Machine-Level Programming V: Advanced Topics

15-213 / 18-213: Introduction to Computer Systems
9th Lecture, Sep. 23, 2014

Instructors:

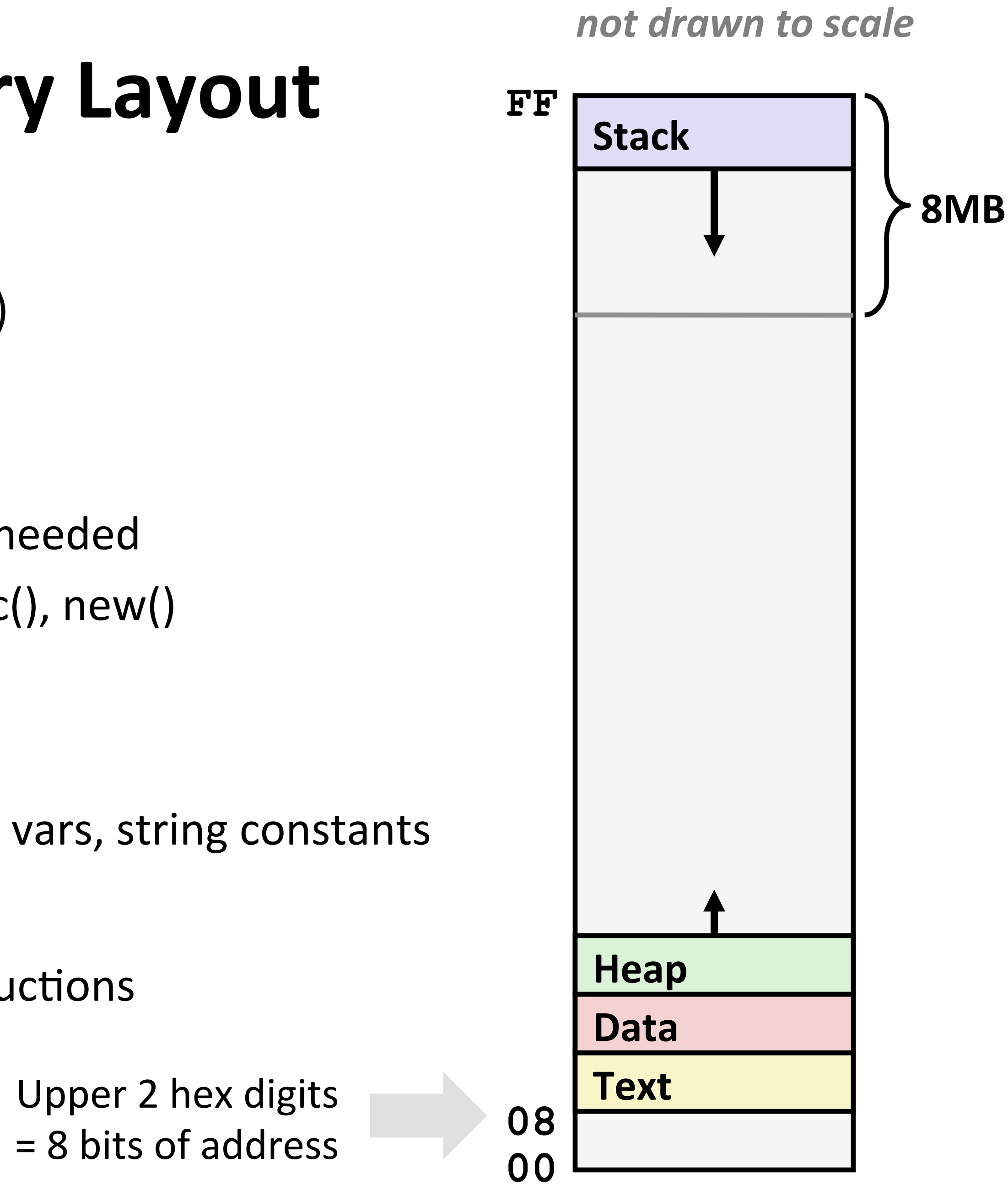
Greg Ganger, Greg Kesden, and Dave O'Hallaron

Today

- **Memory Layout**
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions

IA32 Linux Memory Layout

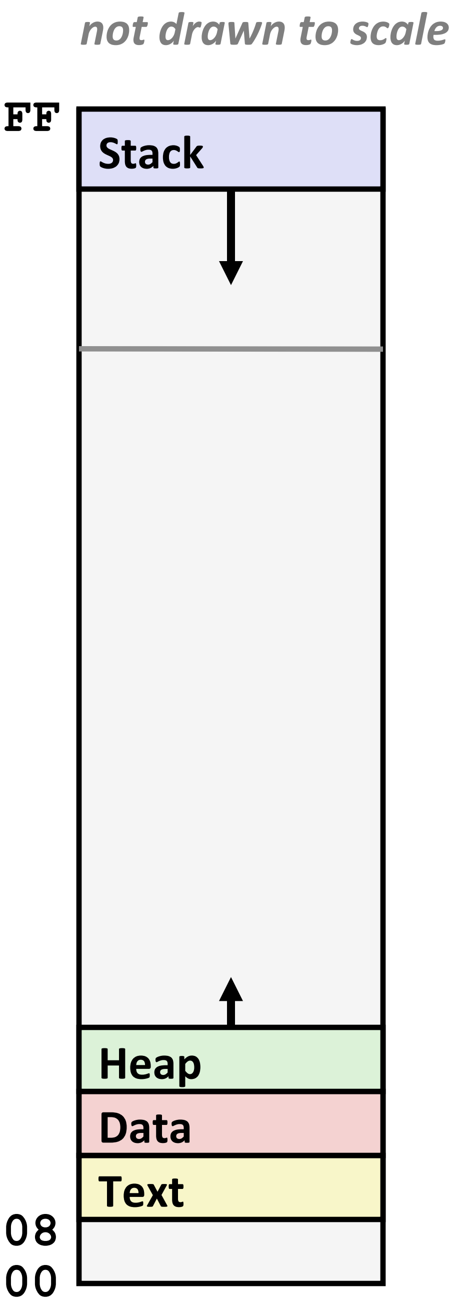
- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text**
  - Executable machine instructions
  - Read-only



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()
{
 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?



