CSSE132
Introduction to Computer Systems

18 : Alignment, Pointers, Bounds

April 9, 2013

Today

- **Structures**
  - Alignment
- **Unions**
- **Pointers**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

Alignment Principles

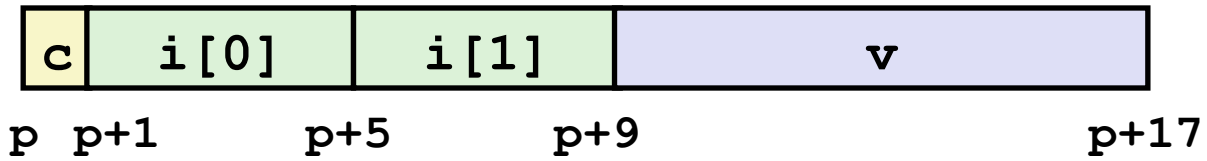
- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (IA32)

- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, `char *`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, …**
  - Windows (and most other OS's & instruction sets):
    - lowest 3 bits of address must be 000_2
  - Linux:
    - lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type
- **12 bytes: `long double`**
  - Windows, Linux:
    - lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type

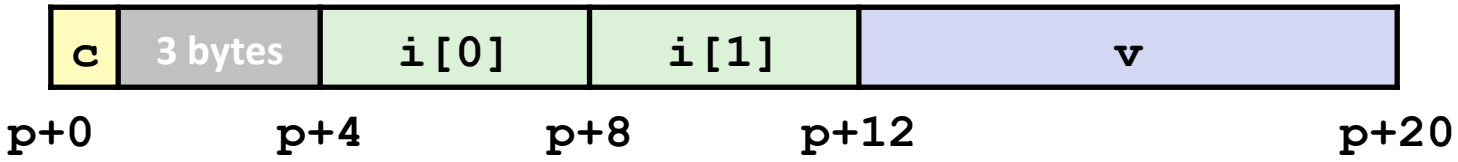
Structures & Alignment

- **Unaligned Data**




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

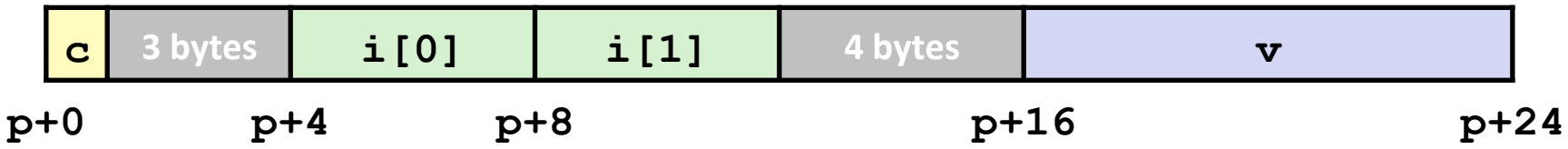
- **IA32 Linux Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - In Linux, `double` treated like a 4-byte data type



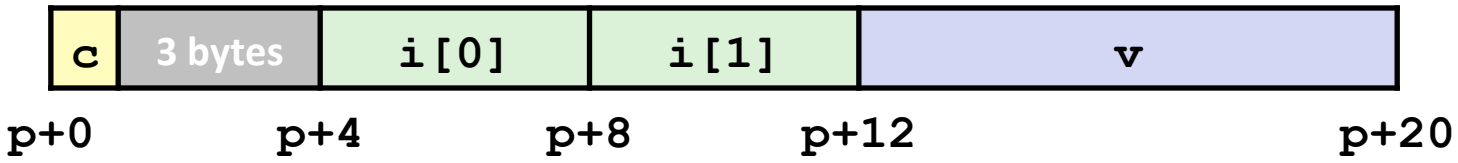
Different Alignment Conventions

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

- **x86-64 or IA32 Windows:**
  - K = 8, due to **`double`** element



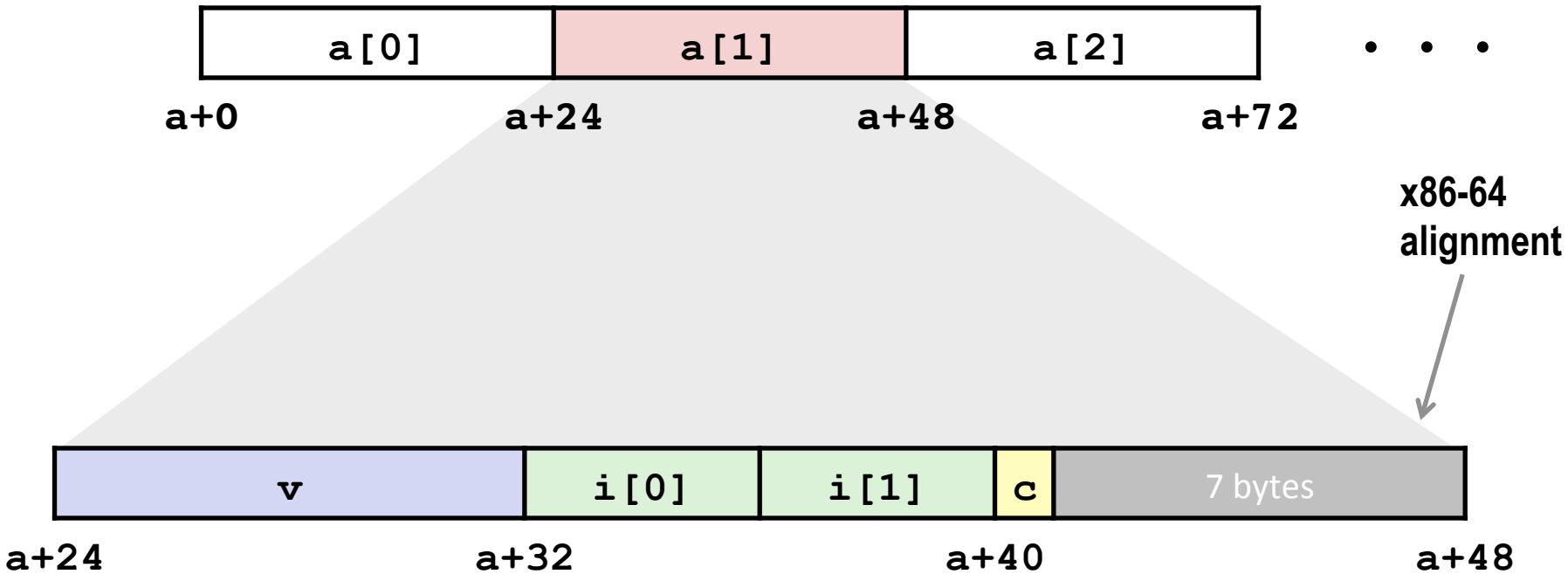
- **IA32 Linux**
  - K = 4; `double` treated like a 4-byte data type



