

Lecture 23

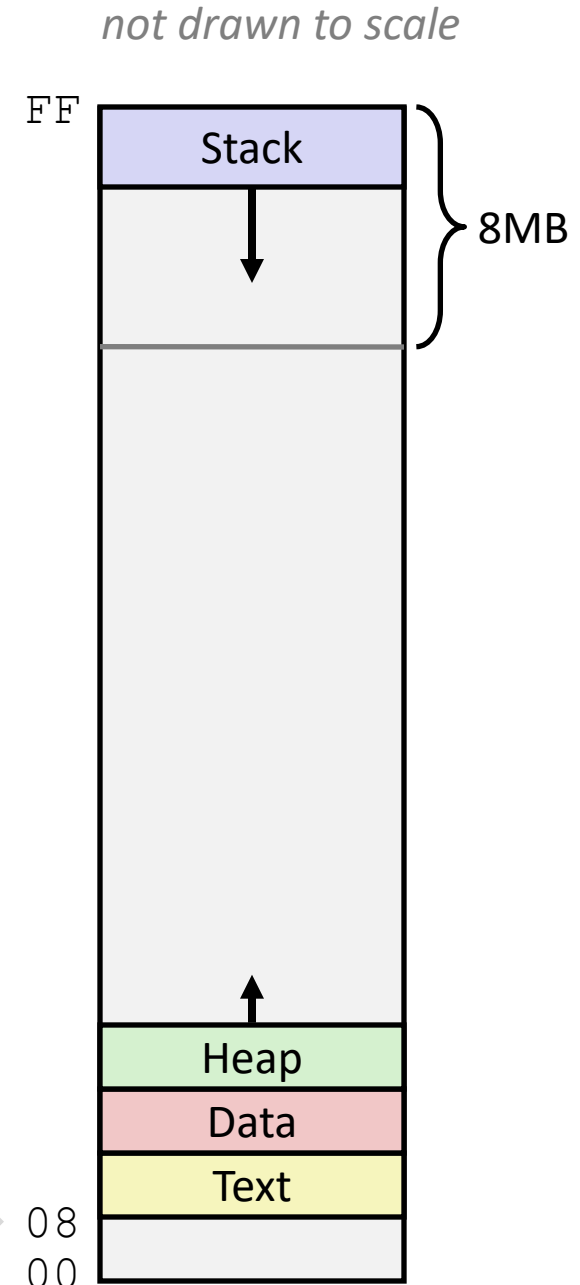
IA-32 Memory Layout

CPSC 275
Introduction to Computer Systems

IA-32 Linux Memory Layout

- Stack
 - Runtime stack (8MB limit on Linux)
 - e. g., local variables
- Heap
 - Dynamically allocated storage
 - When call `malloc`, `calloc`, `new`
- Data
 - Statically allocated data
 - e.g., globals & strings declared in code
- Text
 - Executable machine instructions
 - Read-only (*re-entrant*)

Upper 2 hex digits
= 8 bits of address



Memory Allocation Example

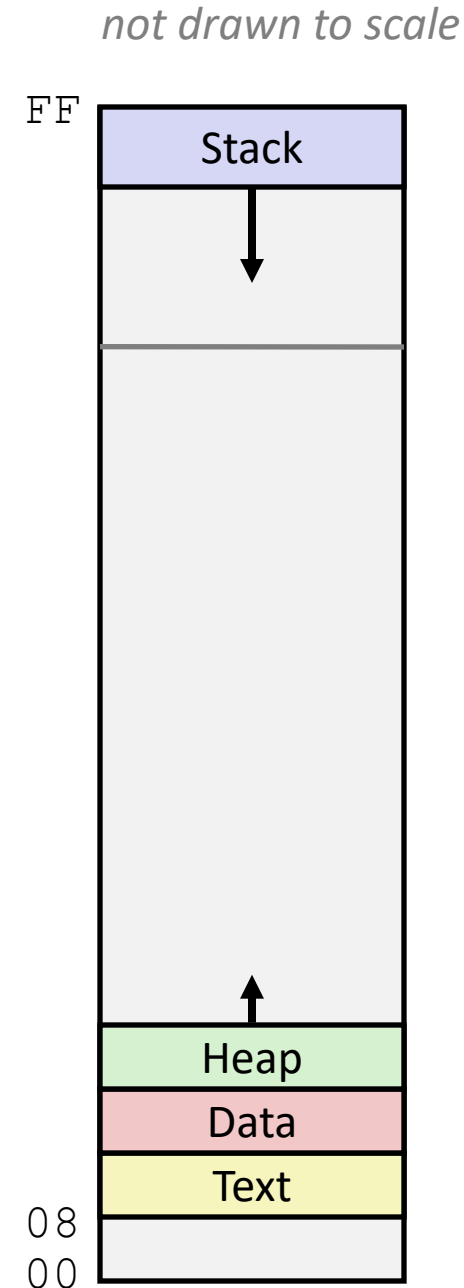
```
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    int i;
    p1 = malloc(1 << 28); /* 256 MB */
    p2 = malloc(1 << 8); /* 256 B */
    p3 = malloc(1 << 28); /* 256 MB */
    p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}
```

