C and Assembly

15-123

Systems Skills in C and Unix

Plan today

- Revisit the ISA
- Write some assembly programs
- Compile them with the assembler
- Run them with the simulator

History of processors

- 4004: 1971, 2300 transistors, 108KHz
  First ever single-chip microprocessor. Designed for desktop calculator

- 8086/8088: 1978, 29K transistors, 5MHz
  Early 16-bit micro. Basis for IBM PC

- i386: 1985, 275K transistors, 16MHz
  Extend x86 from 16 to 32 bits. Basis for current linux systems

- Pentium, …, Pentium 4: Got serious about performance/capacity. Overtook competitors

- Limitations of 32 bits: Only GB of virtual memory. Need to switch to 64 bits

- AMD x86-64: 2002
  Opteron, Athlon

- Extension of x86 to 64 bits. Fully backward compatible.

- Intel Pentium 4 Xeon EM64T (code named Nocona): 2004 100M transistors 3.2GHz

Why learn Assembly

- To understand the machine language execution model
  - Understanding bugs
  - Optimizing the code
  - Creating and fighting malware
    - X86 is the language of choice
- Learn assembly to write
  - Device drivers
  - Game programming
  - OS kernel

Assembly Code Model

CPU

Program Counter: %rip

Registers: %rdi, %rsi, %rdx, %rcx etc..

Memory: Linear array of bytes but split things up

Executable code

Global Data

Stack (Procedure state & data)

Heap (Dynamically allocated data)

Basic operations

- Fetch instruction at %eip
- Read values from registers and/or memory
- Perform arithmetic operation
- Write values to registers and/or memory
- Update %eip to next instruction

Main properties of assembly code

- Textual representation of individual machine Instructions
- Much information from C program missing
  - Variable names
  - Data types
  - Higher level control structures

Assembly Programming

- Assembly programming provides a view of the instruction set architecture from programmers perspective
- Programmer must understand the instruction set architecture of the processor class
  - Eg: iA-32 (x86, x86-32, i386)
- Example

```
#this is in a file first.s
.globl main
main:
    movl $20, %eax
    movl $10, %ebx
ret
```

- **Can compile and run**
  - **gcc first.s**
  - **./a.out**

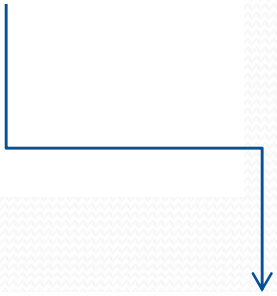
C to Assembly

```c
int main(){
 int x=10,y=15;
 return 0;
}
```

```
.globl main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $10, -8(%rbp)
        movl    $15, -4(%rbp)
        movl    $0, %eax
        leave
        ret
```

Another Example

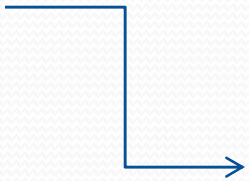
```
long fun(long x, long y, long z)
{
    long t = x * y - z;
    return t;
}
```



```
# IA32 code
# x at 8(%epb), y at 12(%ebp)
# Return value in %eax
fun:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        imull   8(%ebp), %eax
        subl    16(%ebp), %eax
        leave
        ret
```

or the same code in x86-64

```
# x86-64 code
# x in %rdi, y in %rsi, z in %rdx
# return value in %rax
fun:
        imulq   %rsi, %rdi      # x *= y
        subq    %rdx, %rdi      # x -= z
        movq    %rdi, %rax      # Return value = x
        ret                     # return
```



```
# Disassembled Object Code
0000000000400510 <fun>:
  400510:       48 0f af fe             imul    %rsi,%rdi
  400514:       48 29 d7                sub     %rdx,%rdi
  400517:       48 89 f8                mov     %rdi,%rax
  40051a:       c3                      retq
```

Looping with assembly

```
# factorial of 4
# in file factorial.s

        .LC0:
        .string "%d \n"
        .text
.global main
main:
        movl $4, %eax
        movl $1,  %ebx
L1:     cmpl $0, %eax
        je L2
        imul %eax, %ebx
        decl %eax
        jmp L1
L2:     movl    %ebx, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

functions

