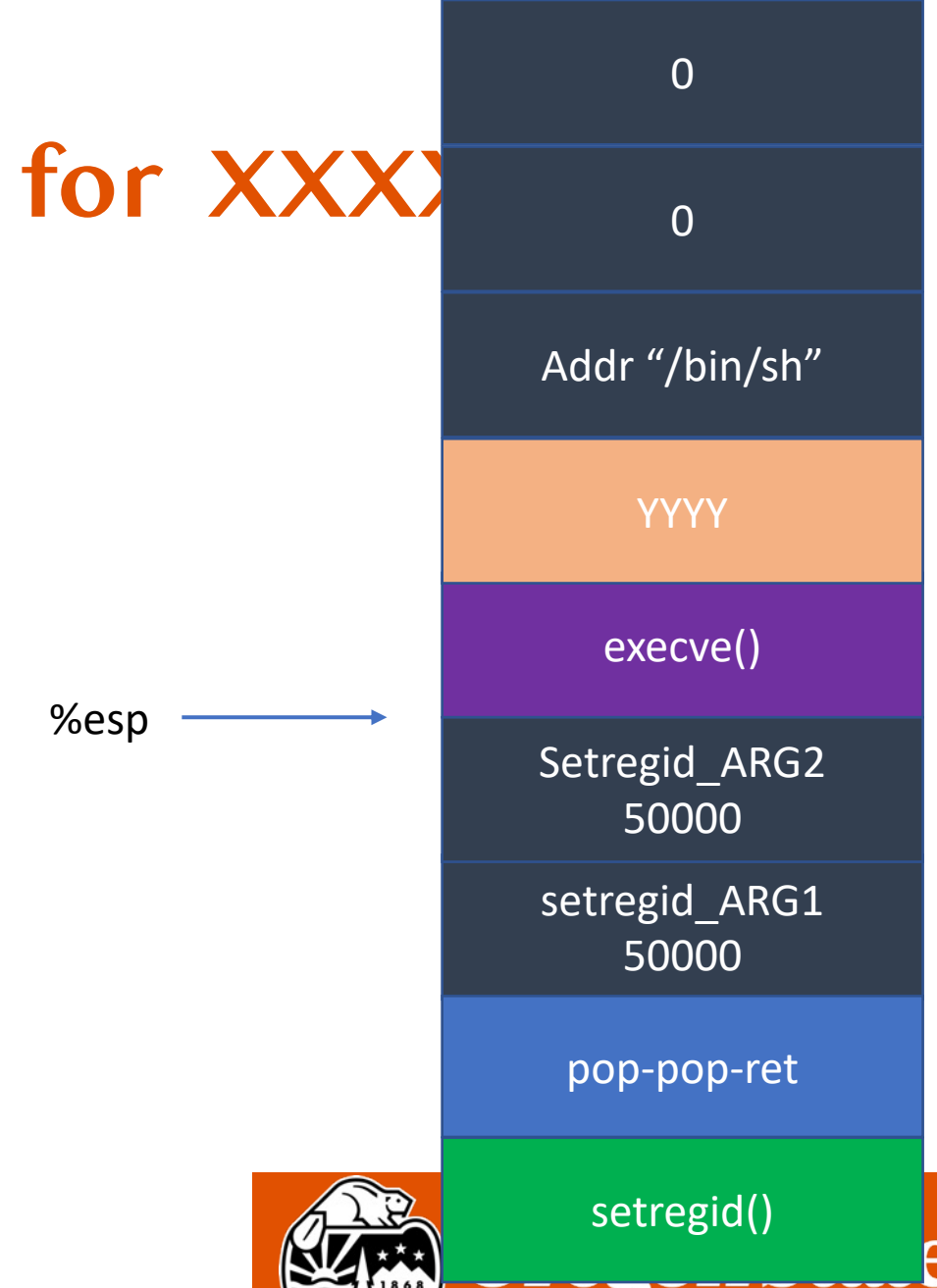
ROP-1-64: Setting Register Values

- We can set register values using pop XXX

What Do You Want to Put for XXXX

- Let's take a 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;
  - …

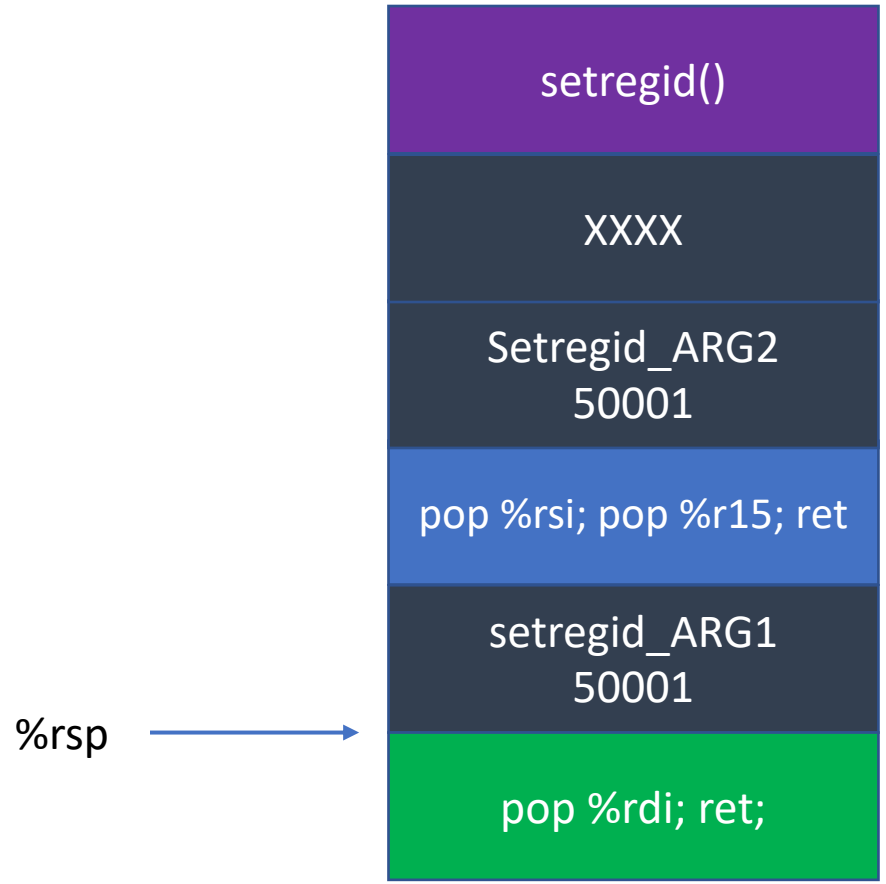
- Yes

ROP-1-64: Setting Register Values

```
$ ROPgadget --binary /home/labs/week5/rop-1-64/rop-1-64
