
WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Machine-Level Programming V: Advanced Topics

18-213/18-613: Introduction to Computer Systems
8th Lecture, February 8, 2024

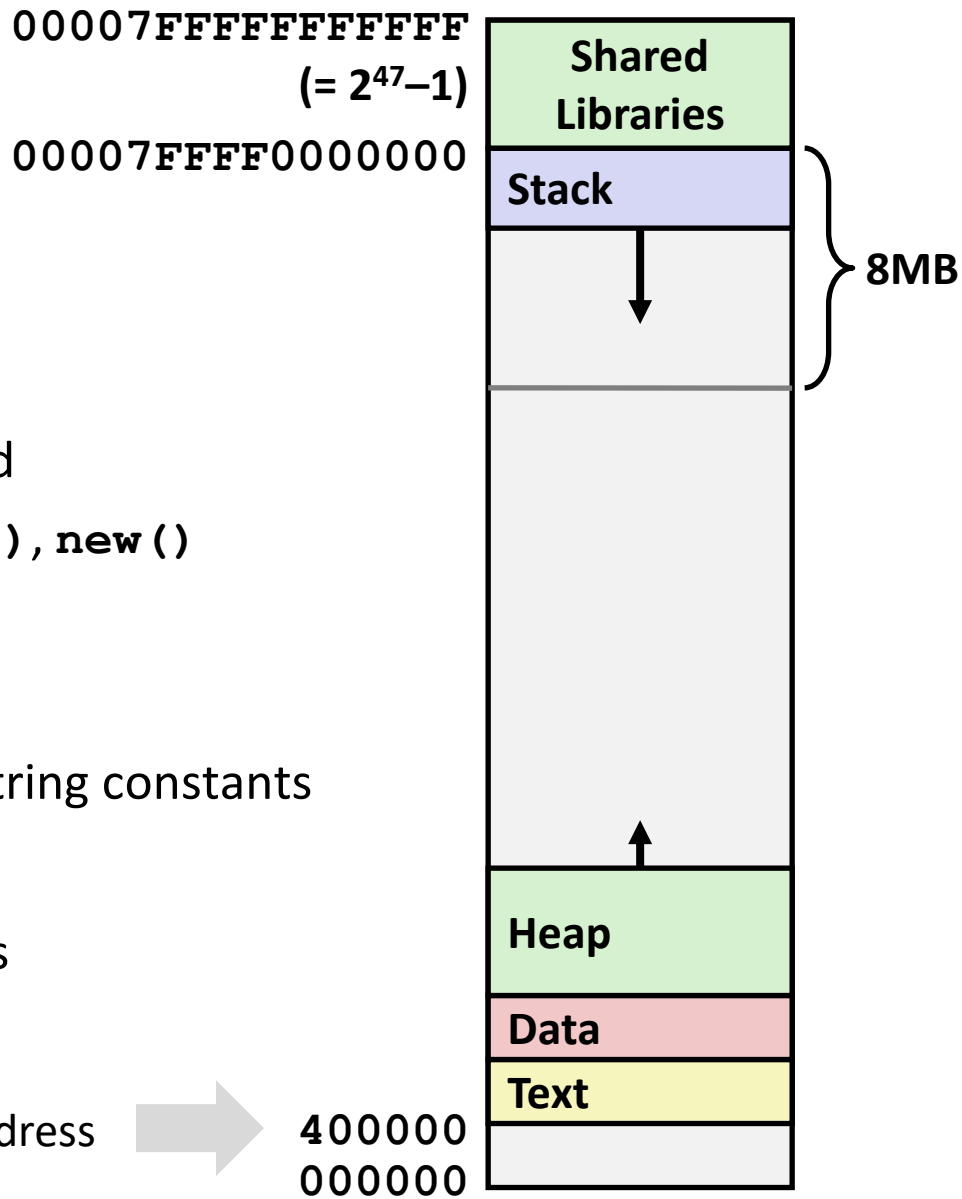
Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability — **CSAPP 3.10.3**
  - Protection — **CSAPP 3.10.4**
- **Unions** — **CSAPP 3.9.2**

x86-64 Linux Memory Layout

not drawn to scale

- **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

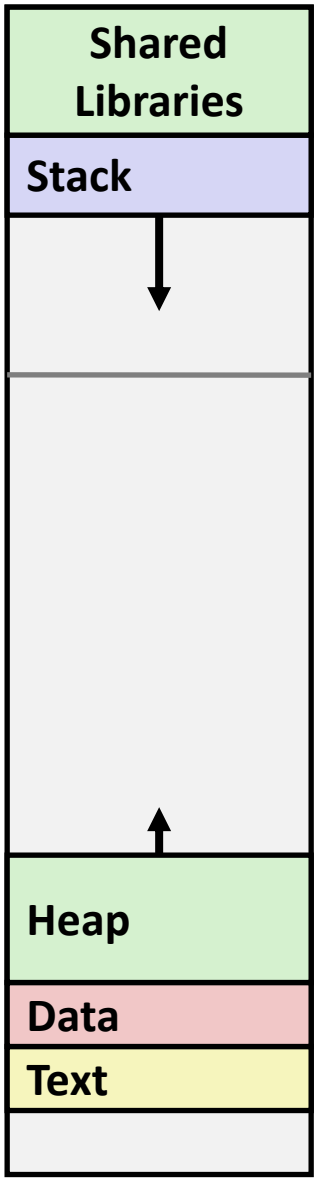


not drawn to scale

Memory Allocation Example

00007FFFFFFFFFFF

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

int global = 0;

int useless() { return 0; }

int main ()
{
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256  B */
    phuge3 = malloc(1L << 32);  /*   4 GB */
    psmall4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



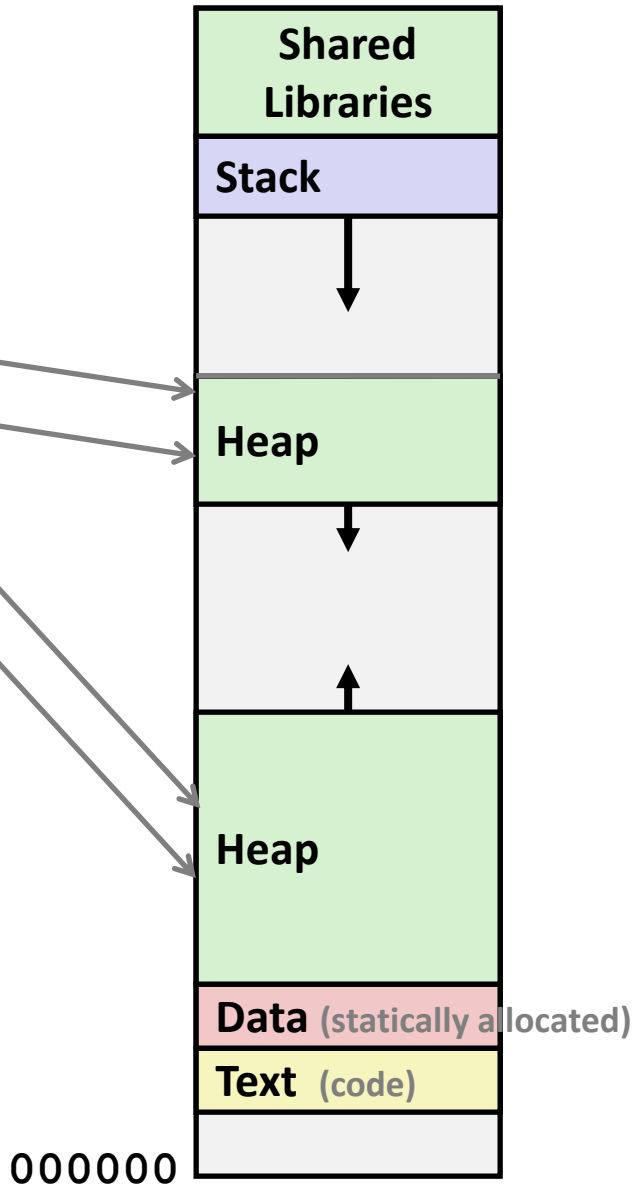
Where does everything go?

x86-64 Example Addresses

address range ~2^{47}

not drawn to scale

`local`	`0x00007ffe4d3be87c`
`phuge1`	`0x00007f7262a1e010`
`phuge3`	`0x00007f7162a1d010`
`psmall4`	`0x000000008359d120`
`psmall2`	`0x000000008359d010`
`big_array`	`0x000000080601060`
`huge_array`	`0x000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x000000000400590`



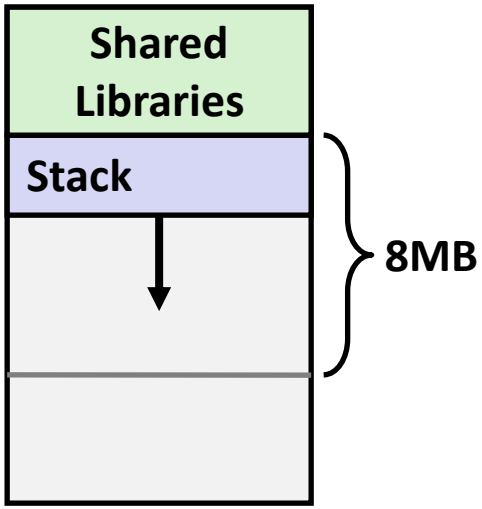
(Exact values can vary)

not drawn to scale

Runaway Stack Example

00007FFFFFFFFFFF

```
int recurse(int x) {
    int a[1<<15];  // 4*2^15 =  128 KiB
    printf("x = %d.  a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```



- Functions store local data on stack in stack frame
- Recursive functions cause deep nesting of frames