Where does everything go?

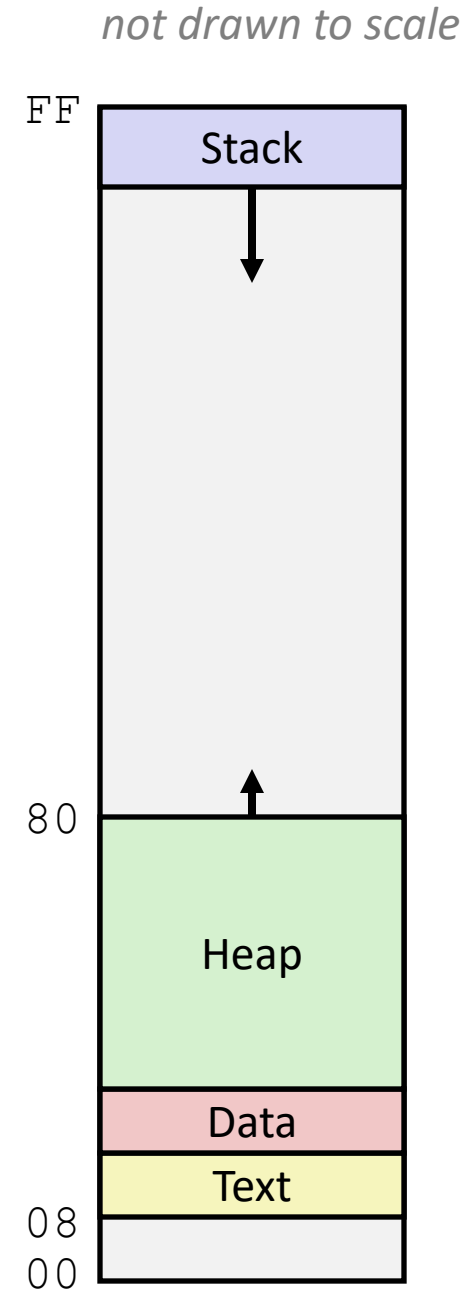


IA-32 Example Addresses

address range $\sim 2^{32}$

&i	0xffffbcd0
p3	0x65586008
p1	0x55585008
p4	0x1904a110
p2	0x1904a008
&p2	0x18049760
&beyond	0x08049744
big_array	0x18049780
huge_array	0x08049760
main()	0x080483c6
useless()	0x08049744
malloc()	0x006be166

malloc() is dynamically linked
address determined at runtime





Buffer Overflow

A Memory Bug Example

```
double fun(int i)
{
    double d = 3.14;
    int a[2];
    a[i] = 1073741824;
    return d;
}
```

```
fun(0)          3.14
fun(1)          3.14
fun(2)          3.14
fun(3)          2.00
fun(4)          3.14
segmentation fault
```

- Result is architecture specific

A Memory Bug Example, cont'd

```
double fun(int i)
{
    double d = 3.14;
    int a[2];
    a[i] = 1073741824;
    return d;
}
```

```
fun(0)          3.14
fun(1)          3.14
fun(2)          3.14
fun(3)          2.00
fun(4)          3.14
segmentation fault
```

Saved State	4	} Location accessed by fun(i)
d7 ... d4	3	
d3 ... d0	2	
a[1]	1	
a[0]	0	

Such problems are a BIG deal

- Generally called a *buffer overflow*
 - when exceeding the memory size allocated for an array
- Why a big deal?
 - Common cause of security vulnerabilities
- Most common form
 - Unchecked lengths on string inputs
 - Particularly for bounded character arrays on the stack
 - sometimes referred to as *stack smashing*

Internet Worm

- November 1988
 - Internet Worm attacks thousands of hosts.
 - How did it happen?
- It was based on *stack buffer overflow* exploits!
 - Many library functions do not check argument sizes.
 - Allows target buffers to overflow.

