Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

University of Minnesota

Stack Overflow

With Slides from Bryant and O'Hallaron

String Library Code

- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```
/* 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;
}
```

- 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);
}
```

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

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

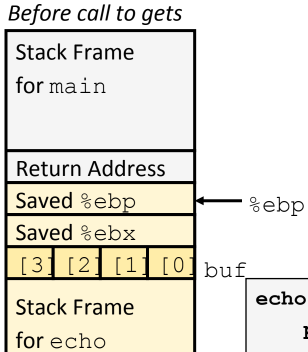
```
80485c5:   55                 push   %ebp
80485c6:   89 e5              mov    %esp,%ebp
80485c8:   53                 push   %ebx
80485c9:   83 ec 14           sub    $0x14,%esp
80485cc:   8d 5d f8           lea 0xfffffff8(%ebp),%ebx
80485cf:   89 1c 24           mov    %ebx,(%esp)
80485d2:   e8 9e ff ff ff     call   8048575 <gets>
80485d7:   89 1c 24           mov    %ebx,(%esp)
80485da:   e8 05 fe ff ff     call   80483e4 <puts@plt>
80485df:   83 c4 14           add    $0x14,%esp
80485e2:   5b                 pop    %ebx
80485e3:   5d                 pop    %ebp
80485e4:   c3                 ret
```

call_echo:

```
80485eb:  e8 d5 ff ff ff     call   80485c5 <echo>
80485f0:  c9                 leave
80485f1:  c3                 ret
```


Buffer Overflow Stack

Before call to gets

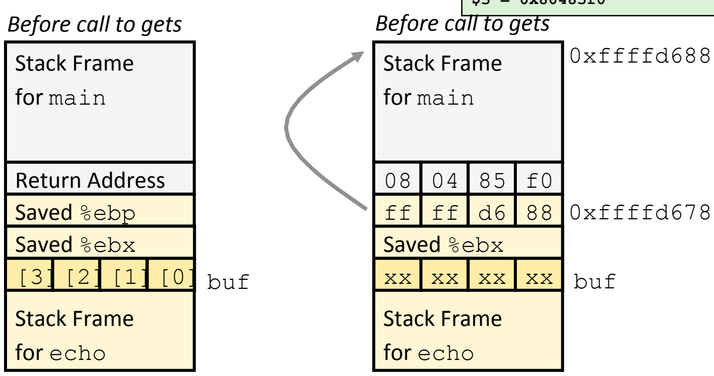



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

Buffer Overflow Stack Example

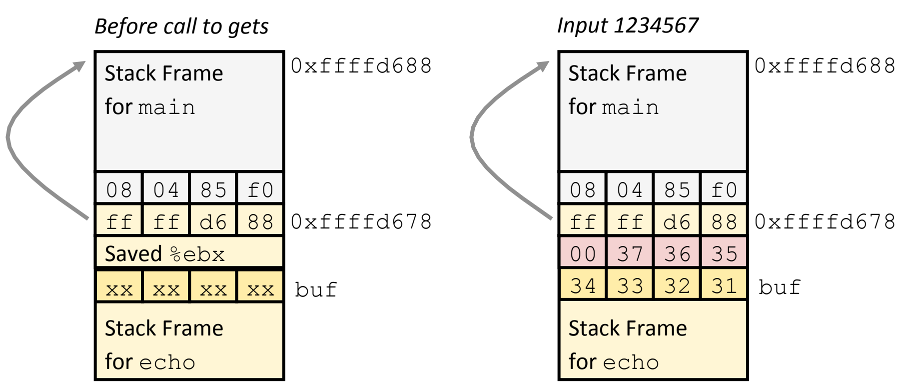
```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
```




```
80485eb:   e8 d5 ff ff ff    call    80485c5 <echo>
80485f0:   c9                leave
```