Gadgets information
============================================================
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`

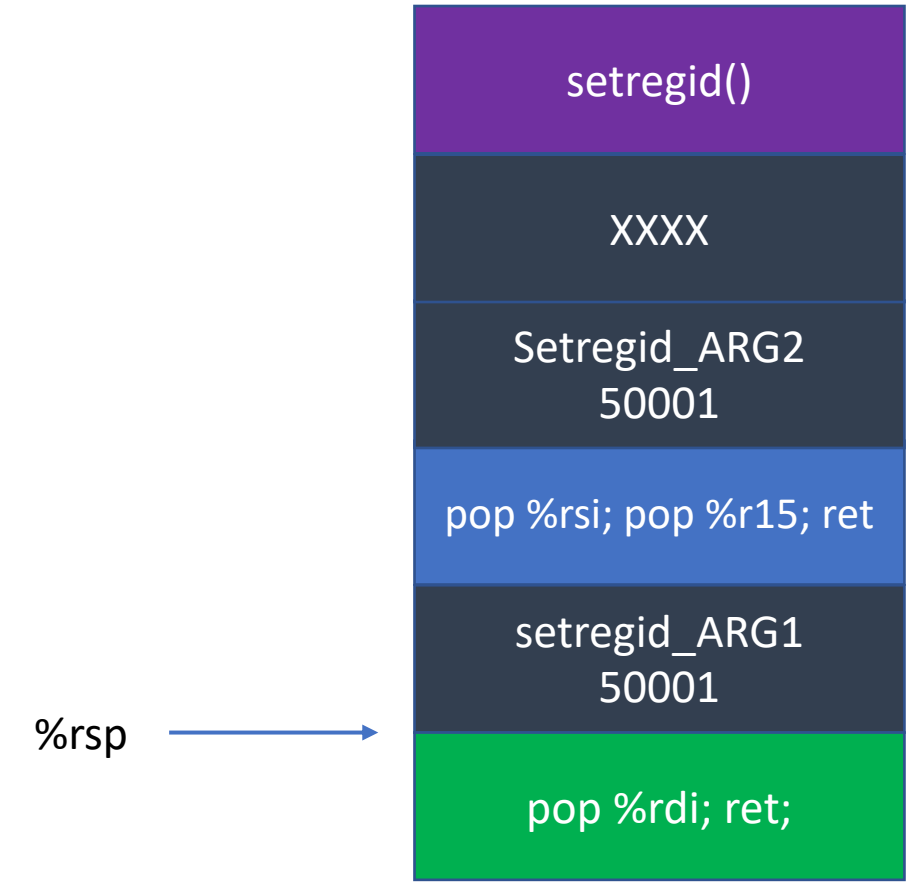



```
0x00000000004007e3 : pop rdi ; ret
0x00000000004006d8 : pop rdx ; nop ; pop rbp ; ret
0x00000000004007e1 : pop rsi ; pop r15 ; ret
```

Oregon State University

