Machine-Level Programming V: Advanced Topic (cont'd)

B&O Readings: 3.9-3.10

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

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 (how big is the array dest??)

- **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:01234567890
01234567890
```

```
unix>./bufdemo-nsp
Type a string:012345678901
012345678901
Segmentation fault (core dumped)
```

Why does the code seg fault with input of >= 12 characters?

Buffer Overflow Disassembly

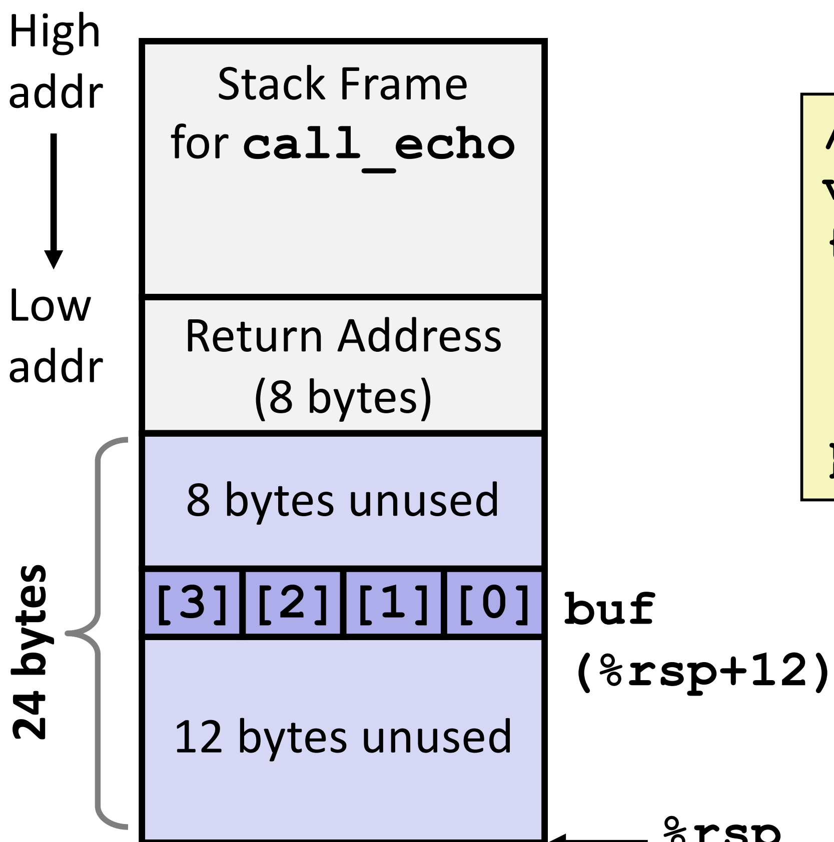
echo:

```
0000000000400694 <echo>:
  400694:       48 83 ec 18             sub    $0x18,%rsp
  400698:       48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  40069d:       e8 a4 ff ff ff          callq  400646 <gets>
  4006a2:       48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  4006a7:       e8 44 fe ff ff          callq  4004f0 <puts@plt>
  4006ac:       48 83 c4 18             add    $0x18,%rsp
  4006b0:       c3                      retq
```

call_echo:

```
4006b1:       48 83 ec 08             sub    $0x8,%rsp
4006b5:       b8 00 00 00 00          mov    $0x0,%eax
4006ba:       e8 d5 ff ff ff          callq  400694 <echo>
4006bf:       48 83 c4 08             add    $0x8,%rsp
4006c3:       c3                      retq
```

Buffer Overflow Stack



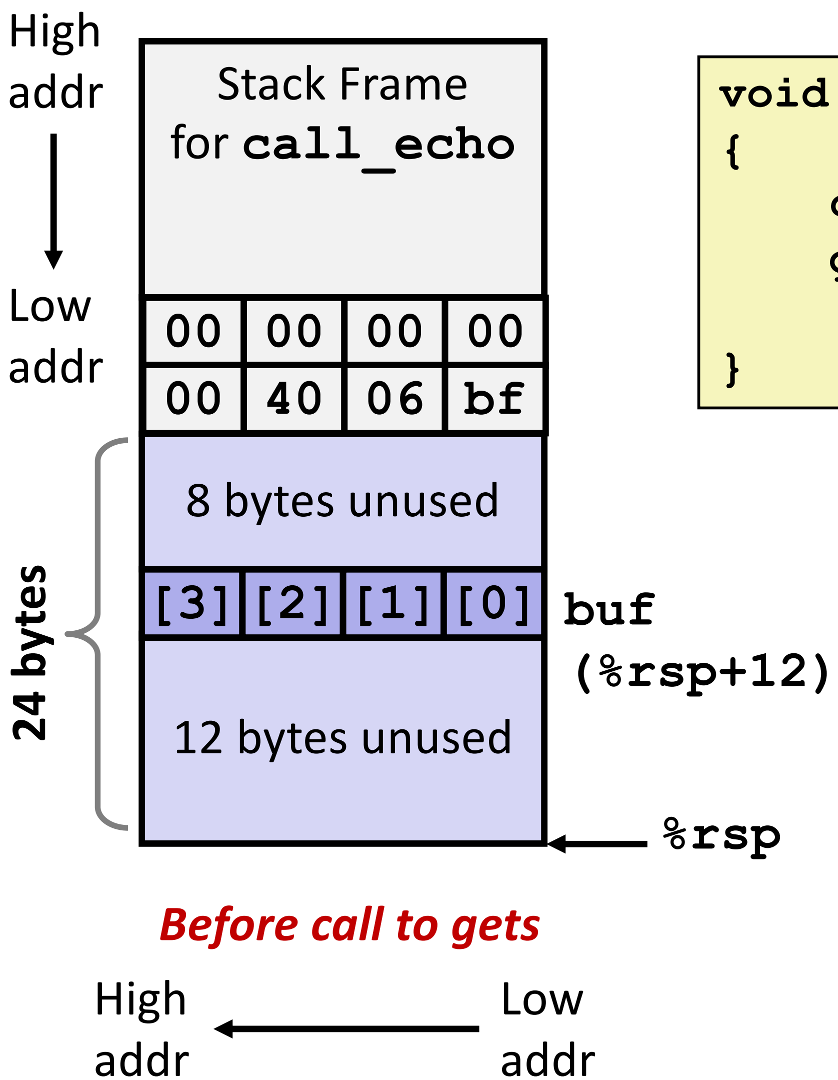

```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Before call to gets

High addr ← Low addr

```
echo:
  subq   $24, %rsp
  movq   12(%rsp), %rdi
  call   gets
  . . .
```