With Slides from Bryant and O'Hallaron

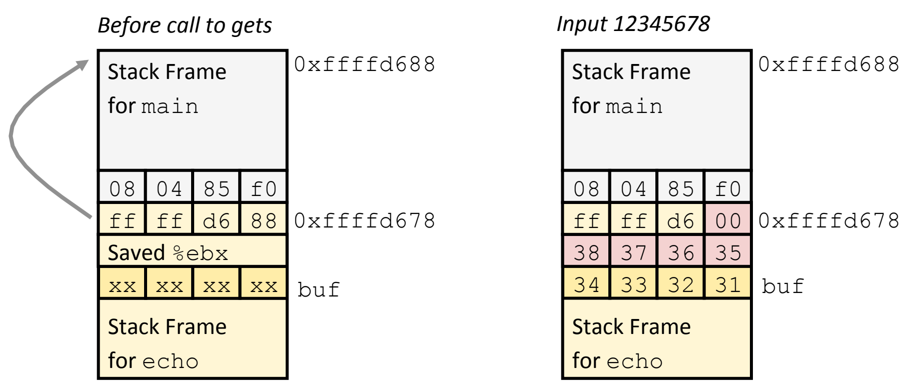
Buffer Overflow Example #1



Overflow buf, and corrupt %ebx,
but no problem

With Slides from Bryant and O'Hallaron

Buffer Overflow Example #2

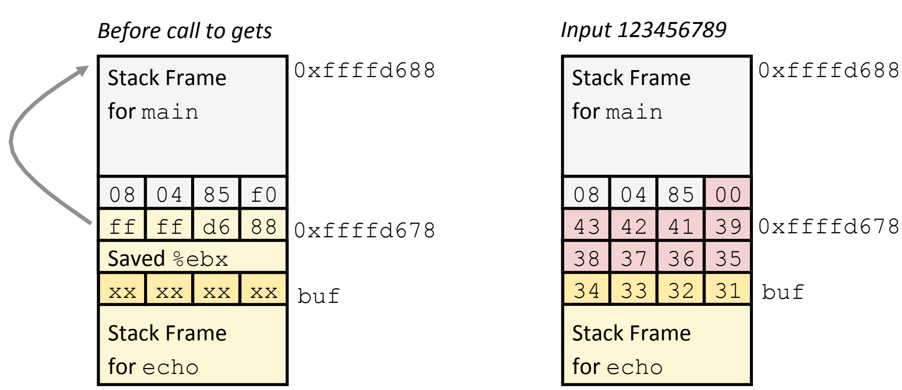


Base pointer corrupted

```
. . .
80485eb:   e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:   c9                leave    # Set %ebp to corrupted value
80485f1:   c3                ret
```

With Slides from Bryant and O'Hallaron

Buffer Overflow Example #3



Return address corrupted

```
80485eb:   e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:   c9                leave   # Desired return point
```

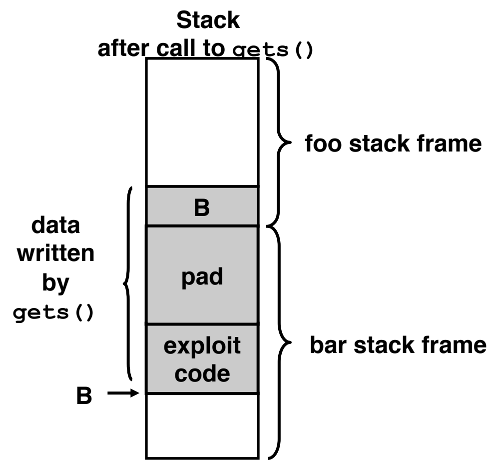
With Slides from Bryant and O'Hallaron

Malicious Use of Buffer Overflow

return address A →

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

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



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

With Slides from Bryant and O'Hallaron

Avoiding Overflow Vulnerability

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

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

With Slides from Bryant and O'Hallaron

Yet Another Example

```
main() {
   unsigned long long ll = 0xdeadbeefbeefdead;
   unsigned int i = 0x12345678;
   printf("%x %x\n", ll, i);
}
```


Yet Another Example

With Slides from Bryant and O'Hallaron

System-Level Protections

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code
- 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

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

With Slides from Bryant and O'Hallaron

Stack Canaries

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

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

With Slides from Bryant and O'Hallaron

Protected Buffer Disassembly

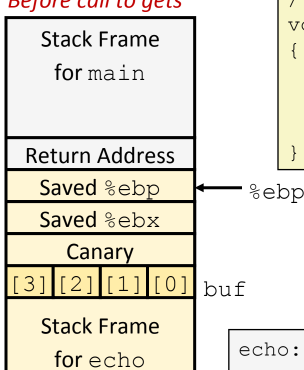
echo:

```
804864d:    55                      push   %ebp
804864e:    89 e5                   mov    %esp,%ebp
8048650:    53                      push   %ebx
8048651:    83 ec 14                sub    $0x14,%esp
8048654:    65 a1 14 00 00 00       mov    %gs:0x14,%eax
804865a:    89 45 f8                mov    %eax,0xfffffff8(%ebp)
804865d:    31 c0                   xor    %eax,%eax
804865f:    8d 5d f4                lea    0xfffffff4(%ebp),%ebx
8048662:    89 1c 24                mov    %ebx,(%esp)
8048665:    e8 77 ff ff ff          call   80485e1 <gets>
804866a:    89 1c 24                mov    %ebx,(%esp)
804866d:    e8 ca fd ff ff          call   804843c <puts@plt>
8048672:    8b 45 f8                mov    0xfffffff8(%ebp),%eax
8048675:    65 33 05 14 00 00 00    xor    %gs:0x14,%eax
804867c:    74 05                   je     8048683 <echo+0x36>
804867e:    e8 a9 fd ff ff          call   804842c <FAIL>
8048683:    83 c4 14                add    $0x14,%esp
8048686:    5b                      pop    %ebx
8048687:    5d                      pop    %ebp
8048688:    c3                      ret
```