```
./runaway 67
x = 67.  a at 0x7ffd18aba930
x = 66.  a at 0x7ffd18a9a920
x = 65.  a at 0x7ffd18a7a910
x = 64.  a at 0x7ffd18a5a900
. . .
x = 4.  a at 0x7ffd182da540
x = 3.  a at 0x7ffd182ba530
x = 2.  a at 0x7ffd1829a520
Segmentation fault (core dumped)
```

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)   ->    3.1400000000
fun(1)   ->    3.1400000000
fun(2)   ->    3.1399998665
fun(3)   ->    2.0000006104
fun(6)   ->    Stack smashing detected
fun(8)   ->    Segmentation fault
```

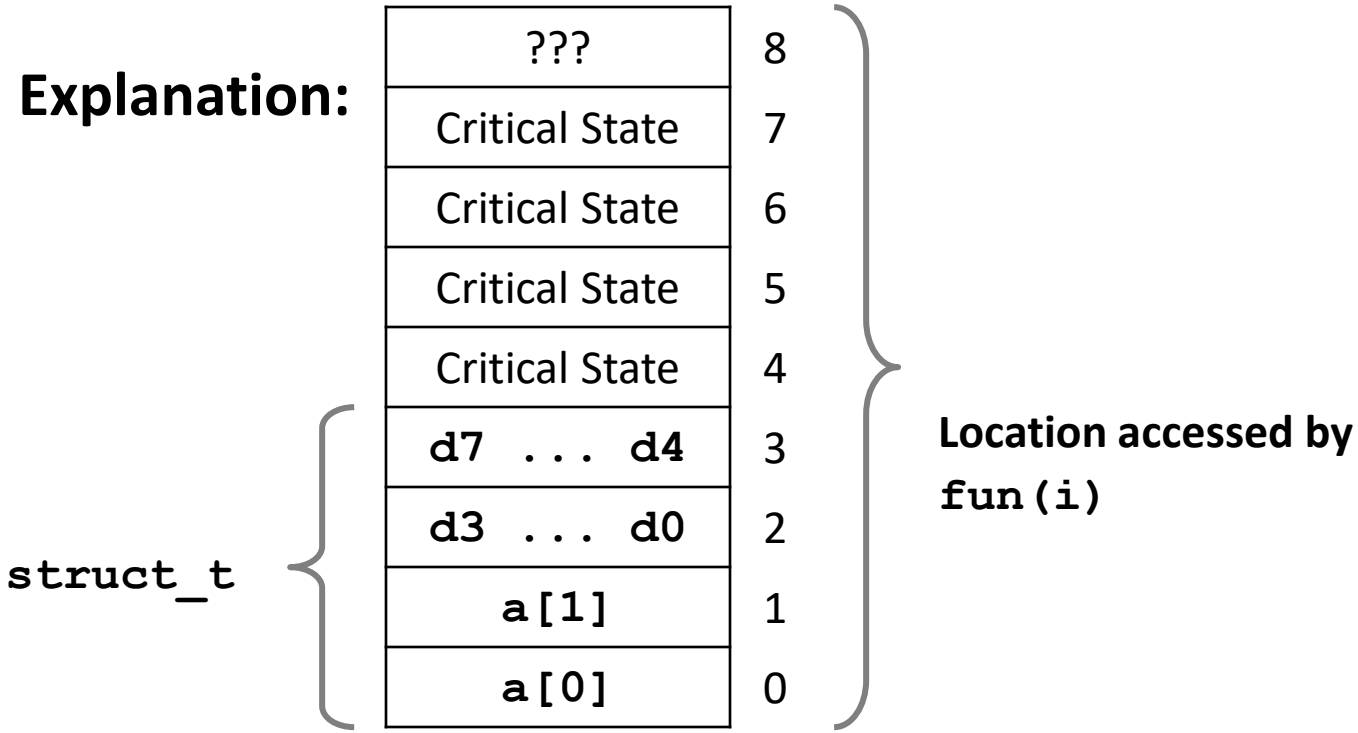
- Result is system specific

Memory Referencing Bug Example

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

```
fun(0)  ->  3.1400000000
fun(1)  ->  3.1400000000
fun(2)  ->  3.1399998665
fun(3)  ->  2.0000006104
fun(4)  ->  Segmentation fault
fun(8)  ->  3.1400000000
```

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
  - What is #1 overall cause?
    - 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

Exploits Based on Buffer Overflows

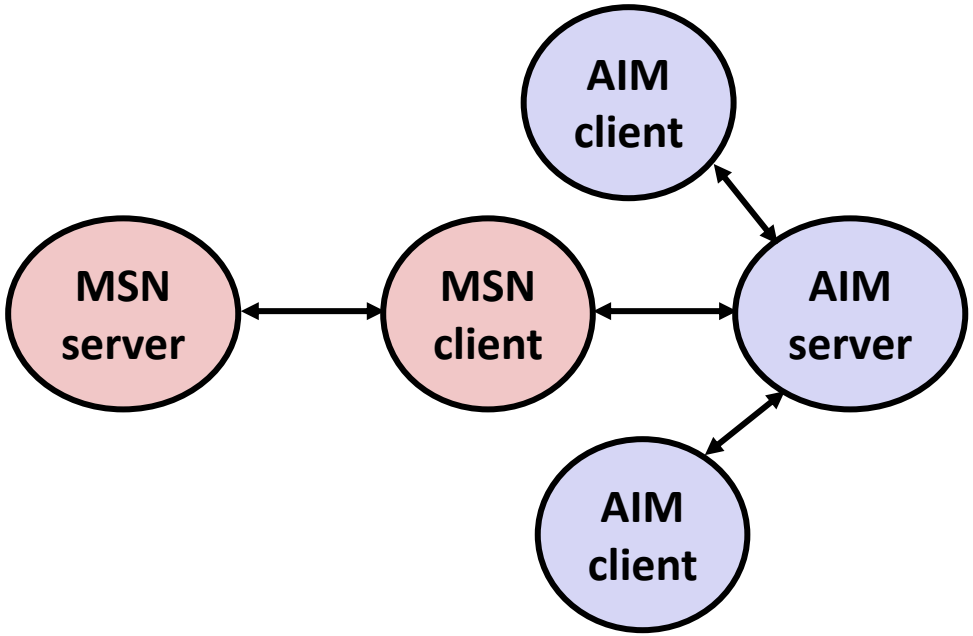
- ***Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real programs**
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- **Examples across the decades**
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more
- **You will learn some of the tricks in attacklab**
  - Hopefully to convince you to never leave such holes in your programs!!

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 droh@cs.cmu.edu`**
  - 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 ☺)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…
  - and CERT was formed… still homed at CMU

Example 2: IM War

- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



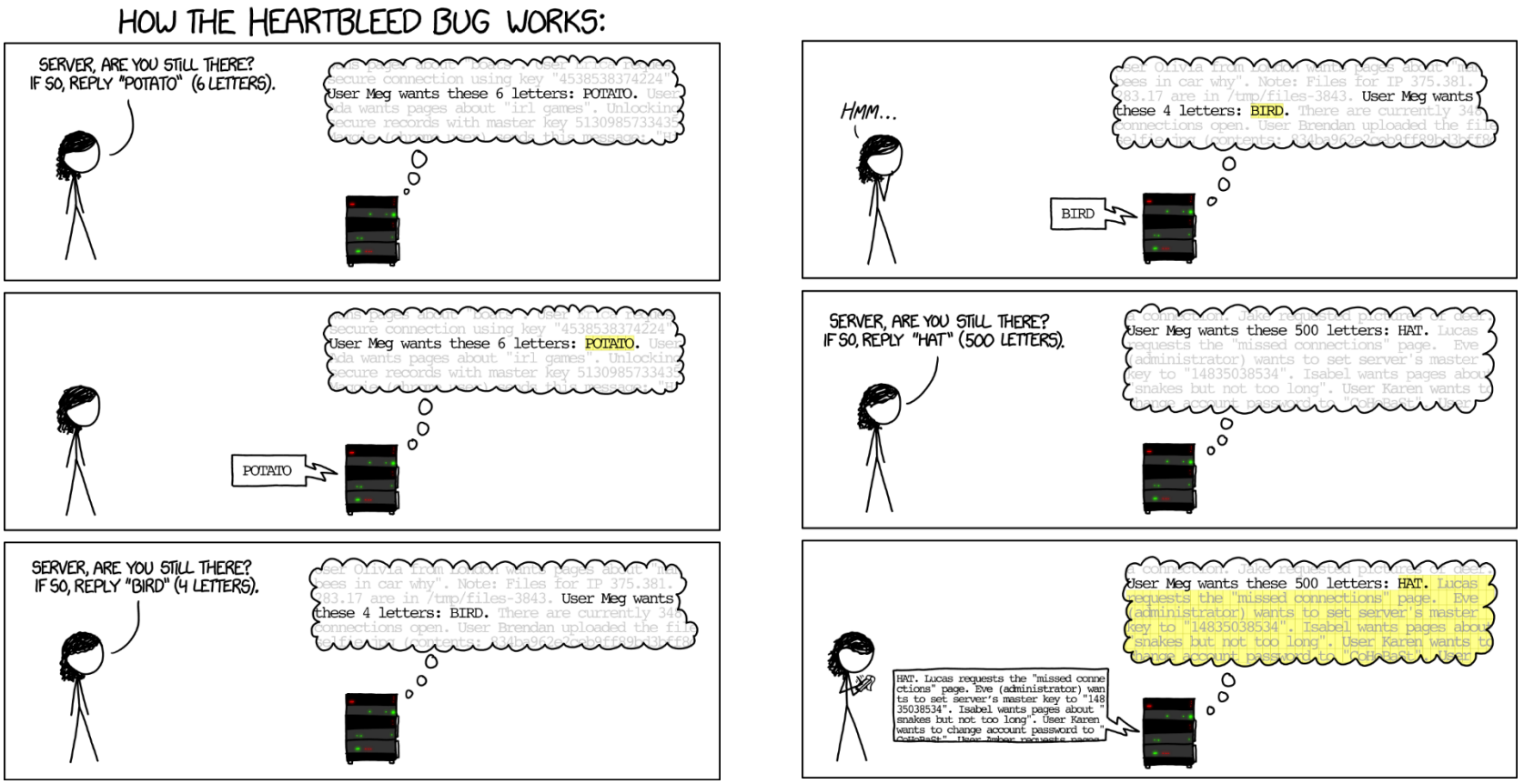
IM War (cont.)