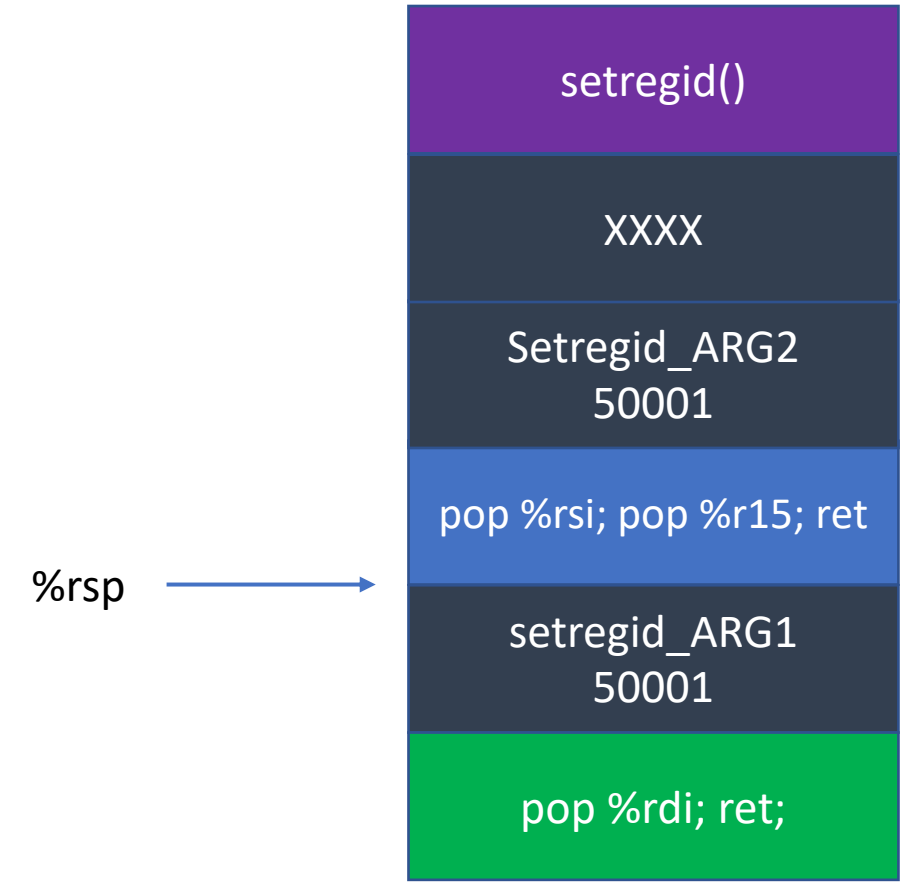
ROP-1-64: Passing Args in amd64

- **`pop %rdi`**
- `ret`
- `pop %rsi`
- `pop %r15`
- `ret`



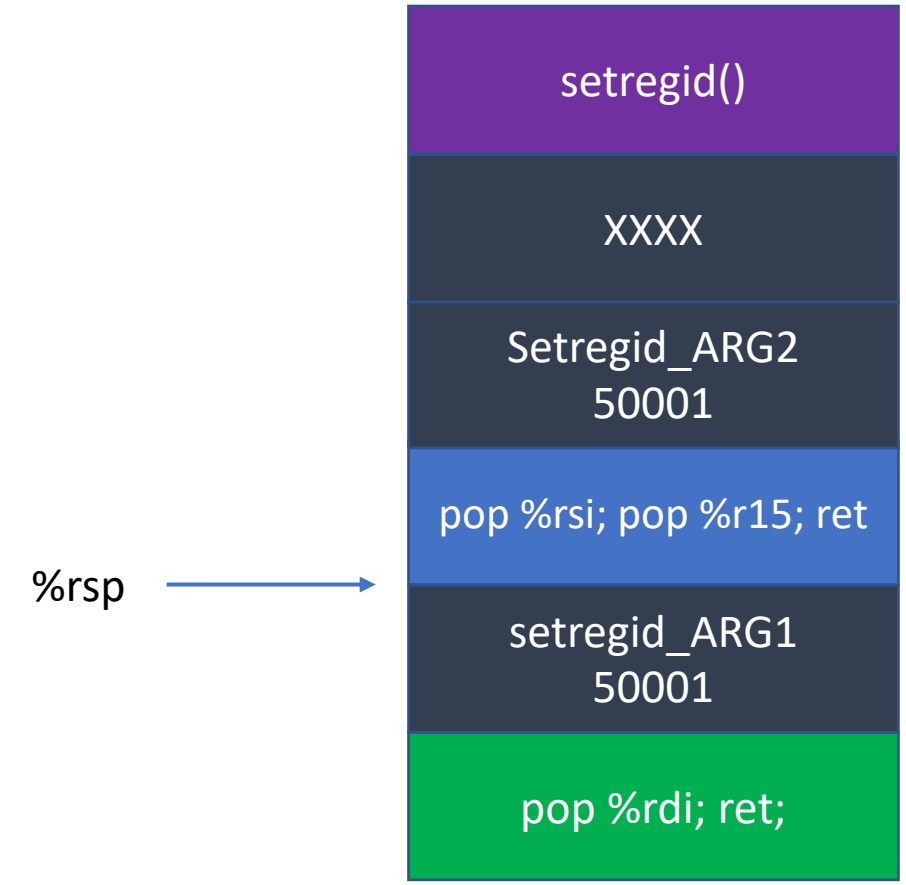
ROP-1-64: Passing Args in amd64

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



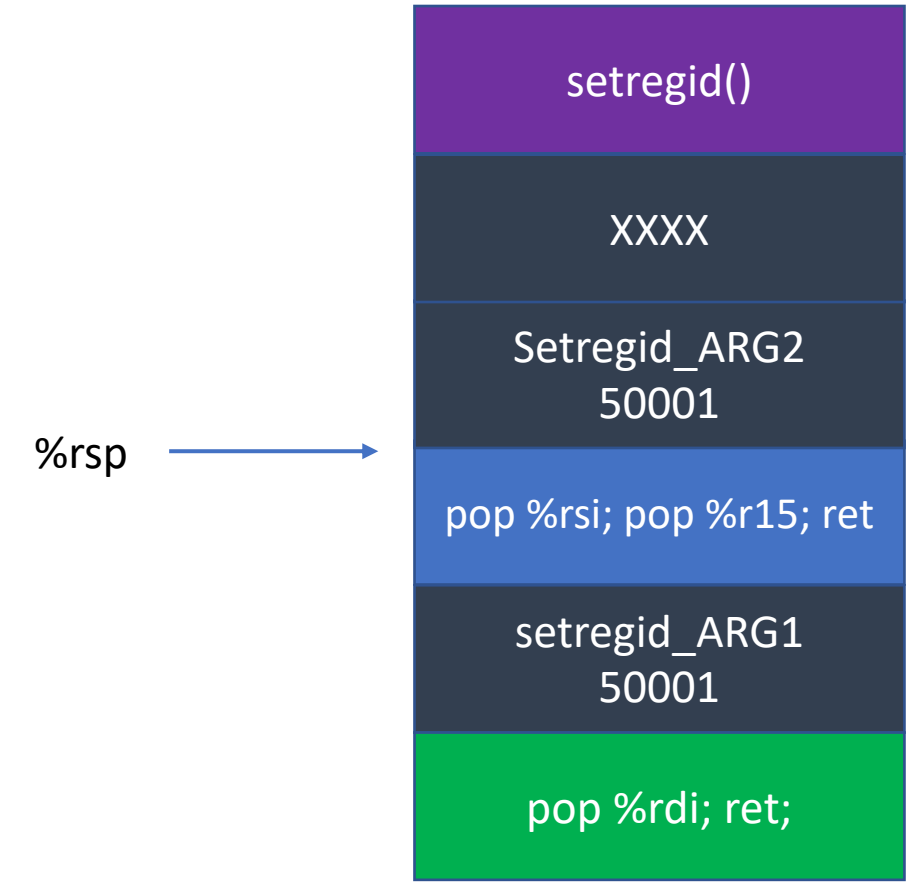
ROP-1-64: Passing Args in amd64