```
int foo(int x){
  return x;
}

int main(){
  int x = 20;
  int y = foo(x);
  return 0;
}
```

```
foo:
pushq   %rbp
movq    %rsp, %rbp
movl    %edi, -4(%rbp)
movl    -4(%rbp), %eax
leave
ret
```

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    $20, -8(%rbp)
    movl    -8(%rbp), %edi
    call    foo
    movl    %eax, -4(%rbp)
    movl    $0, %eax
    leave
    ret
```

Stack

- A special data structure such that
  - An entry can only be removed from top
    - pop()
  - An entry can only be added to the top
    - push(object)
- Manipulating the stack
  - **popl %eax**
  - **pushl %ebp**

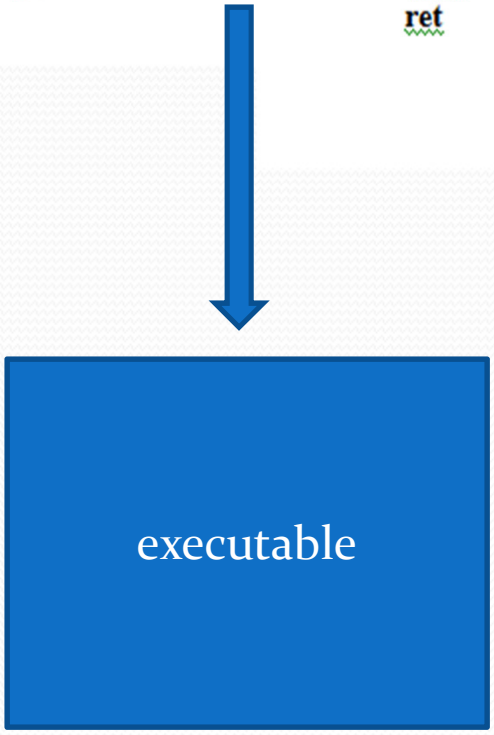
Calling and Returning from a function

- When a function is called
  - Prepare stack and registers to use with the function
- When returning from a function
  - The return address of the calling program is saved
  - Restore the stack and registers to the state before the call
- Function returns a value (if any) through regiser %eax
  - Eg: movl $10 %eax

Mixing C and Assembly

```
// file in main.c
#include <stdio.h>
int main(){
   int i = foo(5);
   printf("The value is %d \n", i);
return 0; }
```

```
# file in foo.s
.global foo
foo:
        movl 4(%esp), %eax    # (esp+4) contains the value 5
        imull %eax, %eax      # multiply the register eax by itself
        ret                   # return values are given back thru eax
```



Global variables

```
// in file global.c
int x = 10;
int main( ) {
    int y  = x;
}
```

```
.globl x
        .data
        .align 4
        .type x, @object
        .size x, 4
x:
        .long 10
        .text
.globl main
        .type main, @function
```

What 15-213 is about

- a programmer's view of how computer systems
  - execute programs
  - store information, and communicate.
- making students to become more effective programmers,
  - issues of performance, portability and robustness.
- Foundation for courses on
  - compilers, networks, operating systems, and computer architecture,
- Topics covered include: machine-level code and its generation by optimizing compilers, performance evaluation and optimization, computer arithmetic, memory organization and management, networking technology and protocols, and supporting concurrent computation.