- **August 1999**
  - Mysteriously, Messenger clients can no longer access AIM servers
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes
    - At least 13 such skirmishes
  - What was really happening?
    - AOL had discovered a buffer overflow bug in their own AIM clients
    - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
    - When Microsoft changed code to match signature, AOL changed signature location

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

It was later determined that this email originated from within Microsoft!

Programmers keep making these mistakes…



https://xkcd.com/1354/

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**

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:012345678901234567890123
012345678901234567890123
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
012345678901234567890123
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

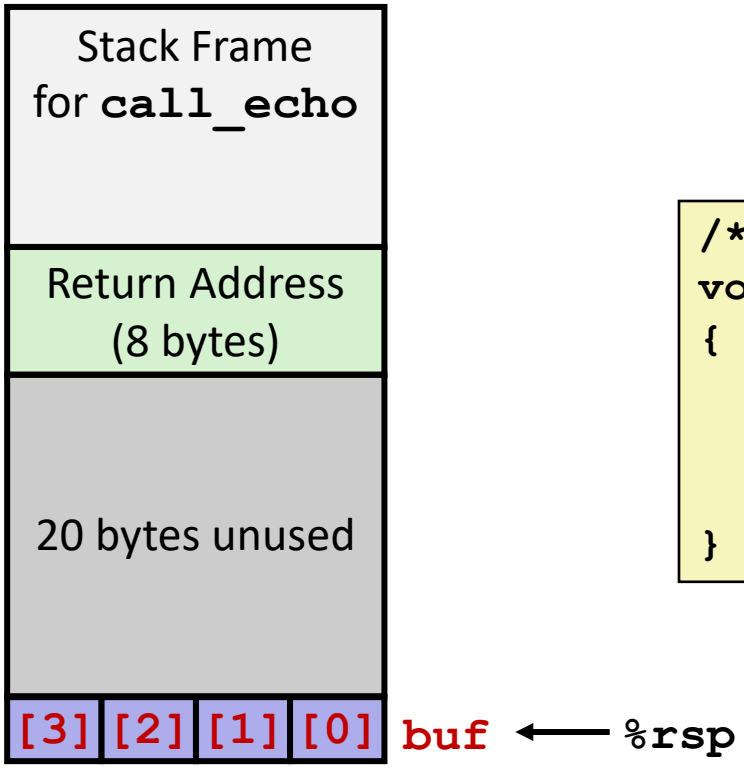
```
000000000040069c <echo>:
  40069c:  48 83 ec 18           sub    $0x18,%rsp
  4006a0:  48 89 e7              mov    %rsp,%rdi
  4006a3:  e8 a5 ff ff ff        callq  40064d <gets>
  4006a8:  48 89 e7              mov    %rsp,%rdi
  4006ab:  e8 50 fe ff ff        callq  400500 <puts@plt>
  4006b0:  48 83 c4 18           add    $0x18,%rsp
  4006b4:  c3                    retq
```

call_echo:

```
  4006b5:  48 83 ec 08           sub    $0x8,%rsp
  4006b9:  b8 00 00 00 00        mov    $0x0,%eax
  4006be:  e8 d9 ff ff ff        callq  40069c <echo>
  4006c3:  48 83 c4 08           add    $0x8,%rsp
  4006c7:  c3                    retq
```

Buffer Overflow Stack Example

Before call to gets

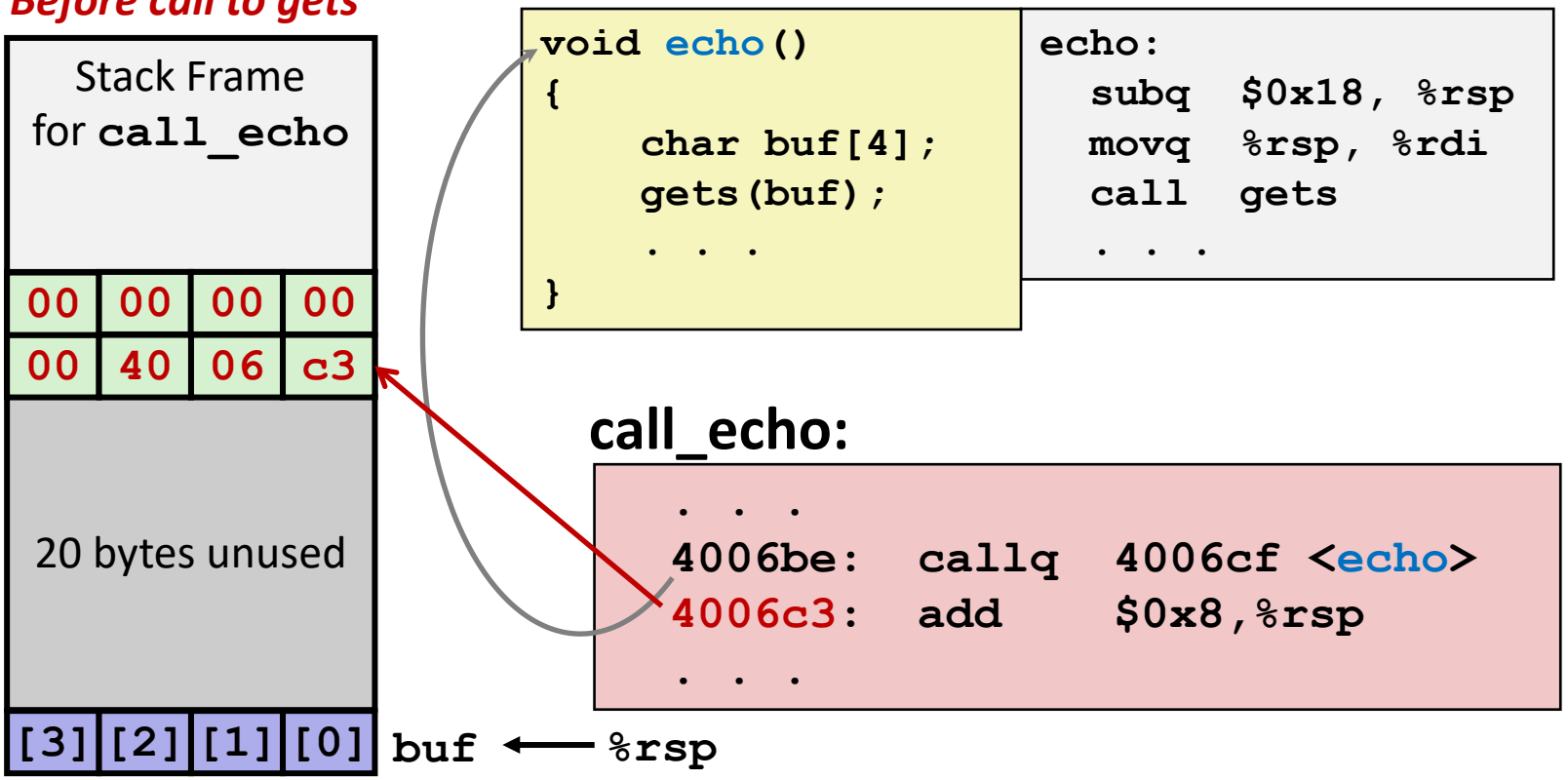



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq $0x18, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example

Before call to gets



Buffer Overflow Stack Example #1

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	c3
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf` ⟵ `%rsp`

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

```
echo:
   subq   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006be:  callq  4006cf <echo>
   4006c3:  add    $0x8,%rsp
   . . .
```

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

"012345678901234567890123\0"

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

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

```
echo:
   subq   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006be:  callq  4006cf <echo>
   4006c3:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901230123456789012345678901230
Segmentation fault
```

Program "returned" to 0x0400600, and then crashed.