String Library Code

- Implementation of C Standard Library function `gets()`
- No way to specify limit on number of characters to read
- Similar problems with other library functions
 - `strcpy`, `strcat`: Copy strings of arbitrary length
 - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

```
/* 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;
}
```

Vulnerable Buffer Code

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

```
/* bufdemo.c */  
int main() {  
    echo();  
}
```

```
$ ./bufdemo  
Type a string:123  
123
```

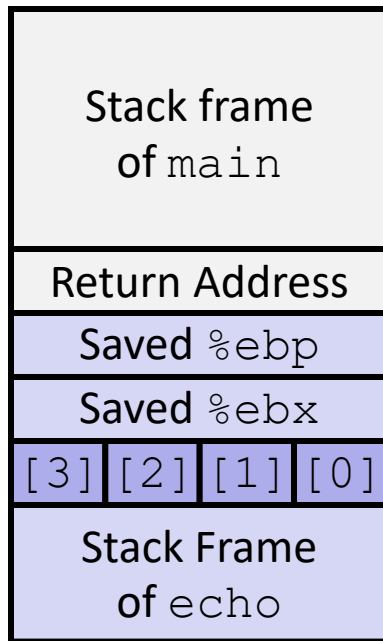
```
$ ./bufdemo  
Type a string:1234567  
Segmentation Fault
```

```
$ ./bufdemo  
Type a string:12345678  
Segmentation Fault
```