With Slides from Bryant and O'Hallaron

Setting Up Canary

Before call to gets



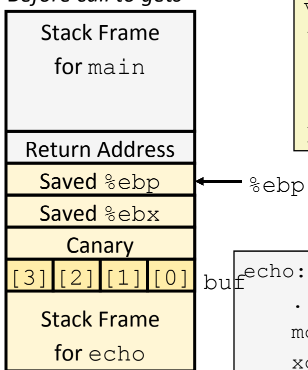

```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movl    %gs:20, %eax      # Get canary
    movl    %eax, -8(%ebp)    # Put on stack
    xorl    %eax, %eax        # Erase canary
    . . .
```

With Slides from Bryant and O'Hallaron

Checking Canary

Before call to gets

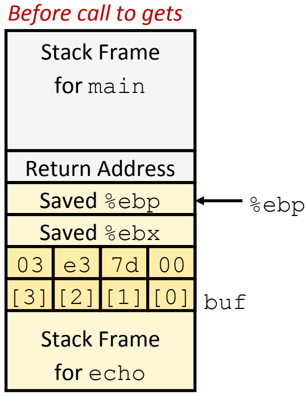



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

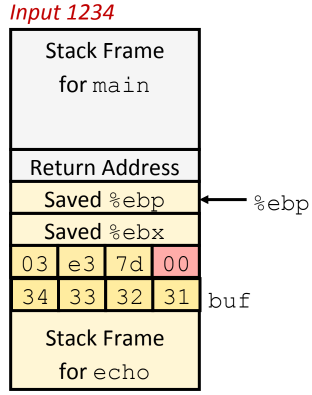
```
echo:
    . . .
    movl    -8(%ebp), %eax      # Retrieve from stack
    xorl    %gs:20, %eax        # Compare with Canary
    je      .L24                # Same: skip ahead
    call    __stack_chk_fail    # ERROR
.L24:
    . . .
```


Canary Example

Before call to gets



Input 1234




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Benign corruption!
(allows programmers to make silent off-by-one errors)

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
  - Add itself to other programs
  - Cannot run independently
- Both are (usually) designed to spread among computers and to wreak havoc

Non-Local Jumps

Nonlocal Jumps: `setjmp/longjmp`

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
  - Controlled to way to break the procedure call / return discipline
  - Useful for error recovery and signal handling
- `int setjmp(jmp_buf j)`
  - Must be called before longjmp
  - Identifies a return site for a subsequent longjmp
  - Called once, returns one or more times
- Implementation:
  - Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in **`jmp_buf`**
  - Return 0

`setjmp/longjmp` (cont)

- `void longjmp(jmp_buf j, int i)`
  - Meaning:
    - return from the **`setjmp`** remembered by jump buffer **`j`** again ...
    - ... this time returning **`i`** instead of 0
  - Called after **`setjmp`**
  - Called once, but never returns
- `longjmp` Implementation:
  - Restore register context (stack pointer, base pointer, PC value) from jump buffer **`j`**
  - Set **`%eax`** (the return value) to **`i`**
  - Jump to the location indicated by the PC stored in jump buf **`j`**

With Slides from Bryant and O'Hallaron

setjmp/longjmp Example

```
#include <setjmp.h>
jmp_buf buf;

main() {
   if (setjmp(buf) != 0) {
      printf("back in main due to an error\n");
   else
      printf("first time through\n");
   p1(); /* p1 calls p2, which calls p3 */
}
...
p3() {
   <error checking code>
   if (error)
      longjmp(buf, 1)
}
```


Limitations of Nonlocal Jumps

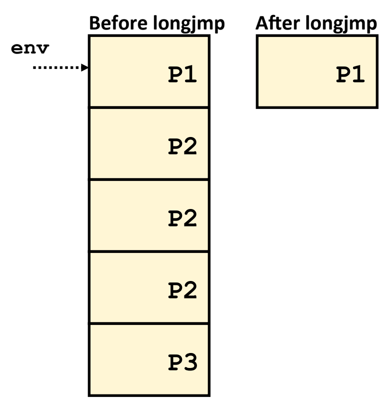
- Works within stack discipline
  - *Can only long jump to environment of function that has been called but not yet completed*

```
jmp_buf env;

P1()
{
  if (setjmp(env)) {
    /* Long Jump to here */
  } else {
    P2();
  }
}

P2()
{  . . . P2(); . . . P3(); }

P3()
{
  longjmp(env, 1);
}
```



Limitations of Long Jumps (cont.)