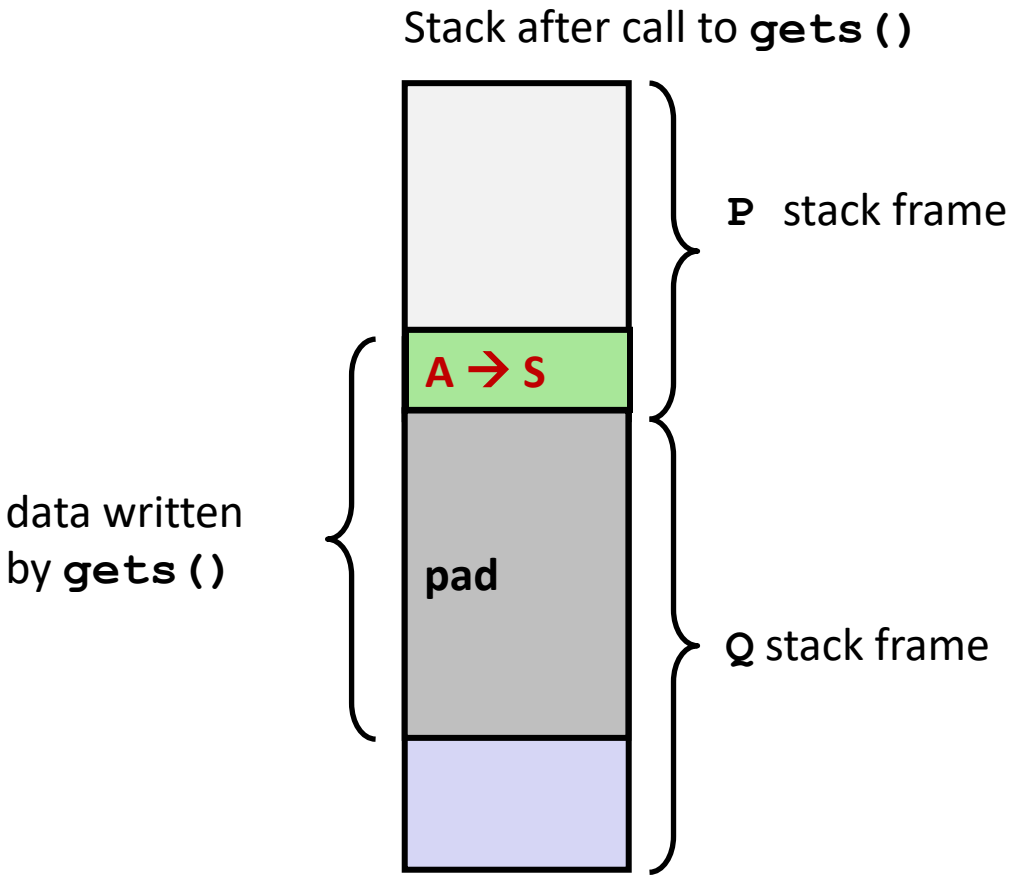
Stack Smashing Attacks

```
void P(){
  Q();
  ...
}
```

return address A

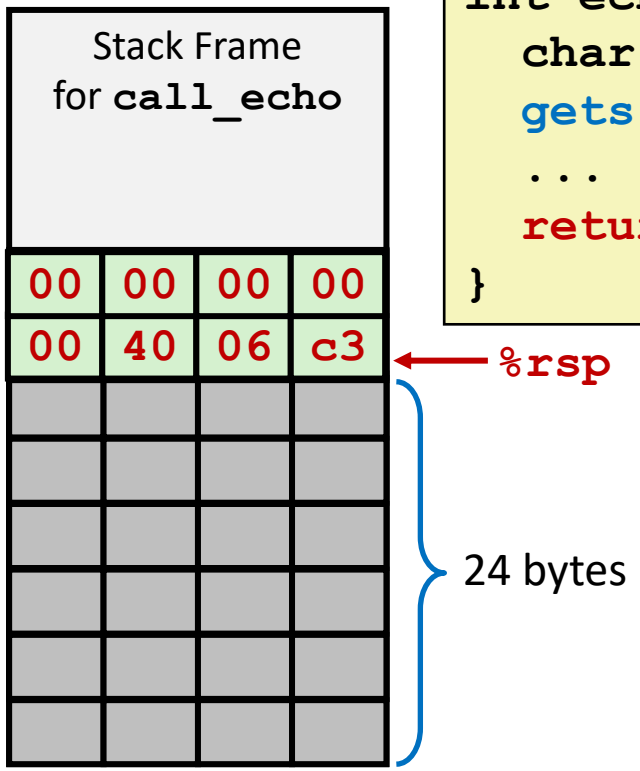
```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

```
void S(){
/* Something
   unexpected */
  ...
}
```



- Overwrite normal return address A with address of some other code S
- When Q executes `ret`, will jump to other code

Crafting Smashing String




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

Target Code

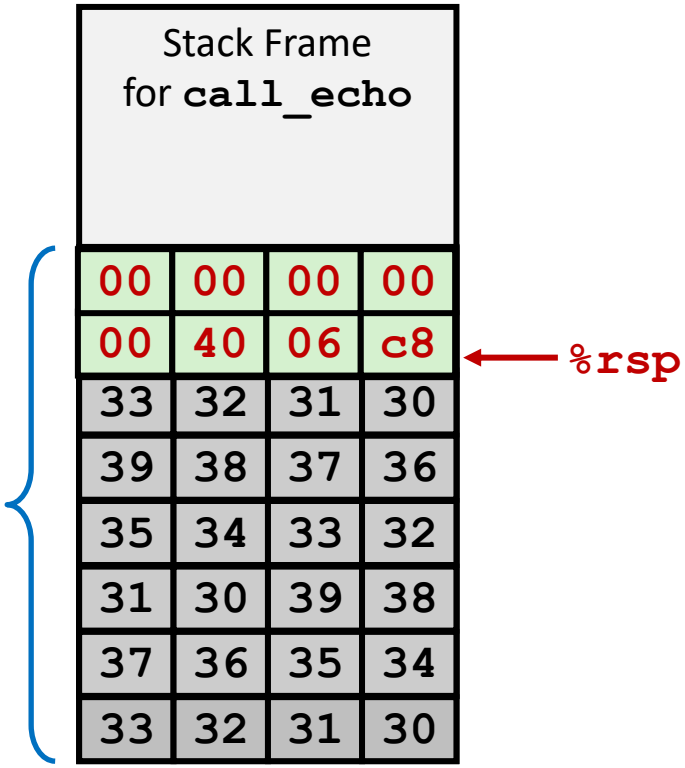
```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
00000000004006c8 <smash>:
  4006c8:       48 83 ec 08
```

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

Smashing String Effect



Target Code

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
00000000004006c8 <smash>:
  4006c8:        48 83 ec 08
```

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

Performing Stack Smash

```
linux> cat smash-hex.txt
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 c8 06 40 00 00 00 00 00
linux> cat smash-hex.txt | ./hexify | ./bufdemo-nsp
Type a string:012345678901234567890123?@
I've been smashed!
```

- **Put hex sequence in file smash-hex.txt**
- **Use hexify program to convert hex digits to characters**
  - Some of them are non-printing
- **Provide as input to vulnerable program**