Arrays of Structures

- Overall structure length multiple of largest K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



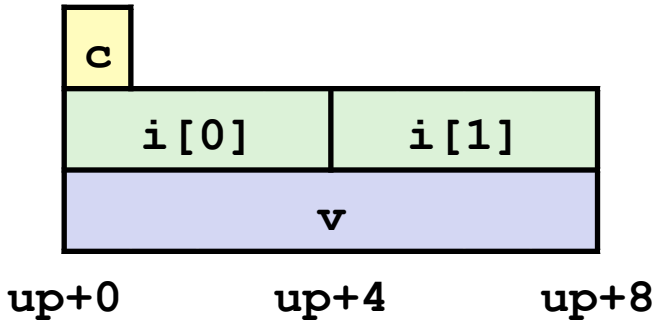
Today

- Structures
  - Alignment
- **Unions**
- Pointers
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection

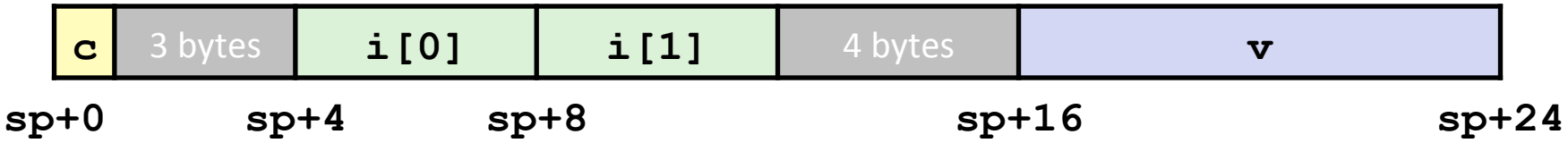
Union Allocation

- 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;
```



Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - 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

Today

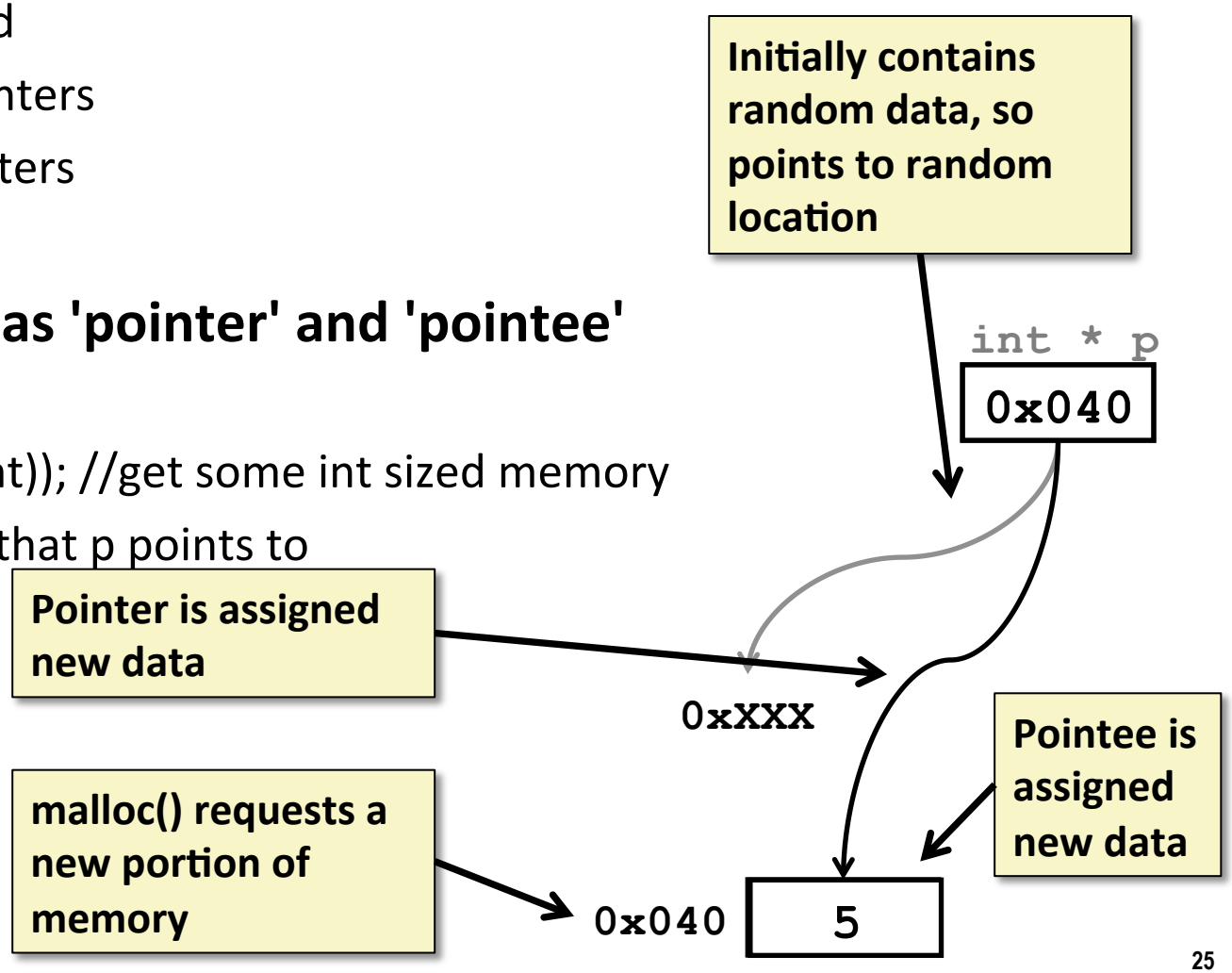
- **Structures**
  - Alignment
- **Unions**
- **Pointers**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

Pointers

- **Data type that represents memory address**
  - Often word sized
  - IA32 : 32 bit pointers
  - x64 : 64 bit pointers

- **Often described as 'pointer' and 'pointee'**