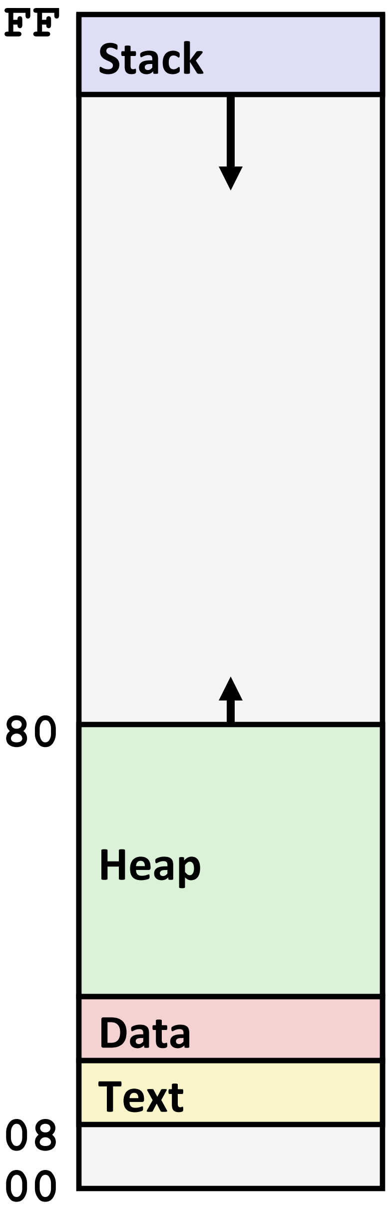
IA32 Example Addresses

address range ~2^{32}

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

`malloc()` is dynamically allocated address determined at runtime

not drawn to scale



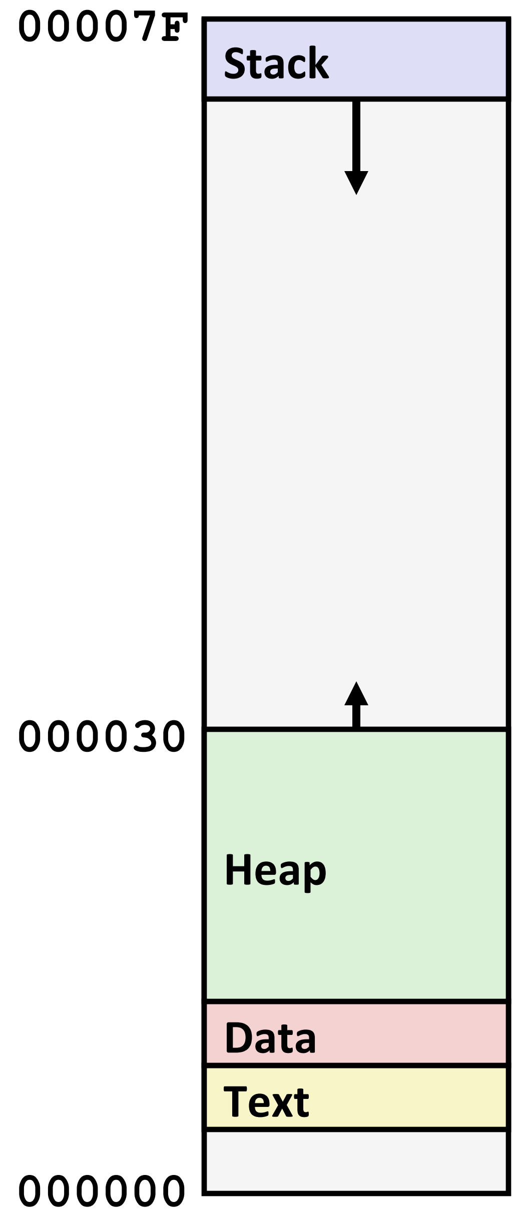
x86-64 Example Addresses

address range ~2^{47}

`$rsp`	`0x00007ffffff8d1f8`
`p3`	`0x00002aaabaadd010`
`p1`	`0x00002aaaaaadc010`
`p4`	`0x0000000011501120`
`p2`	`0x0000000011501010`
`&p2`	`0x0000000010500a60`
`&beyond`	`0x0000000000500a44`
`big_array`	`0x0000000010500a80`
`huge_array`	`0x0000000000500a50`
`main()`	`0x0000000000400510`
`useless()`	`0x0000000000400500`
final `malloc()`	`0x00000386ae6a170`

`malloc()` is dynamically allocated address determined at runtime

not drawn to scale



Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
- Unions

Remember this Memory Bug Example?

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  ➙    3.14
fun(1)  ➙    3.14
fun(2)  ➙    3.1399998664856
fun(3)  ➙    2.00000061035156
fun(4)  ➙    3.14, then segmentation fault
```

- Result is architecture specific

Remember this Memory Bug Example?

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

```
fun(0)  ➙   3.14
fun(1)  ➙   3.14
fun(2)  ➙   3.1399998664856
fun(3)  ➙   2.00000061035156
fun(4)  ➙   3.14, then segmentation fault
```

Explanation:

Saved State	4
d7 ... d4	3
d3 ... d0	2
a[1]	1
a[0]	0

Location accessed by `fun(i)`

Such problems are a BIG deal

- **Generally called a "buffer overflow"**
  - when exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

String Library Code

- **Implementation of Unix function `gets()`**

```
/* 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;
}
```

  - 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

Vulnerable Buffer Code

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

← btw, how big is big enough?

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

```
unix>./bufdemo
Type a string:0123456789a
0123456789a
```

```
unix>./bufdemo
Type a string:0123456789ab
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

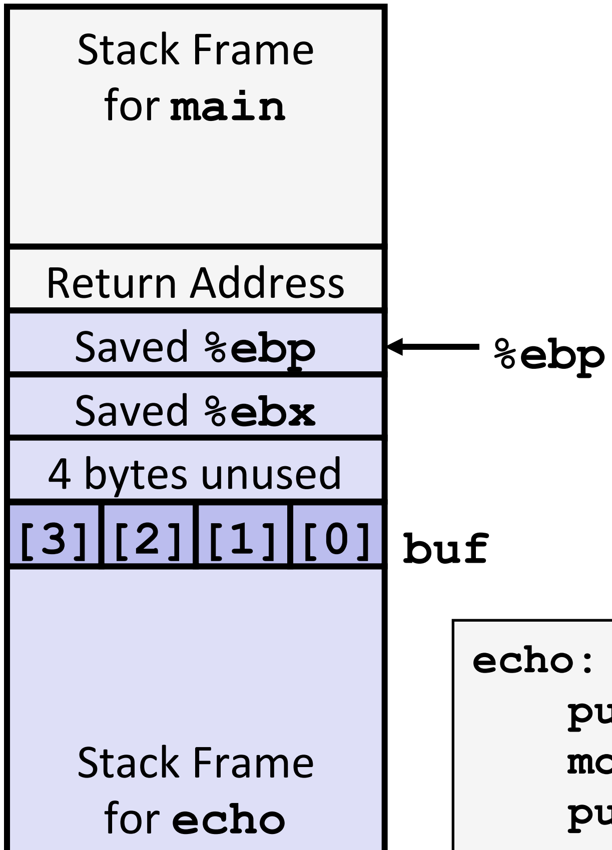
```
080485c3 <echo>:
80485c3:  55                      push   %ebp
80485c4:  89 e5                   mov    %esp,%ebp
80485c6:  53                      push   %ebx
80485c7:  83 ec 24                sub    $0x24,%esp
80485ca:  8d 5d f4                lea    -0xc(%ebp),%ebx
80485cd:  89 1c 24                mov    %ebx,(%esp)
80485d0:  e8 9e ff ff ff          call   8048573 <gets>
80485d5:  89 1c 24                mov    %ebx,(%esp)
80485d8:  e8 2f fe ff ff          call   804840c <puts@plt>
80485dd:  83 c4 24                add    $0x24,%esp
80485e0:  5b                      pop    %ebx
80485e1:  5d                      pop    %ebp
80485e2:  c3                      ret
```

call_echo:

```
 . . .
80485e9:  e8 d5 ff ff ff          call   80485c3 <echo>
80485ee:  c9                      leave
80485ef:  c3                      ret
```