Buffer Overflow Stack



Before call to gets

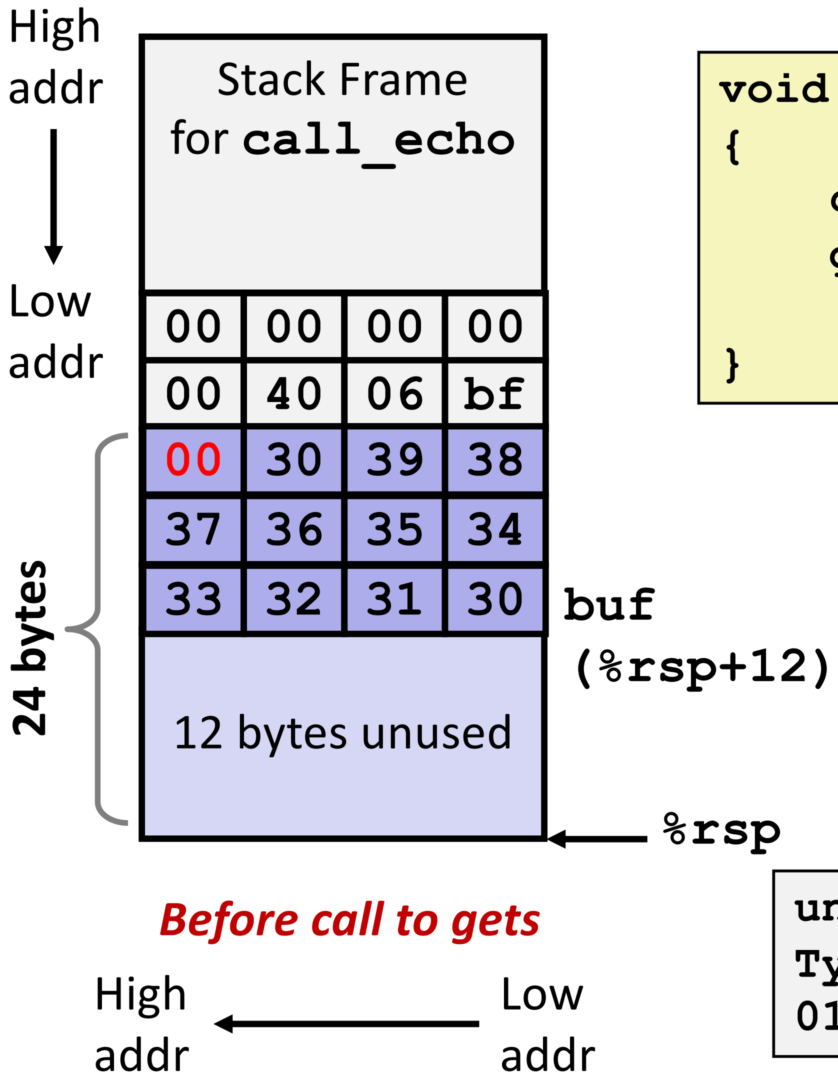
```c
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   12(%rsp), %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006ba:    callq   4006bf <echo>
4006bf:    add     $0x8,%rsp
. . .
```

Buffer Overflow Stack




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

```
echo:
   subq   $24, %rsp
   movq   12(%rsp), %rdi
   call   gets
   . . .
```

call_echo:

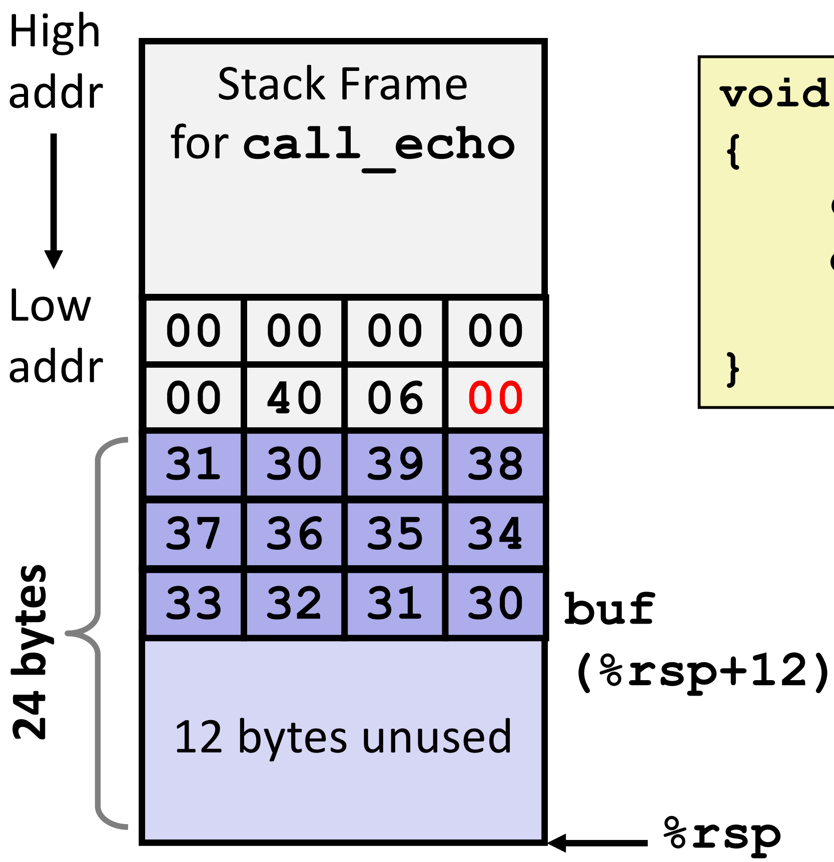
```
. . .
4006ba:     callq   4006bf <echo>
4006bf:     add     $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:01234567890
01234567890
```

12 chars total: C string is null-terminated ('\0')!

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack



Before call to gets

High addr ← Low addr

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

```
echo:
  subq   $24, %rsp
  movq   12(%rsp), %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006ba:    callq   4006bf <echo>
4006bf:    add     $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901
012345678901
Illegal instruction (core dumped)
```

Overflowed buffer and corrupted return address.