```
int * p;  //pointer
p = malloc(sizeof(int)); //get some int sized memory
*p = 5; //set value that p points to
```



Pointers in C

- **Pointers are created with unary &**
  - Generates the address of pointee

```
char c = 5;
char * p = &c; // p gets the address of c
```

- **Pointees are referenced with unary ***
  - Called dereferencing the pointer

```
*p = 4; // set the target of p to 4
```

Pointers in C

- **All pointers have a type**
  - char *
  - int *
  - etc.
- **Arithmetic operations can be performed on pointers**
  - C handles correct size conversions based on type

```
int * p; //make an int pointer
p++;  //increment p's address by 4 bytes
```

- **Casting changes type, but not value**

```
char * c = (char*) p; //c gets p's address
c++;  //increment c's address by 1 byte
```

Pointers in C

- **Arrays and pointers are related**
  - Pointer to contiguous block of 3 ints

```
int * p = malloc(sizeof(int)*3); //get space for 3 ints, p points to first
int second = *(p+1); //add 1 int size unit (4 bytes) to p's address
int third = *(p+2); //add 2 int sizes (8 bytes)
```

  - Array of 3 ints

```
int a[3]; //get space for 3 ints
int second = a[1]; //get second element
int third = a[2]; //get third
```

Function pointers

- **Pointers can point to any location in memory**

- **Functions reside in memory, so…**