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



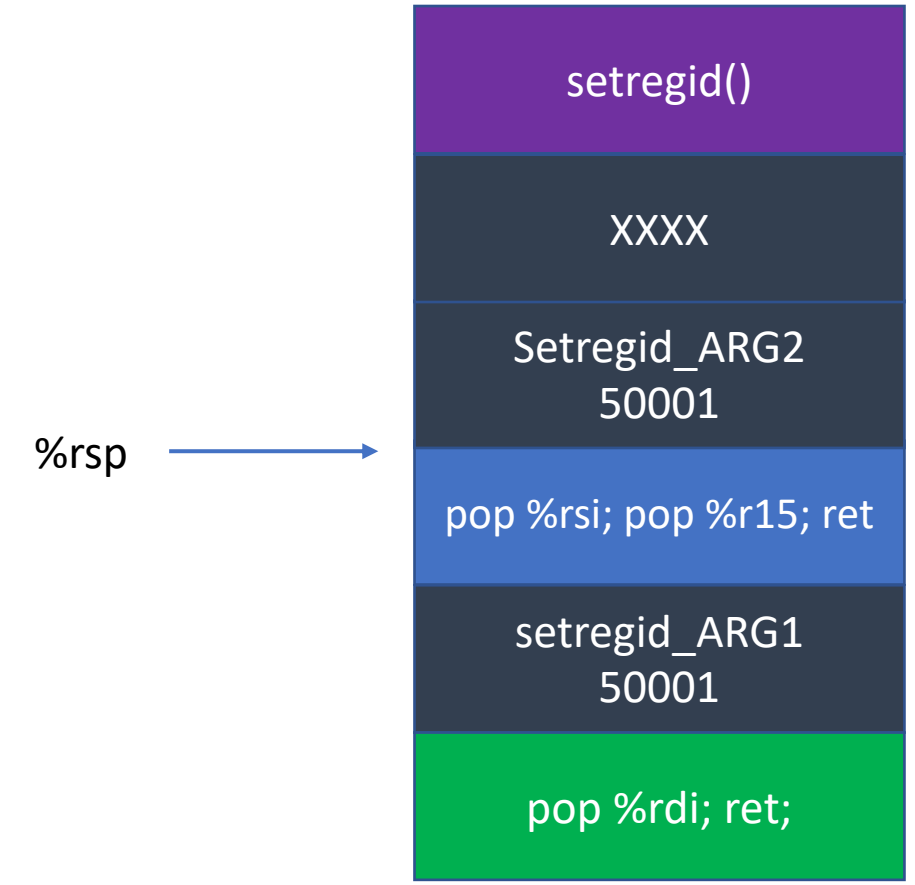
ROP-1-64: Passing Args in amd64

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



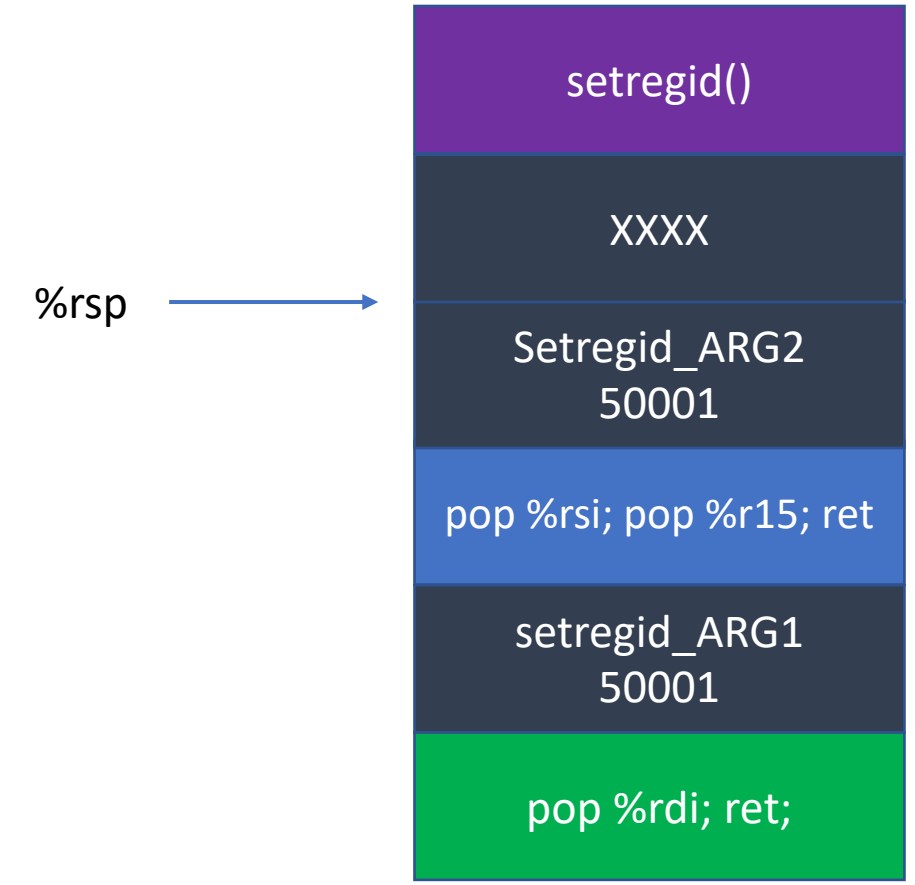
ROP-1-64: Passing Args in amd64

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



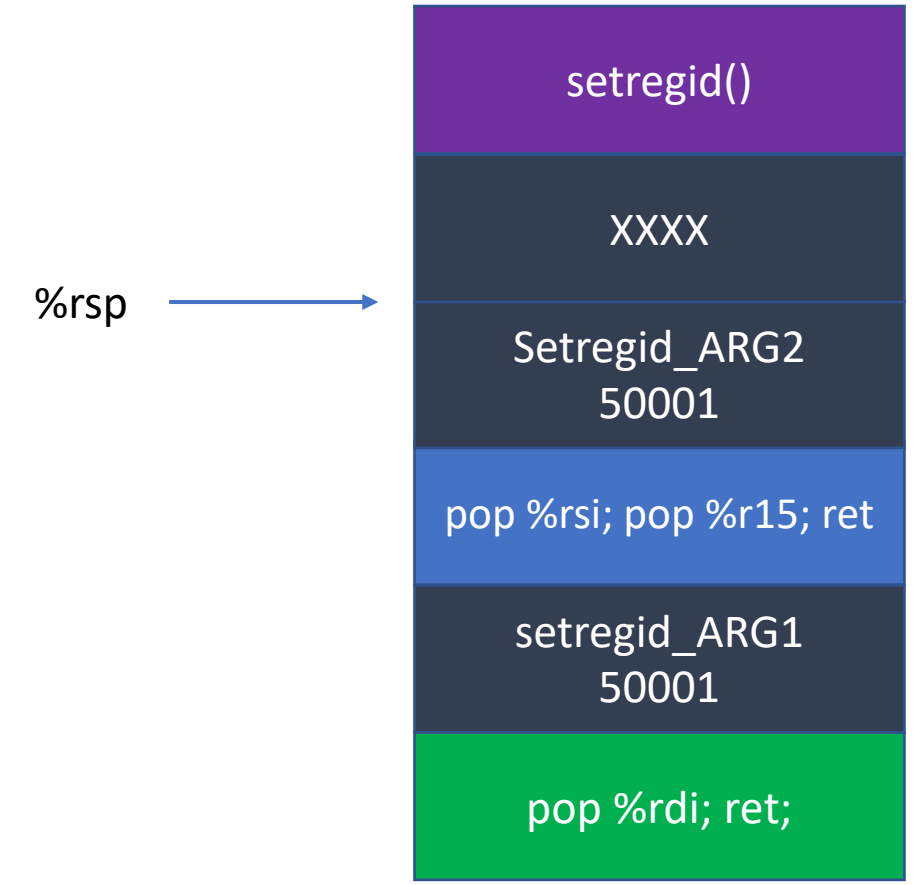
ROP-1-64: Passing Args in amd64