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

Buffer Overflow Stack

Before call to gets



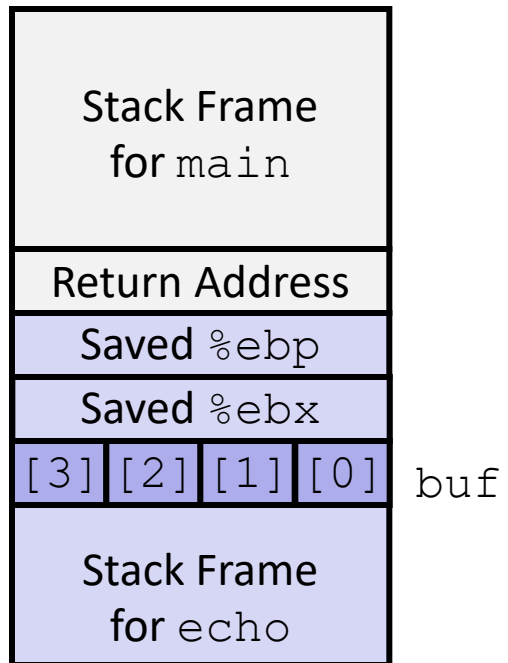
```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

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

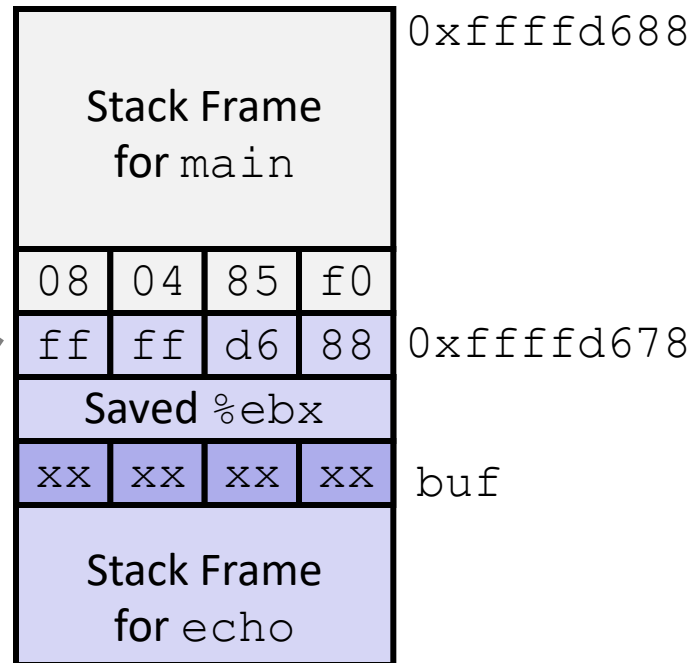
Buffer Overflow Stack Example

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

Before call to gets



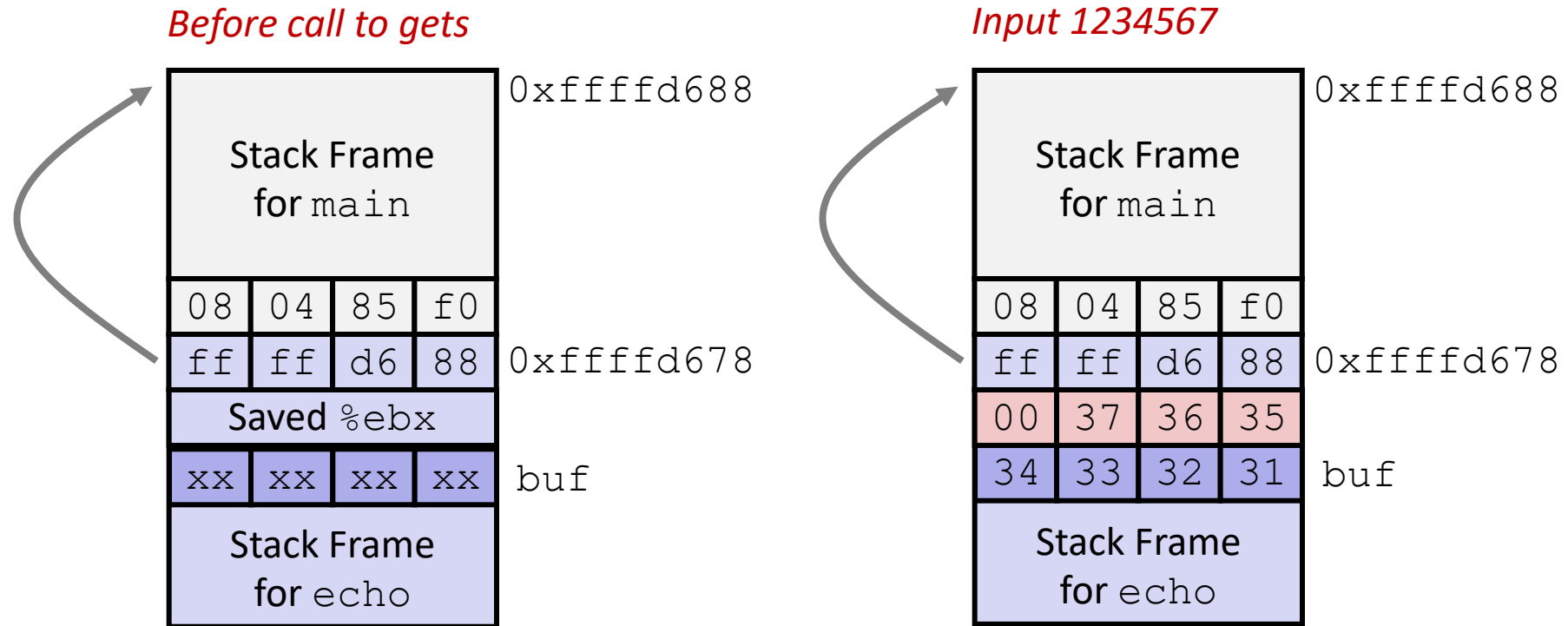
Before call to gets