```
int sum(int a, int b) {return a+b;}
int (*sum_ptr)(int, int);  //declare pointer
sum_ptr = sum; //assign value to pointer
int r = sum_ptr(3, 5); //call sum, result is 8
```

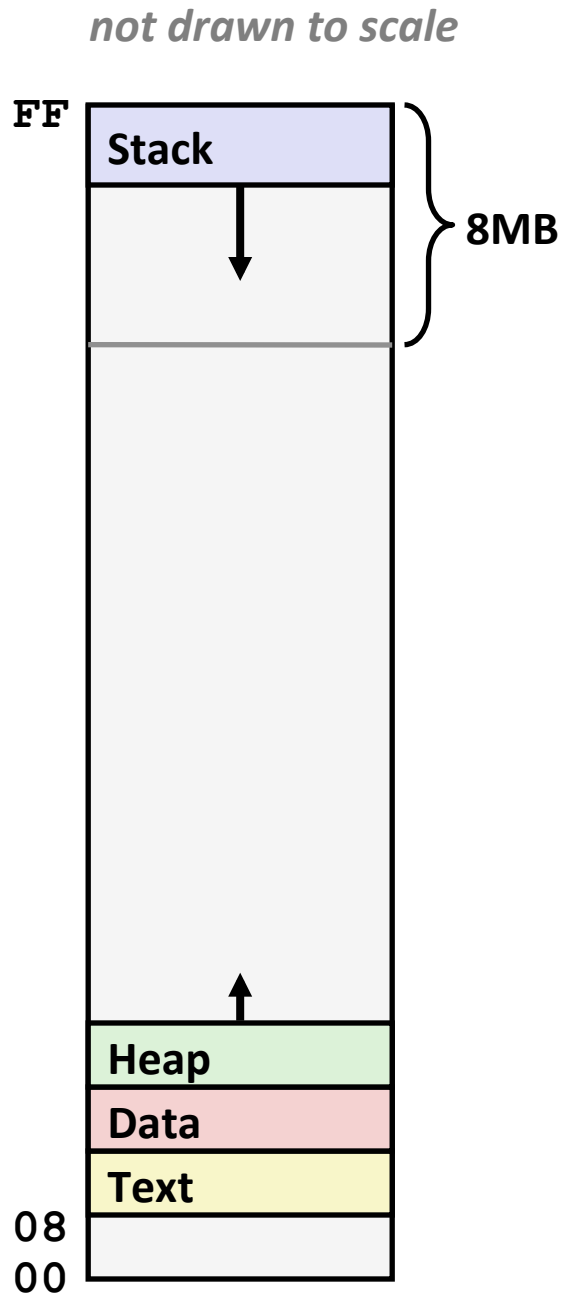
- **Can also pass function pointers as arguments**

Today

- **Structures**
  - Alignment
- **Unions**
- **Pointers**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection

IA32 Linux Memory Layout

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



Memory Allocation Example

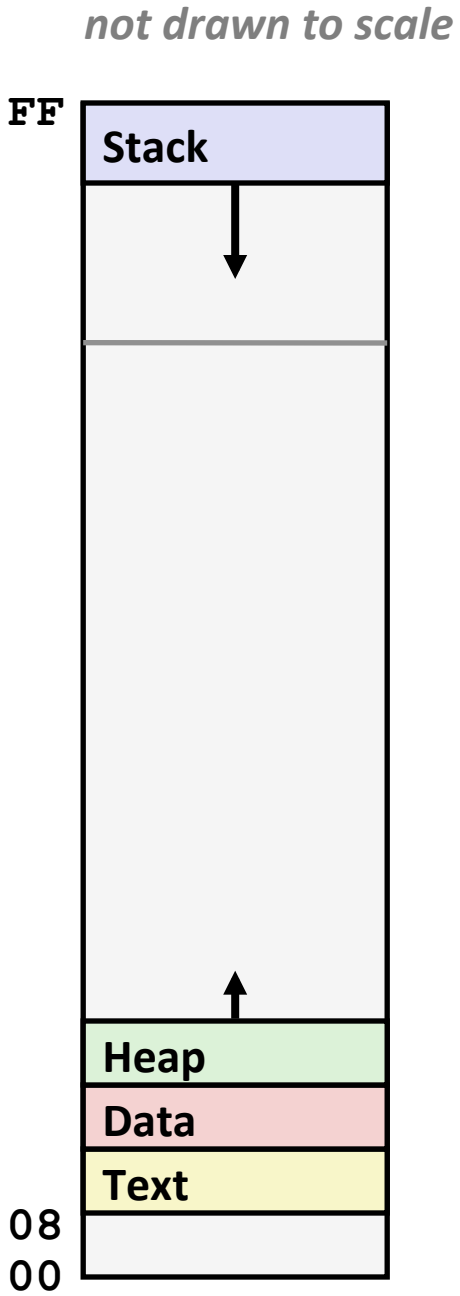
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

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?



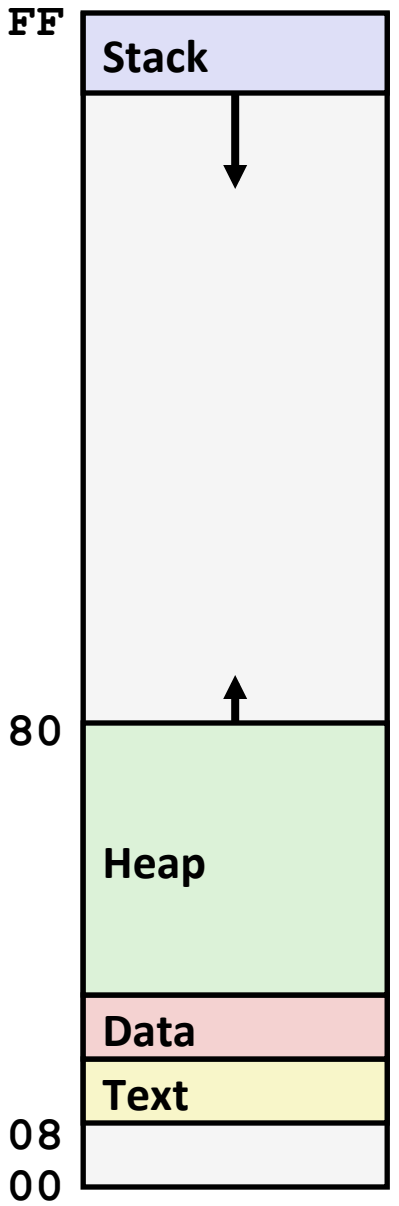
IA32 Example Addresses

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 linked
address determined at runtime

not drawn to scale



Today

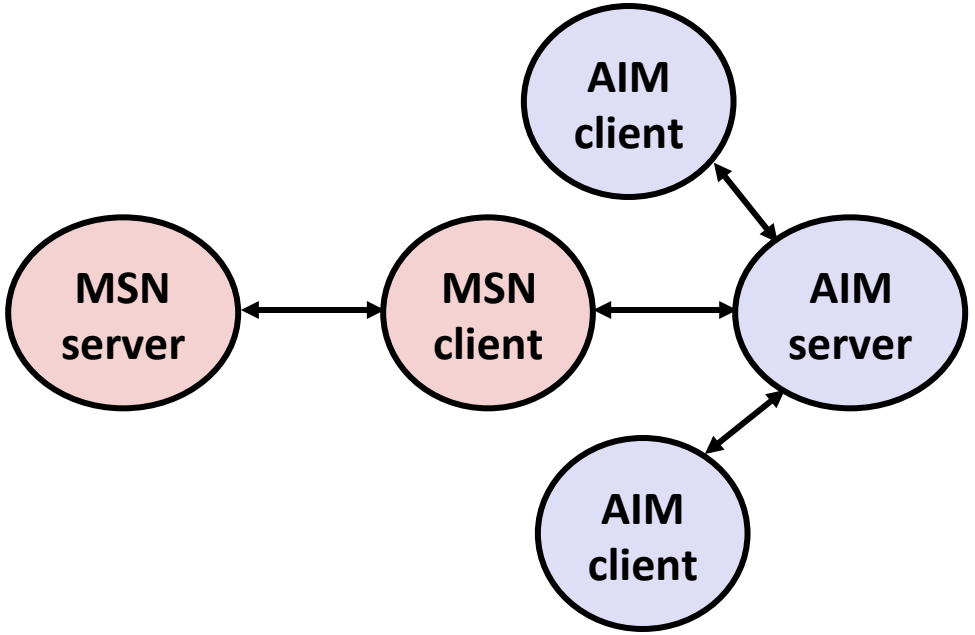
- Structures
  - Alignment
- Unions
- Pointers
- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection

Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?

Internet Worm and IM War

- **November, 1988**
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
- **July, 1999**
  - Microsoft launches MSN Messenger (instant messaging system).
  - Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



Internet Worm and 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.
  - How did it happen?

- **The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!**
    - many library functions do not check argument sizes.
    - allows target buffers to overflow.

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);
}
```