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



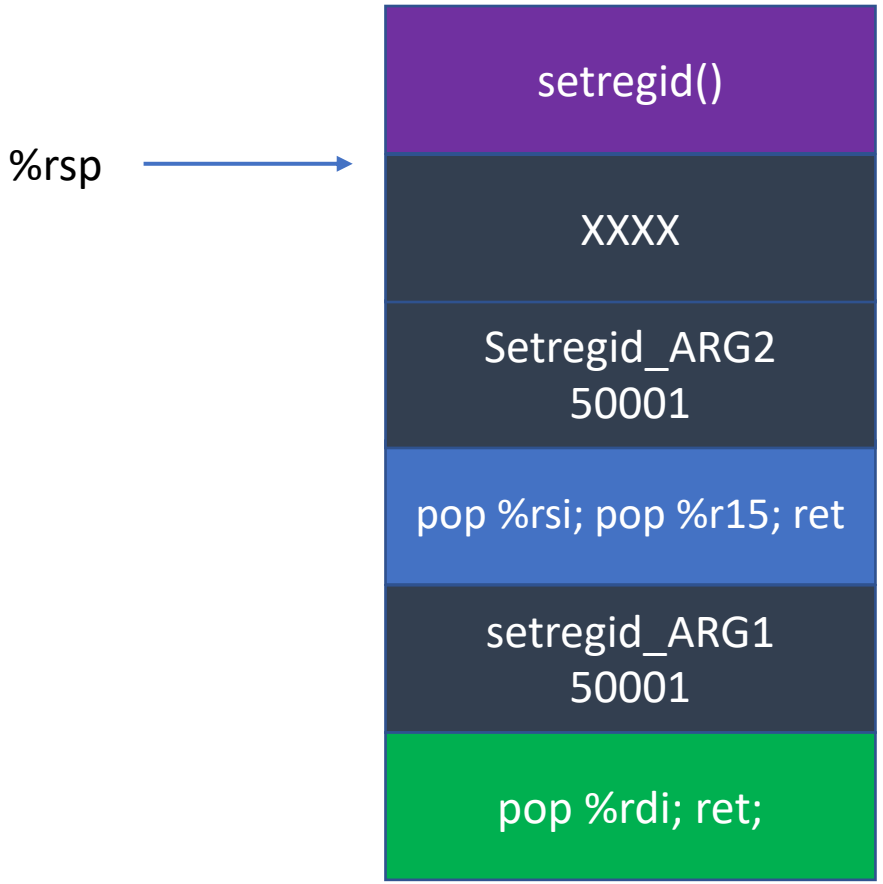
ROP-1-64: Passing Args in amd64

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



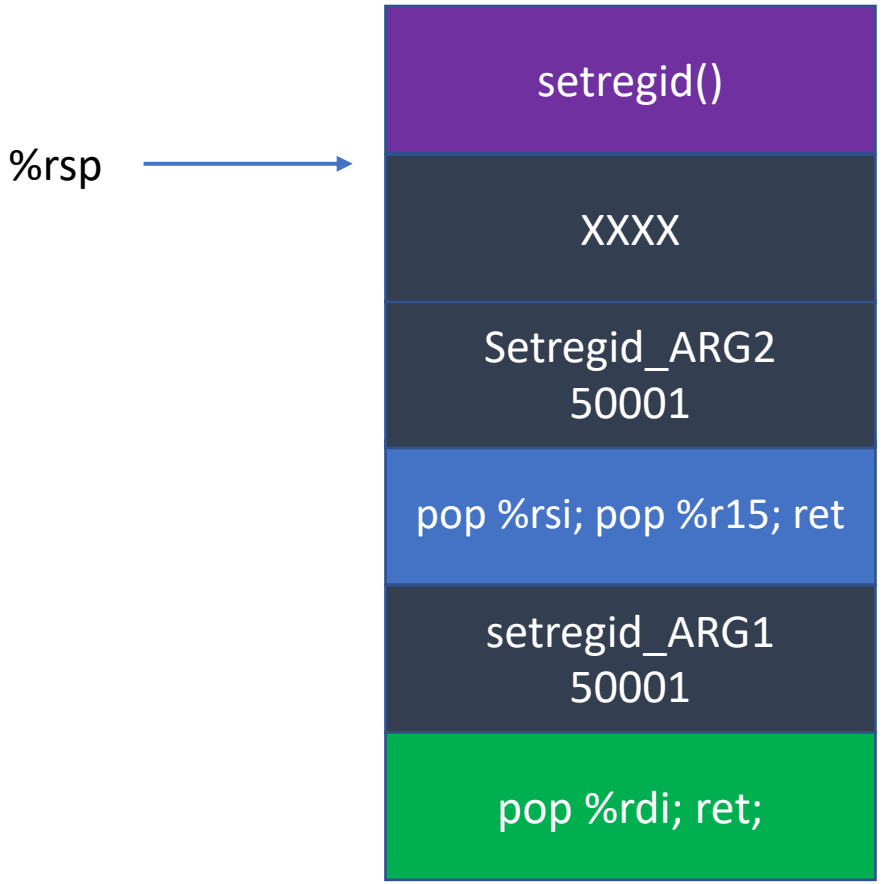
ROP-1-64: Passing Args in amd64

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



ROP-1-64: Passing Args in amd64