Y86

%eax	%esi
%ecx	%edi
%edx	%esp
%ebx	%ebp

ZF	SF	OF

PC

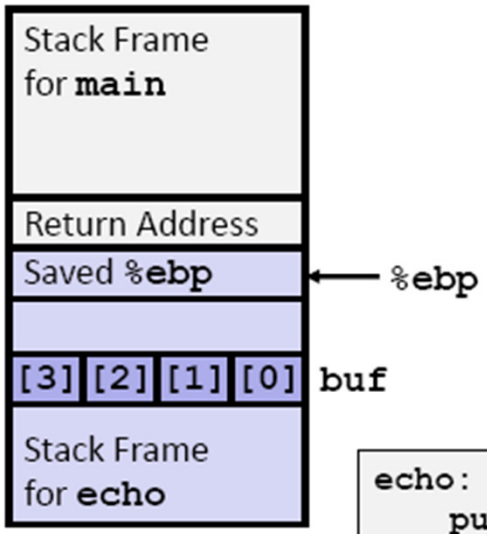
memory

Buffer overflow attacks

Buffer Overflow

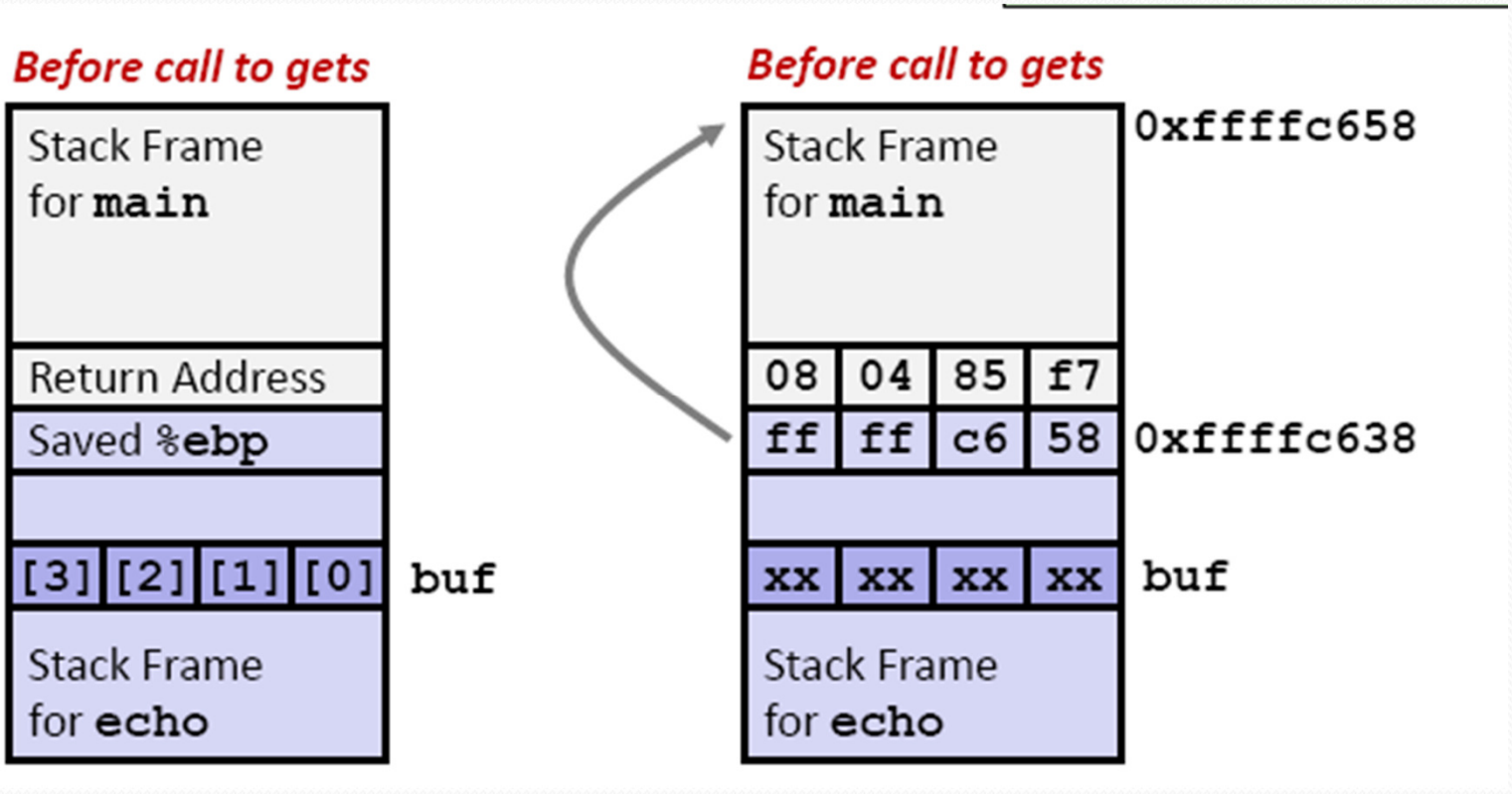
Buffer Overflow Stack

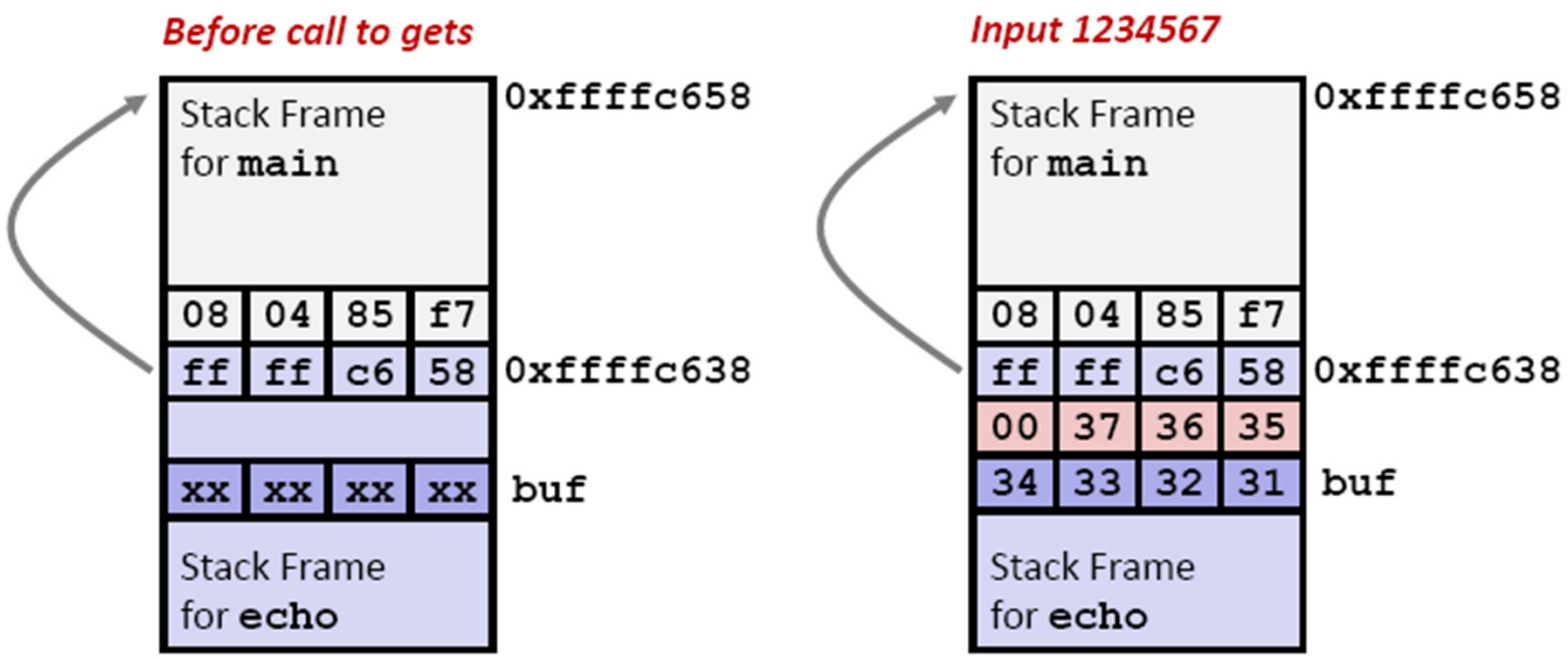
Before call to gets



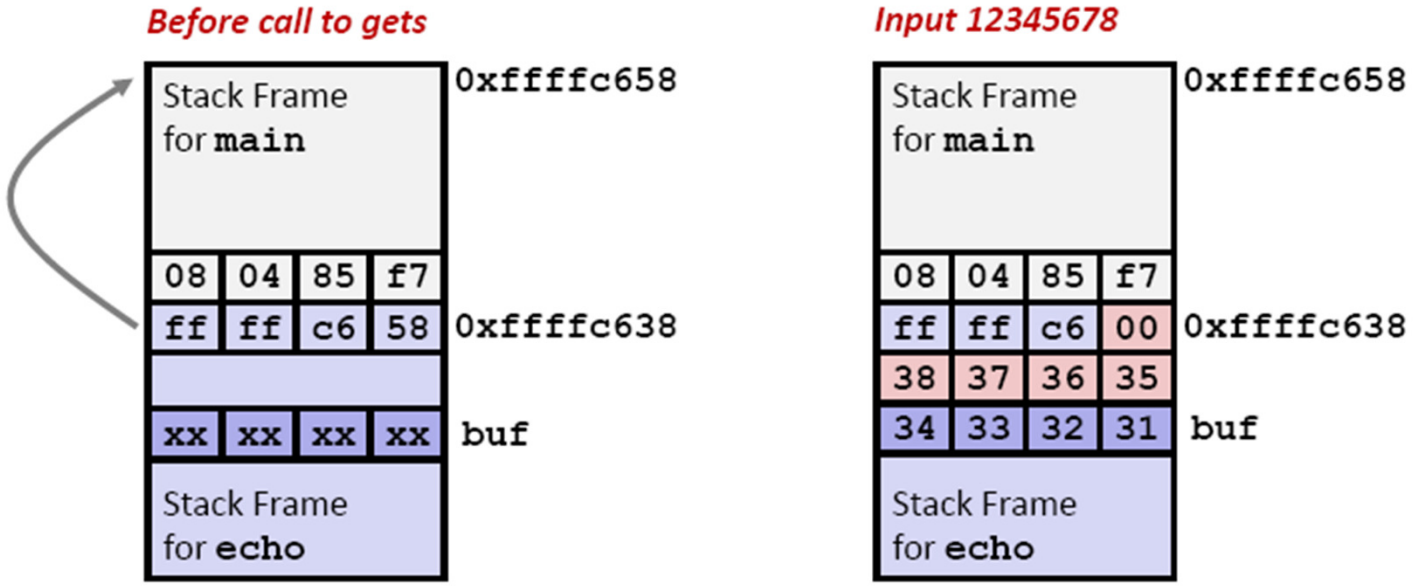