Buffer Overflow Stack

Before call to gets



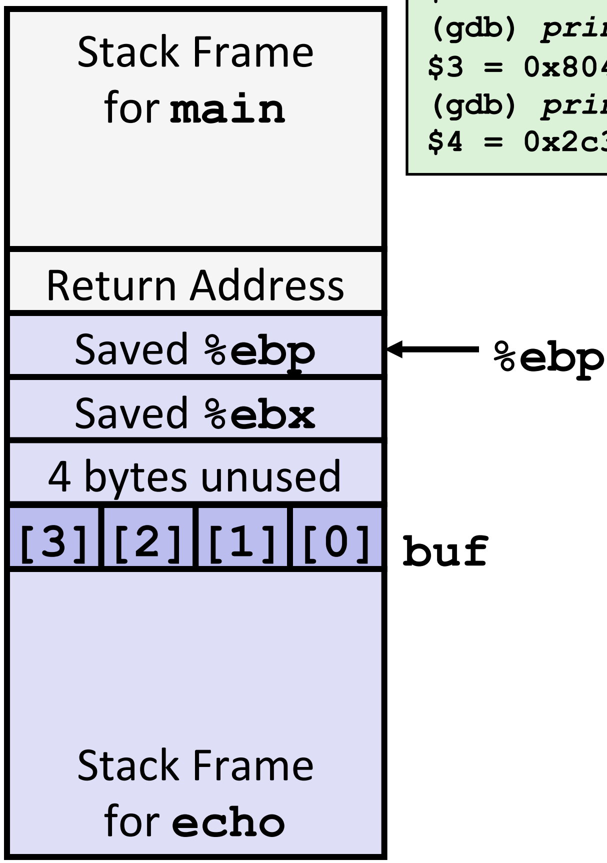

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp            # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx            # Save %ebx
    subl  $36, %esp       # Allocate stack space
    leal  -12(%ebp),%ebx  # Compute buf as %ebp-12
    movl  %ebx, (%esp)    # Push buf on stack
    call  gets            # Call gets
    . . .
```

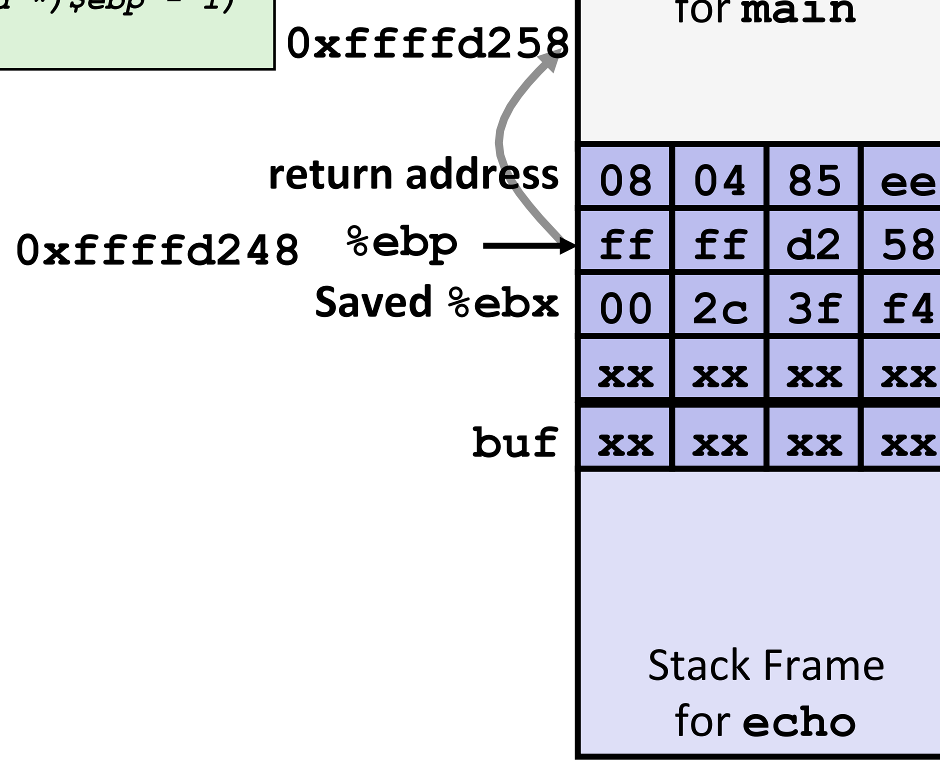
Buffer Overflow Stack Example

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd248
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd258
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485ee
(gdb) print /x *((unsigned *)$ebp - 1)
$4 = 0x2c3ff4
```