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

```
unix>./bufdemo
Type a string:1234567
1234567
```

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

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

Buffer Overflow Disassembly

echo:

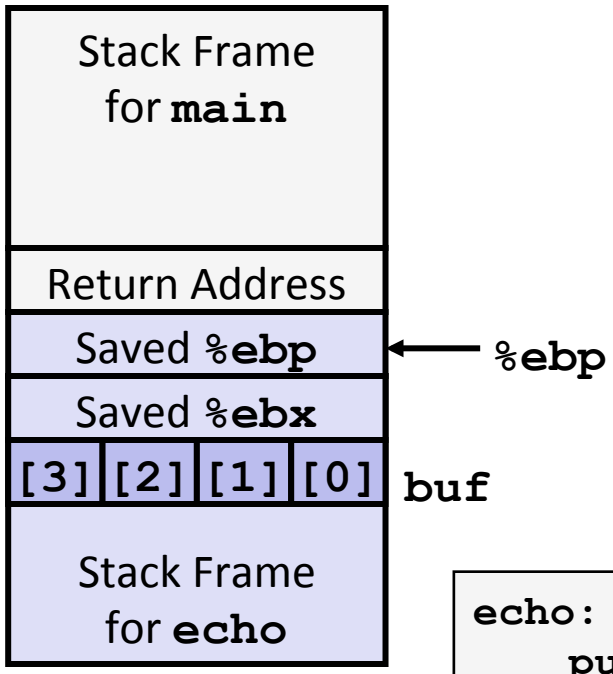
```
80485c5:  55                    push   %ebp
80485c6:  89 e5                 mov    %esp,%ebp
80485c8:  53                    push   %ebx
80485c9:  83 ec 14              sub    $0x14,%esp
80485cc:  8d 5d f8              lea 0xfffffff8(%ebp),%ebx
80485cf:  89 1c 24              mov    %ebx,(%esp)
80485d2:  e8 9e ff ff ff        call   8048575 <gets>
80485d7:  89 1c 24              mov    %ebx,(%esp)
80485da:  e8 05 fe ff ff        call   80483e4 <puts@plt>
80485df:  83 c4 14              add    $0x14,%esp
80485e2:  5b                    pop    %ebx
80485e3:  5d                    pop    %ebp
80485e4:  c3                    ret
```

call_echo:

```
80485eb:  e8 d5 ff ff ff        call   80485c5 <echo>
80485f0:  c9                    leave
80485f1:  c3                    ret
```

