Machine-Level Programming V: Advanced Topics

CS351: Systems Programming
Day 10: Sep. 22, 2022

Instructor:

Nik Sultana

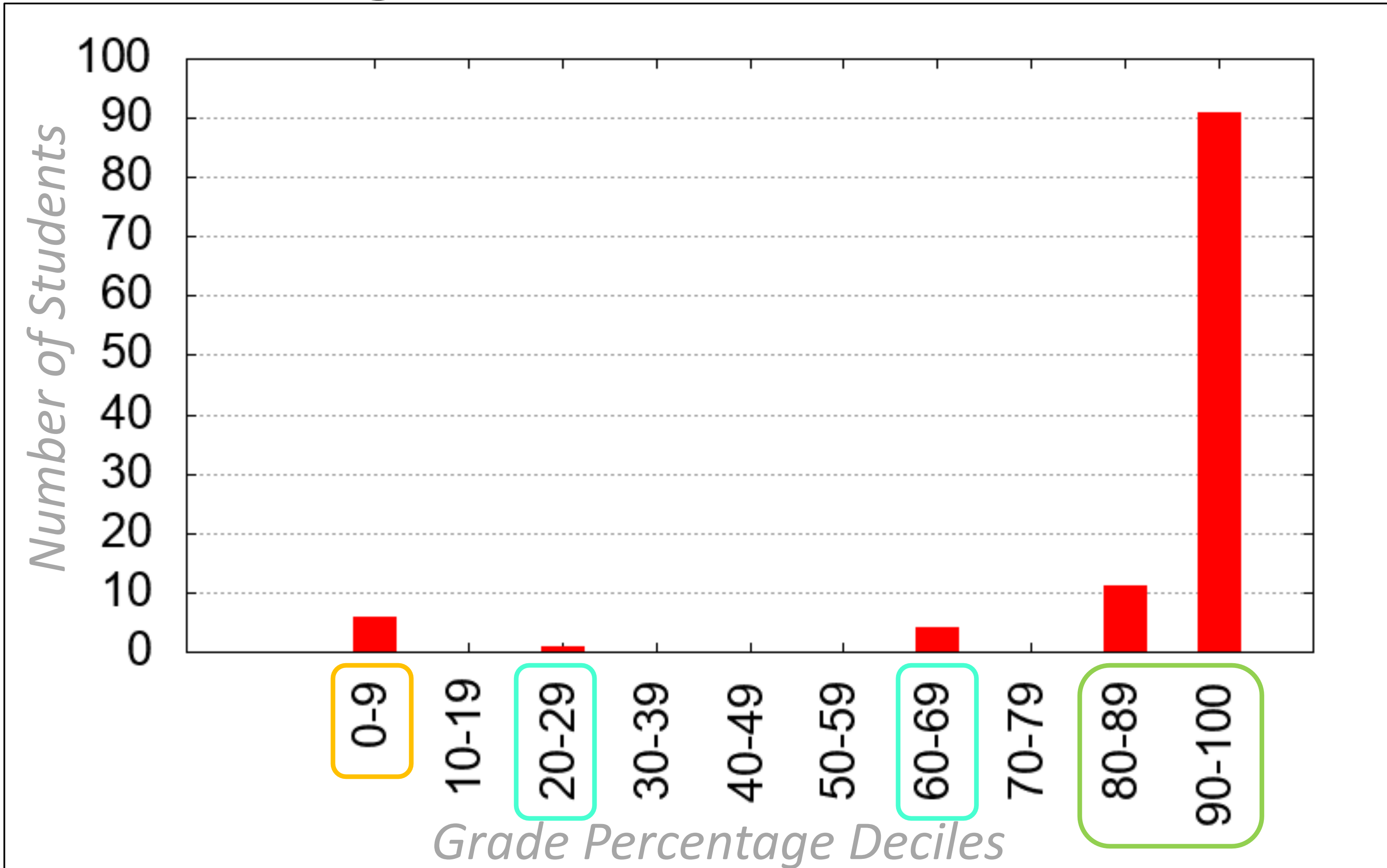
Slides adapted from Bryant and O'Hallaron

Next time: back to in-person in SB104

Sep 20 **LEC 9:** Machine Prog: Data **Preparation:** Read CS:APP 3.8-3.9	Sep 21	Sep 22 **LEC 10:** Machine Prog: **Preparation:** Read CS:
Sep 27 **LEC 11:** C review **Preparation:** Practice CS:APP and K&R exercises. Come prepared with questions.	Sep 28	Sep 29 **LEC 12:** Linking **Preparation:** Read CS:

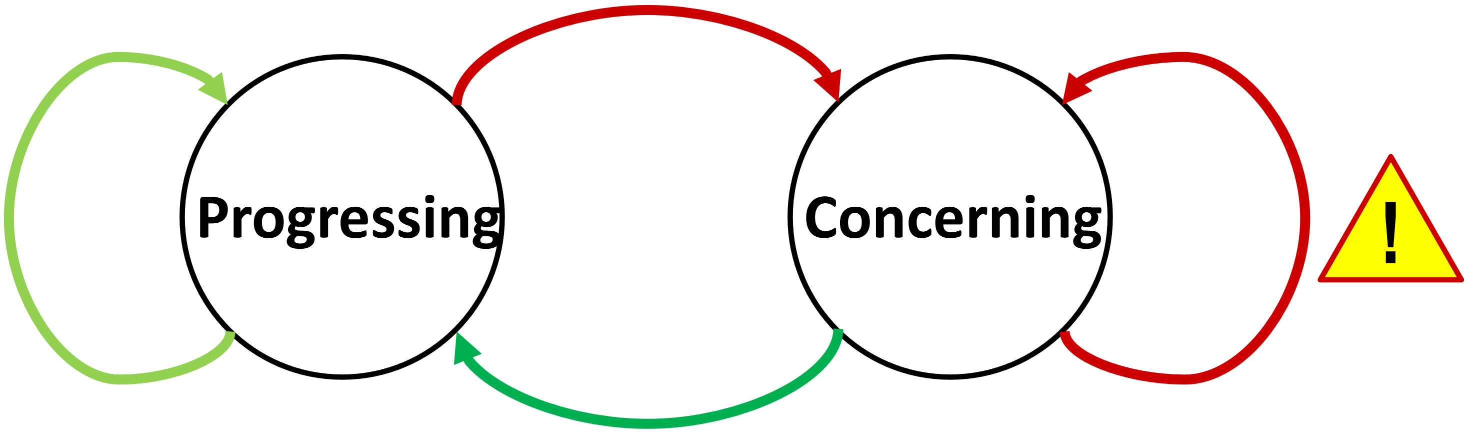
- **Tuesday: TA Kirtan will be giving a review of the C language**
- **Monday: deadline for 2nd lab assignment**