Before call to gets



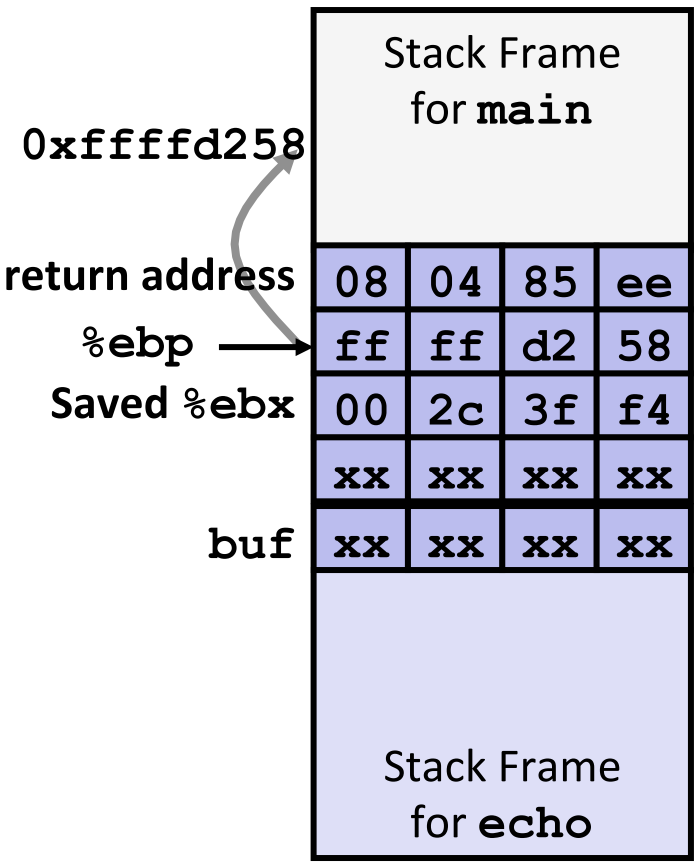
Before call to gets



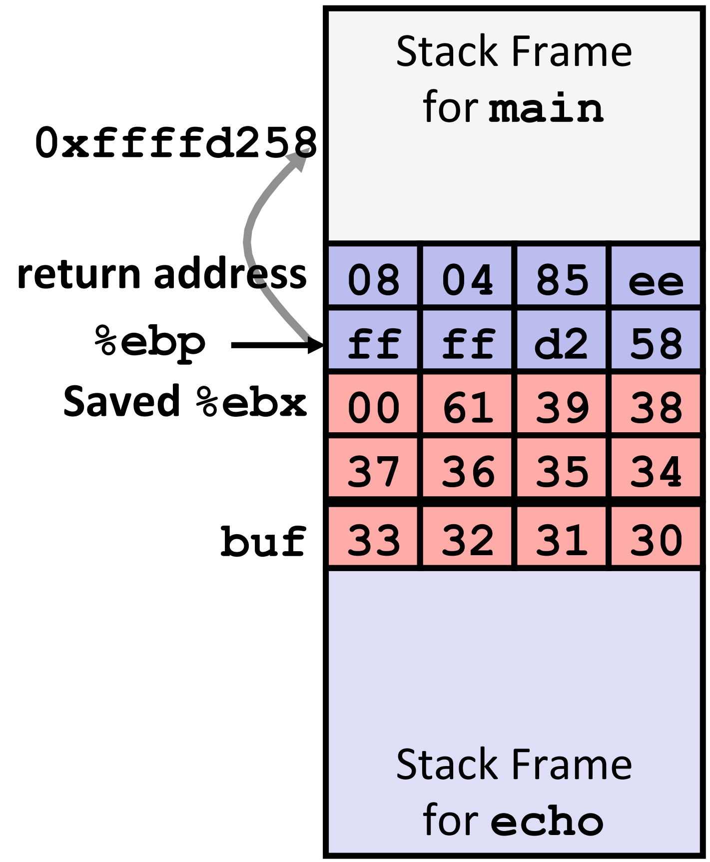

```
80485e9:   e8 d5 ff ff ff          call    80485c3 <echo>
80485ee:   c9                      leave
```

Buffer Overflow Example #1

Before call to gets



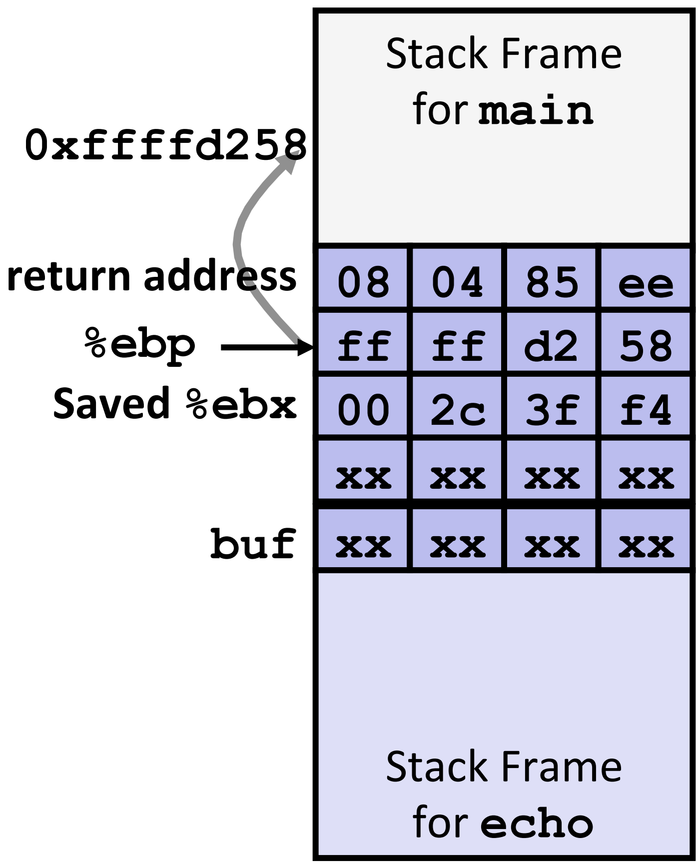
Input 0123456789a



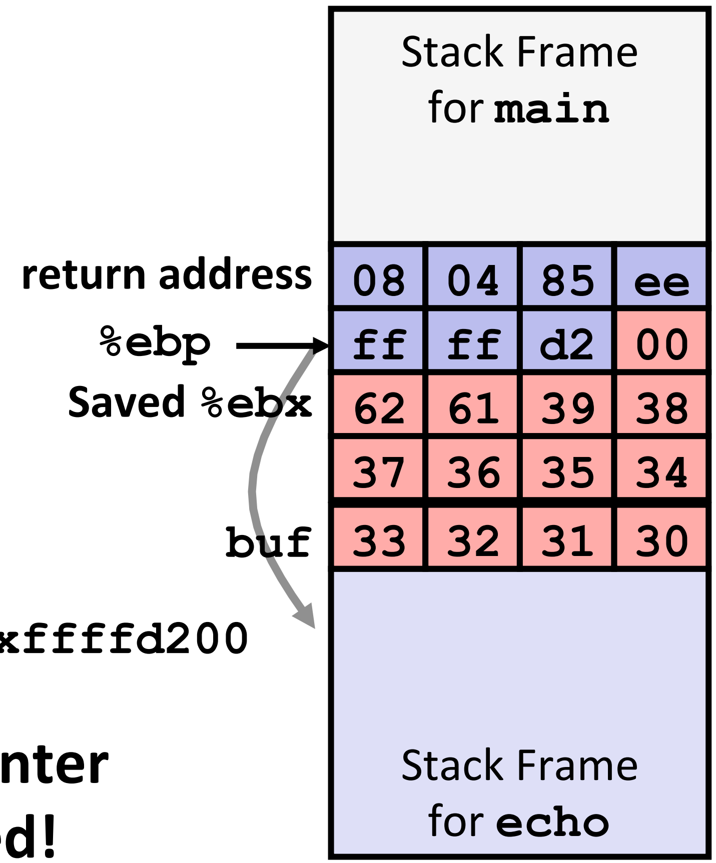
Overflow buf, and corrupt %ebx, but no adverse effects

Buffer Overflow Example #2

Before call to gets



Input 0123456789ab

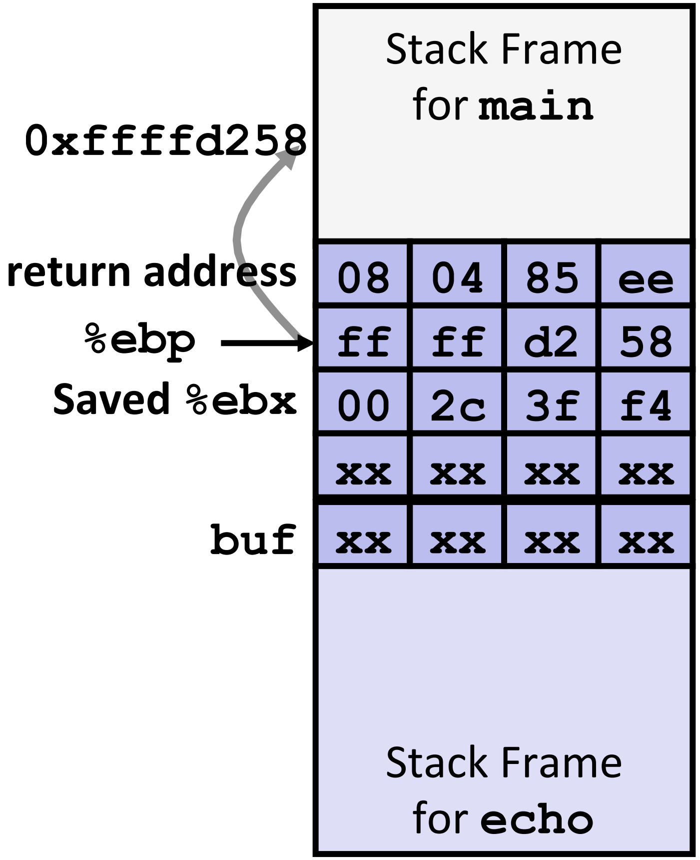


Base pointer corrupted!