Buffer Overflow Stack

Before call to gets

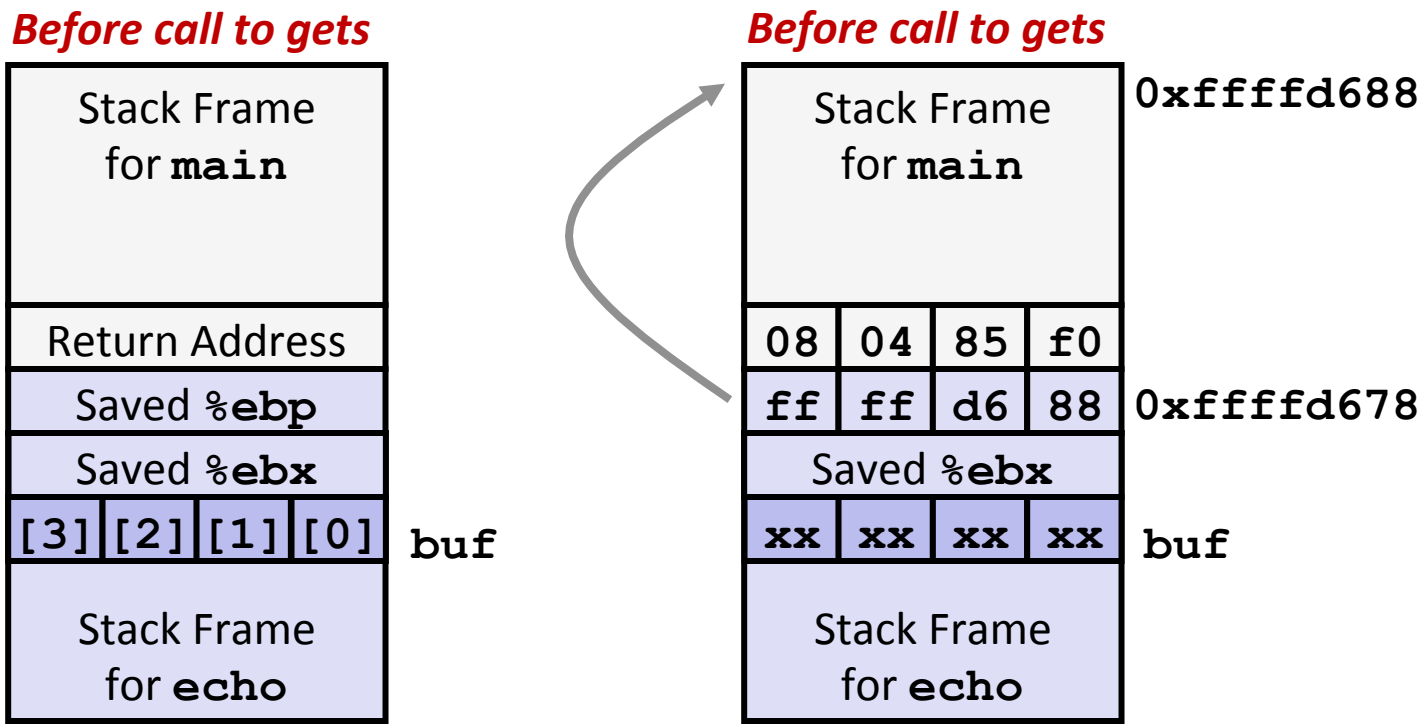



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $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
    . . .
```

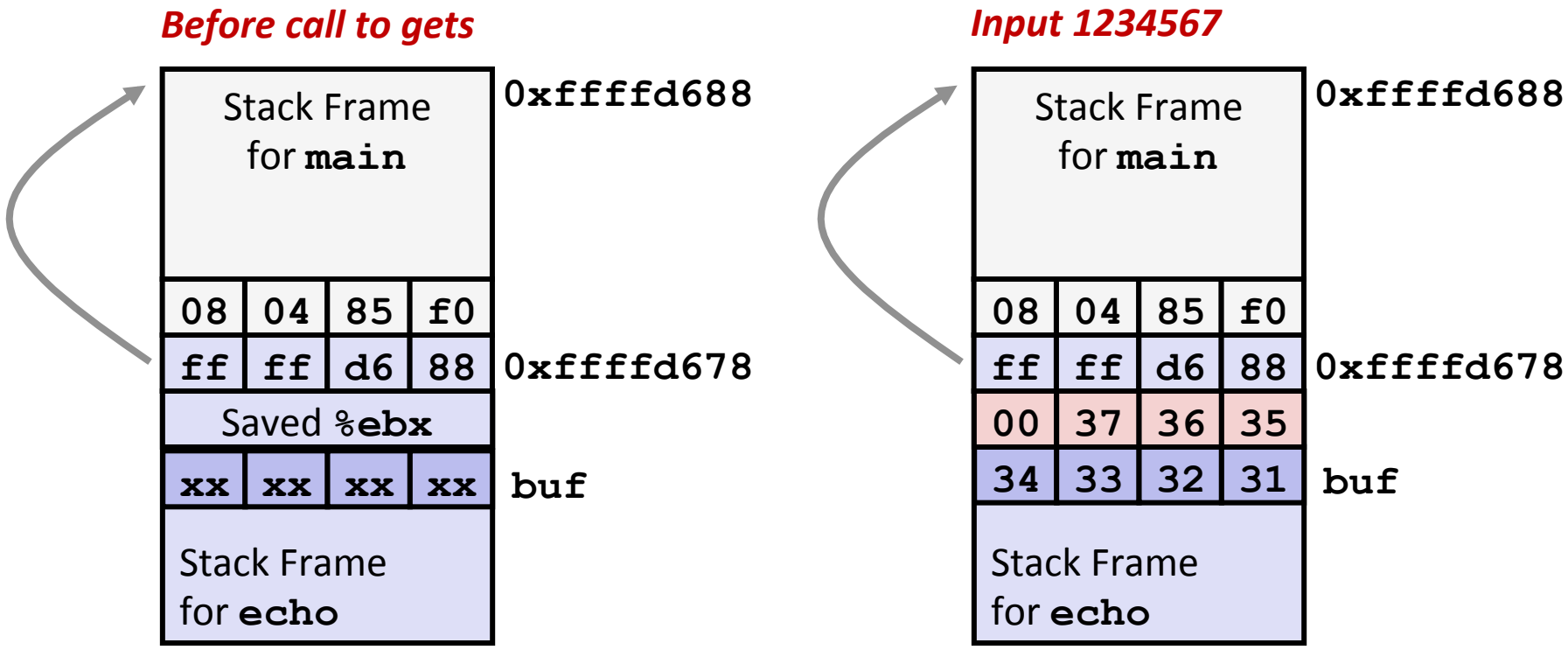
Buffer Overflow Stack Example

```
unix> 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
```



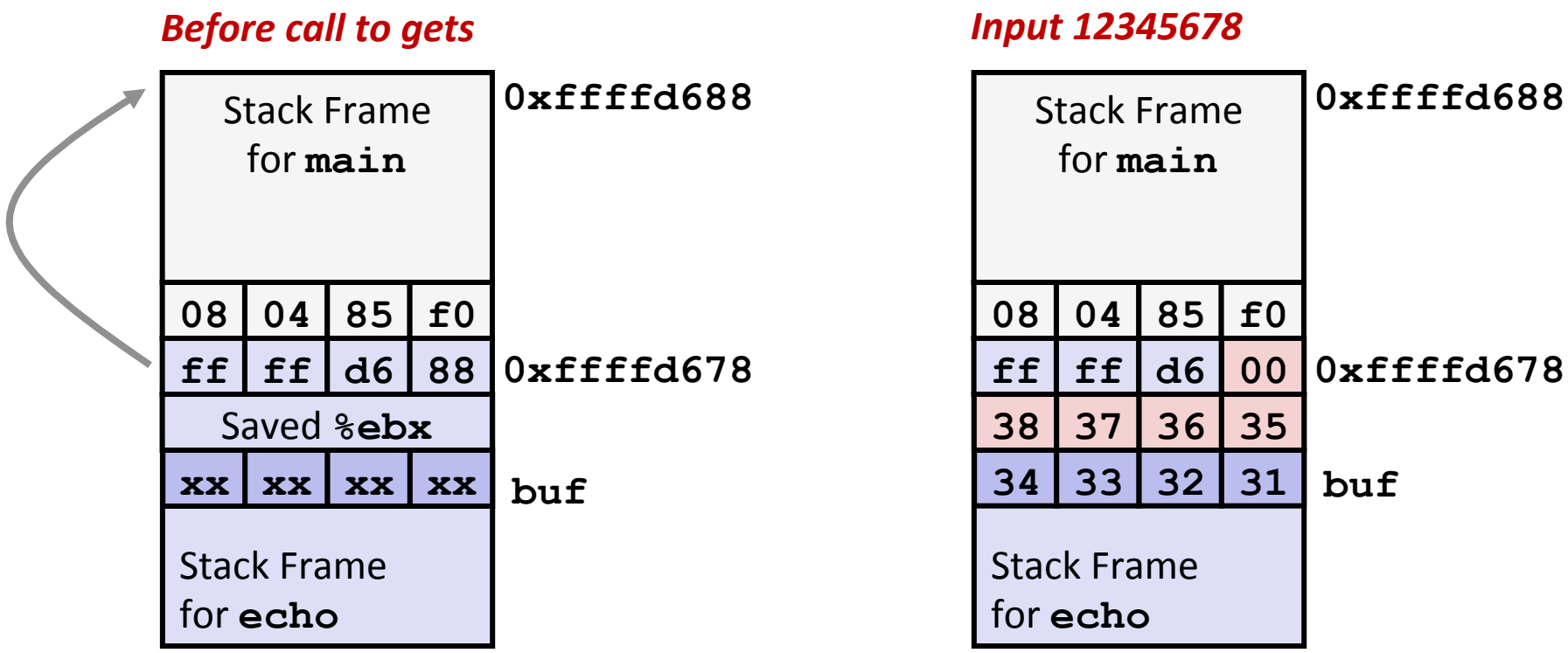