First lab assignment

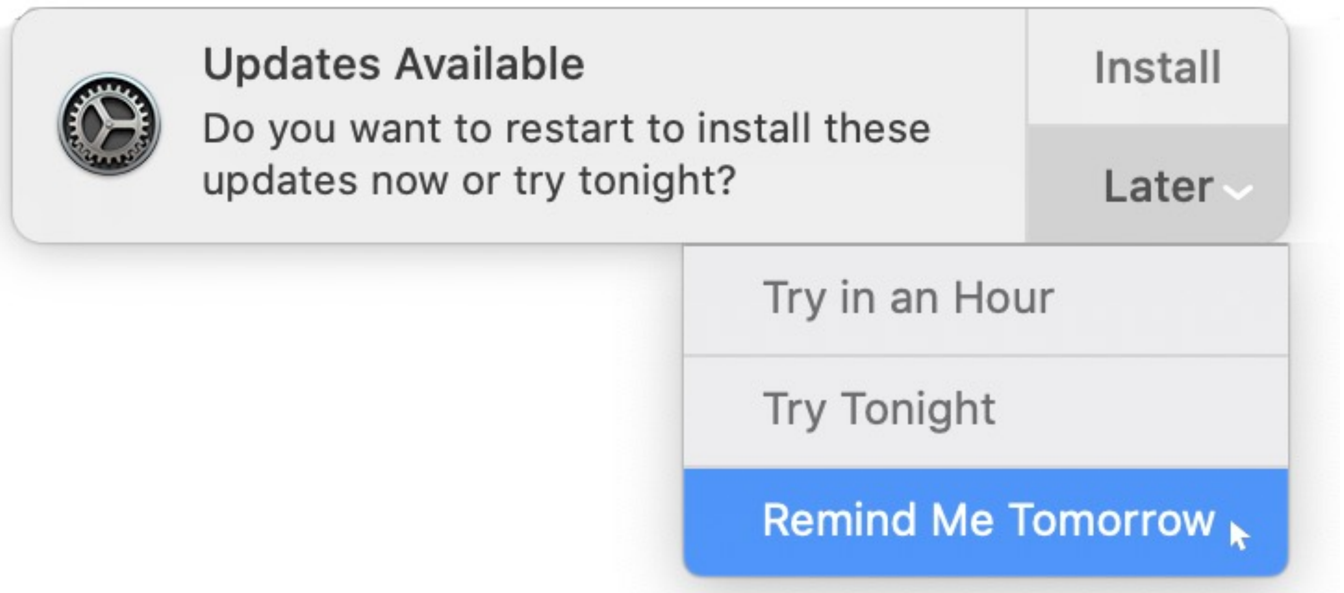


- **Good overall!**
- Zero grades: ensure timely completion of lab.
- Low grades: work with TA to get feedback.

Tracking progress



Start assignments early



- Starting later -> Finishing later
- Avoid last-minute crunch, you'll perform better overall.

Pointers in C

- We encountered pointers several times so far.
  As with any language: **practice makes perfect!**
- K&R Chapter 5 (can get from library – see announcement on Blackboard and at last lecture).

2. Consider the following C declaration:

```
int iarr[100];
void *p = iarr;
```

Which of the following expressions is semantically equivalent to "`iarr[50]`"?

(a) `*(int *)((char *)p + 50 * sizeof(int))`

(b) `*(int *)(p + 50 * sizeof(int *))`

(c) `((int *)((char *)p + 50))[0]`

(d) `*(char *)((int *)p + 50)`

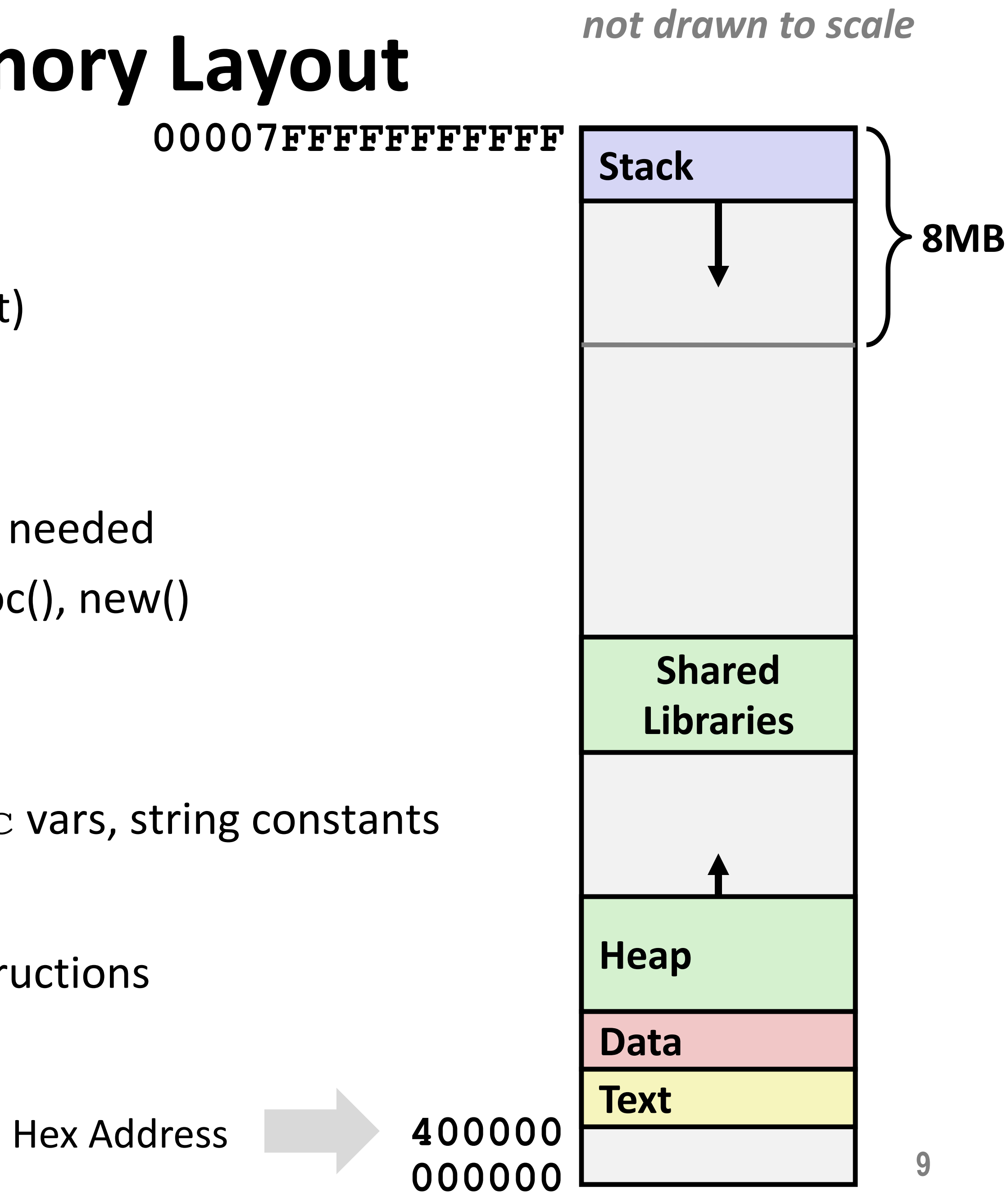
- See past exam questions:
  http://www.cs.iit.edu/~nsultana1/teaching/F22CS351/otherresources.html

Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
- **Unions**

x86-64 Linux Memory Layout

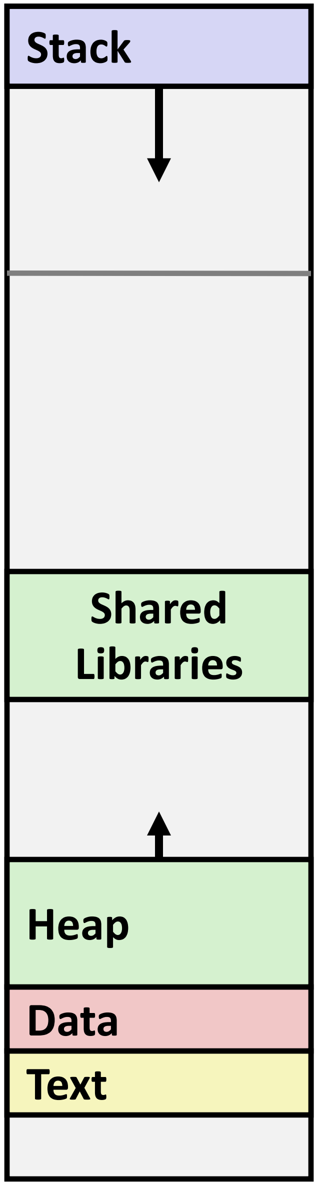
- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



Memory Allocation Example

not drawn to scale

```
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 ... */
}
```



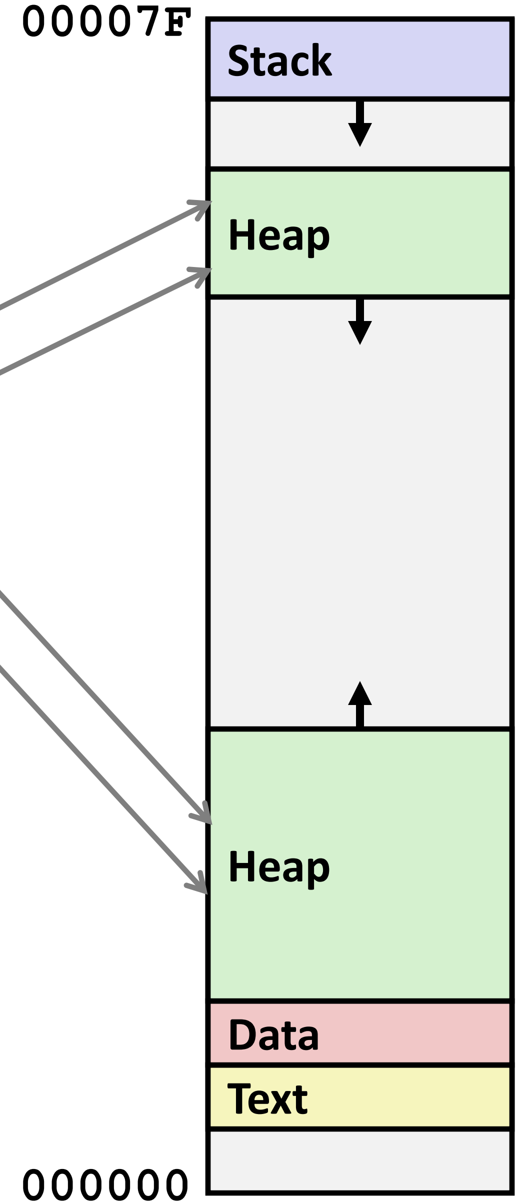
Where does everything go?

x86-64 Example Addresses

address range ~2^{47}

not drawn to scale

`local`	`0x00007ffe4d3be87c`
`p1`	`0x00007f7262a1e010`
`p3`	`0x00007f7162a1d010`
`p4`	`0x000000008359d120`
`p2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`



Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
- Unions

Recall: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  →   3.14
fun(1)  →   3.14
fun(2)  →   3.1399998664856
fun(3)  →   2.00000061035156
fun(4)  →   3.14
fun(6)  →   Segmentation fault
```

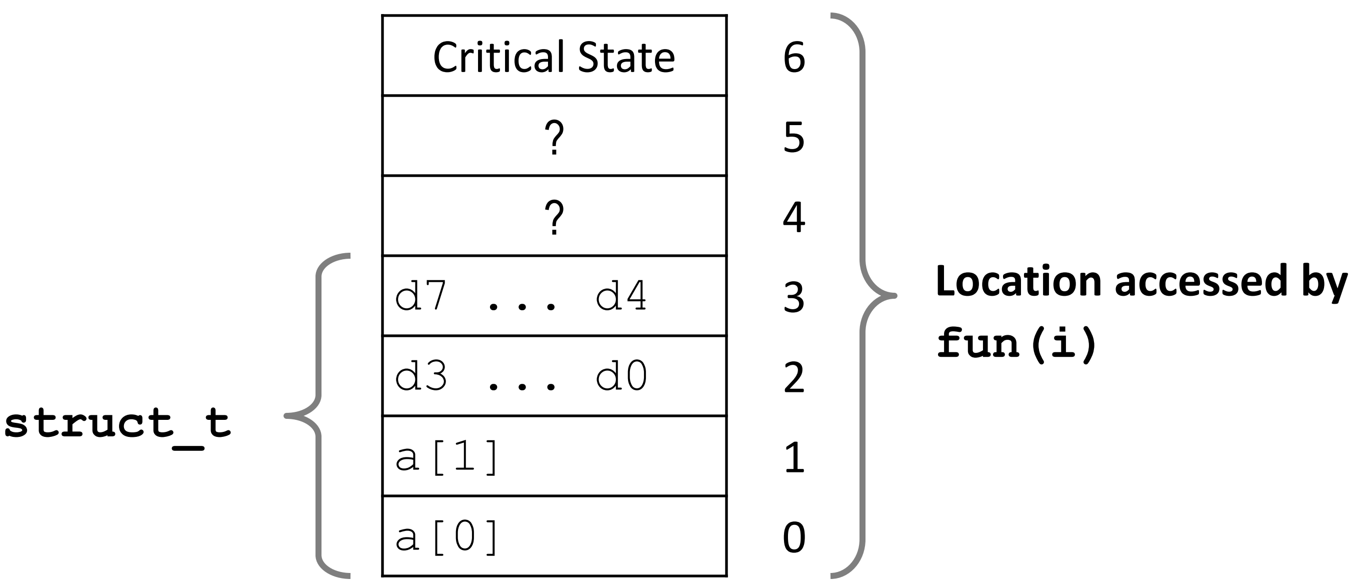
- Result is system specific

Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.00000061035156
fun(4)  →  3.14
fun(6)  →  Segmentation fault
```

Explanation:



Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

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 library functions**
  - **`strcpy`, `strcat`**: Copy strings of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

← btw, how big is big enough?