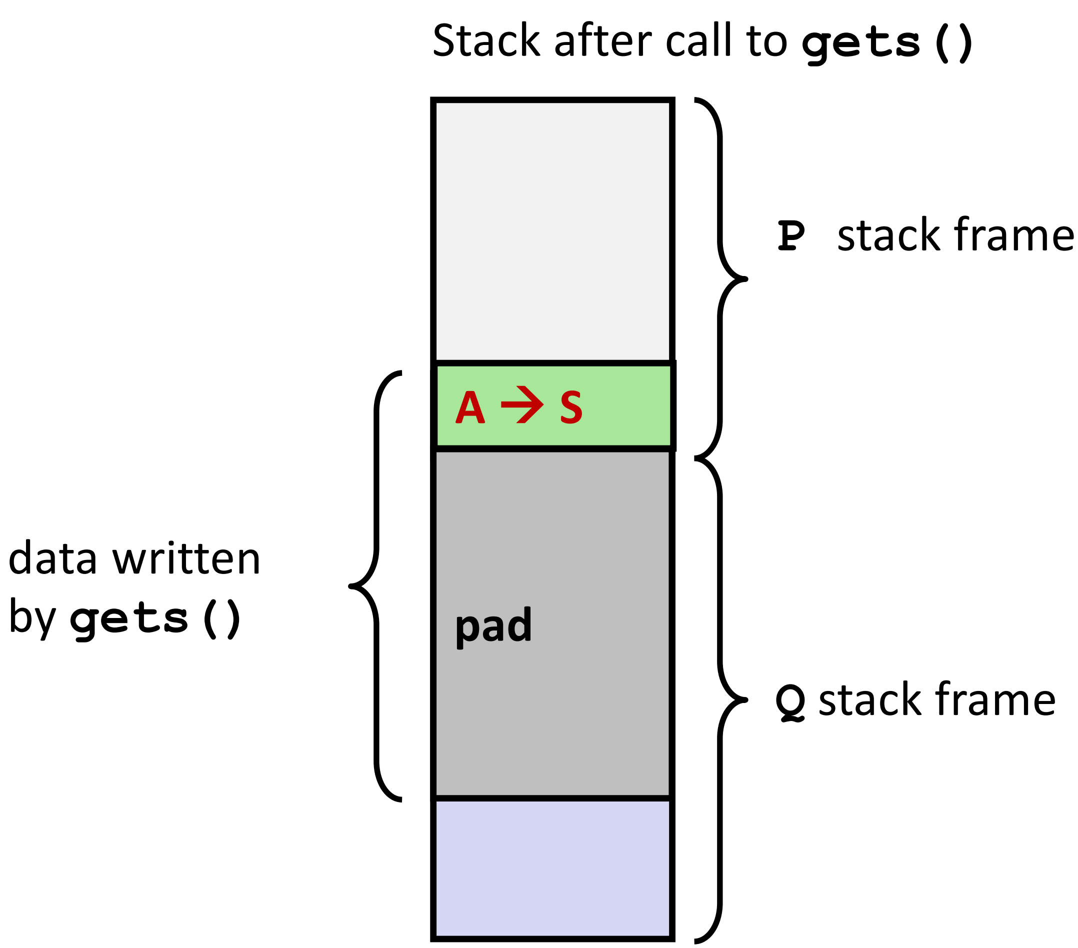
Stack Smashing Attacks

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

return address A

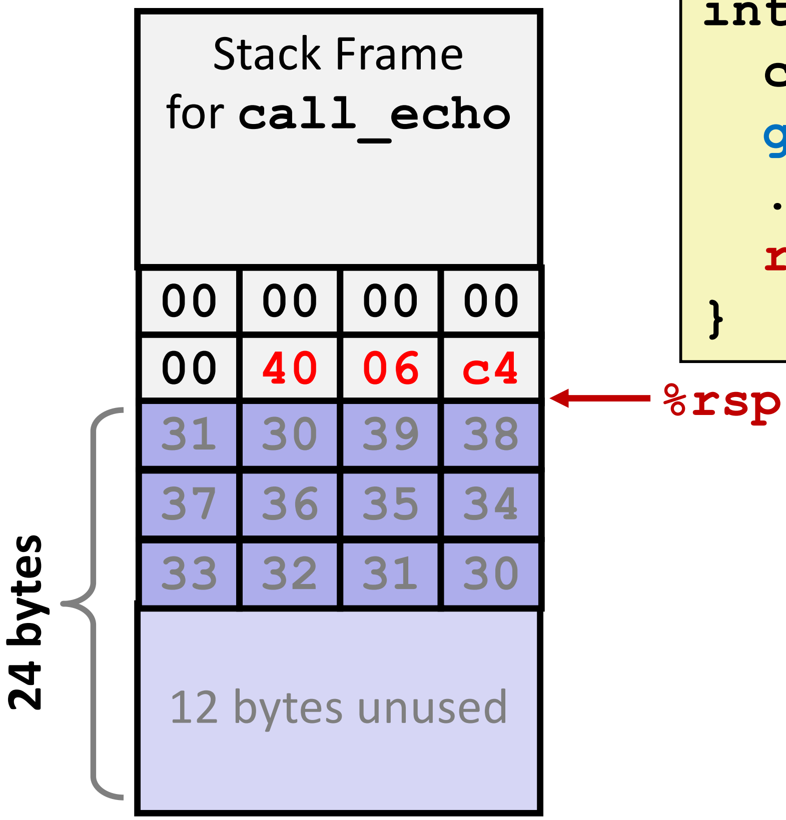
```
int Q() {
   char buf[64];
   gets(buf);
   ...
   return ...;
}
```

```
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);
}
```

```
00000000004006c4 <smash>:
  4006c4:        48 83 ec 08
```