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

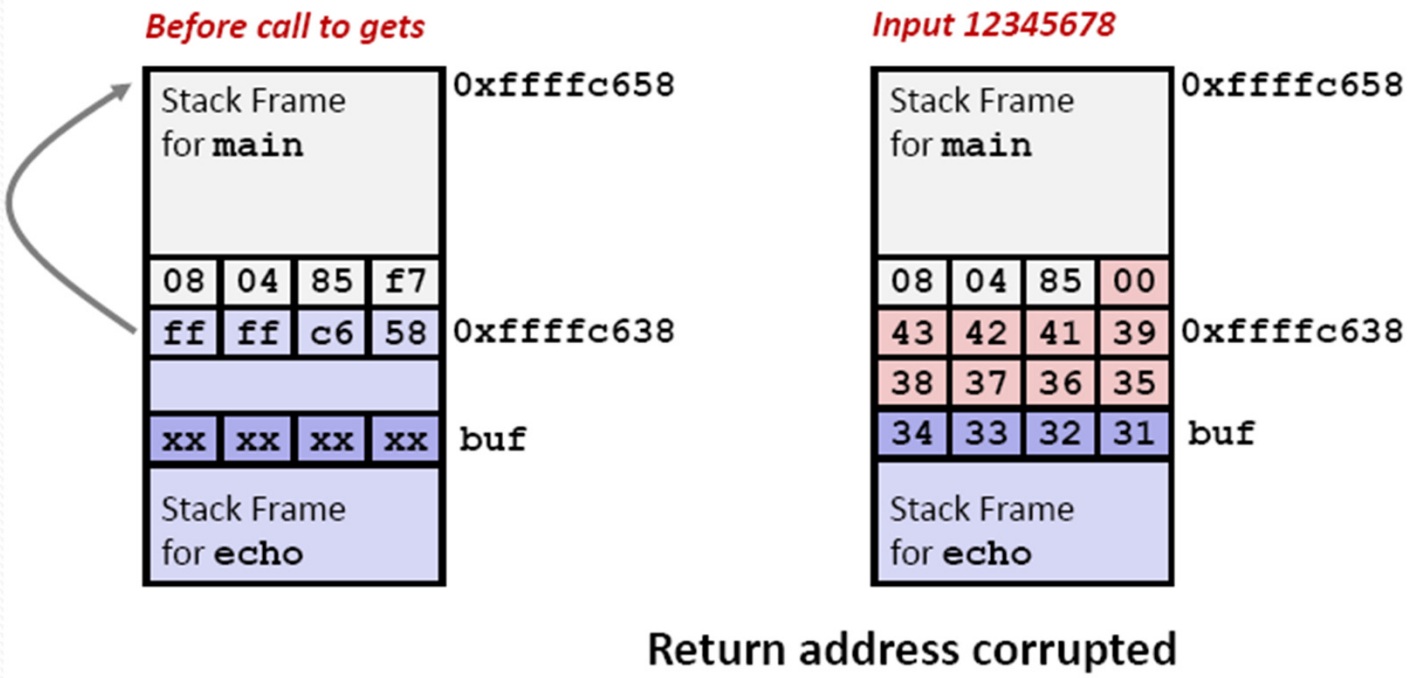

Before call to gets
Stack Frame for main
Return Address
Saved %ebp
[3] [2] [1] [0]
buf
Stack Frame for echo
Before call to gets
Stack Frame for main
0xffffc658
08 04 85 f7
ff ff c6 58
0xffffc638
xx xx xx xx