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

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

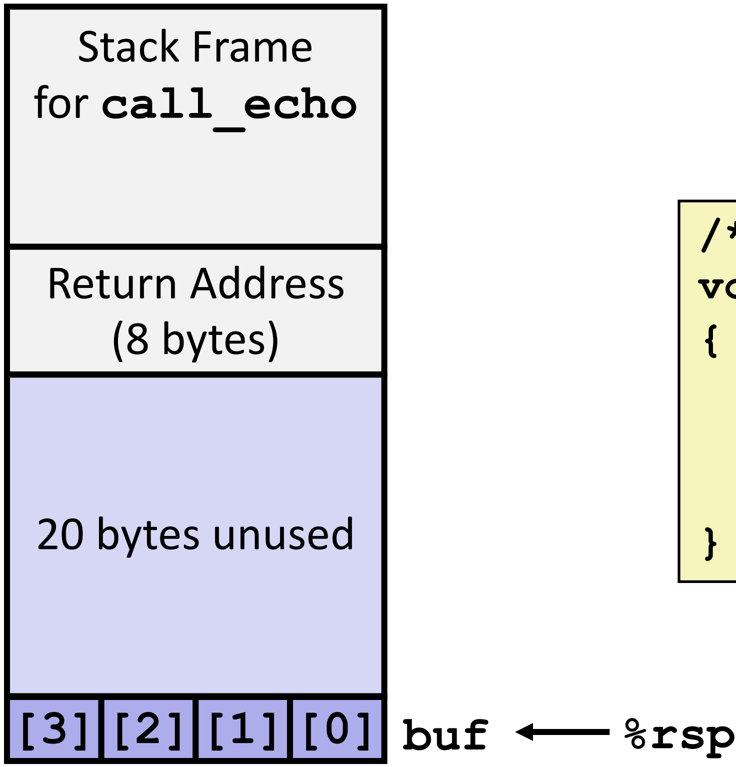
```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18           sub    $0x18,%rsp
 4006d3:  48 89 e7              mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
 4006db:  48 89 e7              mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:  48 83 c4 18           add    $0x18,%rsp
 4006e7:  c3                    retq
```

call_echo:

```
4006e8:   48 83 ec 08           sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00        mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff        callq  4006cf <echo>
 4006f6:  48 83 c4 08           add    $0x8,%rsp
 4006fa:  c3                    retq
```

Buffer Overflow Stack

Before call to gets

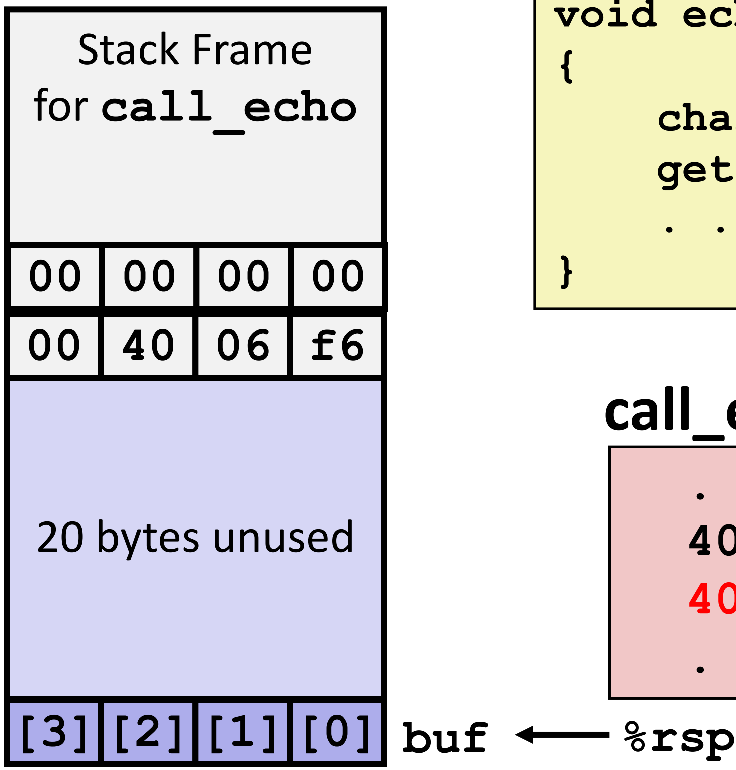



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example

Before call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

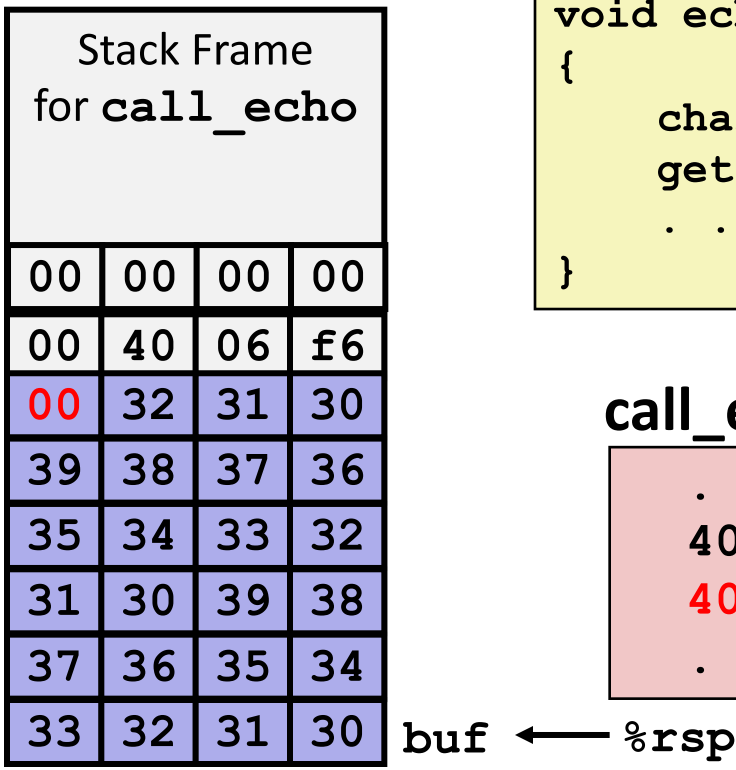
```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $0x8,%rsp
  . . .
```