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 c4 06 40
```

The attach string overwrites the return address to the desired target code.

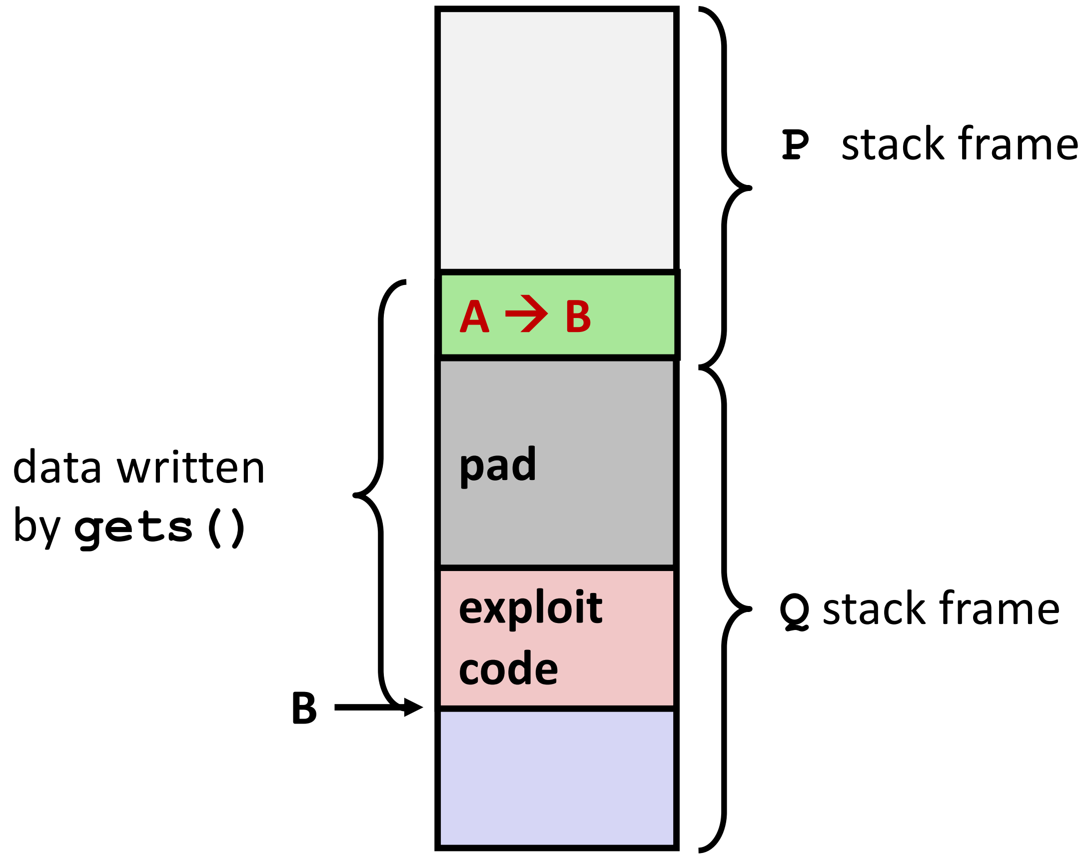
Code Injection Attacks

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

return
address
A

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

Stack after call to **gets()**

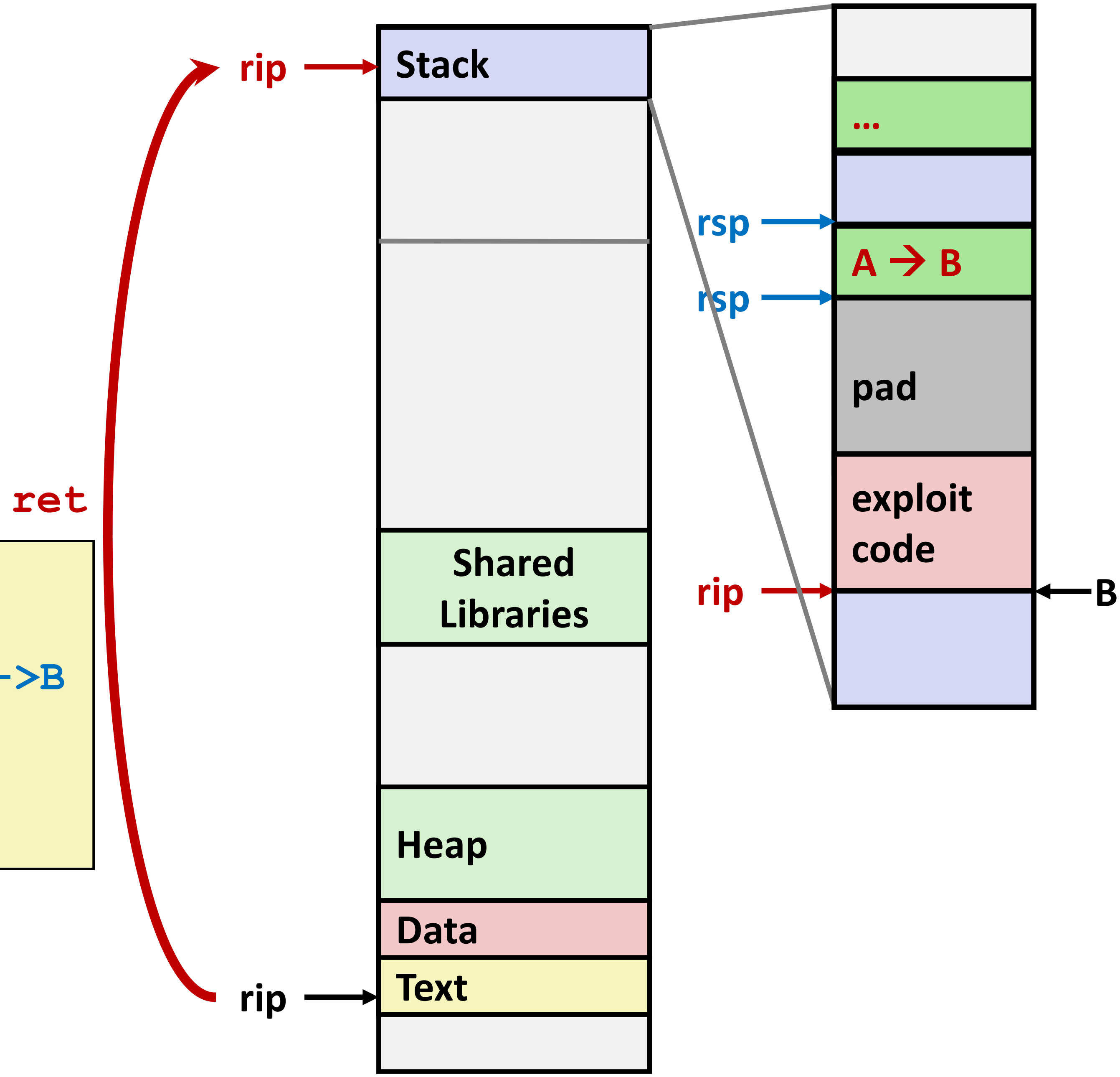


- **Input string contains byte representation of executable code**
- **Overwrite return address A with address of buffer B**
- **When Q executes `ret`, will jump to exploit code**

How Does The Attack Code Execute?

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

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

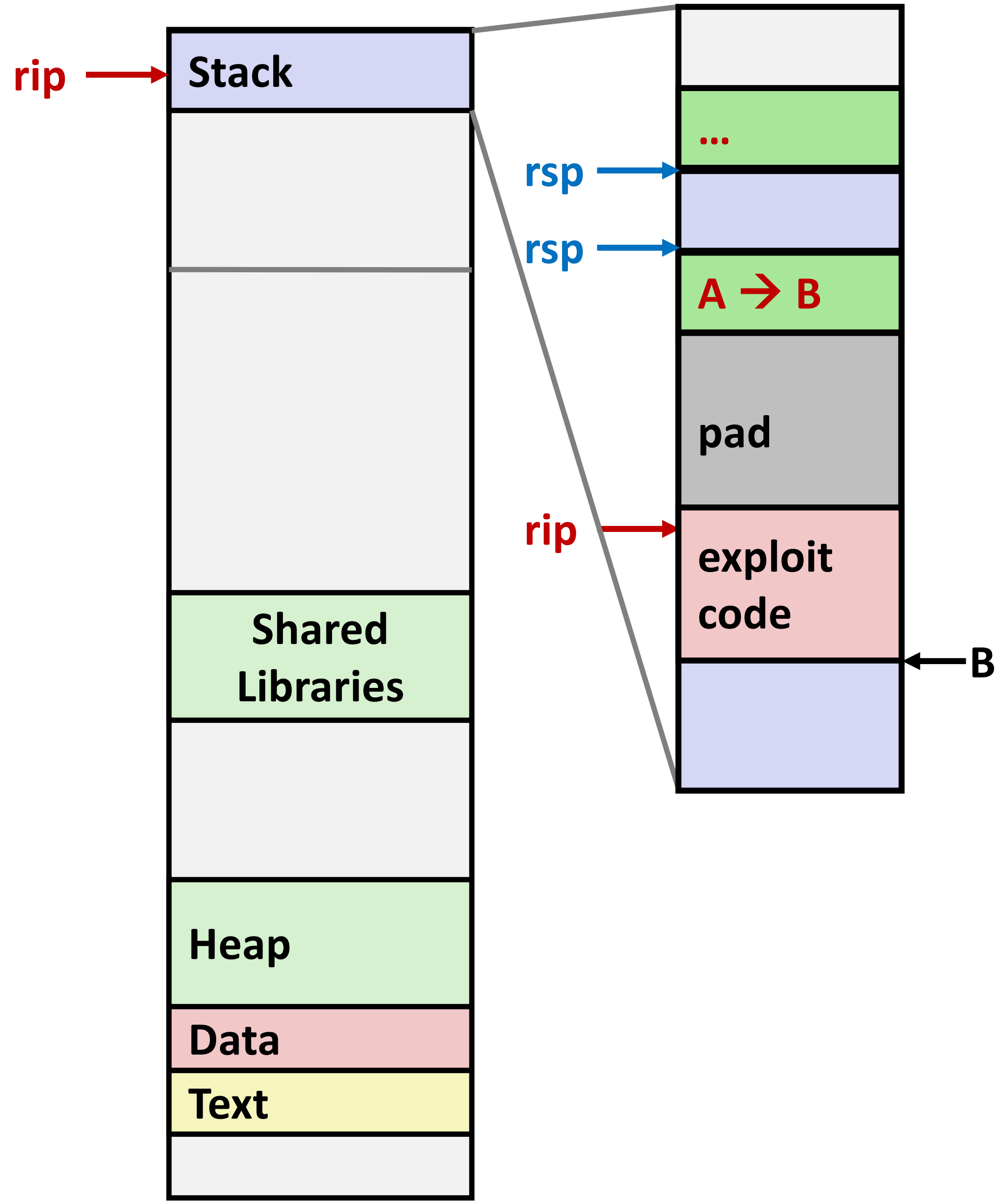


The Attacks Can Be Hidden

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