```
80485eb: e8 d5 ff ff ff
80485f0: c9
```

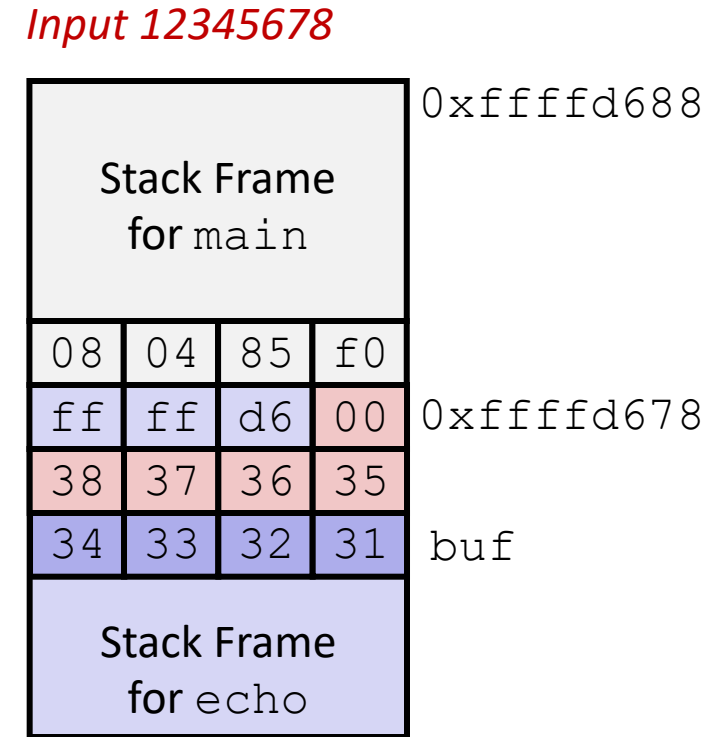
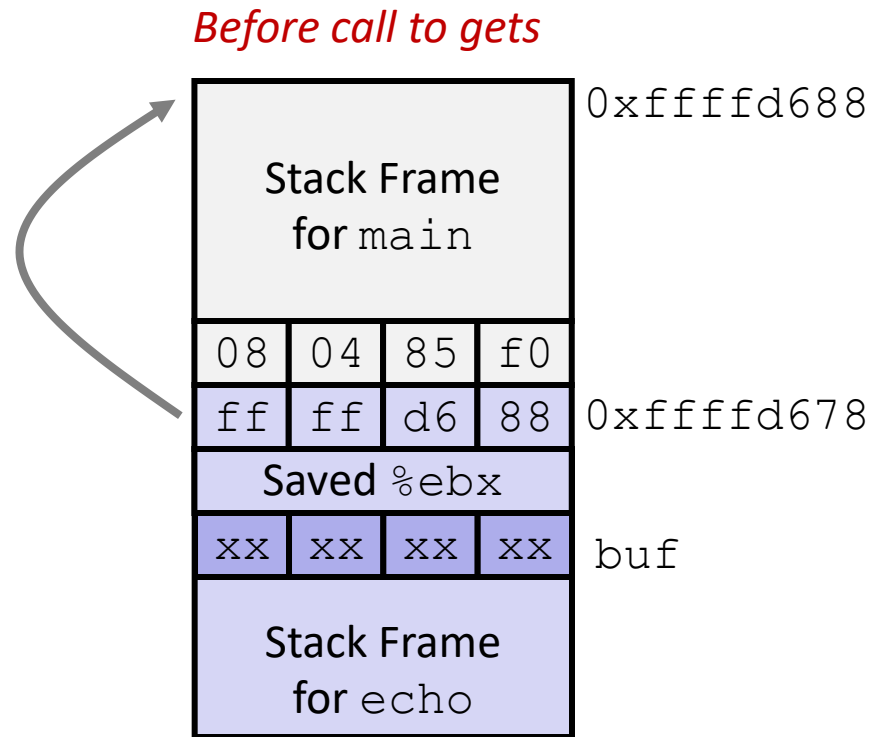
```
call 80485c5 <echo>
leave
```

Buffer Overflow Example #I



Overflow buf and corrupt %ebx

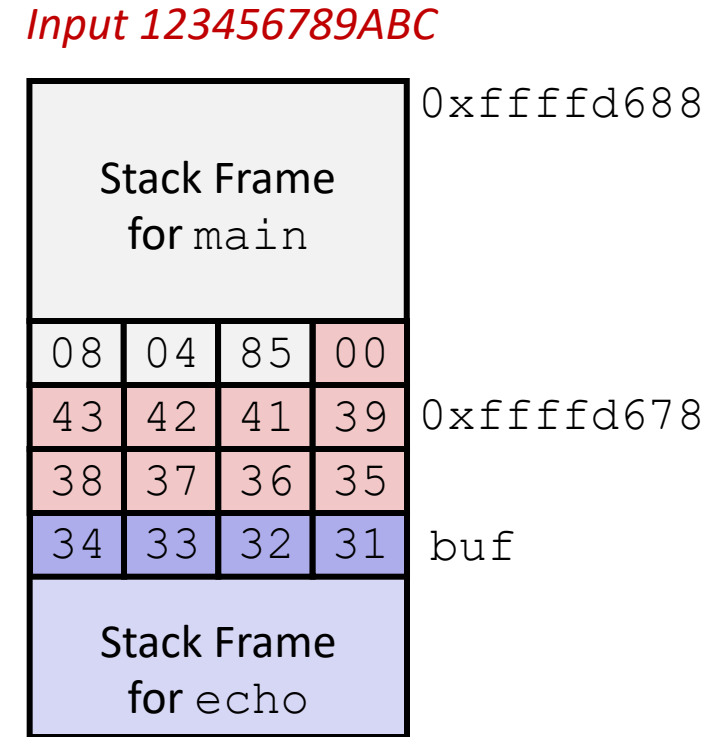
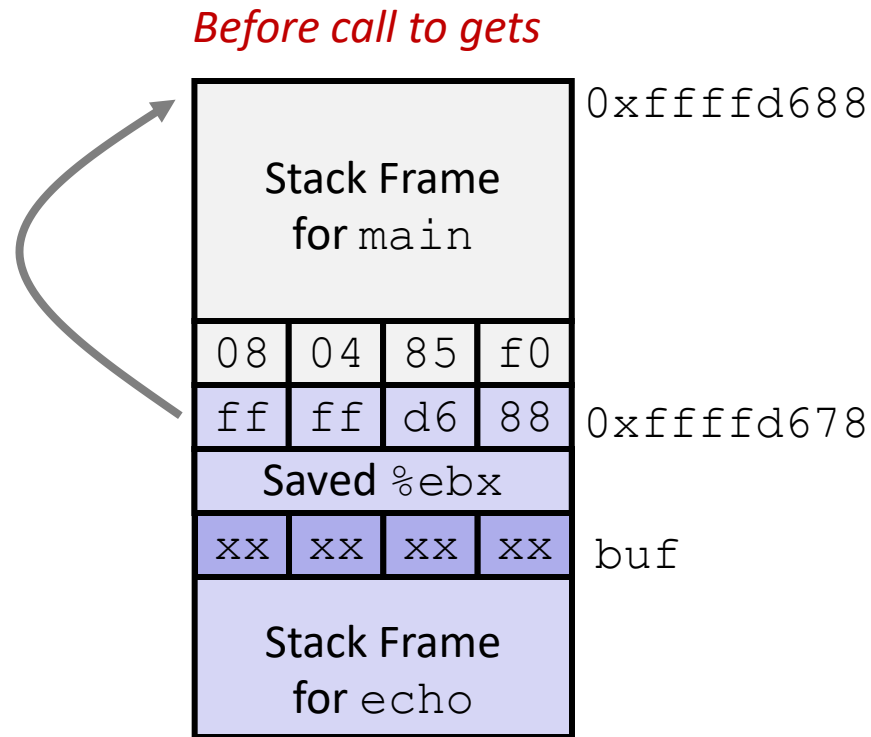
Buffer Overflow Example #2



Base pointer corrupted

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

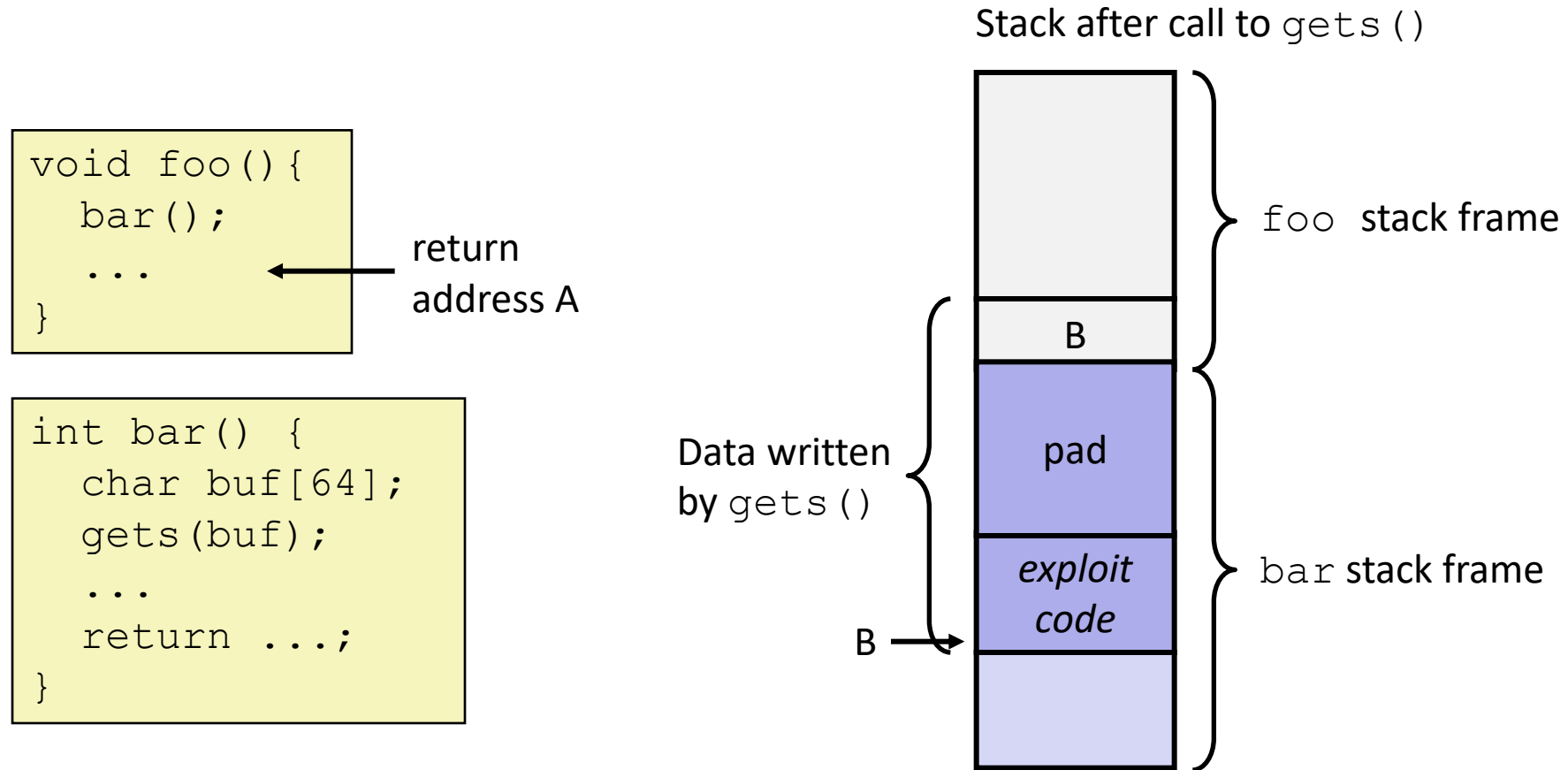

Buffer Overflow Example #3



Return address corrupted

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

Malicious Use of Buffer Overflow



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

Exploits Based on Buffer Overflows

- *Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines*
- Internet worm
 - Early versions of the finger server (**fingerd**) used **gets ()** to read the argument sent by the client:
`finger pyoon@lab.cs.trincoll.edu`
 - Worm attacked **fingerd** server by sending phony argument:
`finger "exploit-code padding new-return-address"`
 - *exploit code*: executed a root shell on the victim machine with a direct TCP connection to the attacker.

Avoiding Overflow Vulnerability

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

- Use library routines that limit string lengths
 - **fgets** instead of **gets**
 - **strncpy** instead of **strcpy**
 - Don't use **scanf** with **%s** conversion specification
 - Use **fgets** to read the string
 - Or use **%ns** where **n** is a suitable integer

System-Level Protections

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