Buffer Overflow Stack Example #1

After call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

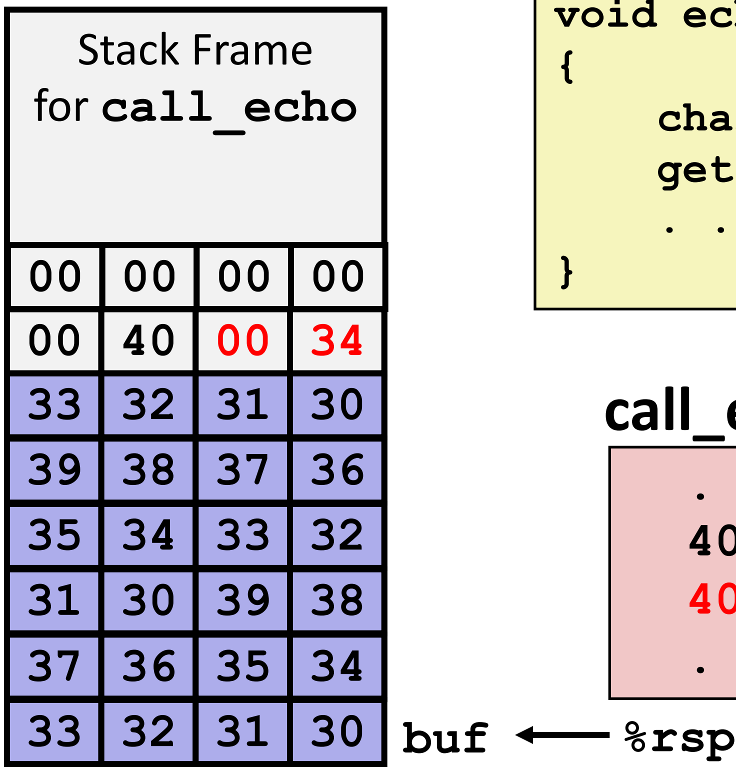
```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

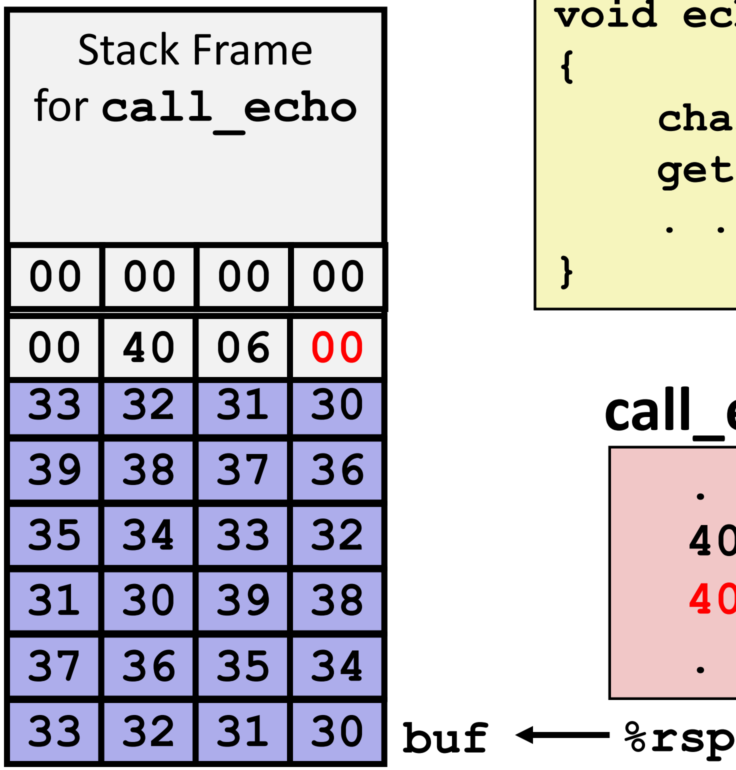
```
 . . .
 4006f1:   callq   4006cf <echo>
 4006f6:   add     $0x8,%rsp
 . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123 4
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

Buffer Overflow Stack Example #3

After call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

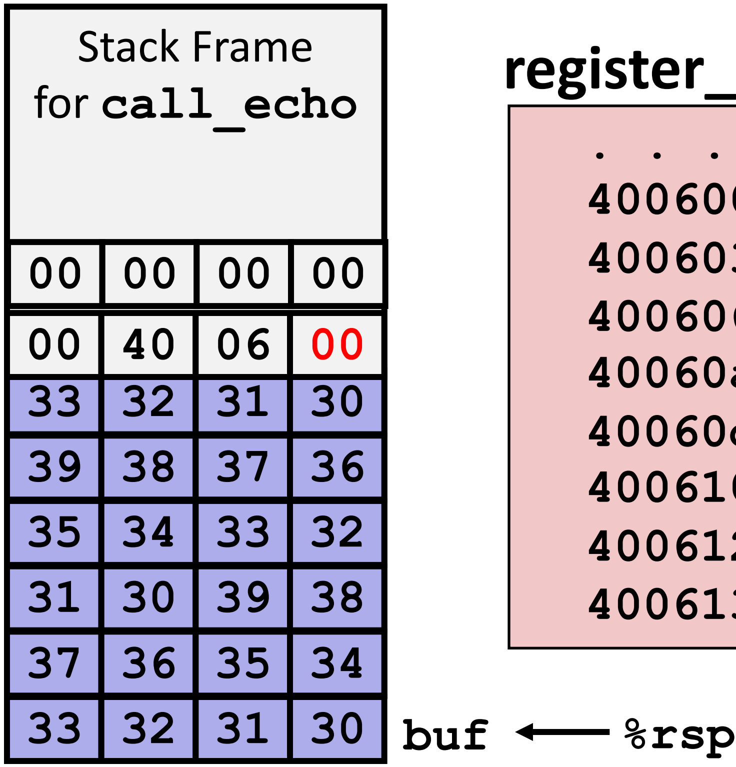
```
 . . .
 4006f1:   callq   4006cf <echo>
 4006f6:   add     $0x8,%rsp
 . . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890123456789012
3
01234567890123456789012
3
```

Overflowed buffer, corrupted return pointer, but program seems to work!