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

As part of the exploit code, move %rsp back to the next return address then call return.

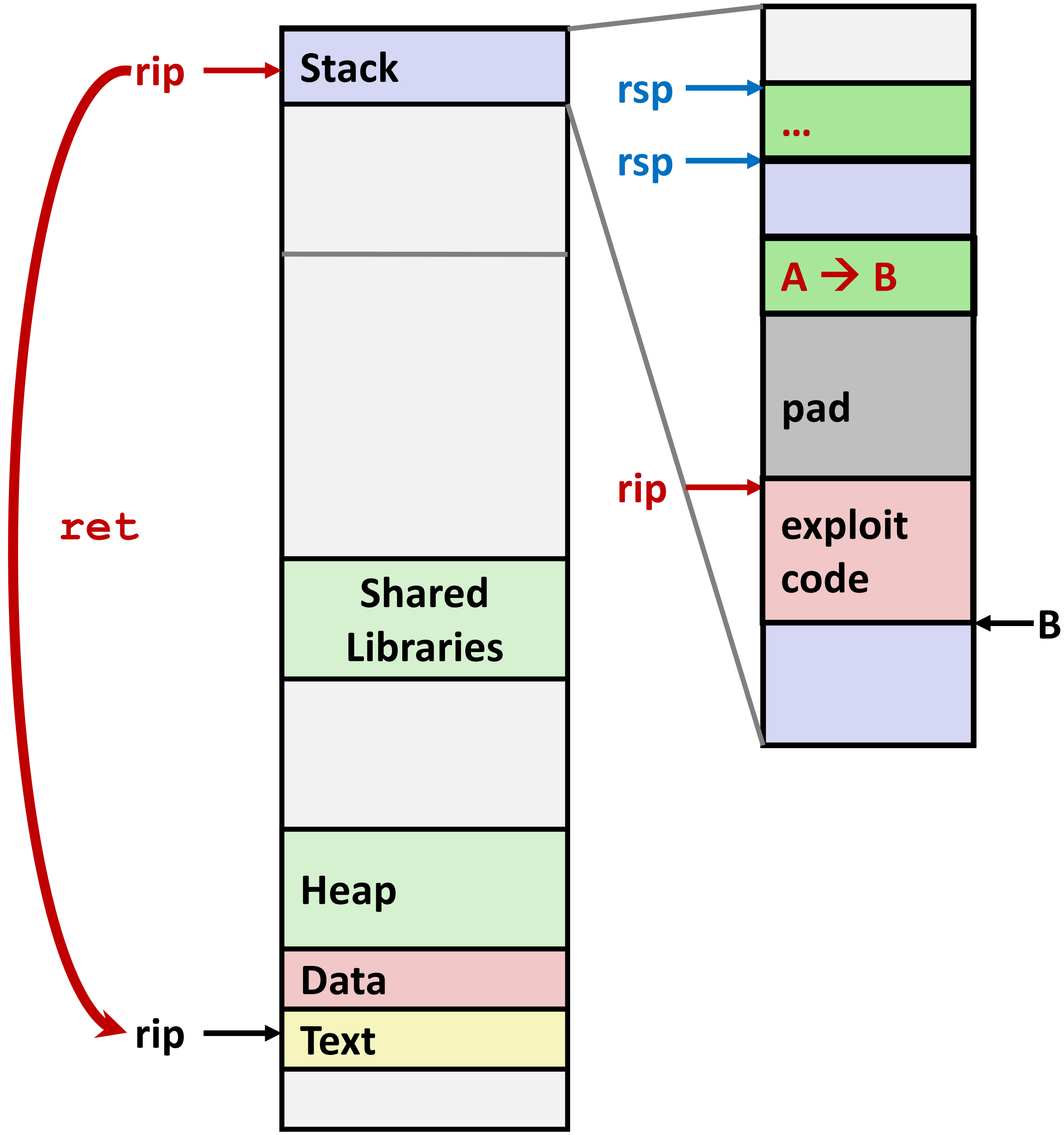


The Attacks Can Be Hidden

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

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

As part of the exploit code, move %rsp back to the next return address then call return.



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
- **You will learn some of the tricks in attacklab**
  - Hopefully to convince you to never leave such holes in your programs!!

What to do about buffer overflow attacks

- **Avoid overflow vulnerabilities**
- **Employ system-level protections**
- **Have compiler use “stack canaries”**

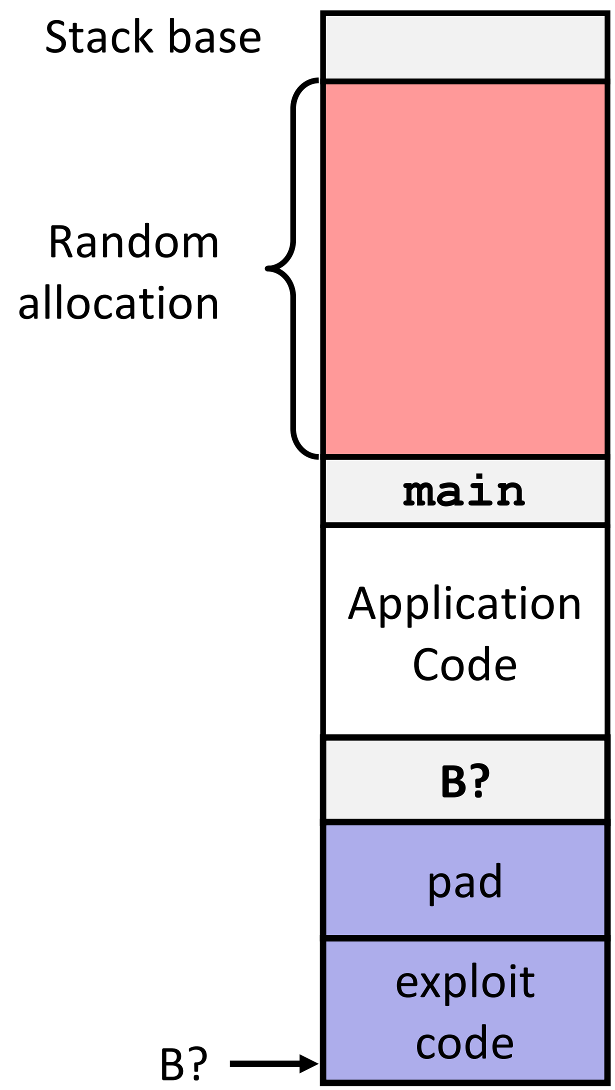
1. Avoid Overflow Vulnerabilities in Code (!)