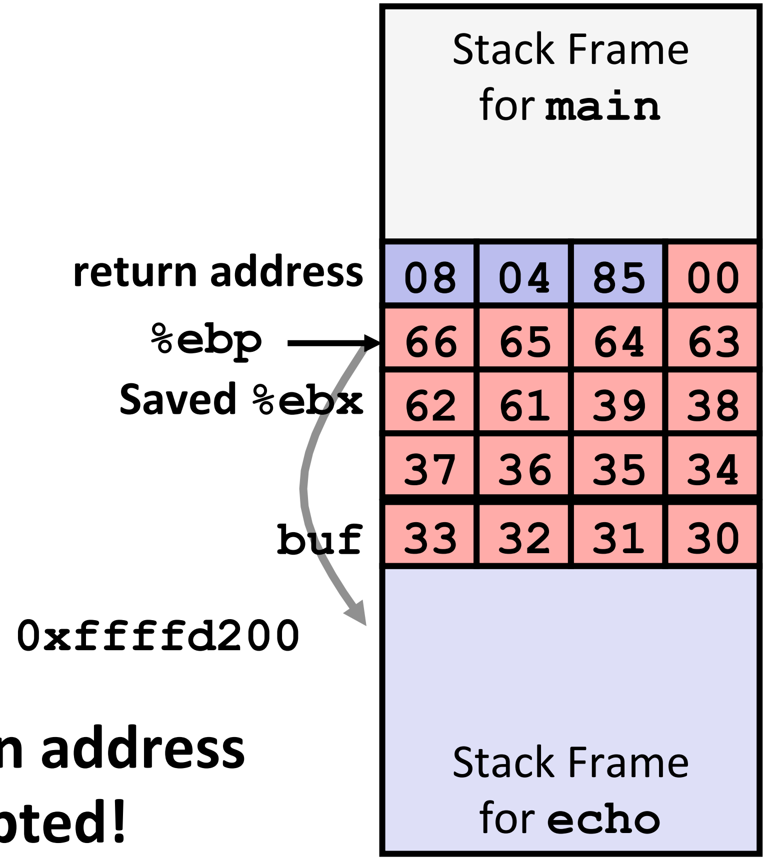
```
. . .
80485e9:   e8 d5 ff ff ff         call   80485c3 <echo>
80485ee:   c9                     leave  # Set %ebp to bad value
80485ef:   c3                     ret
```

Buffer Overflow Example #3

Before call to gets



Input 0123456789abcdef

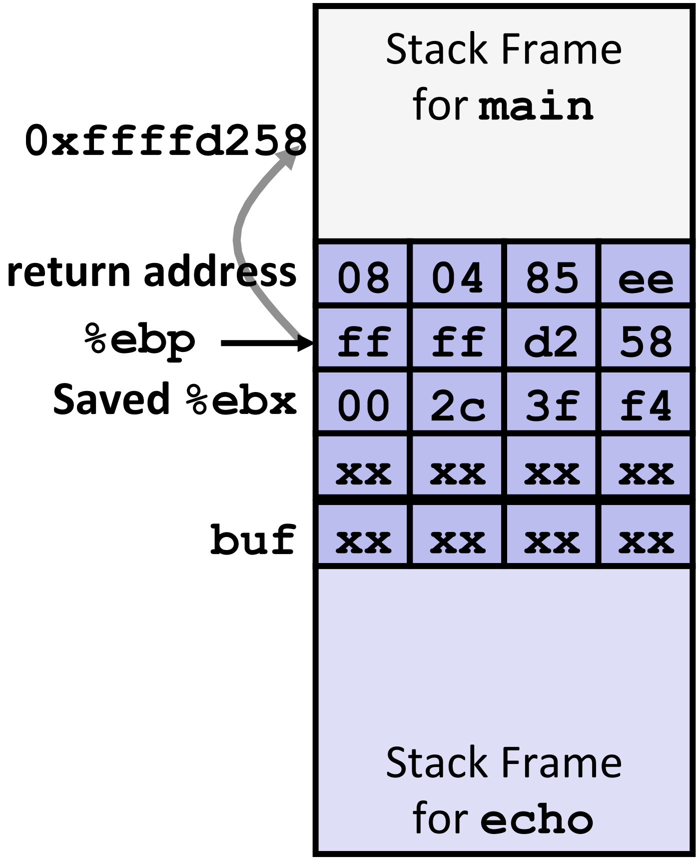


Return address corrupted!

```
. . .
80485e9:   e8 d5 ff ff ff          call   80485c3 <echo>
80485ee:   c9                      leave  # Desired return point
```

Buffer Overflow Example #4

Before call to gets



- **Can we trick program into calling a different function?**

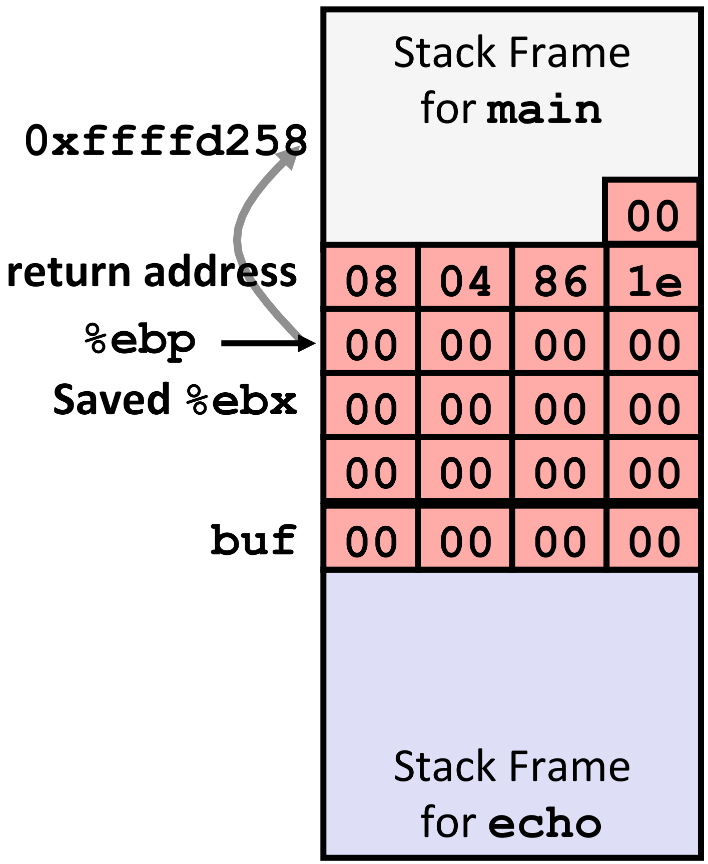
```
void gotcha() {
    printf(
"This function should not get called!\n”
            );
}
```