Buffer Overflow Stack Example #3 Explained

After call to gets



register_tm_clones:

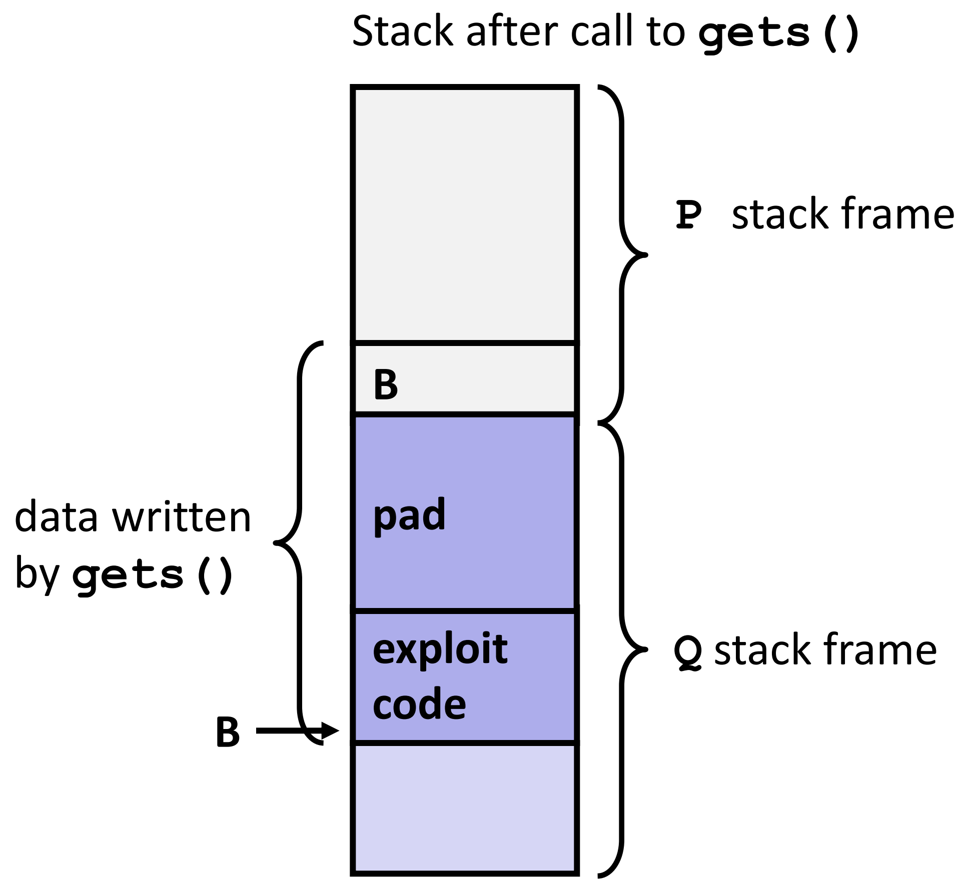
```
. . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

Code Injection Attacks

```
void P(){
  Q();
  ...        ← return address A
}
```

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



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

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real progams**
  - Programmers keep making the same mistakes!
  - Recent measures make these attacks much more difficult
- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more

Example: the original Internet worm (1988)

- **Exploited a few vulnerabilities to spread**
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger user@server`**
  - Worm attacked fingerd server by sending phony argument:
    - **`finger`** ***`"exploit-code padding new-return-address"`***
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- **Once on a machine, scanned for other machines to attack**
  - invaded ~6000 computers in hours (10% of the Internet at the time)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…
  - and CERT was formed, homed at CMU.

Aside: Worms and Viruses

- **Worm: A program that**
  - Can run by itself
  - Can propagate a fully working version of itself to other computers

- **Virus: Code that**
  - Adds itself to other programs
  - Does not run independently

- **Both are (usually) designed to spread among computers and to wreak havoc**

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each…

1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

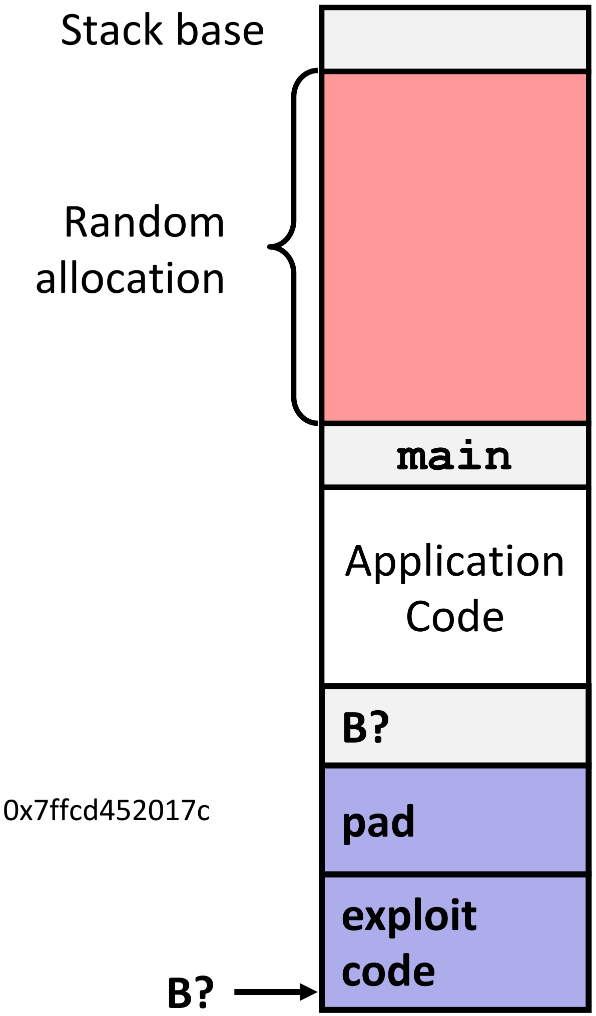
- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

2. System-Level Protections can help

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

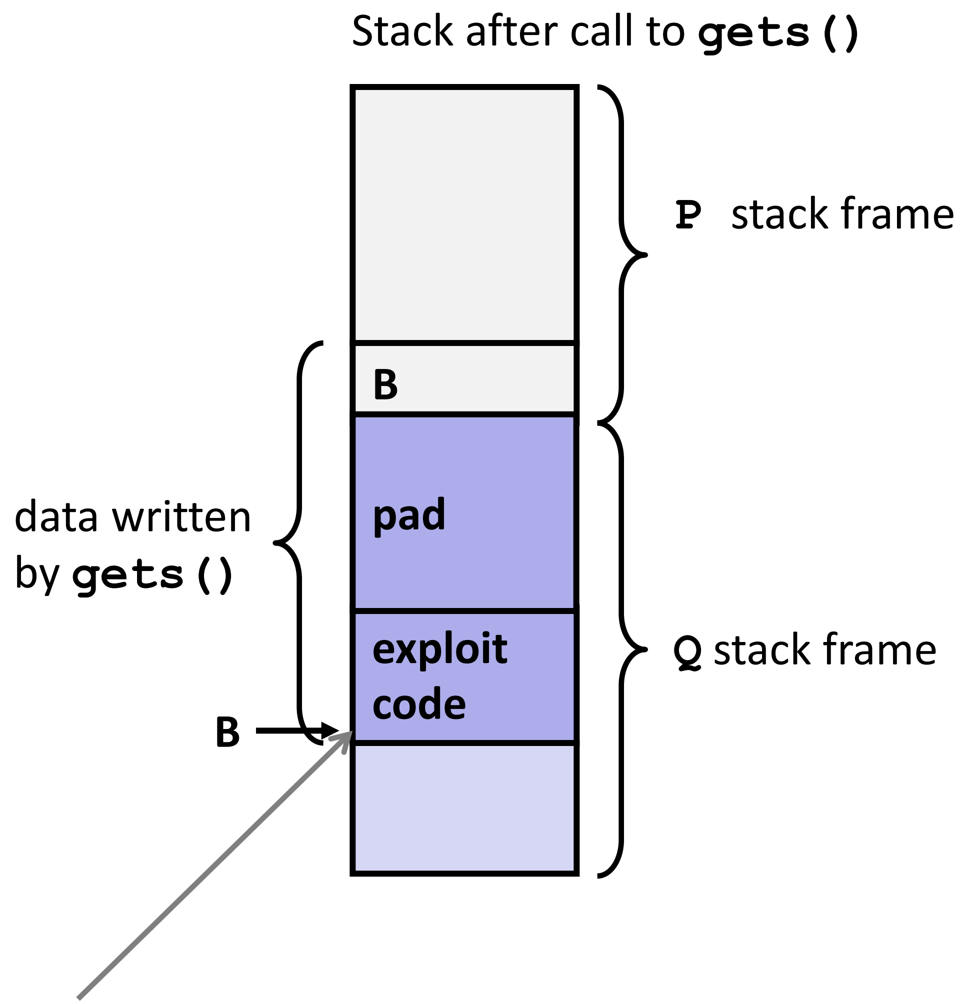
local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

  - Stack repositioned each time program executes



2. System-Level Protections can help

- **Nonexecutable 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



Any attempt to execute this code will fail

3. Stack Canaries can help

- **Idea**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - Now the default (disabled earlier)

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

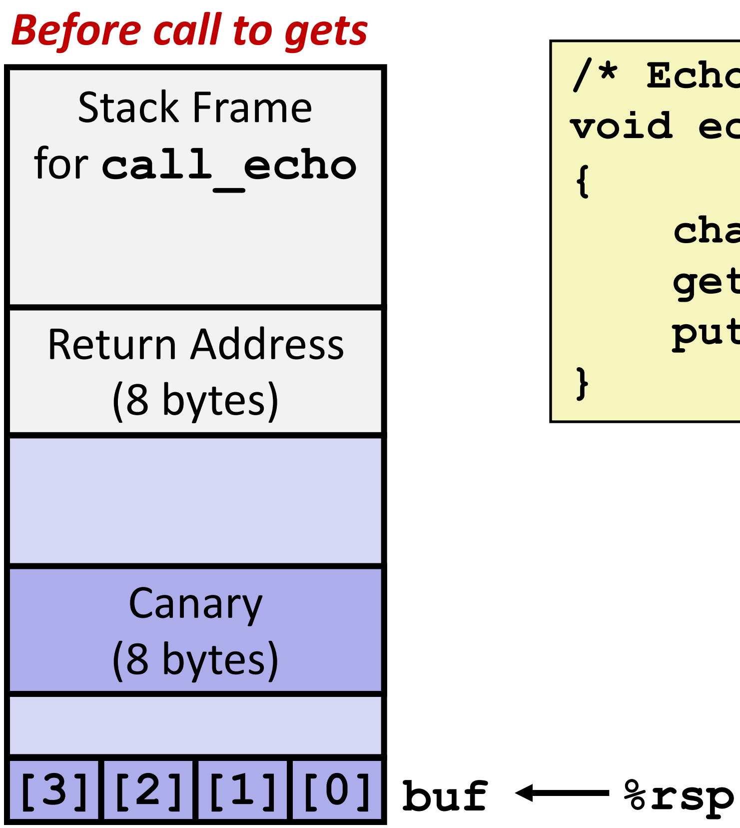
```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Protected Buffer Disassembly