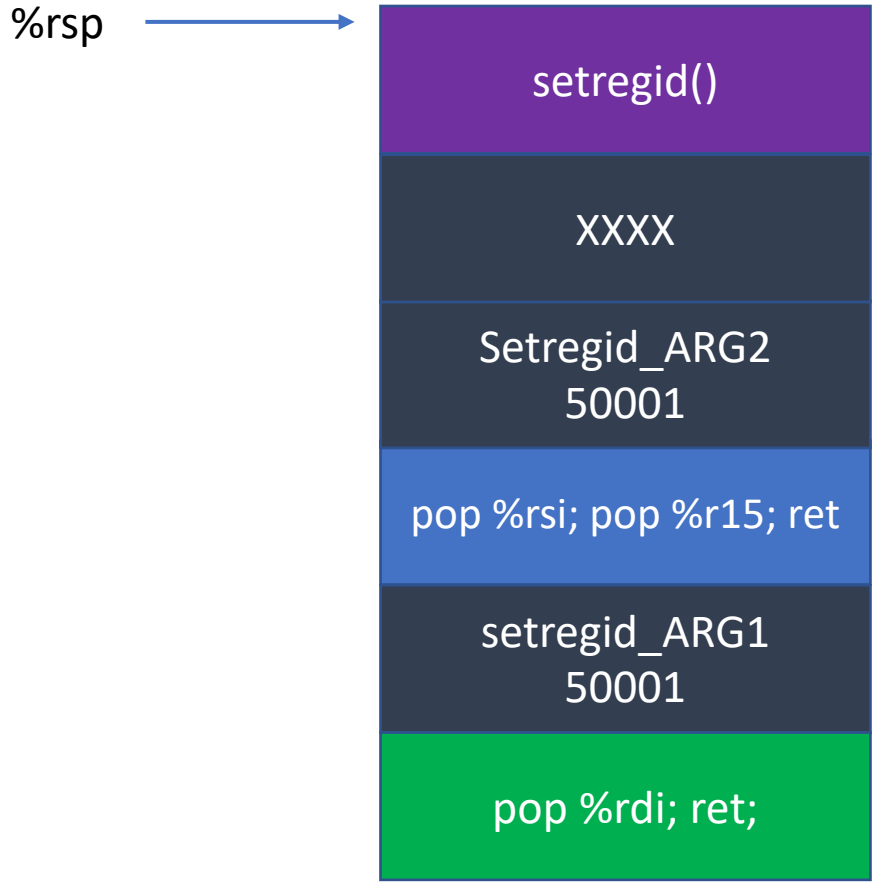
- `pop %rdi = 50001`
- `ret`
- `pop %rsi = 50001`
- `pop %r15 = XXXX`
- **`ret`**



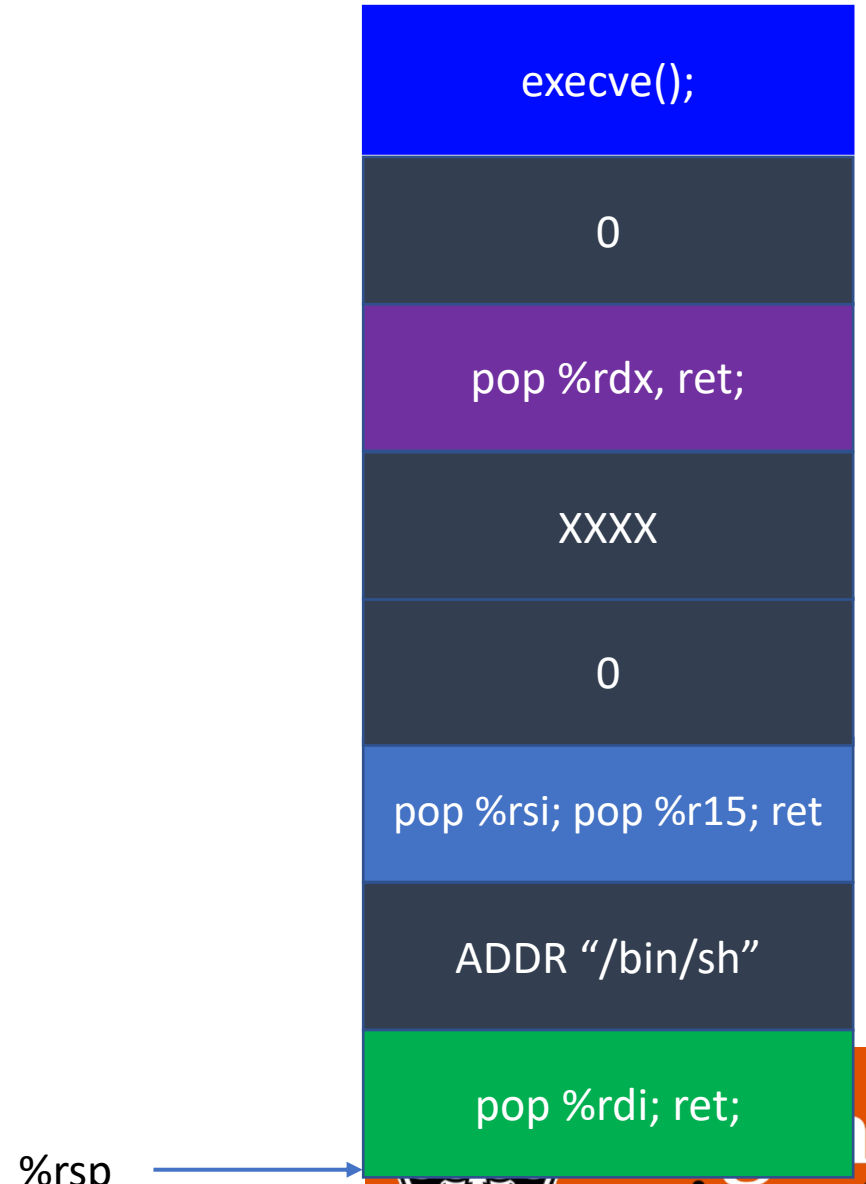
ROP-1-64: Passing Args in amd64

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

- **`setregid(50001, 50001)`**



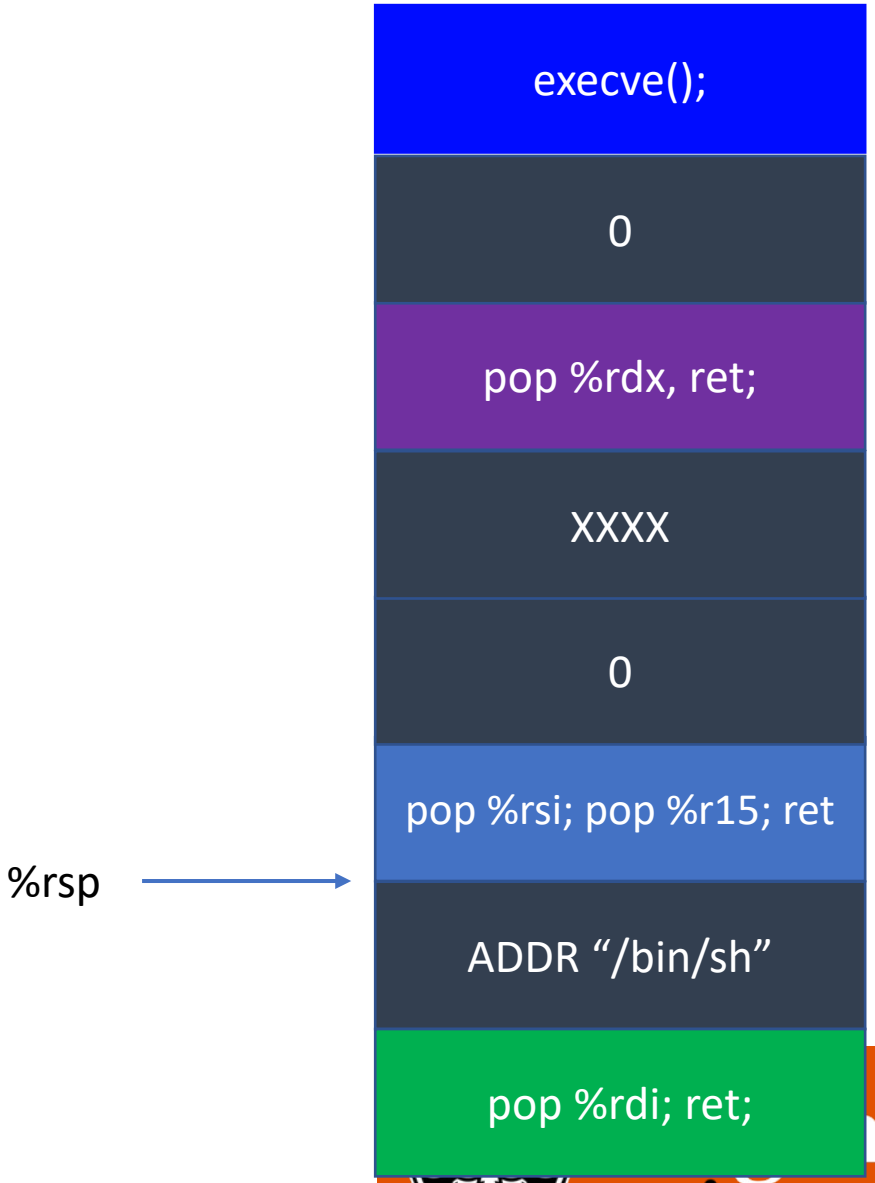
ROP-1-64: Execve?



ROP-1-64: Execve?

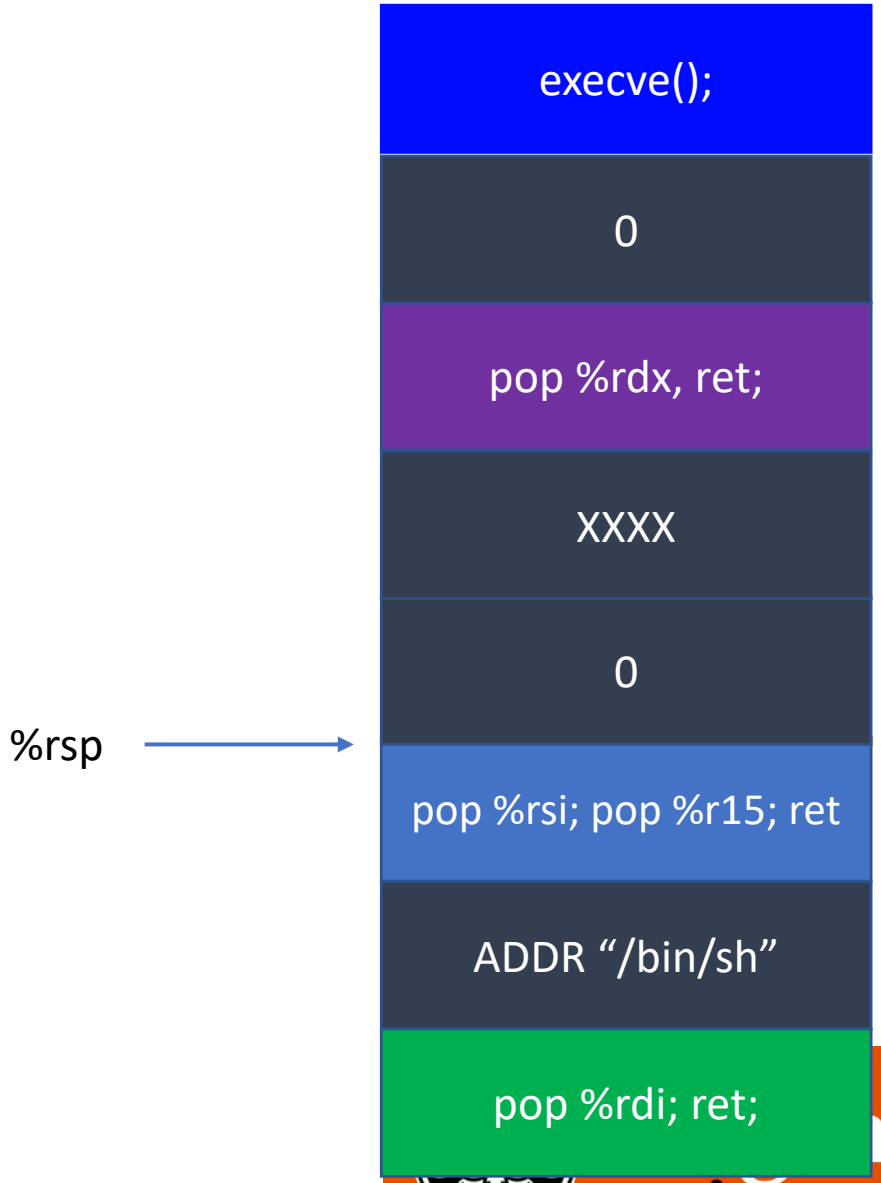
ROP-1-64: Execve?

- rdi = addr of “/bin/sh”



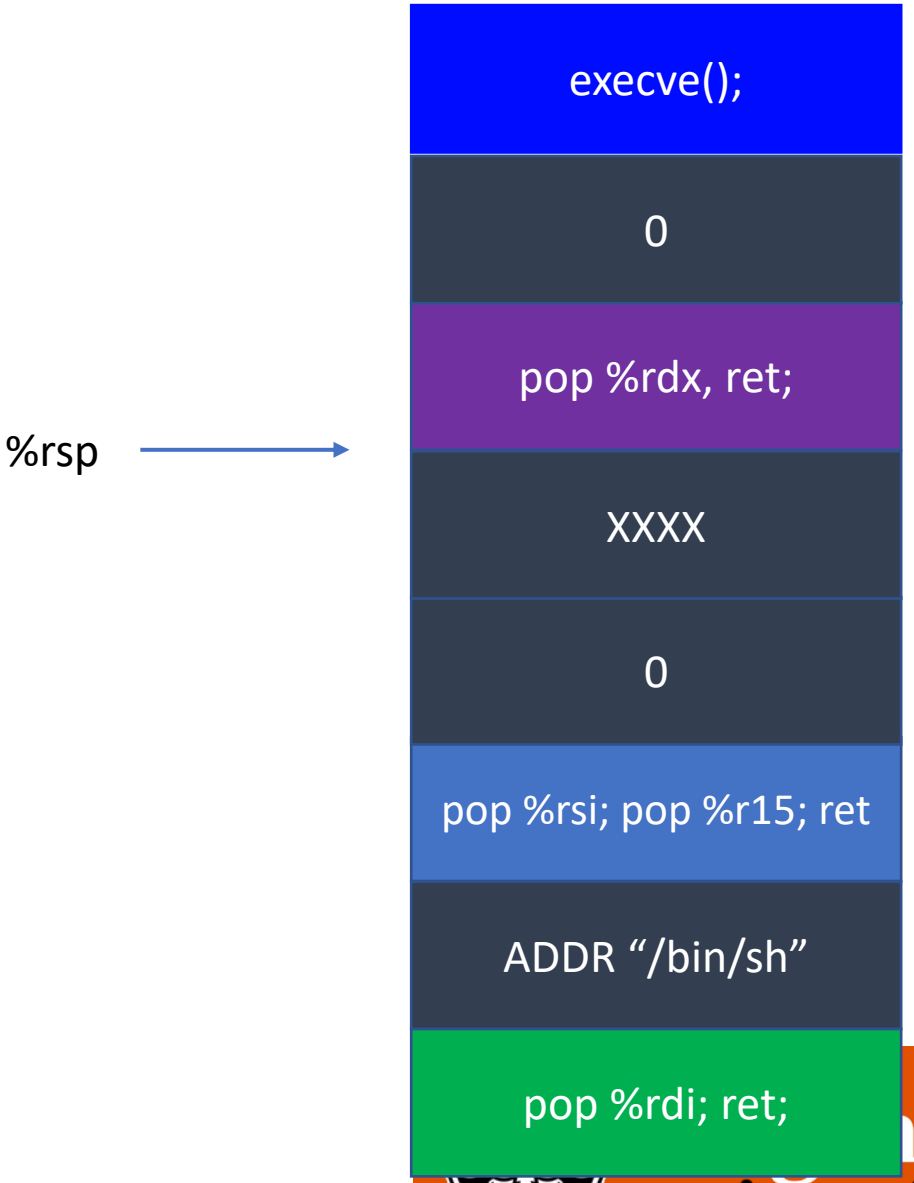
ROP-1-64: Execve?

- rdi = addr of “/bin/sh”



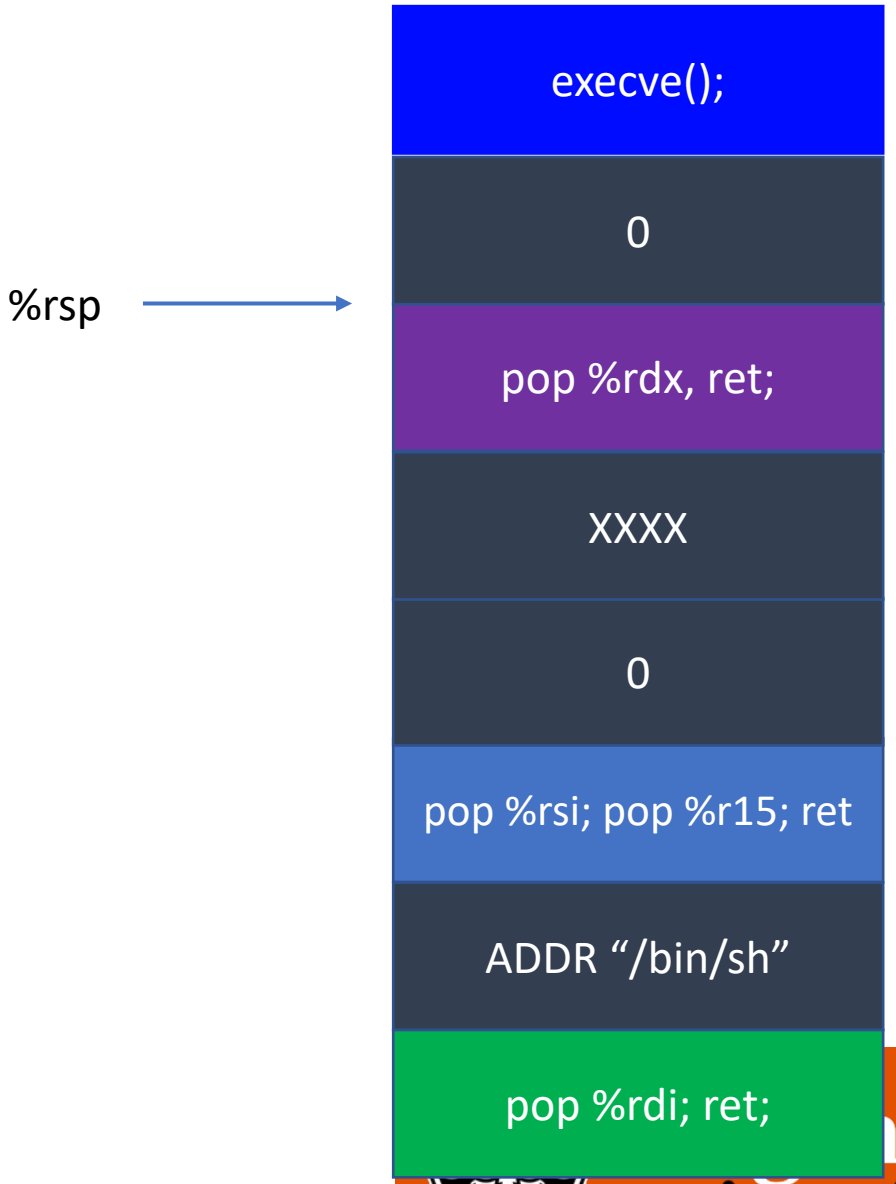
ROP-1-64: Execve?

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



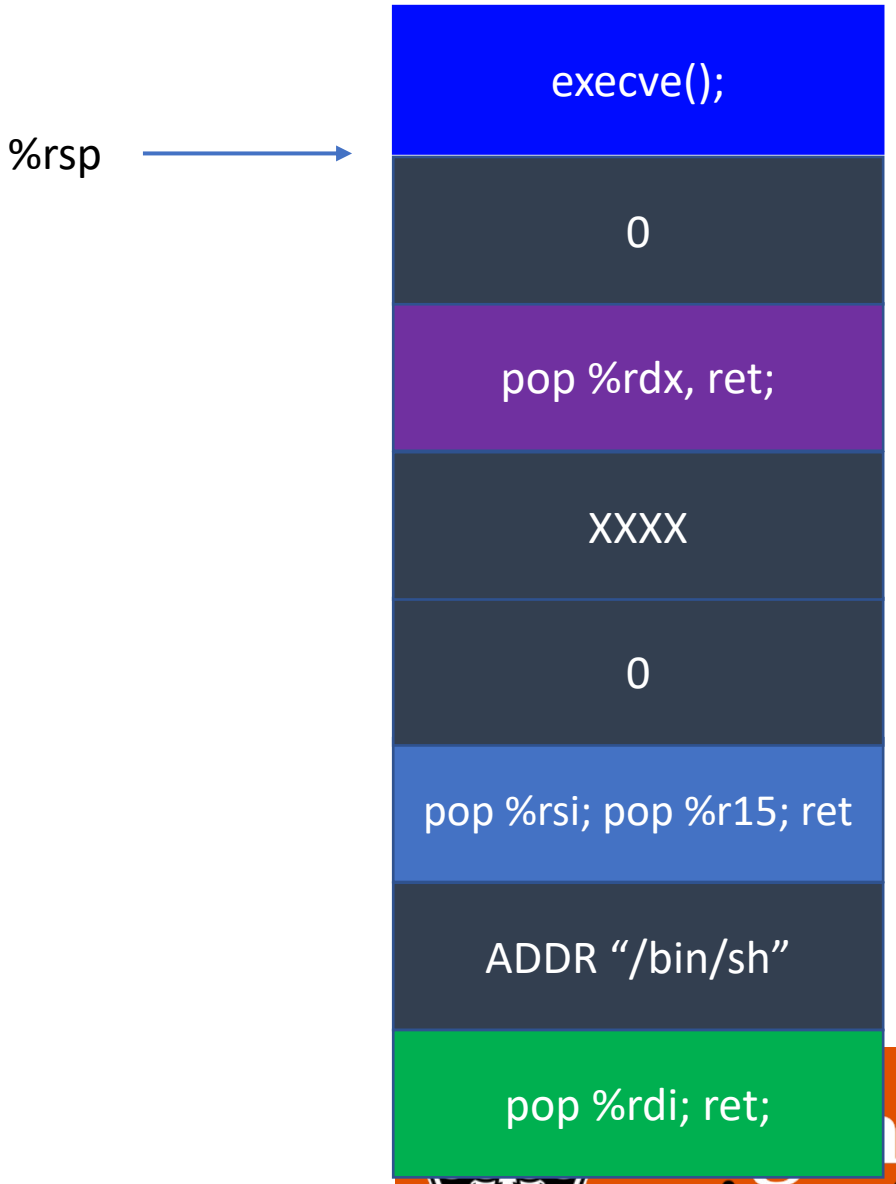
ROP-1-64: Execve?

- 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]

Assignment: Week-5

- ASLR: connect to `vm-ctf2.eecs.oregonstate.edu`
  - Use the same credentials (id, private key, etc.)
- Challenges are in `/home/labs/week5`
  - Use `fetch week5`
- 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’

- **Due: 03/01 11:59pm**