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

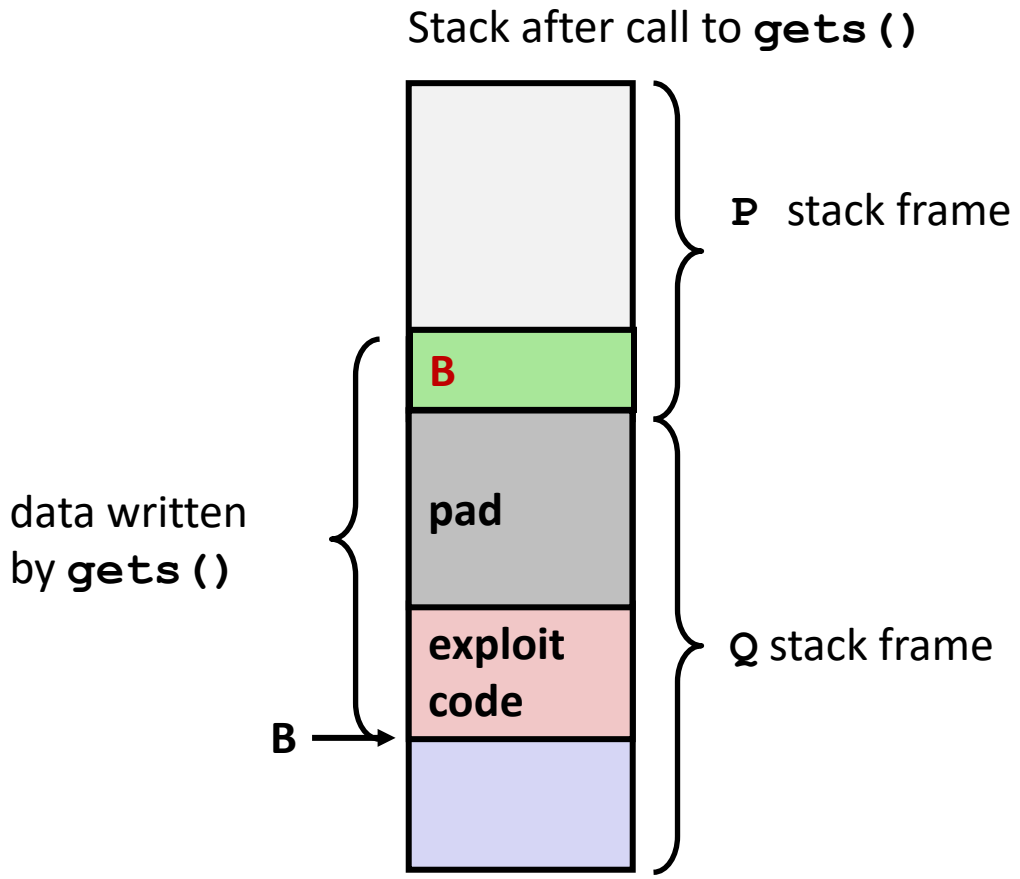
```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

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



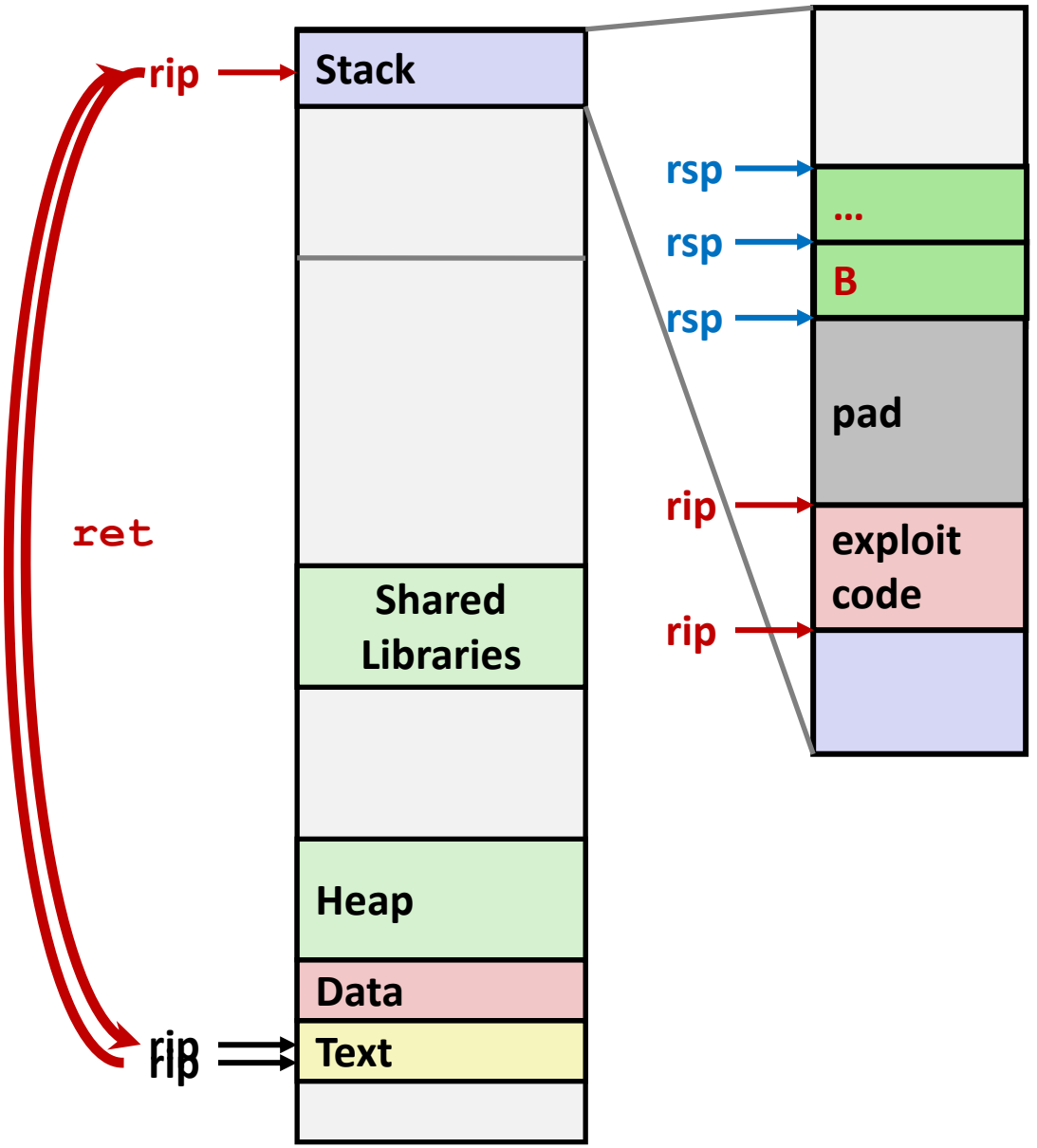
- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

How Does The Attack Code Execute?

```
void P(){
  Q();
  ...
}
```

```
int Q() {
  char buf[64];
  gets(buf); // A->B
  ...
  return ...;
}
```

ret ret



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];
    fgets(buf, 4, stdin);
    puts(buf);
}
```

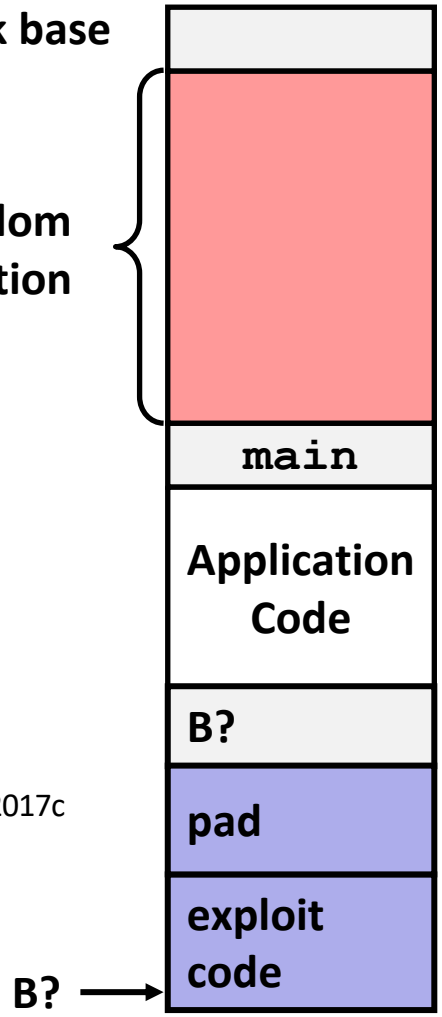
- **For example, use library routines that limit string lengths**
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use **%ns** where **n** is a suitable integer

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

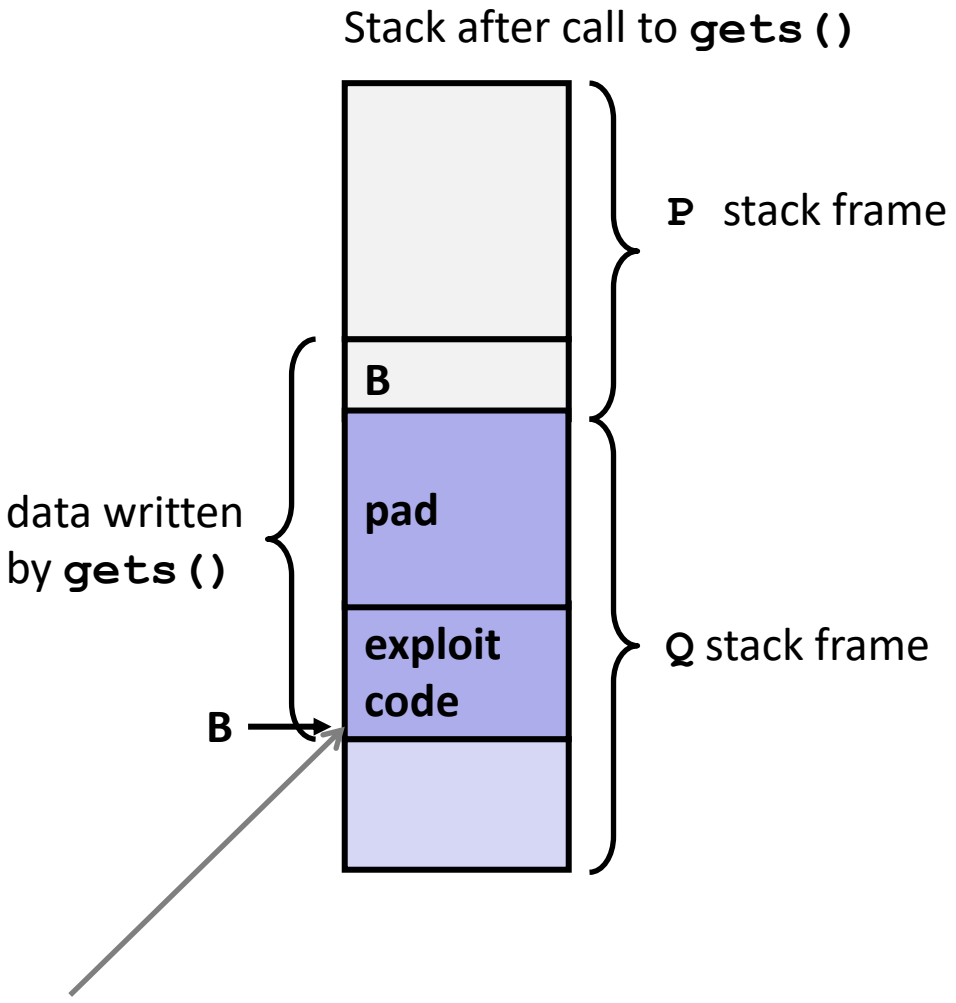
local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