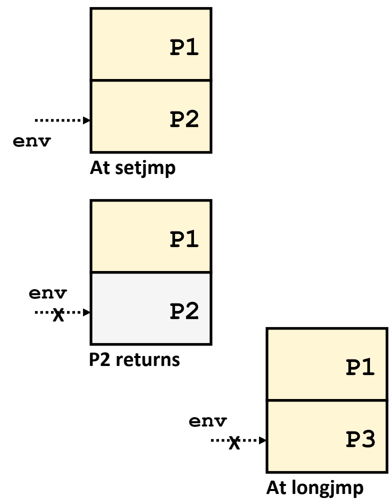
- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

P1()
{
  P2(); P3();
}

P2()
{
   if (setjmp(env)) {
    /* Long Jump to here */
  }
}

P3()
{
  longjmp(env, 1);
}
```



Procedures (x86-64)

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Twice the number of registers that are accessible as 8, 16, 32, 64 bits

With Slides from Bryant and O'Hallaron

x86-64 Integer Registers: Usage Conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

With Slides from Bryant and O'Hallaron

x86-64 Registers

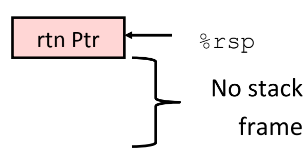
- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- All references to stack frame via stack pointer
  - Eliminates need to update **`%ebp/%rbp`**
- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq    (%rdi), %rdx
   movq    (%rsi), %rax
   movq    %rax, (%rdi)
   movq    %rdx, (%rsi)
   ret
```

- Operands passed in registers
  - First (**`xp`**) in **`%rdi`**, second (**`yp`**) in **`%rsi`**
  - 64-bit pointers
- No stack operations required (except `ret`)
- Avoiding stack
  - Can hold all local information in registers



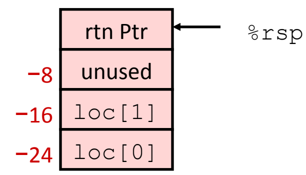
x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq   (%rdi), %rax
  movq   %rax, -24(%rsp)
  movq   (%rsi), %rax
  movq   %rax, -16(%rsp)
  movq   -16(%rsp), %rax
  movq   %rax, (%rdi)
  movq   -24(%rsp), %rax
  movq   %rax, (%rsi)
  ret
```

Avoiding Stack Pointer Change

- Can hold all information within small window beyond stack pointer



x86-64 NonLeaf without Stack Frame

```
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- No values held while swap being invoked
- No callee save registers needed
- `rep` instruction inserted as no-op
  - Based on recommendation from AMD

```
swap_ele:
   movslq %esi,%rsi           # Sign extend i
   leaq   8(%rdi,%rsi,8), %rax # &a[i+1]
   leaq   (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)
   movq   %rax, %rsi           # (2nd arg)
   call   swap
   rep                         # No-op
   ret
```

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee save registers
- Must set up stack frame to save these registers

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

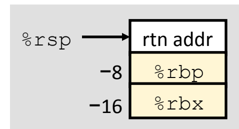

Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)        # Save %rbx
    movq    %rbp, -8(%rsp)         # Save %rbp
    subq    $16, %rsp              # Allocate stack frame
    movslq  %esi,%rax              # Extend i
    leaq    8(%rdi,%rax,8), %rbx   # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp    # &a[i]   (callee save)
    movq    %rbx, %rsi             # 2nd argument
    movq    %rbp, %rdi             # 1st argument
    call    swap
    movq    (%rbx), %rax           # Get a[i+1]
    imulq   (%rbp), %rax           # Multiply by a[i]
    addq    %rax, sum(%rip)        # Add to sum
    movq    (%rsp), %rbx           # Restore %rbx
    movq    8(%rsp), %rbp          # Restore %rbp
    addq    $16, %rsp              # Deallocate frame
    ret
```

With Slides from Bryant and O'Hallaron

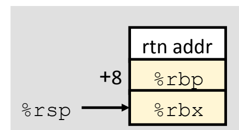
Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)          # Save %rbx
movq    %rbp, -8(%rsp)           # Save %rbp
```




```
subq    $16, %rsp                # Allocate stack frame
```

● ● ●




```
movq    (%rsp), %rbx             # Restore %rbx
movq    8(%rsp), %rbp            # Restore %rbp

addq    $16, %rsp                # Deallocate frame
```


Interesting Features of Stack Frame

Allocate entire frame at once

- All stack accesses can be relative to **%rsp**
- Do by decrementing stack pointer
- Can delay allocation, since safe to temporarily use red zone

Simple deallocation

- Increment stack pointer
- No base/frame pointer needed

x86-64 Procedure Summary

Heavy use of registers

- Parameter passing
- More temporaries since more registers

Minimal use of stack

- Sometimes none
- Allocate/deallocate entire block

Many tricky optimizations

- What kind of stack frame to use
- Various allocation techniques