```
$ gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xfffffc638

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

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

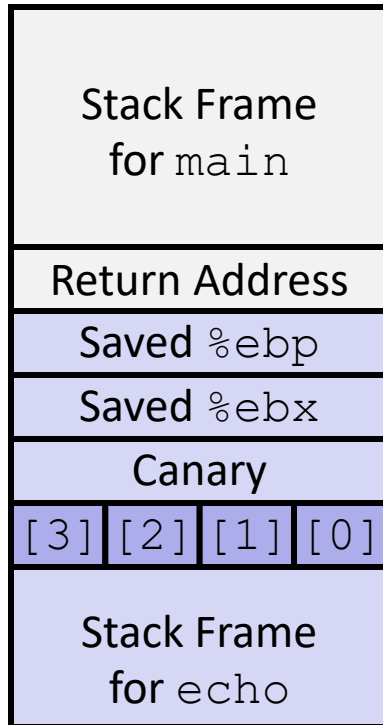
```
int main()
{
    int local;
    printf("local at %p\n", &local);
    return 0;
}
```

Stack Canaries

- Idea
 - Place special value (“canary”) on stack just beyond buffer
 - Check for corruption before exiting function
- gcc implementation
 - **fstack-protector**

Setting Up Canary

Before call to gets

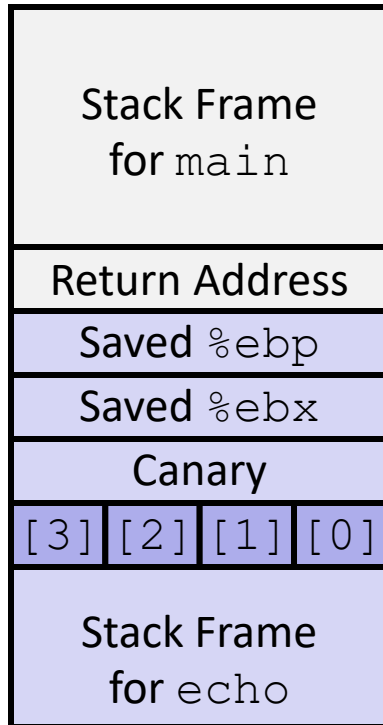


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

Checking Canary

Before call to gets

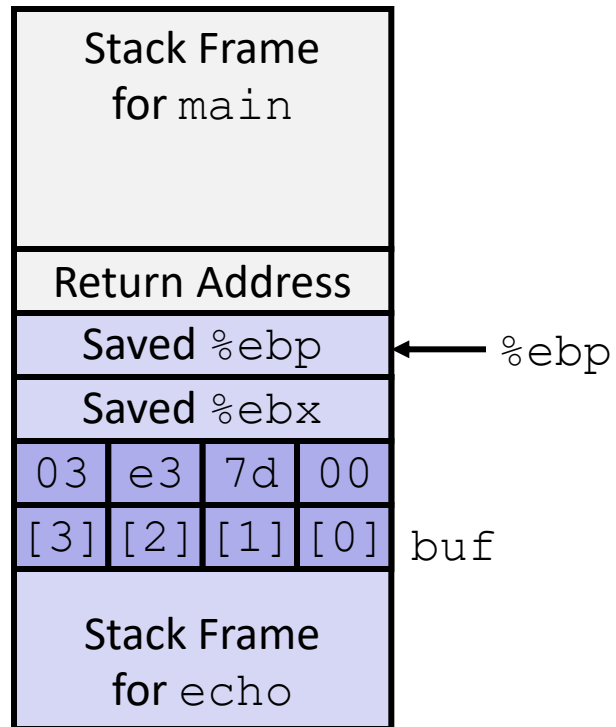


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

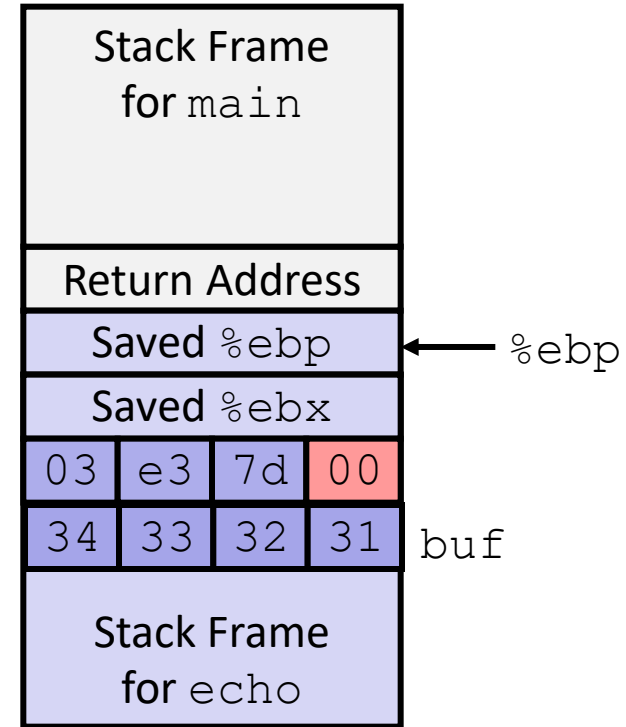
```
echo:
    . . .
    movl    -8(%ebp), %eax    # Retrieve from stack
    xorl    %gs:20, %eax     # Compare with Canary
    je      .L24             # Same: skip ahead
    call    __stack_chk_fail # ERROR
.L24:
    . . .
```


Canary Example

Before call to gets



Input 1234



Worms and Viruses

- Worm: A program that
 - Can run by itself
 - Can propagate a fully working version of itself to other computers
- Virus: Code that
 - Add itself to other programs
 - Cannot run independently
- Both are (usually) designed to spread among computers and to wreak havoc