    - 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:012345678
*** stack smashing detected ***
```

Protected Buffer Disassembly

echo:

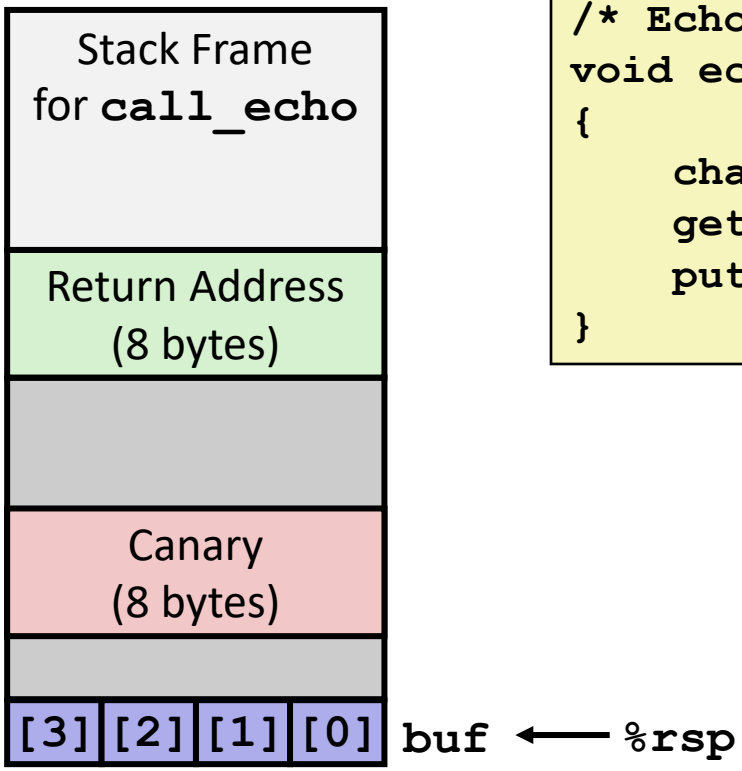
```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761: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

Aside: `%fs:0x28`

- Read from memory using segmented addressing
- Segment is read-only
- Value generated randomly every time program runs

Setting Up Canary

Before call to gets

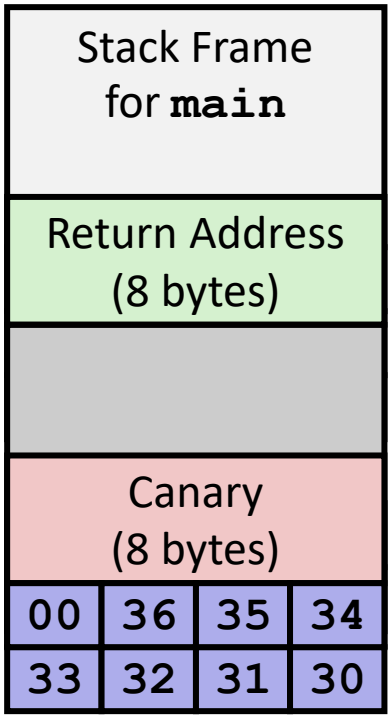



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    mov     %fs:0x28, %rax  # Get canary
    mov     %rax, 0x8(%rsp) # Place on stack
    xor     %eax, %eax      # Erase register
    . . .
```

Checking Canary

After call to gets




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

Input: *0123456*

Some systems:
LSB of canary is 0x00
Allows input 01234567

```
echo:
    . . .
    mov     0x8(%rsp),%rax    # Retrieve from stack
    xor     %fs:0x28,%rax     # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
```

Quiz Time!

Canvas Quiz: Day 8 - Machine Advanced

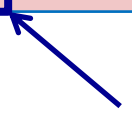
Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*
- **Construct program from *gadgets***
  - Sequence of instructions ending in **`ret`**
    - Encoded by single byte **`0xc3`**
  - Code positions fixed from run to run
  - Code is executable

Gadget Example #1

```
long ab_plus_c
  (long a, long b, long c)
{
   return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```



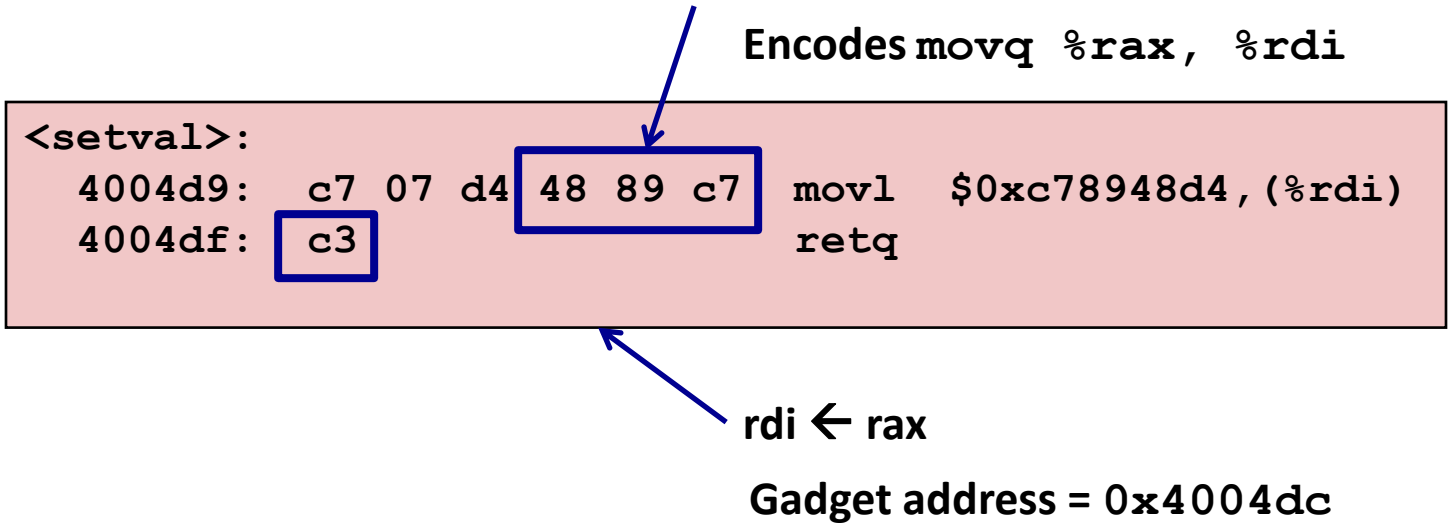
rax ← rdi + rdx

Gadget address = 0x4004d4

- Use tail end of existing functions

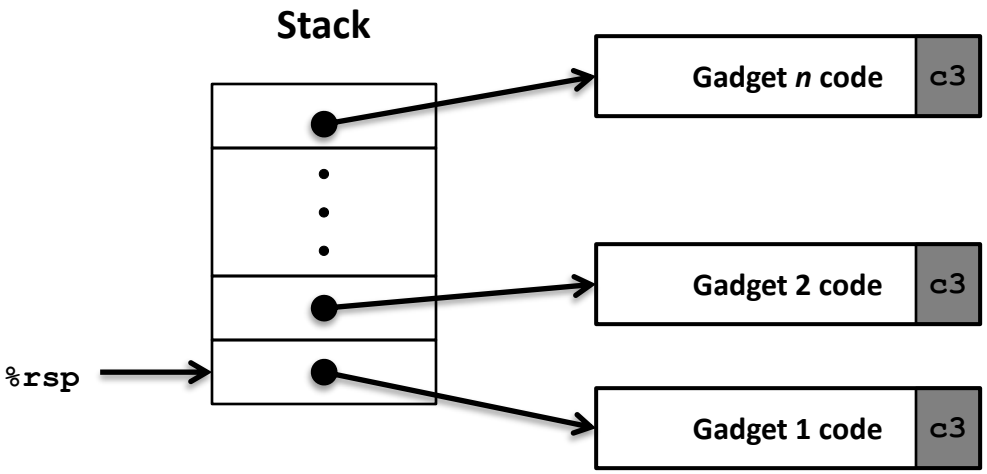
Gadget Example #2

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



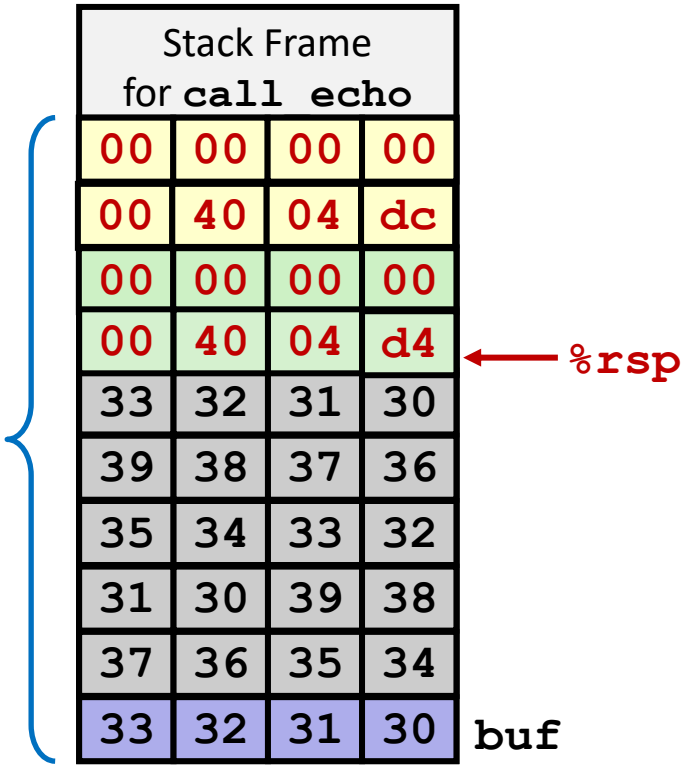
- Repurpose byte codes

ROP Execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**
  - `ret`: pop address from stack and jump to that address