```
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave
```

Buffer Overflow Example #1



Overflow buf, and corrupt %ebx, but no problem

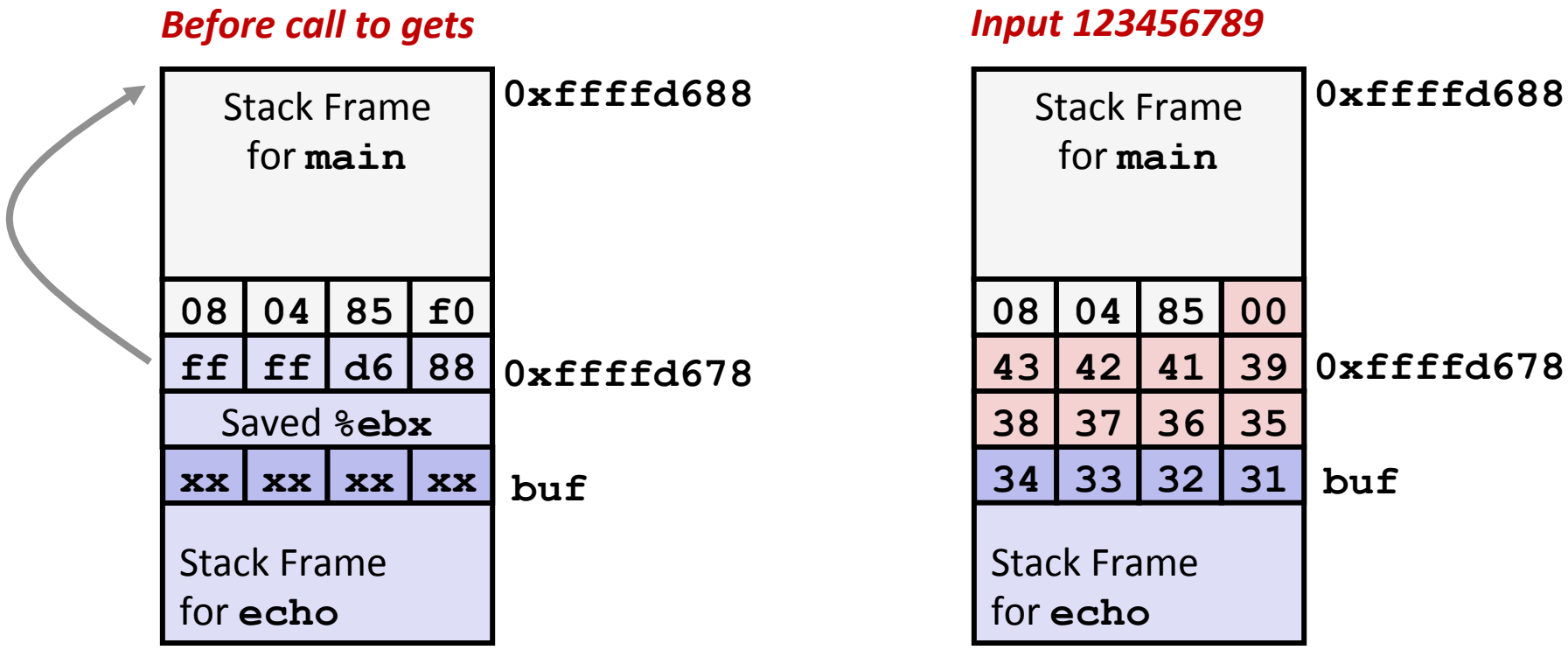
Buffer Overflow Example #2



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

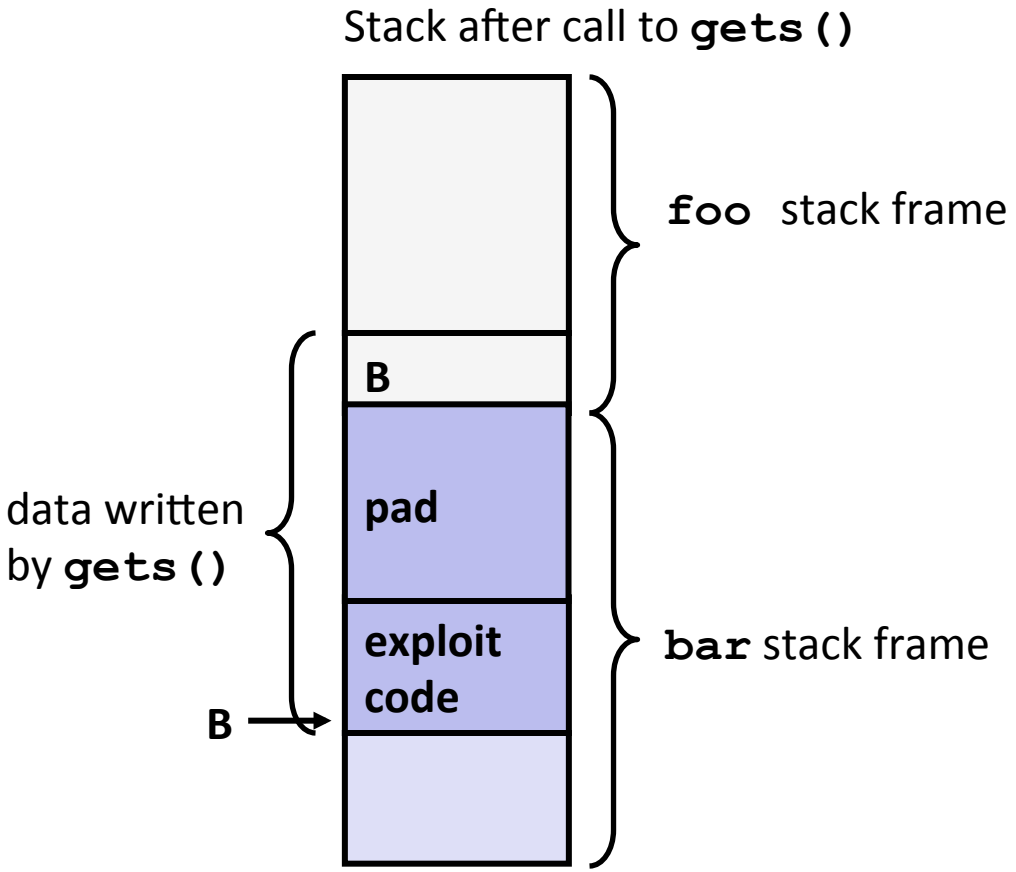
```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave   # Desired return point
```

Malicious Use of Buffer Overflow