buf
Stack Frame for echo



Overflow buf, but no problem



Base pointer corrupted

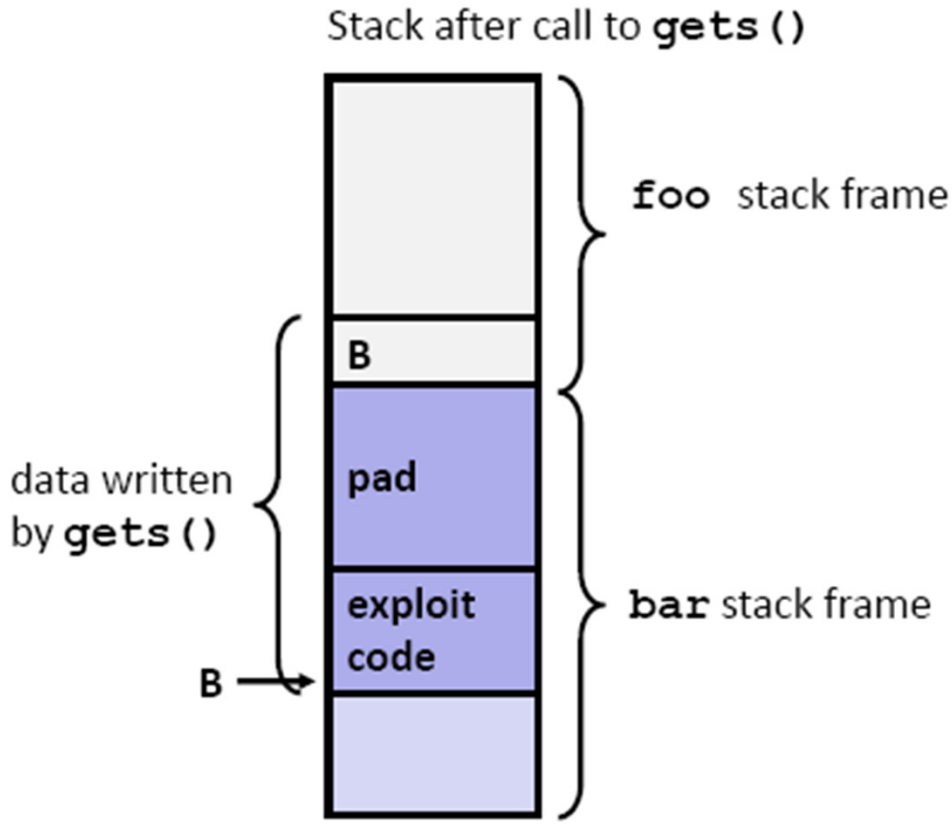

Before call to gets
Stack Frame for main
0xffffc658
08 04 85 f7
ff ff c6 58
0xffffc638
xx xx xx xx
buf
Stack Frame for echo
Input 12345678
Stack Frame for main
0xffffc658
08 04 85 00
43 42 41 39
0xffffc638
38 37 36 35
34 33 32 31
buf
Stack Frame for echo
Return address corrupted

Malicious Use of Buffer Overflow

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

return address A

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



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