Crafting an ROP Attack String



- **Gadget #1**
  - `0x4004d4` rax ← rdi + rdx
- **Gadget #2**
  - `0x4004dc` rdi ← rax
- **Combination**

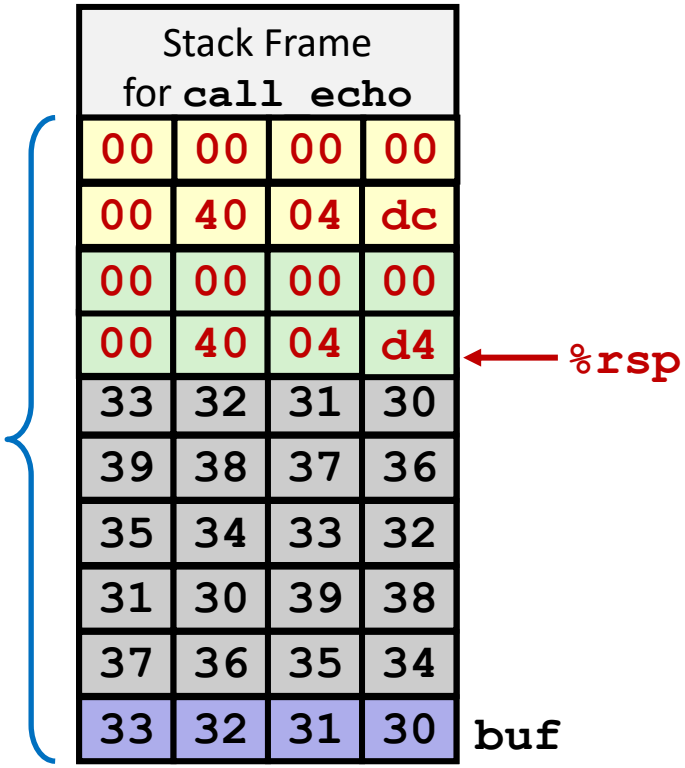
  rdi ← rdi + rdx

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00 dc 04 40 00 00 00 00 00
```

Multiple gadgets will corrupt stack upwards

What Happens when `echo` returns?



1. Echo executes `ret`
   - Starts Gadget #1
2. Gadget #1 executes `ret`
   - Starts Gadget #2
3. Gadget #2 executes `ret`
   - Goes off somewhere …

Multiple gadgets will corrupt stack upwards

ROP Defense: IBT

“Indirect Branch Tracking (IBT) is a control flow integrity technology for x86-64 processors that creates a special "branch target" instruction that has no function other than to mark a location as a valid indirect branch target, with the processor capable of being put into a mode where it will raise an exception if an indirect branch is made to a location without a branch target instruction.

IBT is designed to protect against computer security exploits that use indirect branch instructions to jump into code in unintended ways, such as return-oriented programming.”

-- https://en.wikipedia.org/wiki/Indirect_Branch_Tracking

ROP Defense: IBT, cont.

- "If IBT is enabled, the CPU will ensure that every indirect branch lands on a special instruction (endbr32 or endbr64), which executes as a no-op; if anything else is found, the processor will raise a control-protection (#CP) exception." (https://lwn.net/Articles/889475/)

- "[x86-64's] IBT cannot ensure that the target of an indirect branch matches the caller's expectations, but it can ensure that the target was meant to be reached in this way [i.e. via an indirect jump, but not necessarily the intended indiret jump]." (https://lwn.net/Articles/889475/)

- Complete compliance w.r.t. marking indirect jump targets is require while IBT is enable, which can be tricky, for example, w.r.t. library code, firmware code, and future code.
  - IBT is most likely turned off before any access to firmware, for example.

ROP Defense: IBT, cont.

- "The ENDBRANCH (see Section 73 for details) is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. This instruction opcode is selected to be one that is a NOP on legacy machines such that programs compiled with ENDBRANCH new instruction continue to function on old machines without the CET enforcement. On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the processor pipeline to detect control flow violations. The CPU implements a state machine that tracks indirect jmp and call instructions. When one of these instructions is seen, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state. In WAIT_FOR_ENDBRANCH state the next instruction in the program stream must be an ENDBRANCH. If an ENDBRANCH is not seen the processor causes a control protection exception (#CP), else the state machine moves back to IDLE state."

  - Control-flow Enforcement Technology Specification , Section 1.2, Page 11, Document Number: 334525-003, Revision 3.0, Intel, May 2019.

ROP Defense: IBT, cont.

- "The ENDBRANCH (see Section 73 for details) is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. This instruction opcode is selected to be one that is a NOP on legacy machines such that programs compiled with ENDBRANCH new instruction continue to function on old machines without the CET enforcement. On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the processor pipeline to detect control flow violations. The CPU implements a state machine that tracks indirect jmp and call instructions. When one of these instructions is seen, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state. In WAIT_FOR_ENDBRANCH state the next instruction in the program stream must be an ENDBRANCH. If an ENDBRANCH is not seen the processor causes a control protection exception (#CP), else the state machine moves back to IDLE state."

  - Control-flow Enforcement Technology Specification , Section 3.1, Page 19, Document Number: 334525-003, Revision 3.0, Intel, May 2019.

ROP Defense: IBT, cont.