```
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 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 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.

Exploits Based on Buffer Overflows

- ***Buffer overflow bugs allow remote machines to execute arbitrary code on victim machines***
- **IM War**
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 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.

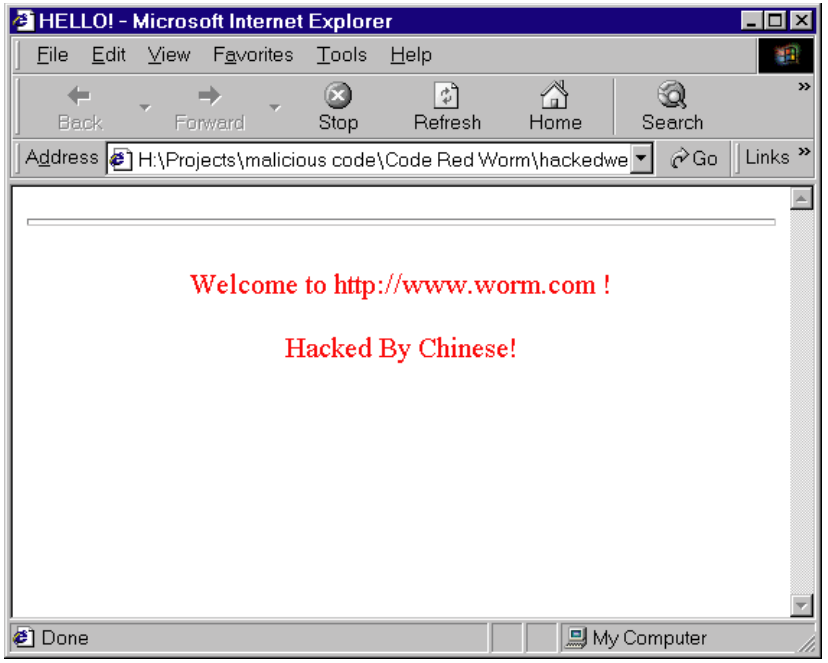
I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.
...
It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their 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!

Code Red Exploit Code

- **Starts 100 threads running**
- **Spread self**
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- **Attack www.whitehouse.gov**
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
  - Between 21st & 27th of month
- **Deface server's home page**
  - After waiting 2 hours



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

- **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

```
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
```

Stack Canaries

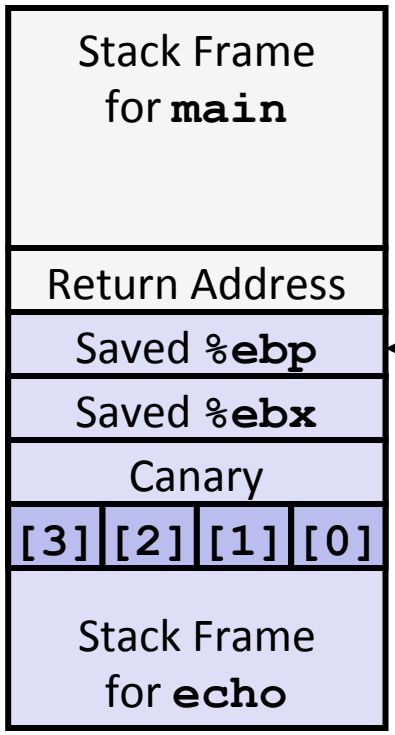
- **Idea**
  - Place special value ("canary") on stack just beyond buffer
  - Check for corruption before exiting function
- **GCC Implementation**
  - `-fstack-protector`
  - `-fstack-protector-all`

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

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:
    . . .
```

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**

Today

- **Structures**
  - Alignment
- **Unions**
- **Pointers**
- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection