Executables & Buffer Overflows

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

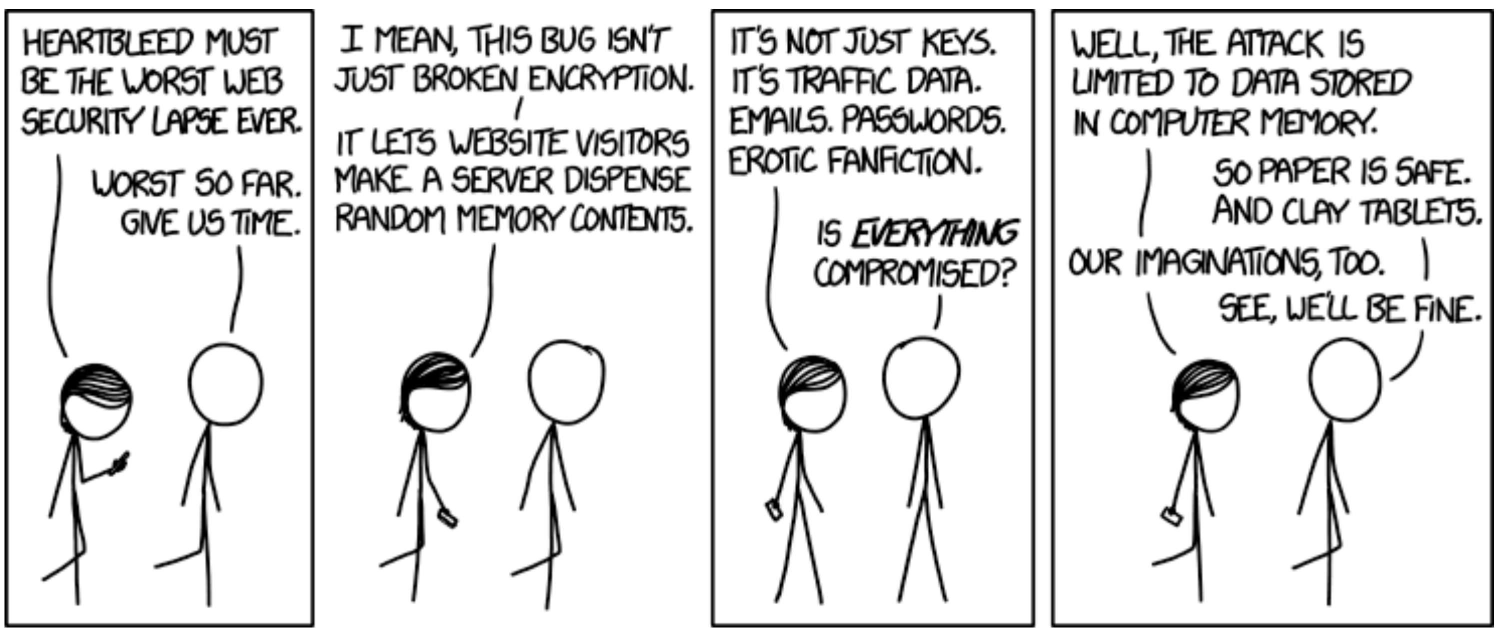
Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu



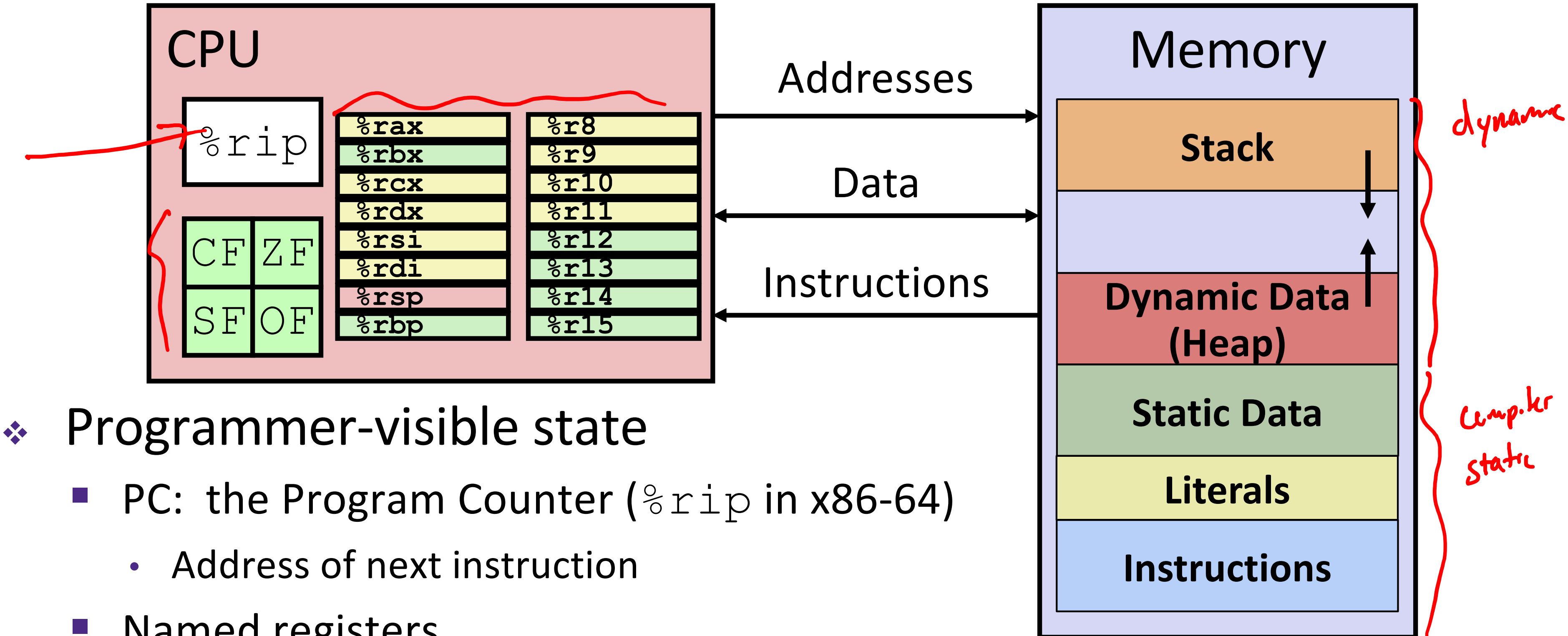
http://xkcd.com/1353/

Administrivia

- Mid-quarter survey due tomorrow (2/14)
- Homework 3 due Friday (2/15)
- Lab 3 releasing today, due next Friday (2/22)

- Midterm **tomorrow** (2/14) take home
  - Will be posted in the morning
  - Due 11:59PM same day
  - See Piazza for rules
    - course material
    - collaboration

Assembly Programmer's View



- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- **Executables**
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

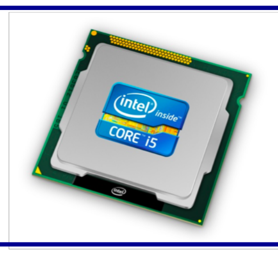
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

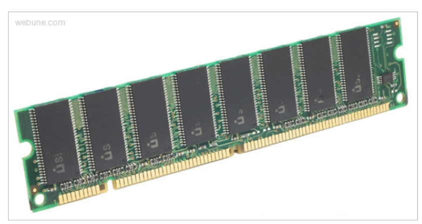
OS:

Windows 10

OS X Yosemite

Computer system:

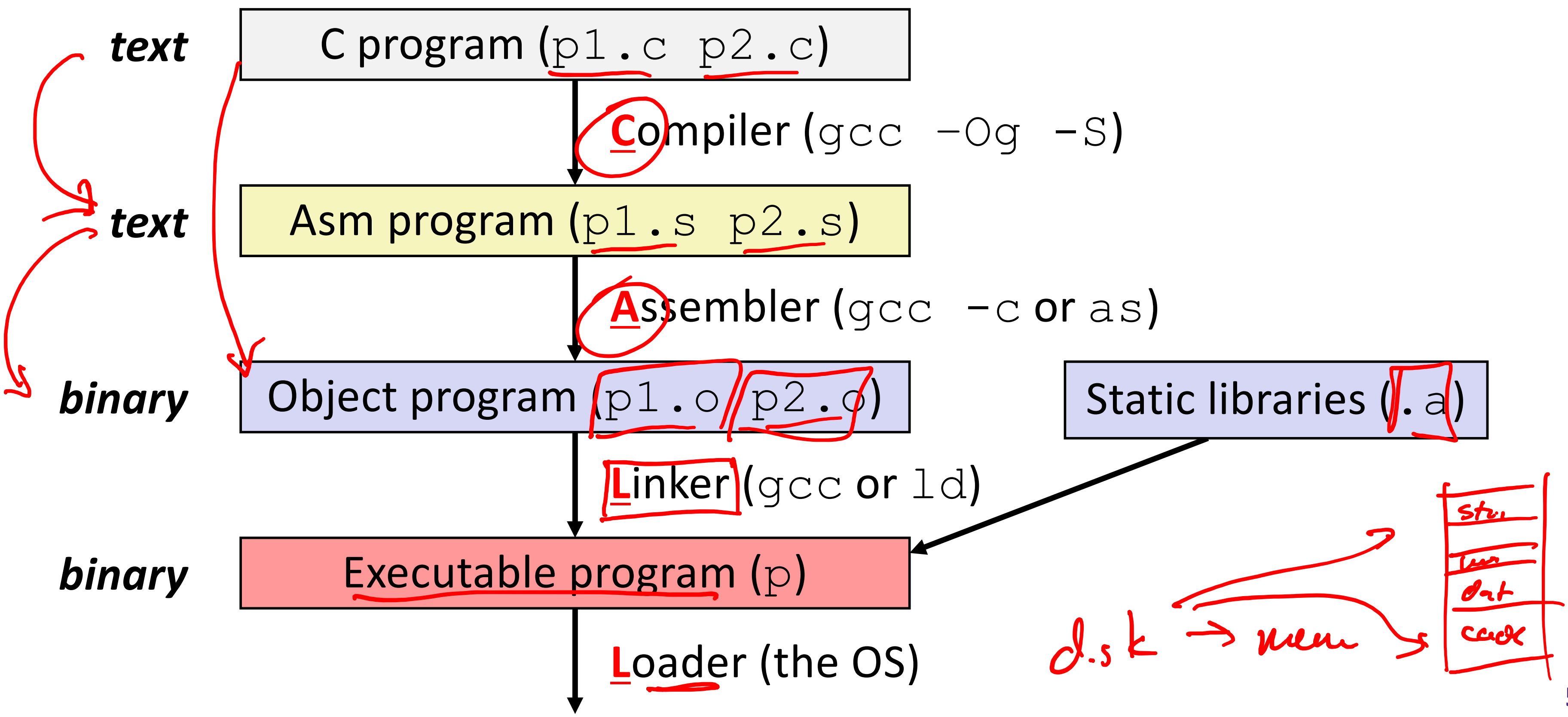




Building an Executable from a C File

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file `p`
- Run with command: `./p`

CALL



Compiler

- **Input:** Higher-level language code (*e.g.* C, Java)
  - `foo.c`
- **Output:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`

#define #ifdef

- First there's a preprocessor step to handle #directives
  - Macro substitution, plus other specialty directives
  - If curious/interested: http://tigcc.ticalc.org/doc/cpp.html
- Super complex, whole courses devoted to these!
- Compiler optimizations -O0 -Og -O1 -O2 -O3
  - "Level" of optimization specified by capital '`O`' flag (*e.g.* `-Og`, `-O3`)
  - Options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compiling Into Assembly

- C Code (`sum.c`)

```
void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}
```

- x86-64 assembly (`gcc -Og -S sum.c`)

sum.s

```
sumstore(long, long, long*):
  addq    %rdi, %rsi
  movq    %rsi, (%rdx)
  ret
```

godbolt.org

Warning: You may get different results with other versions of `gcc` and different compiler settings

Assembler

- **Input:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`
- **Output:** Object files (*e.g.* ELF, COFF)
  - `foo.o`
  - Contains *object code* and *information tables*
- Reads and uses *assembly directives*
  - *e.g.* `.text`, `.data`, `.quad`
  - x86: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
- Produces "machine language"
  - Does its best, but object file is *not* a completed binary
- Example: `gcc -c foo.s`

Producing Machine Language

- **Simple cases:** arithmetic and logical operations, shifts, etc. (addq $1, %rax)
  - All necessary information is contained in the instruction itself
- What about the following?
  - Conditional jump labels
  - Accessing static data (*e.g.* global var or jump table) variable name
  - `call` label
- Addresses and labels are problematic because the final executable hasn't been constructed yet!
  - So how do we deal with these in the meantime?

Object File Information Tables

- **Symbol Table** holds list of "items" that may be used by other files What I have .L2 .L8
  - *Non-local labels* – function names for `call`
  - *Static Data* – variables & literals that might be accessed across files
- **Relocation Table** holds list of "items" that this file needs the address of later (currently undetermined) What I need
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external
- Each file has its own symbol and relocation tables

Object File Format

1) object file header: size and position of the other pieces of the object file table of contents
2) text segment: the machine code (instructions)
3) data segment: data in the source file (binary)
4) relocation table: identifies lines of code that need to be "handled" need
5) symbol table: list of this file's labels and data that can be referenced have
6) debugging information

- More info: ELF format
  - http://www.skyfree.org/linux/references/ELF_Format.pdf

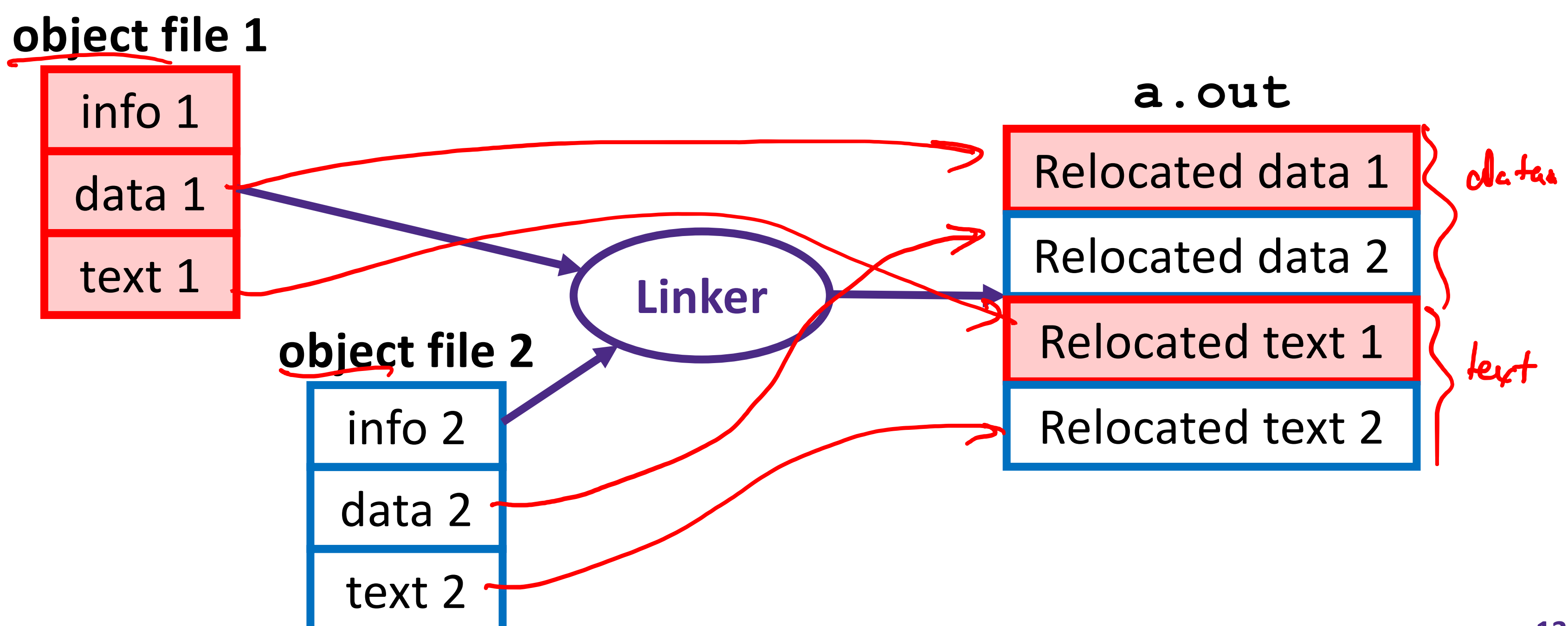
Linker

- **Input:** Object files (e.g. ELF, COFF)
  - `foo.o` lib.o bar.o
- **Output:** executable binary program
  - `a.out` ./a.out

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program
  - But you might have to relink

Linking

1) Take text segment from each .o file and put them together
2) Take data segment from each .o file, put them together, and concatenate this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry using Symbol Tables



Disassembling Object Code

- Disassembled:

```
0000000000400536 <sumstore>:
  400536:   48 01 fe        add    %rdi,%rsi
  400539:   48 89 32        mov    %rsi,(%rdx)
  40053c:   c3              retq
```

- **Disassembler** (`objdump -d sum`)
  - Useful tool for examining object code (`man 1 objdump`)
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on either `a.out` (complete executable) or `.o` file

Disassembling Object Code

- Executable has **addresses**

assembler

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00     mov    $0x0,%eax
  4004fb:  48 85 ff           test   %rdi,%rdi
  4004fe:  74 13              je     400513 <pcount_r+0x1d>
  400500:  53                 push   %rbx
  400501:  48 89 fb           mov    %rdi,%rbx
  400504:  48 d1 ef           shr    %rdi
  400507:  e8 ea ff ff ff     callq  4004f6 <pcount_r>
  40050c:  83 e3 01           and    $0x1,%ebx
  40050f:  48 01 d8           add    %rbx,%rax
  400512:  5b                 pop    %rbx
  400513:  f3 c3              rep ret
```

  - `gcc -g pcount.c –o pcount`
  - `objdump –d pcount`

A Picture of Memory (64-bit view)

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00    mov    $0x0,%eax
  4004fb:  48 85 ff          test   %rdi,%rdi
  4004fe:  74 13             je     400513 <pcount_r+0x1d>
  400500:  53                push   %rbx
  400501:  48 89 fb          mov    %rdi,%rbx
  400504:  48 d1 ef          shr    %rdi
  400507:  e8 ea ff ff ff    callq  4004f6 <pcount_r>
  40050c:  83 e3 01          and    $0x1,%ebx
  40050f:  48 01 d8          add    %rbx,%rax
  400512:  5b                pop    %rbx
  400513:  f3 c3             rep ret
```

0|8	1|9	2|a	3|b	4|c	5|d	6|e	7|f	
								0x00
								0x08
								0x10
			...					...
						b8	00	0x4004f0
00	00	00	48	85	ff	74	13	0x4004f8
53	48	89	fb	48	d1	ef	e8	0x400500
ea	ff	ff	ff	83	e3	01	48	0x400508
01	d8	5b	f3	c3				0x400510

Loader

- **Input:** executable binary program, command-line arguments
  - `./a.out arg1 arg2`
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
- Memory sections (Instructions, Static Data, Stack) are set up
- Registers are initialized

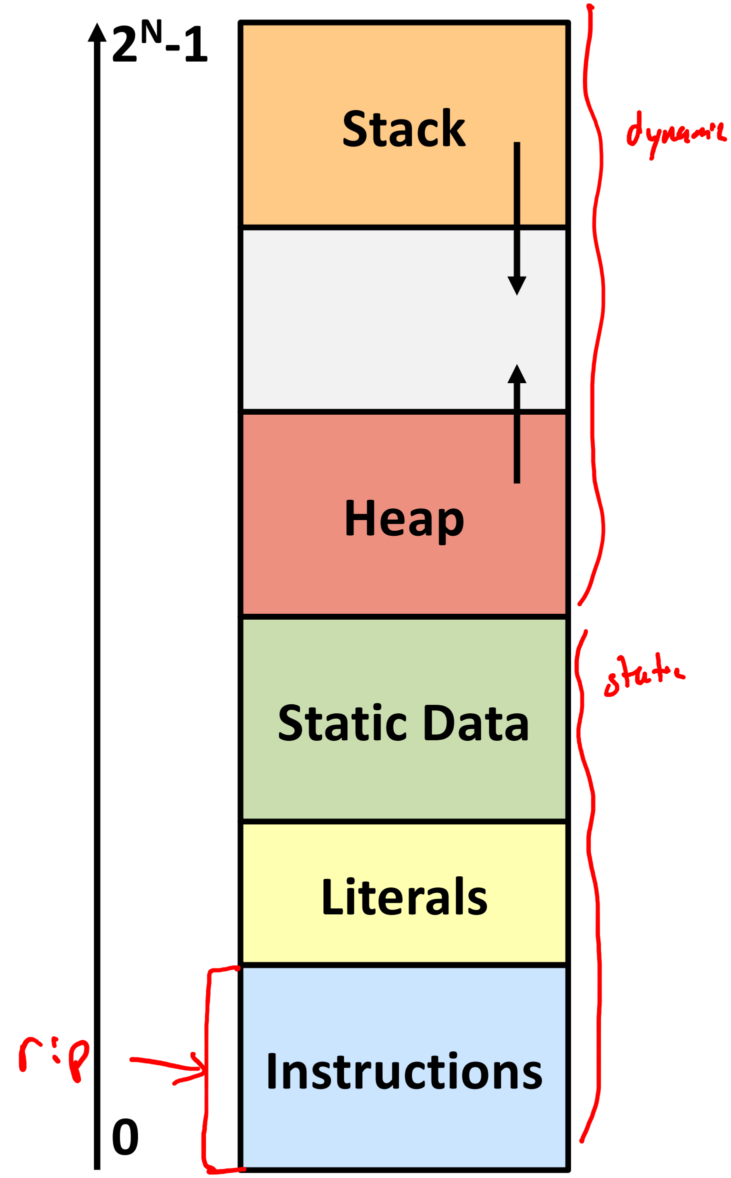
Buffer Overflows

- Address space layout (more details!)
- Input buffers on the stack
- Overflowing buffers and injecting code
- Defenses against buffer overflows

not drawn to scale

Review: General Memory Layout

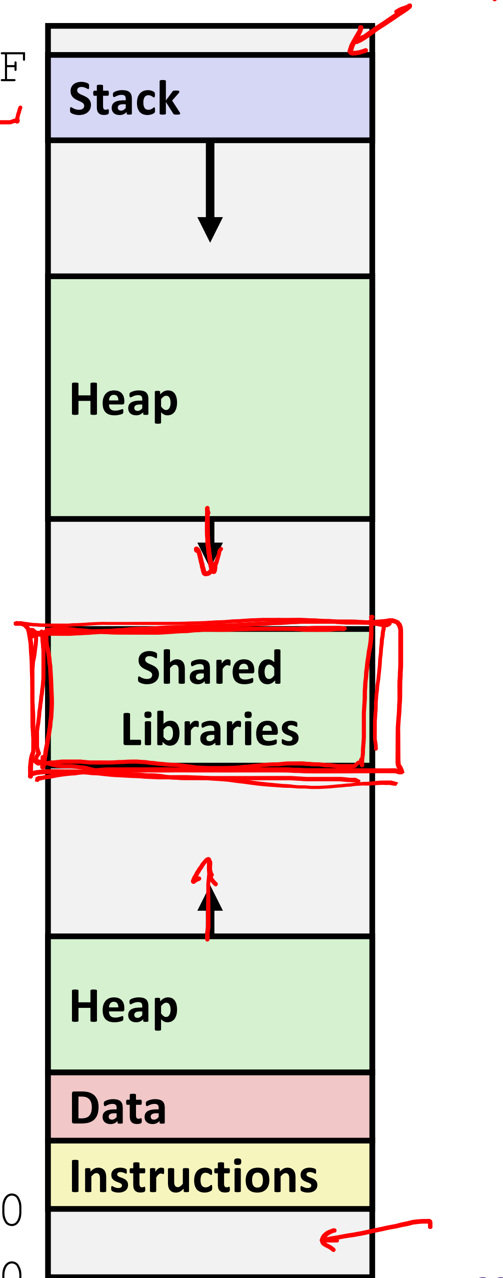
- Stack
  - Local variables (procedure context)
- Heap
  - Dynamically allocated as needed
  - `malloc()`, `calloc()`, `new`, …
- Statically allocated Data
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)
- Code/Instructions
  - Executable machine instructions
  - Read-only



not drawn to scale

x86-64 Linux Memory Layout

- Stack
  - Runtime stack has 8 MiB limit
- Heap
  - Dynamically allocated as needed
  - `malloc(), calloc(), new, …`
- Statically allocated data (Data)
  - Read-only: string literals
  - Read/write: global arrays and variables
- Code / Shared Libraries
  - Executable machine instructions
  - Read-only

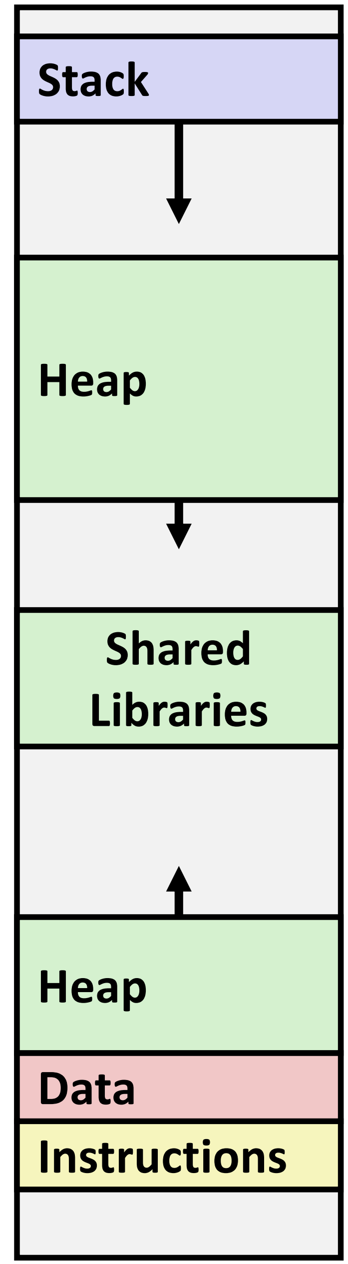


Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```

Where does everything go?

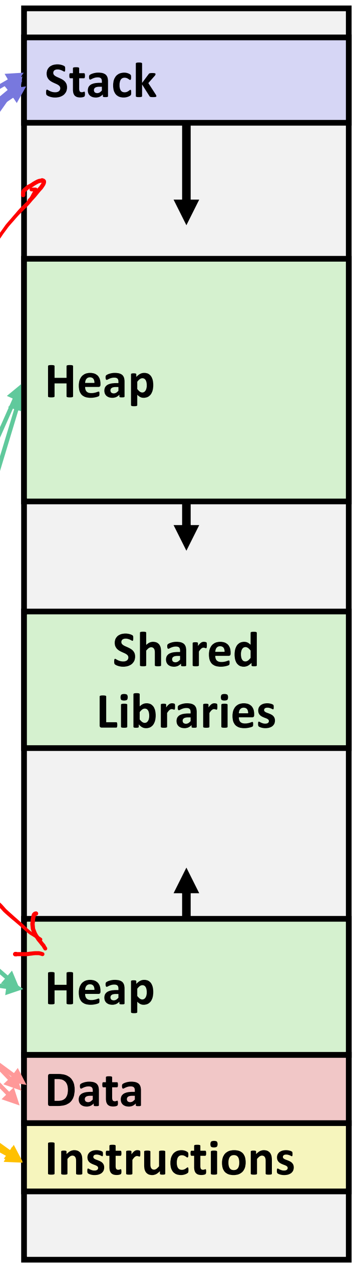
not drawn to scale



not drawn to scale

Memory Allocation Example

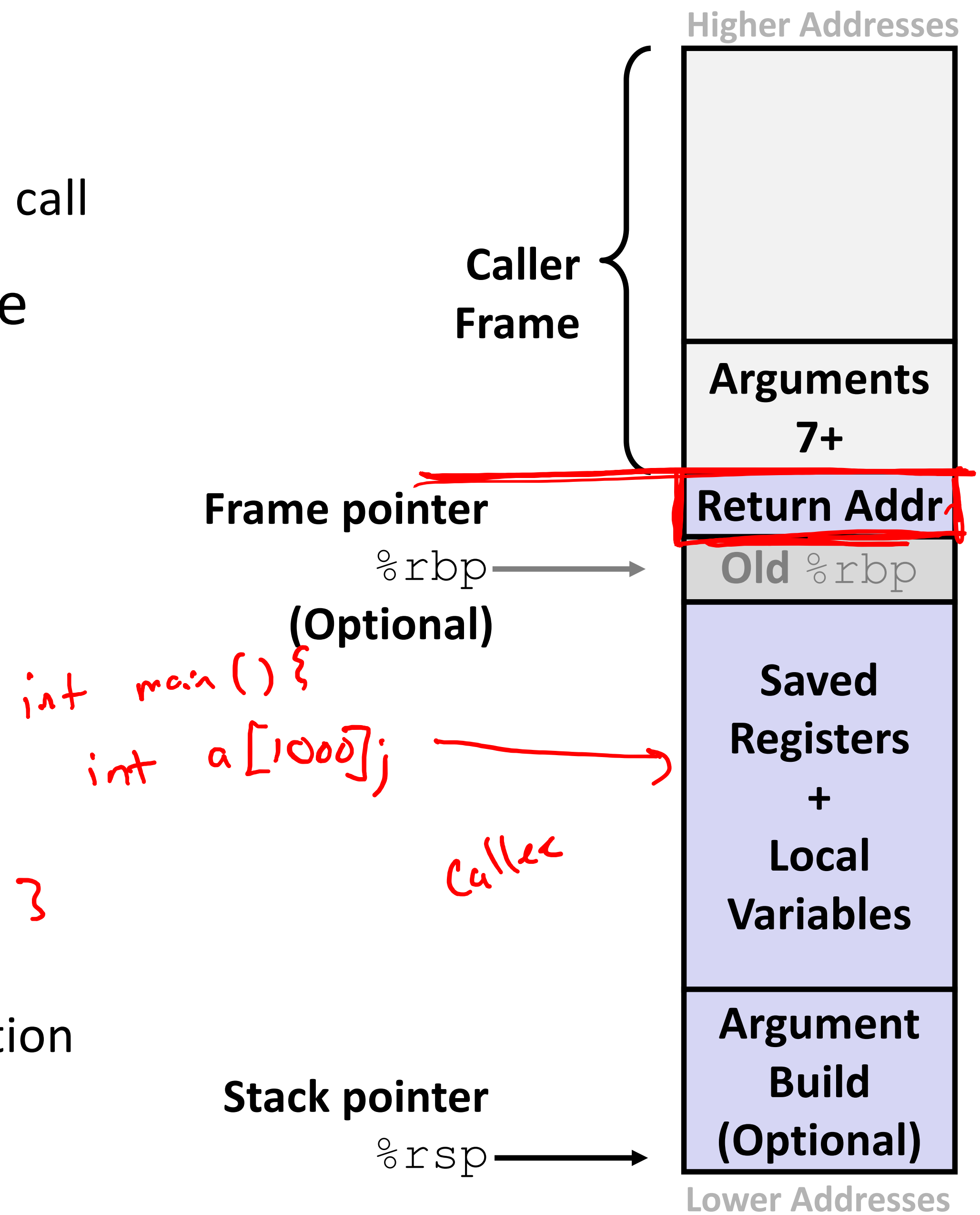
```
char big_array[1L<<24];   /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```



Where does everything go?

Reminder: x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Arguments (if > 6 args) for this call
- Current/ Callee Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables (if can't be kept in registers)
  - "Argument build" area (If callee needs to call another function -parameters for function about to call, if needed)



Buffer Overflow in a Nutshell

- Characteristics of the traditional Linux memory layout provide opportunities for malicious programs
  - Stack grows "backwards" in memory
  - Data and instructions both stored in the same memory
- C does not check array bounds
  - Some Unix/Linux/C functions don't check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)

Buffer Overflow in a Nutshell

- Buffer overflows on the stack can overwrite "interesting" data → return address ← change
  - Attackers just choose the right inputs
- Simplest form (sometimes called "stack smashing")
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure
- Why is this a big deal?
  - It is (was?) the #1 *technical* cause of security vulnerabilities
    - #1 *overall* cause is social engineering / user ignorance

String Library Code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

pointer to start of an array

same as:

```
*p = c;
p++;
```

  - What could go wrong in this code?

String Library Code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char* gets(char* dest) {
    int c = getchar();
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify **limit** on number of characters to read

- Similar problems with other Unix functions:
  - `strcpy`: Copies string of arbitrary length to a dst
  - `scanf`, `fscanf`, `sscanf`, when given `%s` specifier

Vulnerable Buffer Code

```
/* Echo Line */
void echo() {
    char buf[8];   /* Way too small! */
    gets(buf);  ← write
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

```
unix> ./buf-nsp
Enter string: 1234567890123456
Illegal instruction
```

```
unix> ./buf-nsp
Enter string: 12345678901234567
Segmentation Fault
```

Buffer Overflow Disassembly (buf-nsp)

24 bytes 16 bytes

echo:

```
0000000000400597 <echo>:
  400597:  48 83 ec 18           sub    $0x18,%rsp
    ...                            ... other stuff ...
  4005aa:  48 8d 7c 24 08        lea    0x8(%rsp),%rdi
  4005af:  e8 d6 fe ff ff        callq  400480 <gets@plt>
  4005b4:  48 89 7c 24 08        lea    0x8(%rsp),%rdi
  4005b9:  e8 b2 fe ff ff        callq  4004a0 <puts@plt>
  4005be:  48 83 c4 18           add    $0x18,%rsp
  4005c2:  c3                    retq
```

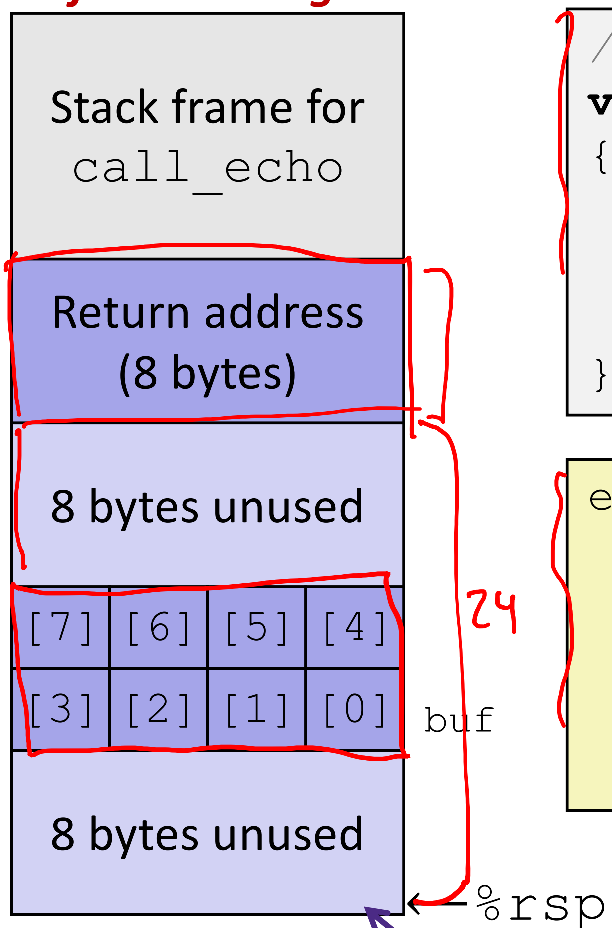
call_echo:

```
00000000004005c3 <call_echo>:
  4005c3:  48 83 ec 08           sub    $0x8,%rsp
  4005c7:  b8 00 00 00 00        mov    $0x0,%eax
  4005cc:  e8 c6 ff ff ff        callq  400597 <echo>
  4005d1:  48 83 c4 08           add    $0x8,%rsp
  4005d5:  c3                    retq
```

return address (4005d1)

Buffer Overflow Stack

Before call to gets



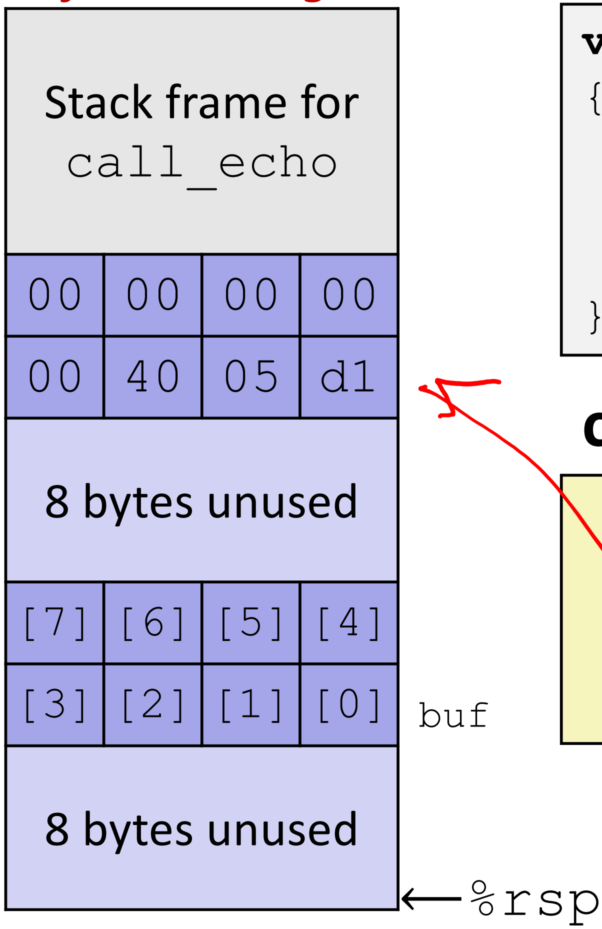

```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  call   gets
   ...
```

Note: addresses increasing right-to-left, bottom-to-top

Buffer Overflow Example

Before call to gets




```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

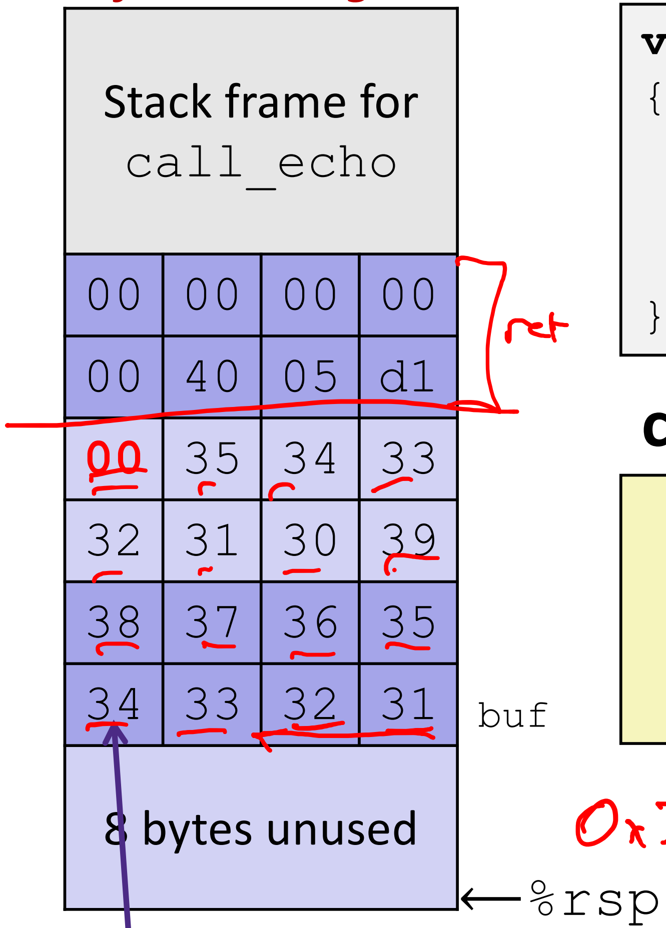
```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  call   gets
   ...
```

call_echo:

```
  . . .
  4005cc:   callq   400597 <echo>
  4005d1:   add     $0x8,%rsp
  . . .
```

Buffer Overflow Example #1

After call to gets



Note: Digit "*N*" is just 0x3*N* in ASCII!

0x31 = '1'

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  call   gets
   ...
```

call_echo:

```
  . . .
  4005cc:   callq   400597 <echo>
  4005d1:   add     $0x8,%rsp
  . . .
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Example #2

After call to gets

Stack frame for call_echo			
00	00	00	00
00	40	05	00
36	35	34	33
32	31	30	39
38	37	36	35
34	33	32	31
8 bytes unused			

`buf` (beside the row 34 33 32 31); `←%rsp` (at the bottom of the frame)

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
   ...
  leaq  8(%rsp), %rdi
  call  gets
   ...
```

call_echo:

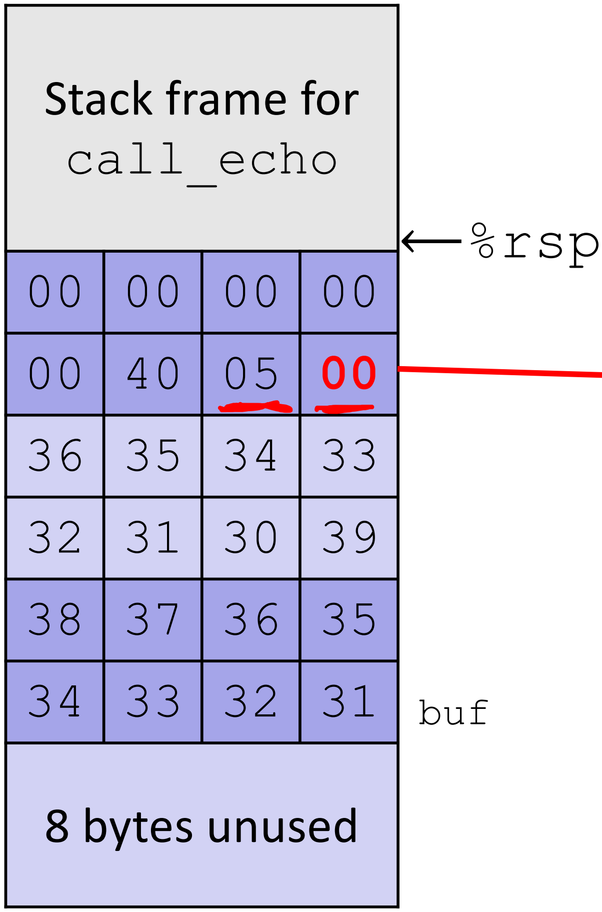
```
 . . .
4005cc:  callq  400597 <echo>
4005d1:  add    $0x8,%rsp
 . . .
```

```
unix> ./buf-nsp
Enter string: 1234567890123456
Illegal instruction
```

Overflowed buffer and corrupted return pointer

Buffer Overflow Example #2 Explained

After return from echo




```
00000000004004f0 <deregister_tm_clones>:
  4004f0:   push   %rbp
  4004f1:   mov    $0x601040,%eax
  4004f6:   cmp    $0x601040,%rax
  4004fc:   mov    %rsp,%rbp
  4004ff:   je     400518
  400501:   mov    $0x0,%eax
  400506:   test   %rax,%rax
  400509:   je     400518
  40050b:   pop    %rbp
  40050c:   mov    $0x601040,%edi
  400511:   jmpq   *%rax
  400513:   nopl   0x0(%rax,%rax,1)
  400518:   pop    %rbp
  400519:   retq
```

"Returns" to a byte that is not the beginning of an instruction, so program signals `SIGILL`, `Illegal instruction`

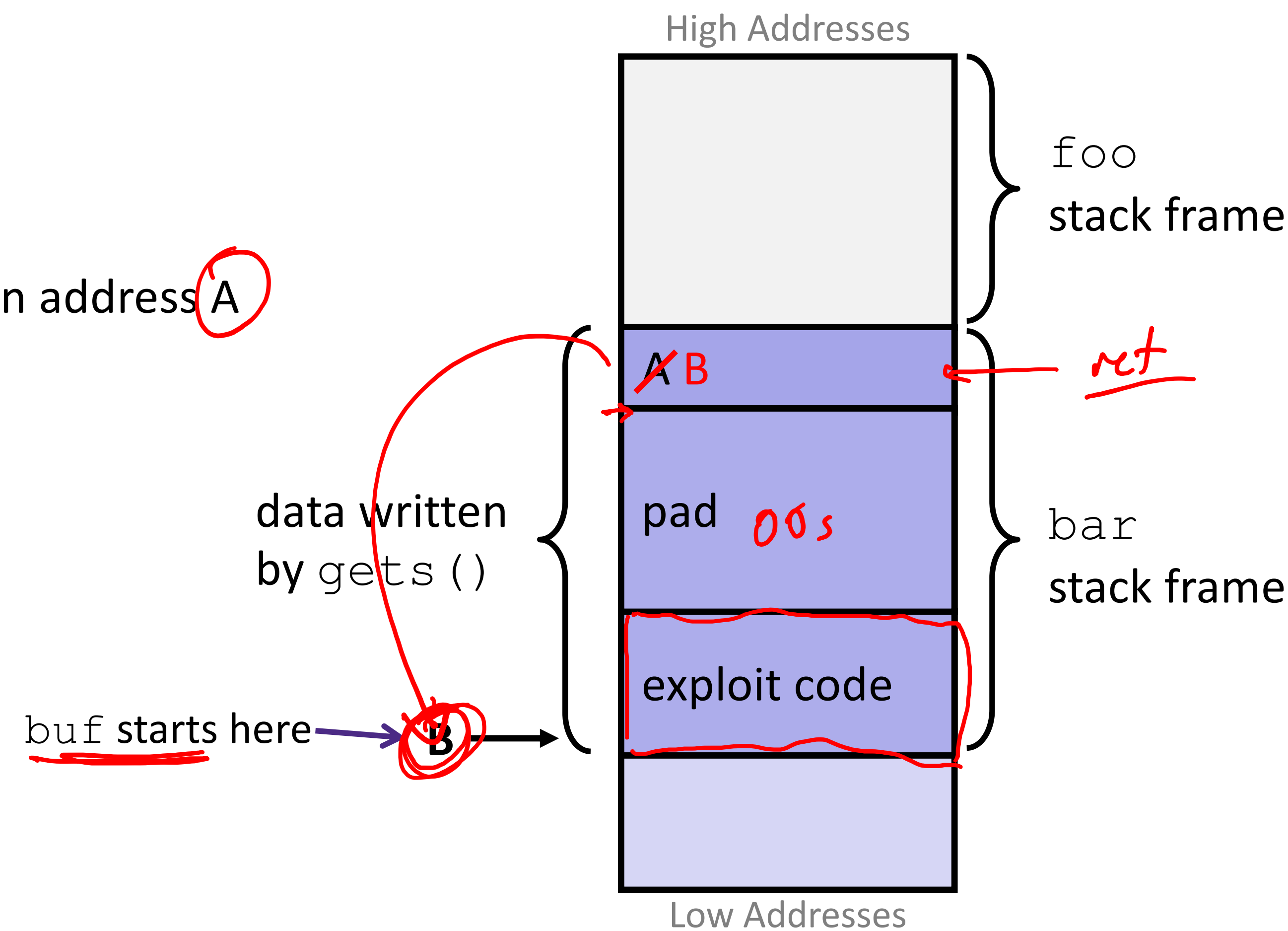
Malicious Use of Buffer Overflow: Code Injection Attacks

```
void foo(){
  bar();
}
```

return address A

```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

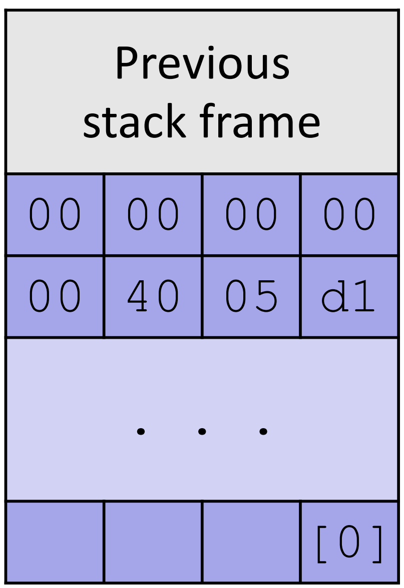
Stack after call to `gets()`



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When `bar()` executes `ret`, will jump to exploit code

Peer Instruction Question

- `smash_me` is vulnerable to stack smashing!
- What is the maximum number of characters that `gets` can safely read without corrupting the return address to a stack address (in x86-64 Linux)?



64 − 16 = 48   47 + '\0'

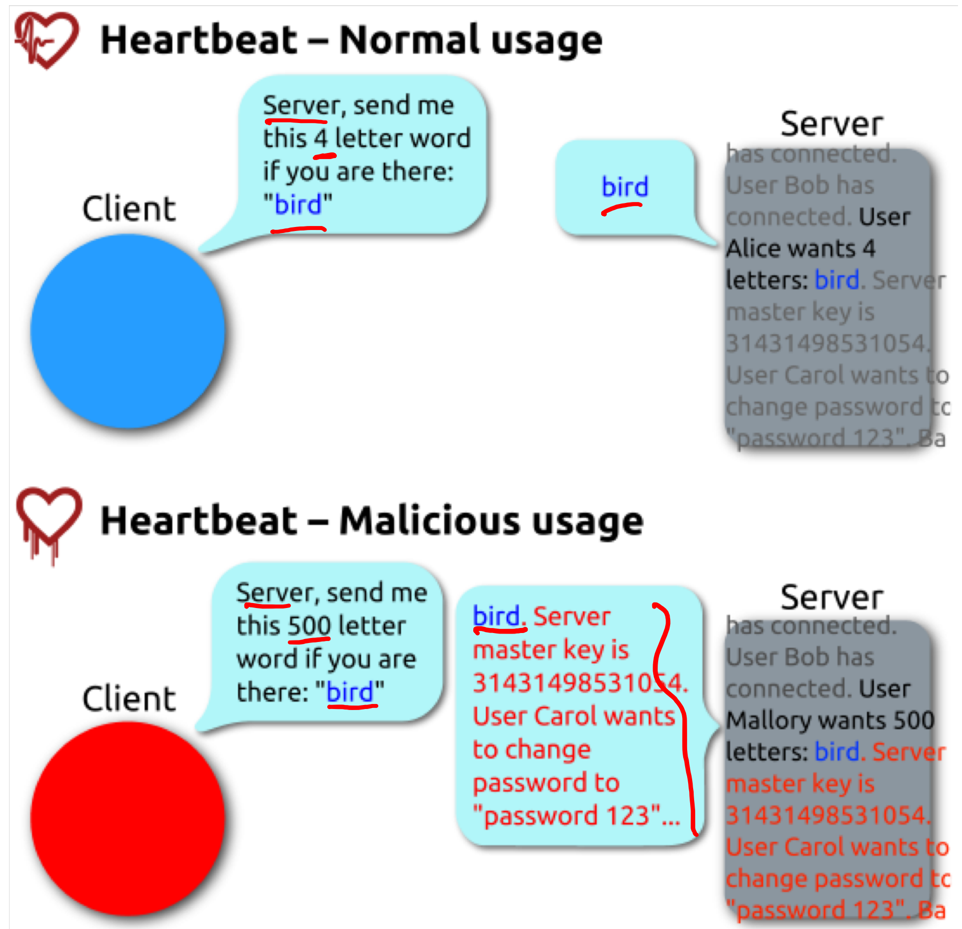
```
smash_me:
   subq   $0x40, %rsp
    ...
   leaq   16(%rsp), %rdi
   call   gets
    ...
```

Exploits Based on Buffer Overflows

- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - *Still happens!!*
    - Heartbleed (2014, affected 17% of servers)
    - Cloudbleed (2017)
  - *Fun:* Nintendo hacks
    - Using glitches to rewrite code: https://www.youtube.com/watch?v=TqK-2jUQBUY
    - FlappyBird in Mario: https://www.youtube.com/watch?v=hB6eY73sLV0

Heartbleed (2014)

- Buffer over-read in OpenSSL
  - Open source security library
  - Bug in a small range of versions
- "Heartbeat" packet
  - Specifies length of message
  - Server echoes it back
  - Library just "trusted" this length
  - Allowed attackers to read contents of memory anywhere they wanted
- Est. 17% of Internet affected
  - "Catastrophic"
  - Github, Yahoo, Stack Overflow, Amazon AWS, ...



By FenixFeather - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32276981

Dealing with buffer overflow attacks

1) Avoid overflow vulnerabilities
2) Employ system-level protections
3) Have compiler use "stack canaries"

1) Avoid Overflow Vulnerabilities in Code

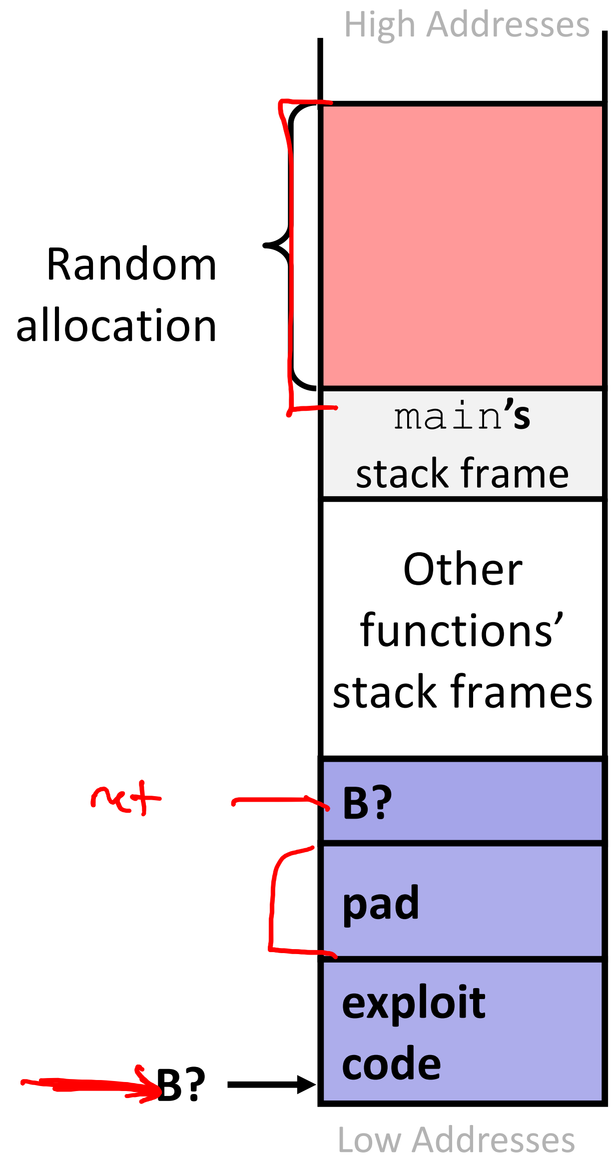
```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    fgets(buf, 8, stdin);
    puts(buf);
}
```

- Use library routines that limit string lengths
  - `fgets` instead of `gets` (2nd argument to `fgets` sets limit)
  - `strncpy` instead of `strcpy`
  - Don't use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer

buffer

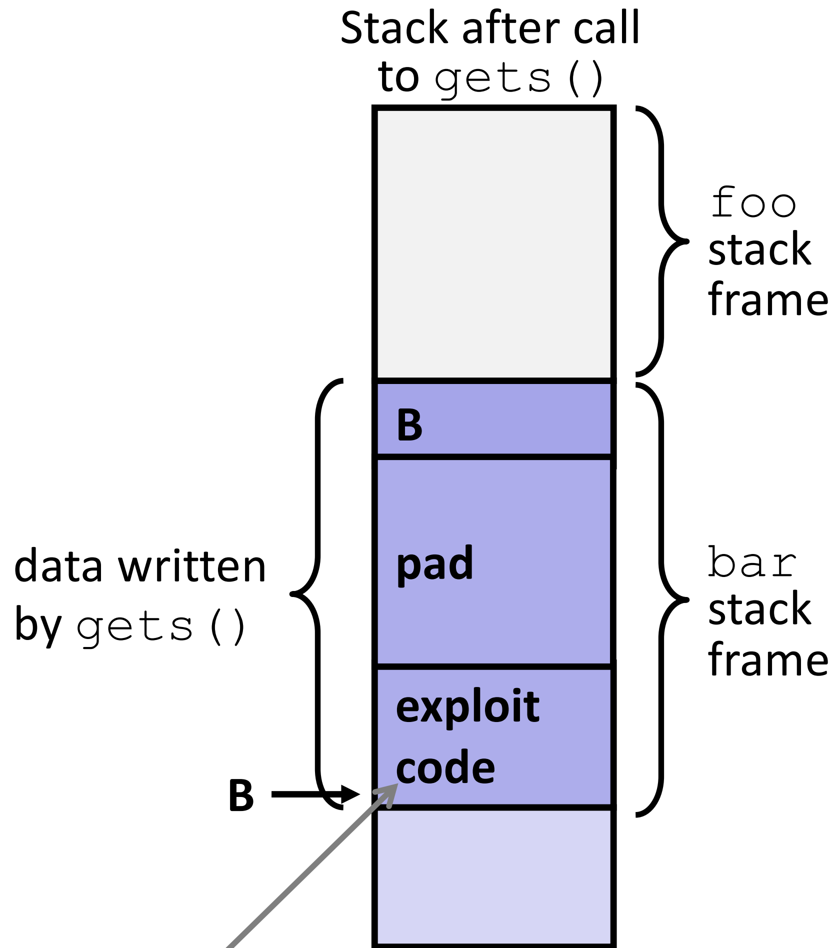
2) System-Level Protections

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code
- Example: Code from Slide 6 executed 5 times; address of variable `local` =
  - `0x7ffd19d3f8ac`
  - `0x7ffe8a462c2c`
  - `0x7ffe927c905c`
  - `0x7ffefd5c27dc`
  - `0x7fffa0175afc`
  - Stack repositioned each time program executes



2) System-Level Protections

- **Non-executable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - x86-64 added explicit "execute" permission
  - Stack marked as non-executable
    - Do *NOT* execute code in Stack, Static Data, or Heap regions
    - Hardware support needed



Any attempt to execute this code will fail

3) Stack Canaries

- Basic Idea: place special value ("canary") on stack just beyond buffer
  - *Secret* value known only to compiler
  - "After" buffer but before return address
  - Check for corruption before exiting function
- GCC implementation (now default)
  - `-fstack-protector`
  - Code back on Slide 14 (`buf-nsp`) compiled with `-fno-stack-protector` flag

```
unix>./buf
Enter string: 12345678
12345678
```

```
unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```

Summary

1) Avoid overflow vulnerabilities
   - Use library routines that limit string lengths

2) Employ system-level protections
   - Randomized Stack offsets
   - Code on the Stack is not executable

3) Have compiler use "stack canaries"