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

Setting Up Canary

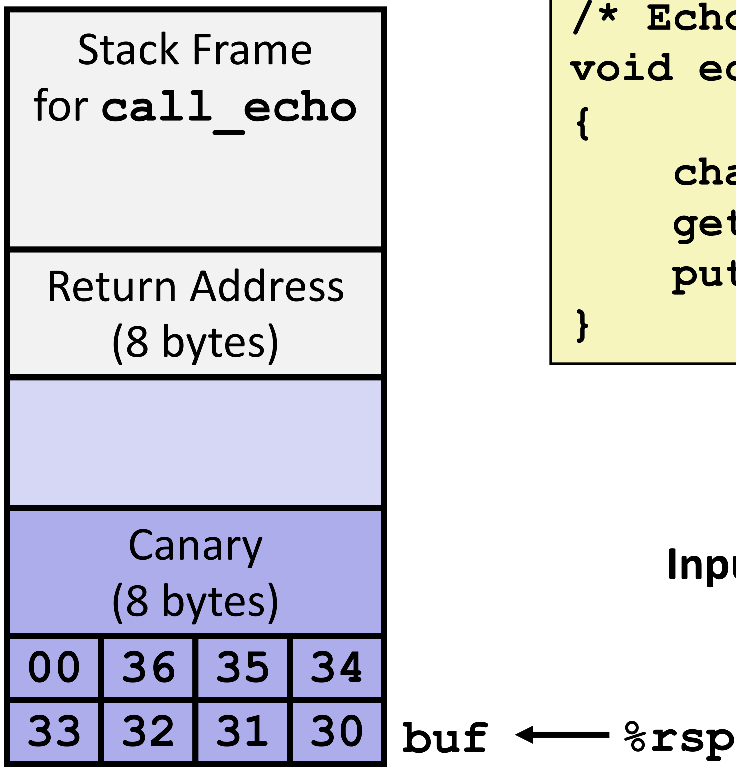



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo: . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Checking Canary

After call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: ***0123456***

```
echo: . . .
    movq    8(%rsp), %rax      # Retrieve from
stack
    xorq    %fs:40, %rax       # Compare to canary
    je      .L6                # If same, OK
    call    __stack_chk_fail   # FAIL
.L6: . . .
```

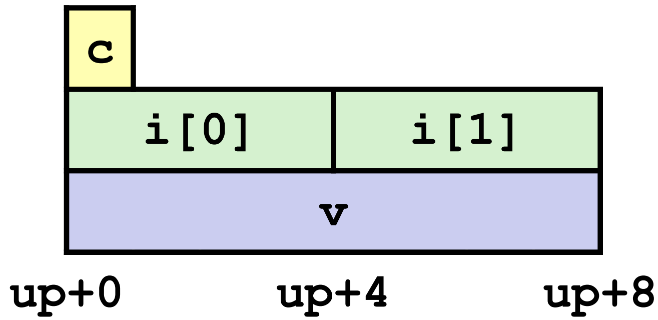
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- **Unions**

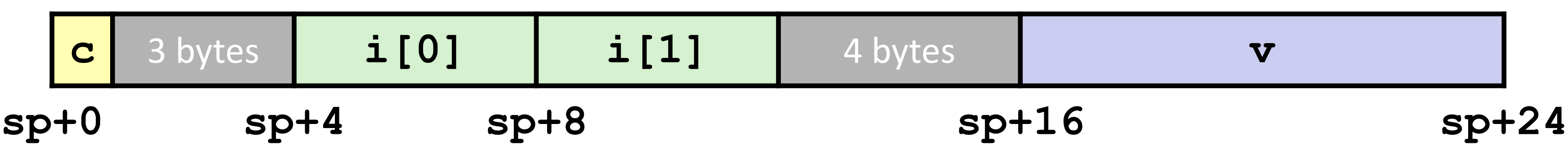
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

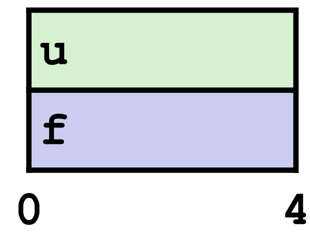



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as `(float) u` ?

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(unsigned) f` ?

Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big Endian**
  - Most significant byte has lowest address
  - Sparc
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- **Bi Endian**
  - Can be configured either way
  - ARM

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

64-bit

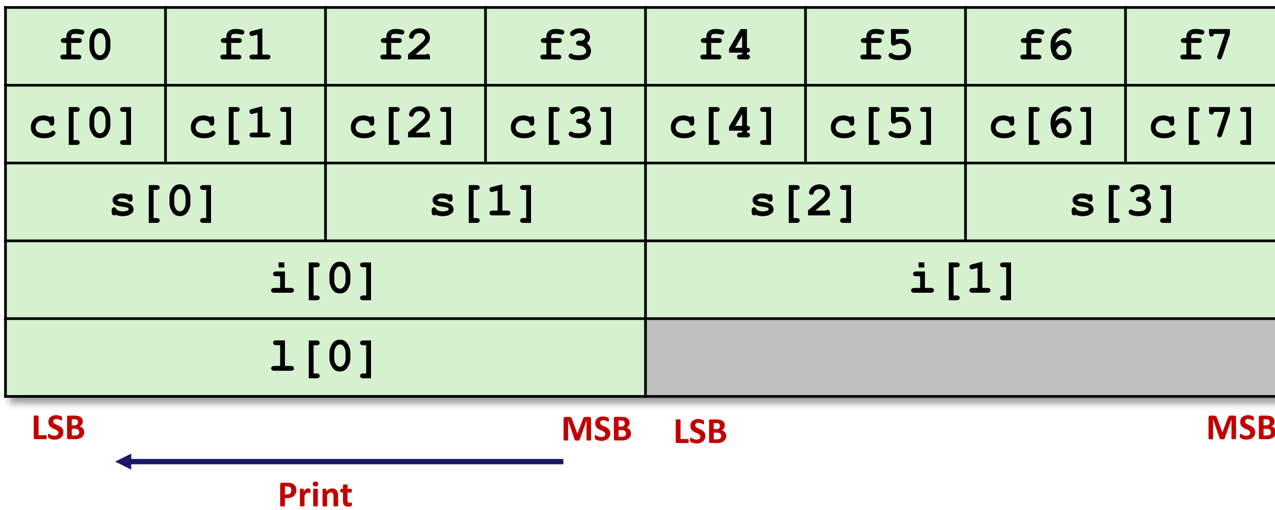
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

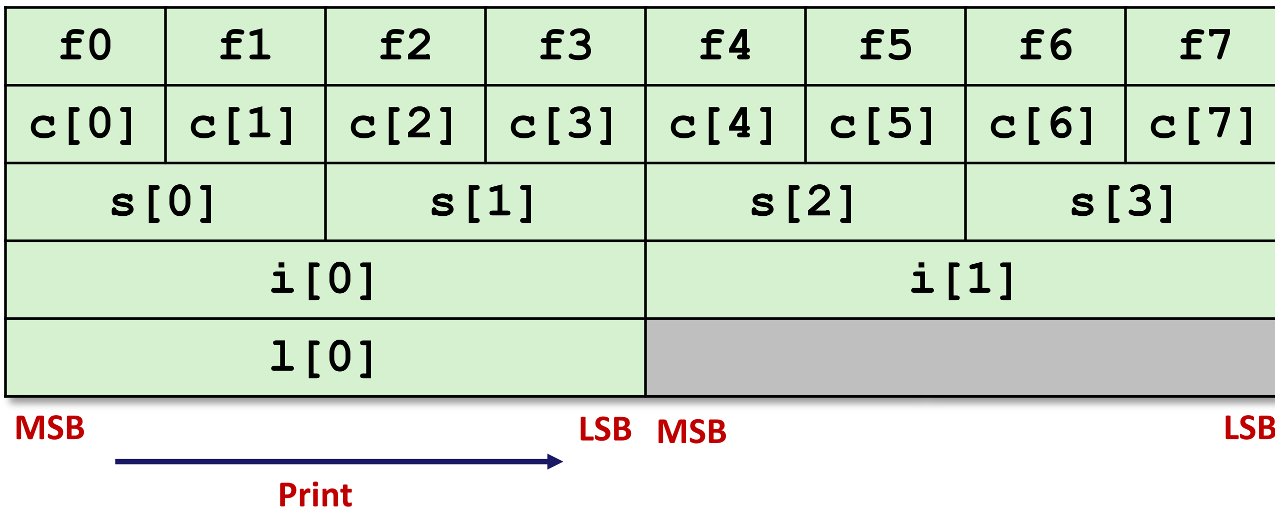


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian



Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

LSB ← Print ← MSB

Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Summary of Compound Types in C

- **Arrays**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Way to circumvent type system

Next time: back to in-person in SB104

Sep 20 **LEC 9:** Machine Prog: Data **Preparation:** Read CS:APP 3.8-3.9	Sep 21	Sep 22 **LEC 10:** Machine Prog: **Preparation:** Read CS:
Sep 27 **LEC 11:** C review **Preparation:** Practice CS:APP and K&R exercises. Come prepared with questions.	Sep 28	Sep 29 **LEC 12:** Linking **Preparation:** Read CS:

- **Tuesday: TA Kirtan will be giving a review of the C language**
- **Monday: deadline for 2nd lab assignment**

Per-lecture feedback

If there are terms you don't understand, please be specific in the feedback.

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- **The form is anonymous.**
  (It checks that you're at Illinois Tech to filter abuse, but I don't see who submitted any of the forms.)
- https://forms.gle/qoeEbBuTYXo5FiU1A
- I'll remind about this at each lecture.