```
/* 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

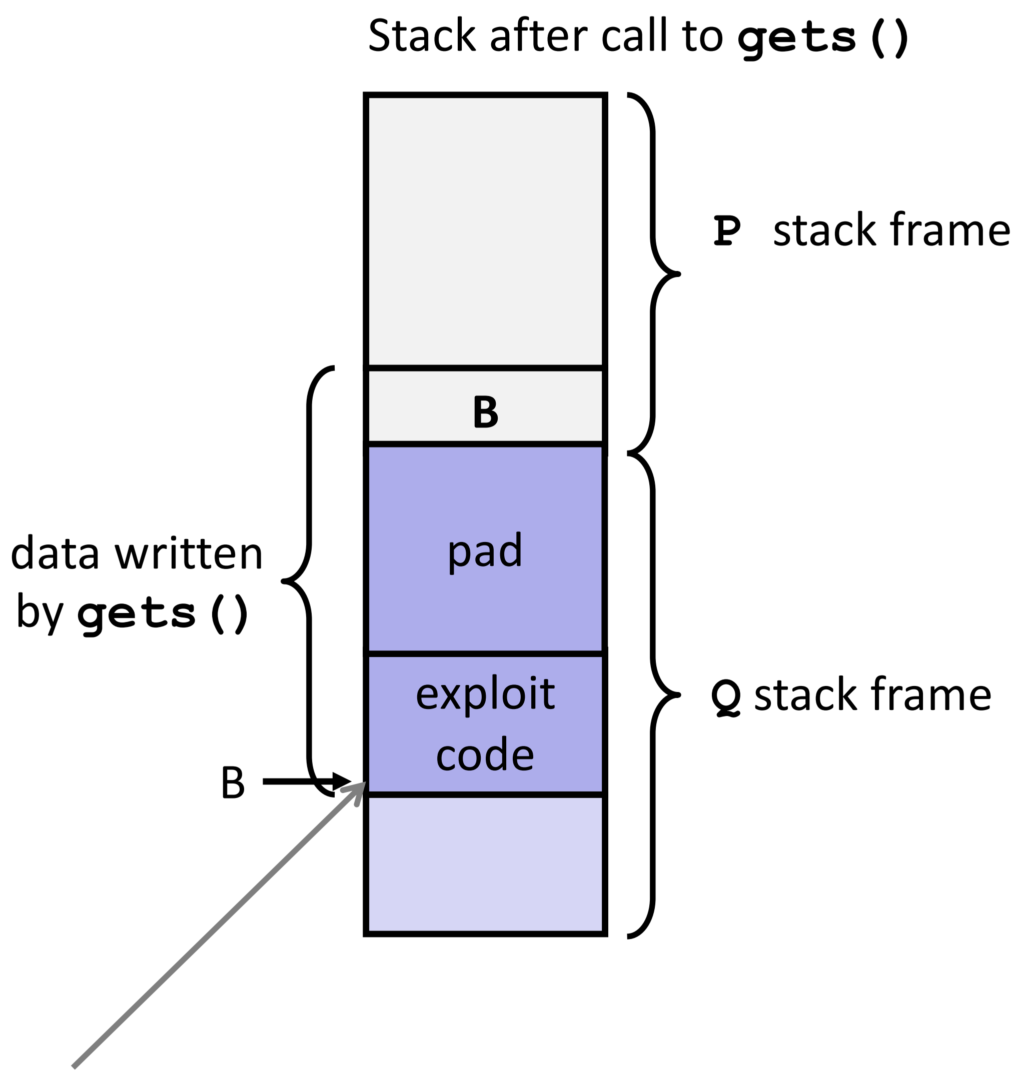
2. System-Level Protections can help

- **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
    - 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**
  - It is the default on some system; on linuxlabs, I enabled it using **`-fstack-protector-all`**

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

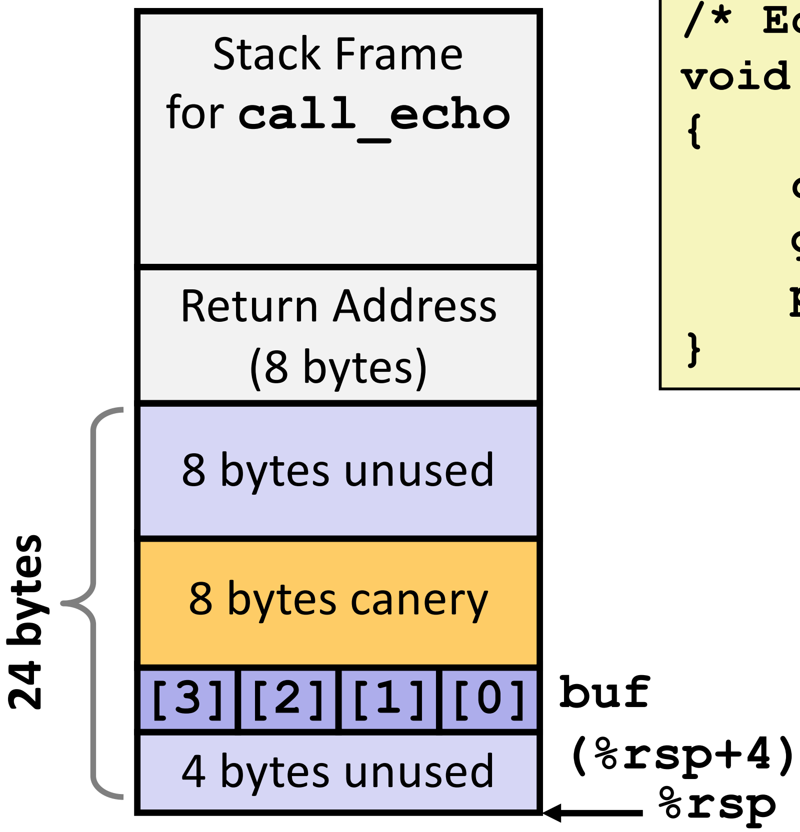
```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

Protected Buffer Disassembly

echo:

```
400739:  sub    $0x18,%rsp
40073d:  mov    %fs:0x28,%rax
400746:  mov    %rax,0x8(%rsp)
40074b:  xor    %eax,%eax
40074d:  lea    0x4(%rsp),%rdi
400752:  callq  4006c6 <gets>
400757:  lea    0x4(%rsp),%rdi
40075c:  callq  400560 <puts@plt>
400761:  mov    0x8(%rsp),%rax
400766:  xor    %fs:0x28,%rax
40076f:  jne    400776 <echo+0x3d>
400771:  add    $0x18,%rsp
400775:  retq
400776:  callq  400570 <__stack_chk_fail@plt>
```

Setting Up Canary

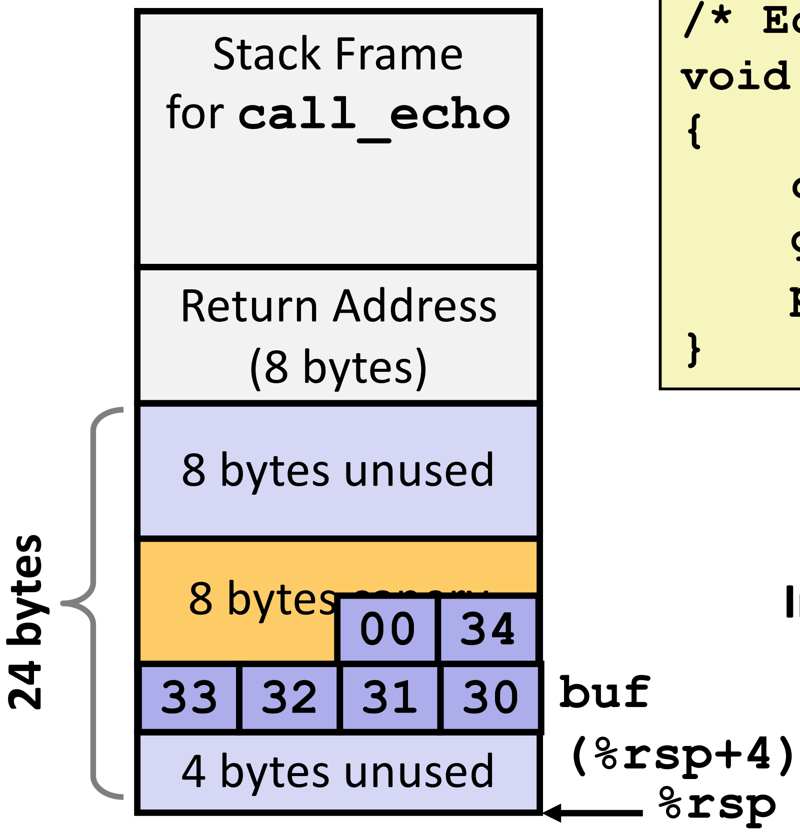



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Before call to gets

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

Checking Canary




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

Input: ***01234***

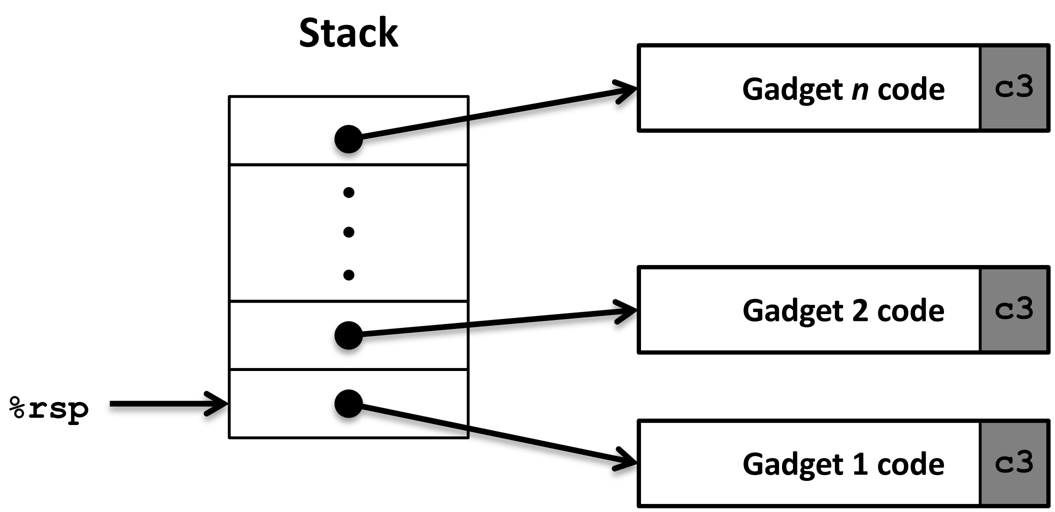
After call to gets

```
echo:
    . . .
    movq    8(%rsp), %rax      # Retrieve from stack
    xorq    %fs:40, %rax       # Compare to canary
    jne     .L6                # If not equal, jump
                               # to __stack_chk_fail
```

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

ROP Execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

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;
}
```

Encodes `movq %rax, %rdi`

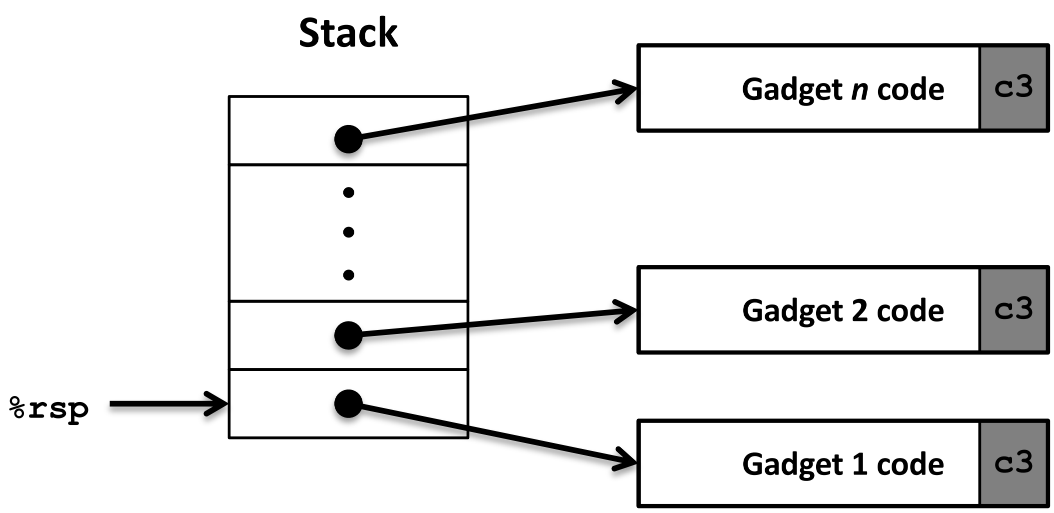
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

Gadget address = `0x4004dc`

- Repurpose byte codes

ROP Execution



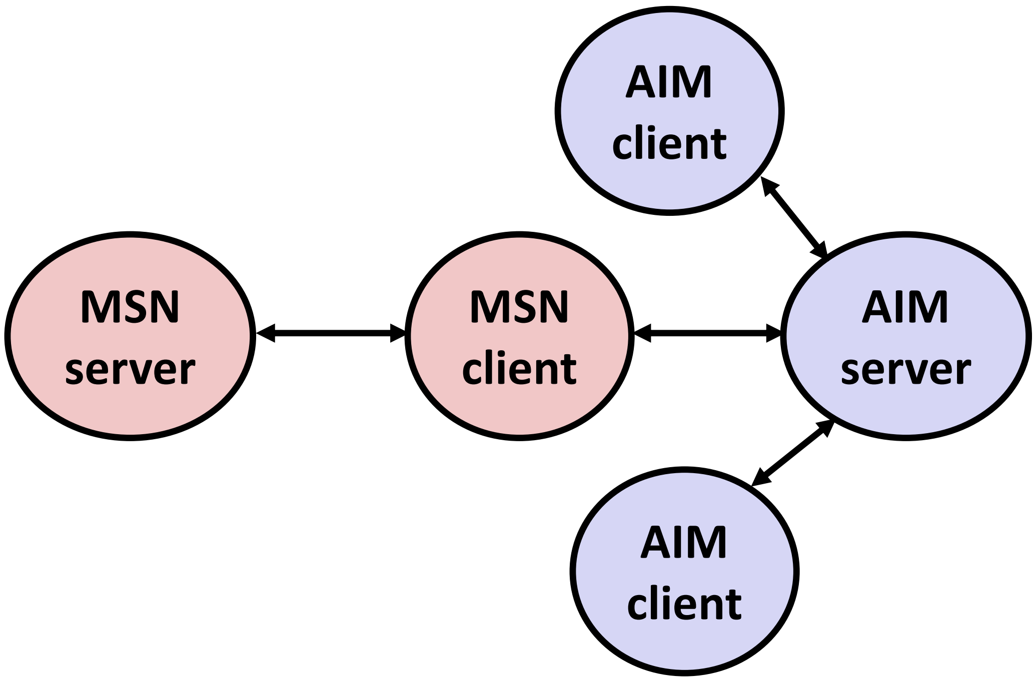
- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

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 (Computer Emergency Response Team)was formed

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**
  - 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
  - Mysteriously, Messenger clients can no longer access AIM servers
  - 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
    - Interesting recounting of events by an engineer at Microsoft at the time: https://nplusonemag.com/issue-19/essays/chat-wars/

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!

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 (i.e., infect another software)
  - Does not run independently

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

Program Optimization

B&O Readings: 5

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Today

- **Overview**
- **Machine-Independent Optimizations**
  - Code motion/precomputation
  - Strength reduction
  - Common Subexpression Elimination
  - Removing unnecessary procedure calls
- **Optimization Blockers**
  - Procedure calls
  - Memory aliasing
- **Putting it all together**

Performance Realities

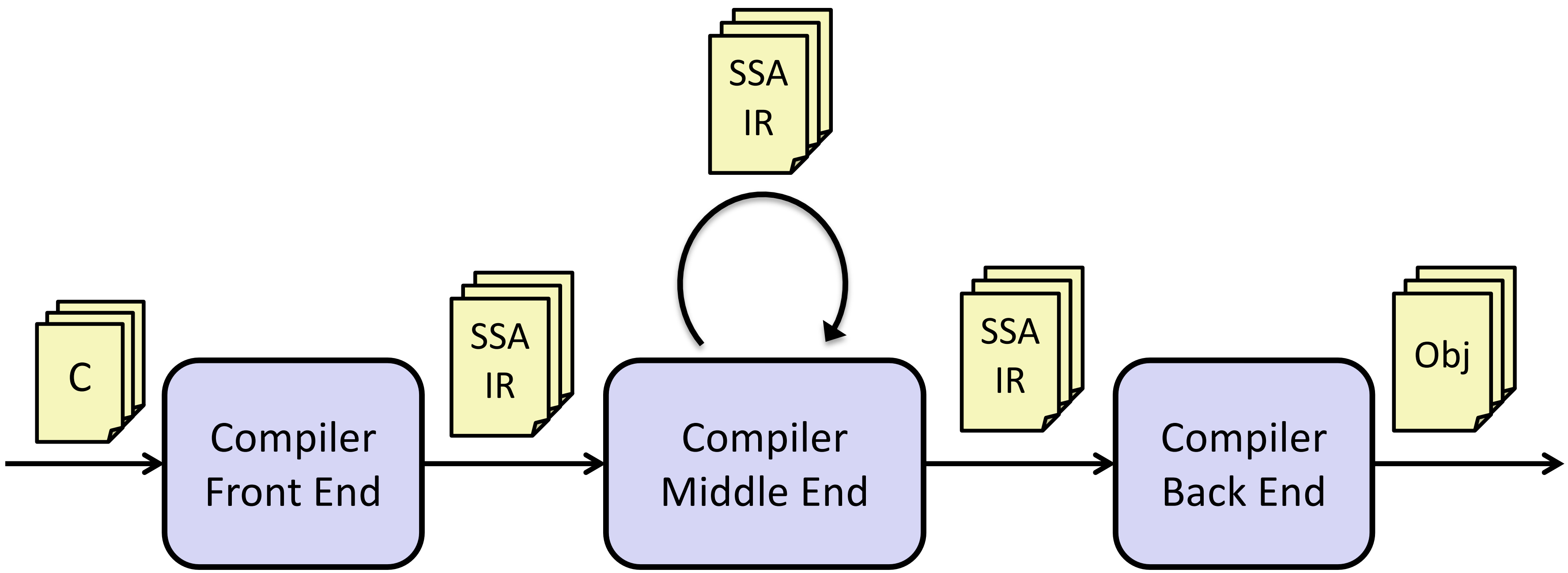
- ***There's more to performance than asymptotic complexity***

- **Constant factors matter!**
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Compilers do 2 things:

- **Code generation**
  - Translate high-level language to machine instructions, 1 statement at a time
- **Optimization**
  - Preserve meaning but improve performance
  - Active research area, but some standard optimizations

Bird's Eye View of Compiler Optimizations



- C: your ordinary C programs
- SSA IR: Static Single Assignment Intermediate Representation
- Obj: object files that are ISA dependent

Optimizing Compilers

- **Provide efficient mapping of program to machine**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **Don’t (usually) improve asymptotic efficiency**
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- **Have difficulty overcoming “optimization blockers”**
  - potential memory aliasing
  - potential procedure side-effects

Generally Useful Optimizations

- **Optimizations that you or the compiler should do regardless of processor / compiler**
  - Code Motion
  - Strength Reduction
  - Common Subexpression Eliminiation
  - Tail Recursion Elimination

Code Motion

- **Reduce frequency with which computation performed**
  - If it will always produce same result
  - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
         a[n*i+j] = b[j];
}
```

→

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
         a[ni+j] = b[j];
```

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
         a[n*i+j] = b[j];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
         *rowp++ = b[j];
```

```
set_row:
        testq    %rcx, %rcx              # Test n
        jle      .L1                     # If 0, goto done
        imulq    %rcx, %rdx              # ni = n*i
        leaq     (%rdi,%rdx,8), %rdx     # rowp = A + ni*8
        movl     $0, %eax                # j = 0
.L3:                                     # loop:
        movsd    (%rsi,%rax,8), %xmm0    # t = b[j]
        movsd    %xmm0, (%rdx,%rax,8)    # M[A+ni*8 + j*8] = t
        addq     $1, %rax                # j++
        cmpq     %rcx, %rax              # j:n
        jne      .L3                     # if !=, goto loop
.L1:                                     # done:
        rep ; ret
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x   -->   x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

1 multiplication: i*n

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Tail Recursion Elimination

- Varies across languages
- Optional in C

```c
int fact(int n) {
  if(n <= 0)
     return 1;
  else {
      int n1_fact = fact(n-1);
      return n1_fact * n;
  }
}
```

-O1 →

```
fact:
        pushq   %rbx
        movl    %edi, %ebx
        movl    $1, %eax
        testl   %edi, %edi
        jle     .L2
        leal    -1(%rdi), %edi
        call    fact
        imull   %ebx, %eax
.L2:
        popq    %rbx
        ret
```

-O2 →

```
fact:
        testl   %edi, %edi
        movl    $1, %eax
        jle     .L2
.L3:
        imull   %edi, %eax
        subl    $1, %edi
        jne     .L3
.L2:
        ret
```

With –O2:

- Compiler doesn't create stack frame for tail recursion
- No call to `fact`, just a jmp
- No longer need to store away caller-saved registers!