```
foo:
.LFB6:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        cmpl    $6, -8(%rbp)
        ja      .L2
        movl    -8(%rbp), %eax
        leaq    0(,%rax,4), %rdx
        leaq    .L4(%rip), %rax
        movl    (%rdx,%rax), %eax
        cltq
        leaq    .L4(%rip), %rdx
        addq    %rdx, %rax
      notrack jmp      *%rax
        .section        .rodata
        .align 4
      .align 4
```

- Note the handling of the entry to function calls and the jump for a switch statement above

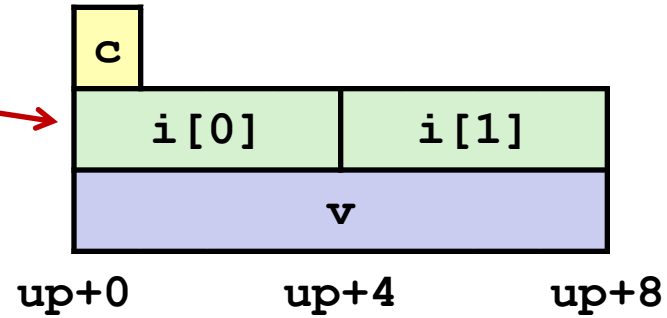
Today

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

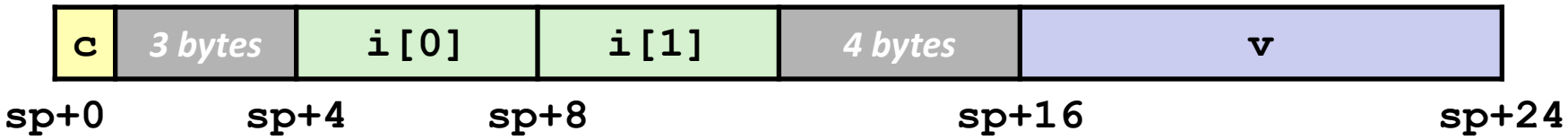
Union Allocation

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

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

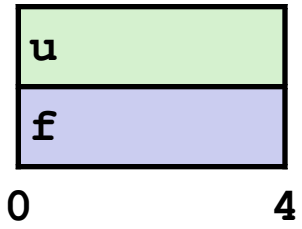



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



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, *Internet*
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- **Bi Endian**
  - Can be configured either way
  - ARM

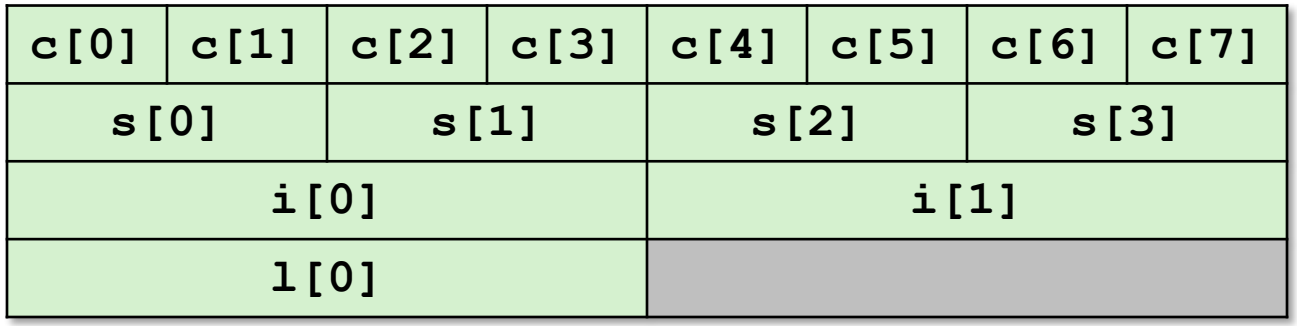
Byte Ordering Example

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

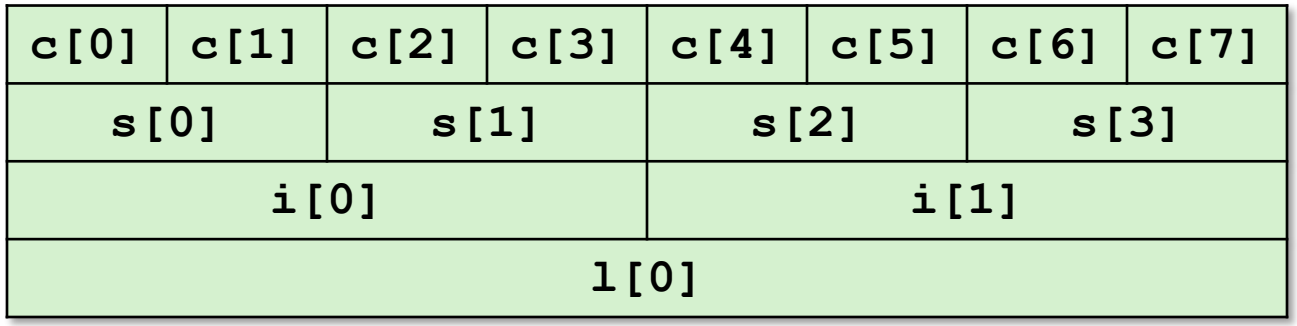
How are the bytes inside short/int/long stored?

Memory addresses growing →

32-bit



64-bit

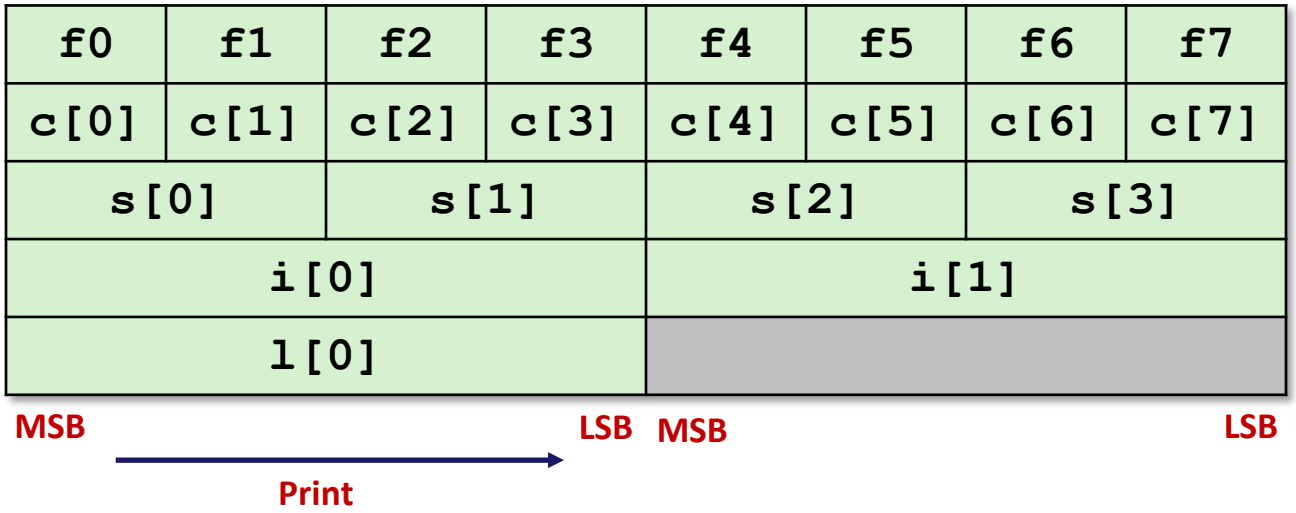


Byte Ordering Example (Cont).

```
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 Sun

Big Endian

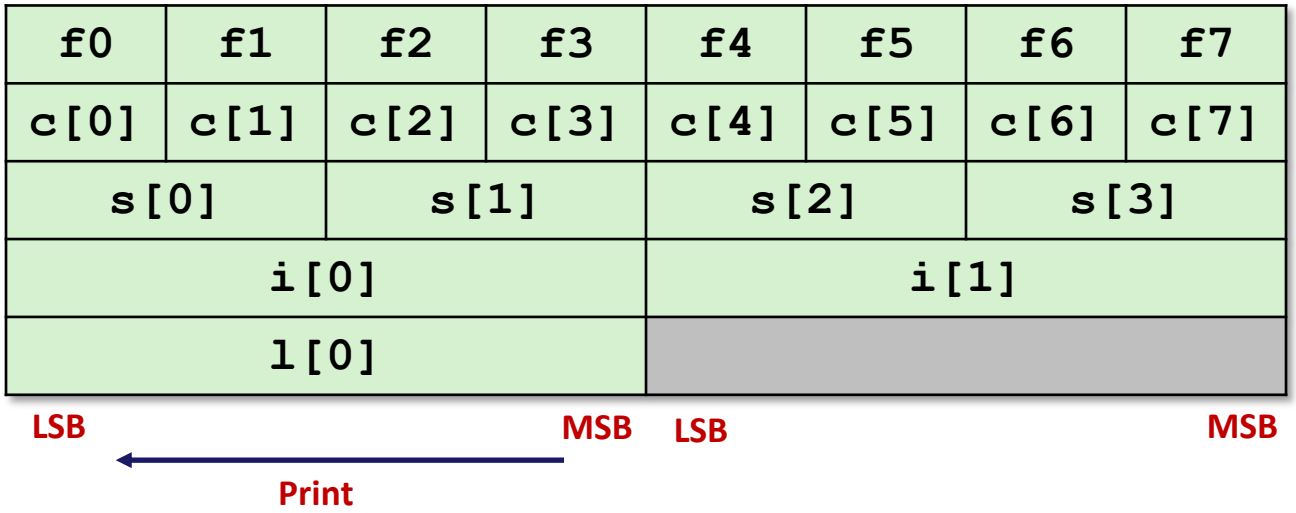


Output on Sun:

```
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 IA32

Little Endian

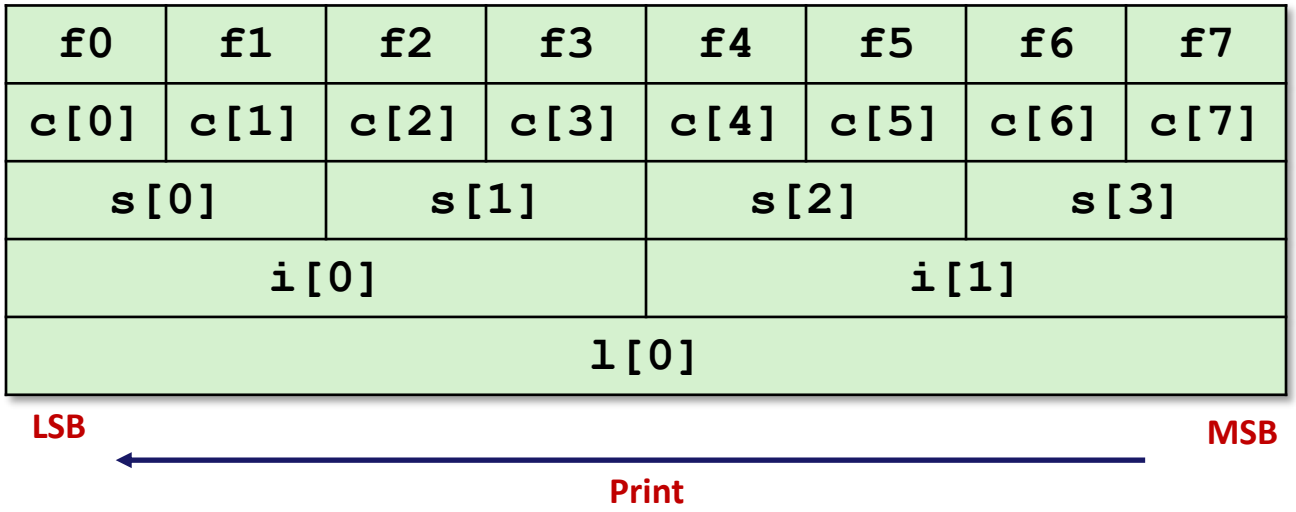


Output:

```
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 x86-64

Little Endian



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

Summary

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Code Injection Attack
  - Return Oriented Programming
- **Unions**