- **Idea: Alter return address on stack**

```
0804861e <gotcha>:
 804861e: 55   push   %ebp
 . . .
```

Buffer Overflow Example #4

Before call to gets



- **Alter return address on stack**

```
0804861e <gotcha>:
 804861e: 55   push   %ebp
 . . .
```

- **Exploit string:**

  00 00 00 00 00 00 00 00 (8X)

  00 00 00 00 00 00 00 00 (8X)

  1e 86 04 08 (Little Endian)

- **Must supply as raw bytes**
  - E.g., via tool hex2raw
    - See Buffer Lab writeup
  - Note: must know address of gotcha()
    - which is already part of program

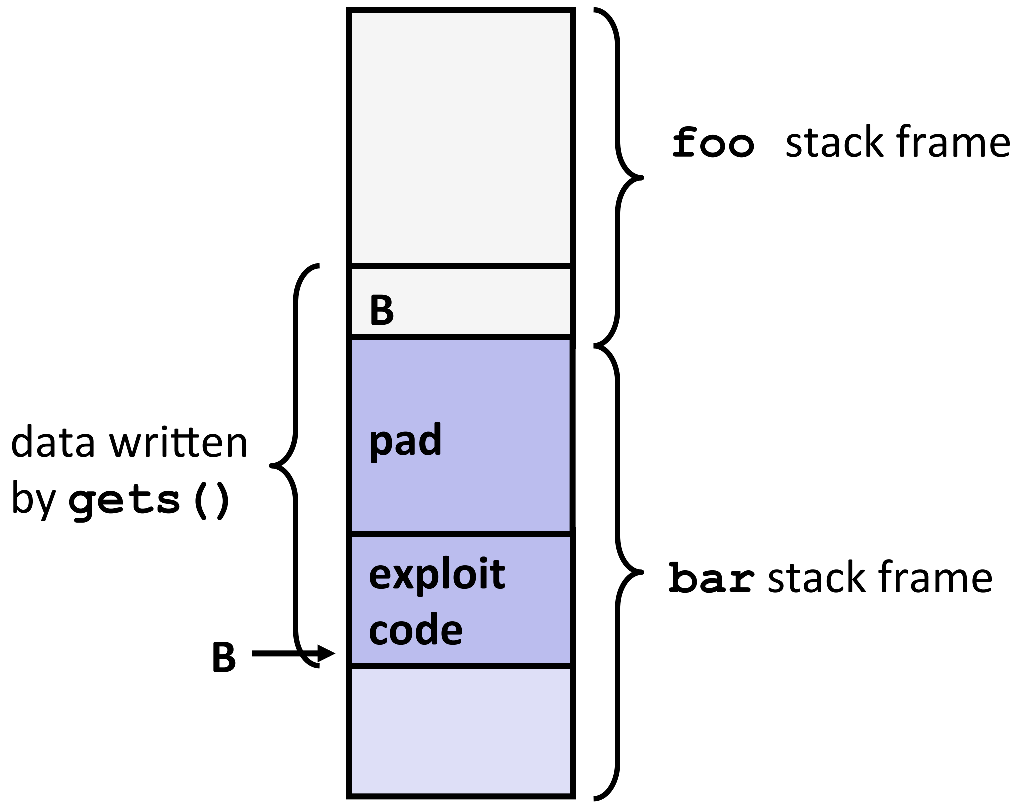
Worse: can insert new code and take over

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

return address A

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

Stack after call to `gets()`



- Have input string contain 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

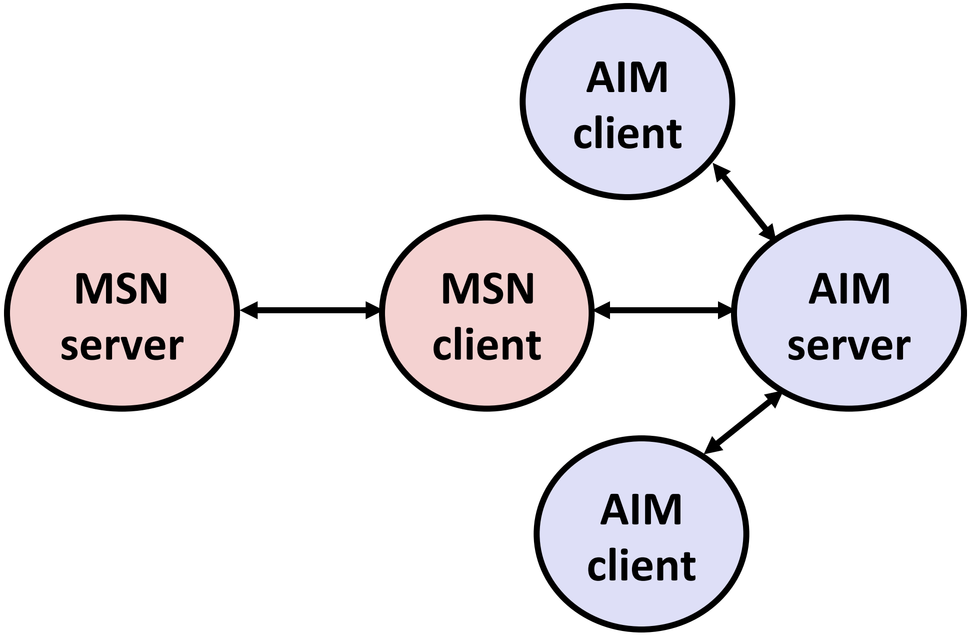
- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **Distressingly common in real progams**
  - Programmers keep making the same mistakes ☹
- **Examples across the decades**
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more
- **You will learn how this is done in buflab**
  - Hopefully to convince you to never leave such holes in your programs!!

Example: the original Internet worm (1988)

- **Exploited a few vulnerabilities to spread**
  - Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.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.
- **Once on a machine, scanned for other machines to attack**
  - invaded ~6000 computers in hours (10% of the Internet ☺)
    - see June 1989 article in *Comm. of the ACM*
  - the young author of the worm was prosecuted…
  - and CERT was formed… still homed at CMU

Example 2: IM War

- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



IM War (cont.)

- **August 1999**
  - Mysteriously, Messenger clients can no longer access AIM servers
  - Microsoft and AOL begin the IM war:
    - AOL changes server to disallow Messenger clients
    - Microsoft makes changes to clients to defeat AOL changes
    - At least 13 such skirmishes
  - What was really happening?
    - AOL had discovered a buffer overflow bug in their own AIM clients
    - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
    - When Microsoft changed code to match signature, AOL changed signature location

```
Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>
Subject: AOL exploiting buffer overrun bug in their own software!
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you
might find interesting because you are an Internet security expert with
experience in this area. I have also tried to contact AOL but received
no response.

I am a developer who has been working on a revolutionary new instant
messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself
this might not be the end of the world, as MS surely has had its share.
But AOL is now *exploiting their own buffer overrun bug* to help in
its efforts to block MS Instant Messenger.
....
Since you have significant credibility with the press I hope that you
can use this information to help inform people that behind AOL's
friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com
```

It was later determined that this email originated from within Microsoft!

Aside: 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**
  - Adds itself to other programs
  - Does not run independently
- **Both are (usually) designed to spread among computers and to wreak havoc**

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use "stack canaries"
- Lets talk about each…

1. Avoid Overflow Vulnerabilities in Code (!)

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

- **For example, 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

2. System-Level Protections can help

- **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
  - Currently disabled on shark machines

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

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

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

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

- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - X86-64 added explicit "execute" permission

3. Stack Canaries can help

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

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

```
unix>./bufdemo-protected
Type a string:1234
*** stack smashing detected ***
```

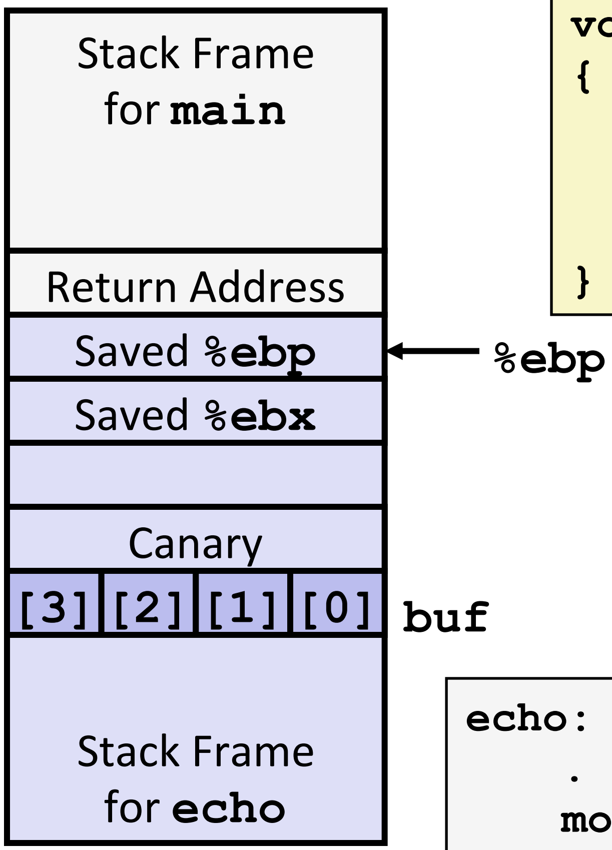
Protected Buffer Disassembly

echo:

```
804864b:   55                     push   %ebp
804864c:   89 e5                  mov    %esp,%ebp
804864e:   53                     push   %ebx
804864f:   83 ec 24               sub    $0x24,%esp
8048652:   65 a1 14 00 00 00      mov    %gs:0x14,%eax
8048658:   89 45 f4               mov    %eax,-0xc(%ebp)
804865b:   31 c0                  xor    %eax,%eax
804865d:   8d 5d f0               lea    -0x10(%ebp),%ebx
8048660:   89 1c 24               mov    %ebx,(%esp)
8048663:   e8 77 ff ff ff         call   80485df <gets>
8048668:   89 1c 24               mov    %ebx,(%esp)
804866b:   e8 f0 fd ff ff         call   8048460 <puts@plt>
8048670:   8b 45 f4               mov    -0xc(%ebp),%eax
8048673:   65 33 05 14 00 00 00   xor    %gs:0x14,%eax
804867a:   74 05                  je     8048681 <echo+0x36>
804867c:   e8 cf fd ff ff         call   8048450 <...fail...>
8048681:   83 c4 24               add    $0x24,%esp
8048684:   5b                     pop    %ebx
8048685:   5d                     pop    %ebp
8048686:   c3                     ret
```

Setting Up Canary

Before call to gets

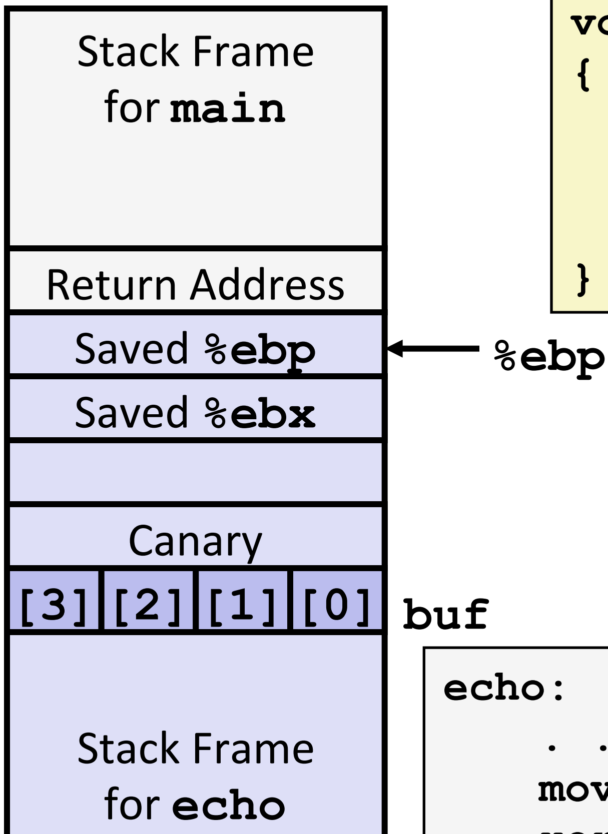



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movl    %gs:20, %eax        # Get canary
    movl    %eax, -12(%ebp)     # Put on stack
    xorl    %eax, %eax          # Erase canary
    . . .
```

Checking Canary

Before call to gets

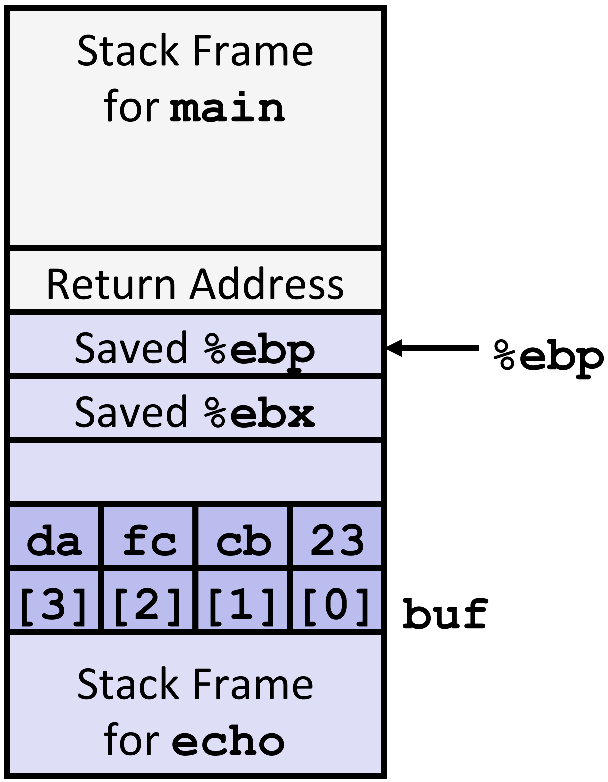



```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

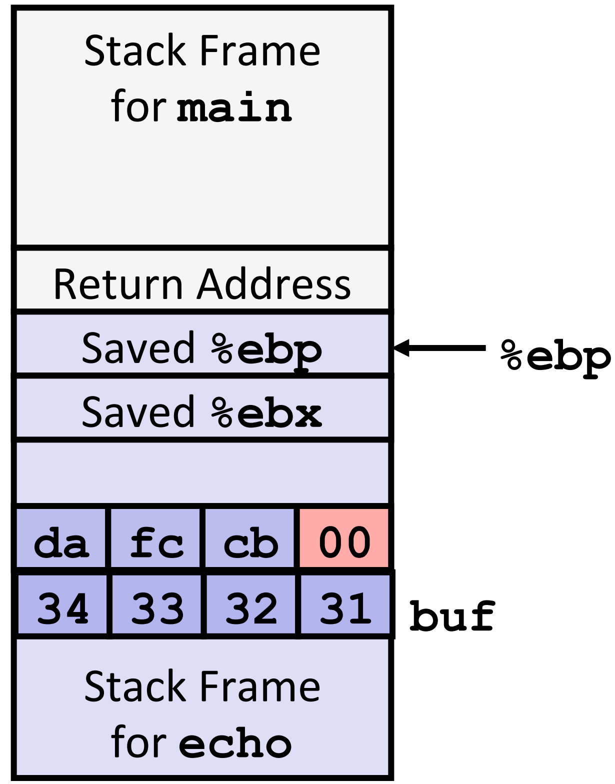
```
echo:
    . . .
    movl    -12(%ebp), %eax     # Retrieve from stack
    xorl    %gs:20, %eax        # Compare with Canary
    je      .L24                # Same: skip ahead
    call    __stack_chk_fail    # ERROR
.L24:
    . . .
```

Canary Example

Before call to gets



Input 1234




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 3)
$1 = dafccb23
```

Canary corrupted

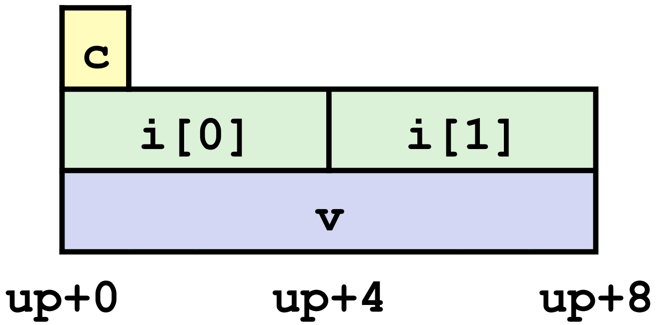
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- **Unions**

Union Allocation

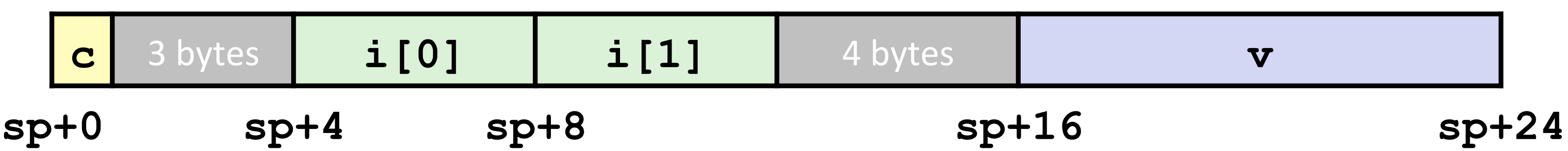
- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```



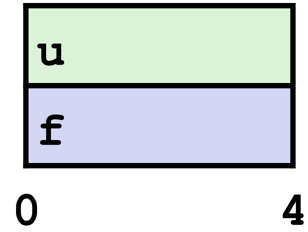

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

(Windows or x86-64)



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float) u` ?

Same as `(unsigned) f` ?

Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- **Big Endian**
  - Most significant byte has lowest address
  - Sparc
- **Little Endian**
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- **Bi Endian**
  - Can be configured either way
  - ARM

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

64-bit

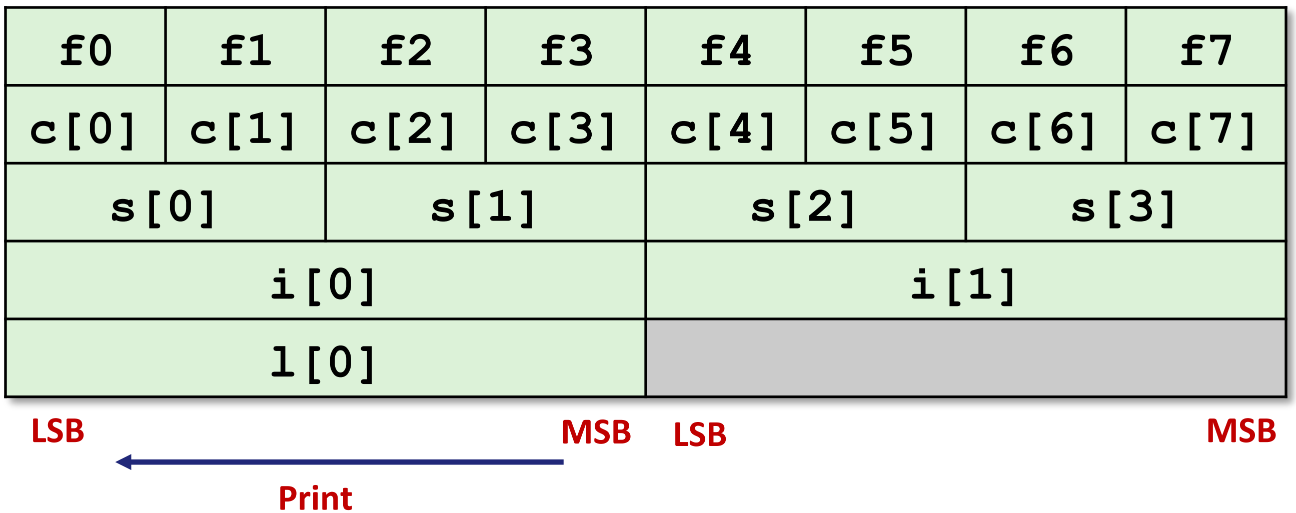
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==  [0x%x,0x%x,0x%x,0x%x,
0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian



Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

MSB LSB MSB LSB

Print

Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

LSB ← Print — MSB

Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
    - IA32 Linux unusual in only requiring 4-byte alignment for 8-byte data
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Way to circumvent type system