15-213
"The course that gives CMU its Zip!"

Machine-Level Programming V: Miscellaneous Topics
Feb 10, 2004

Topics
- Linux Memory Layout
- Understanding Pointers
- Buffer Overflow
- Floating Point Code

Linux Memory Allocation

Initially
Linked
Some Heap
More Heap

Linux Memory Layout

Stack
- Runtime stack (8MB limit)

Heap
- Dynamically allocated storage
  - When call malloc, calloc, new

DLLs
- Dynamically Linked Libraries
- Library routines (e.g., printf, malloc)
- Linked into object code when loaded

Data
- Statically allocated data
  - E.g., arrays & strings declared in code

Text
- Executable machine instructions
  - Read-only

Text & Stack Example

Initially

(gdb) break main
(gdb) run
Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
$3 = (void *) 0xbffffc78

Main
- Address 0x804856f should be read
  0x804856f

Stack
- Address 0xbffffc78
Dynamic Linking Example

(gdb) print malloc
$1 = (<text variable, no debug info>)
   0x8048454 <malloc>
(gdb) run
Program exited normally.
(gdb) print malloc
$2 = (void *(unsigned int))
   0x40006240 <malloc>

Initially
- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

Final
- Code in DLL region

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 ... */
}

Example Addresses

$esp  0xbffffc78
p3   0x500b5008
p1   0x400b4008
Final malloc 0x40006240
p4   0x1904a640
p2   0x1904a538
beyond 0x1904a524
big_array 0x1804a520
huge_array 0x0804a510
main() 0x0804856f
useless() 0x08048560
Initial malloc 0x08048454

&p2? 0x1904a42c

C operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ) [ ]</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* /</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; ^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>&amp; &amp;</td>
</tr>
<tr>
<td>? : = += -= *= /= &amp;= ^= !== &lt;&lt;= &gt;&gt;=</td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= &amp;= ^= != == += &lt;&lt;= &gt;&gt;= = right to left</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Note: Unary +, -, and * have higher precedence than binary forms.
**C pointer declarations**

- `int *p`  
  p is a pointer to int
- `int *p[13]`  
  p is an array[13] of pointer to int
- `int *(p[13])`  
  p is an array[13] of pointer to int
- `int **p`  
  p is a pointer to a pointer to an int
- `int (*)(p[13])`  
  p is a pointer to an array[13] of int
- `int *f()`  
  f is a function returning a pointer to int
- `int (*)(f)()`  
  f is a pointer to a function returning int
- `int (*)(f())[13]()`  
  f is a function returning ptr to an array[13] of pointers to functions returning int
- `int (*)(*x[3])()[5]`  
  x is an array[3] of pointers to functions returning pointers to array[5] of ints

---

**Avoiding Complex Declarations**

Use `typedef` to build up the decl

Instead of

```c
int (*(*x[3])())[5] : 

typedef int fiveints[5];

typedef fiveints* p5i; 

typedef p5i (*f_of_p5is)();

f_of_p5is x[3];
```

X is an array of 3 elements, each of which is a pointer to a function returning an array of 5 ints.

---

**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 Unix functions do not check argument sizes.
- allows target buffers to overflow.
### String Library Code
- Implementation of Unix function `gets`
  - No way to specify limit on number of characters to read

```c
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

- Similar problems with other Unix functions
  - `strcpy`: Copies string of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

### Vulnerable Buffer Code

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

```c
int main()
{
    printf("Type a string:");
echo();
    return 0;
}
```

### Buffer Overflow Executions

- `./bufdemo` Type a string: 123 123
- `./bufdemo` Type a string: 12345 Segmentation Fault
- `./bufdemo` Type a string: 12345678 Segmentation Fault

### Buffer Overflow Stack

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

```
#include <stdio.h>

int main()
{
    printf("Type a string:");
echo();
    return 0;
}
```

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

```
#include <stdio.h>

int main()
{
    printf("Type a string:");
echo();
    return 0;
}
```
unix> gdb bufdemo  
(gdb) break echo  
Breakpoint 1 at 0x8048583  
(gdb) run  
Breakpoint 1, 0x8048583 in echo ()  
(gdb) print *(unsigned *)%ebp  
$1 = 0xbffff8f8  
(gdb) print *((unsigned *)%ebp + 1)  
$3 = 0x804864d

8048648: call 804857c <echo>  
804864d: mov 0xffffffe8(%ebp),%ebx  
# Return Point

Before call to gets

Stack Frame for main

Return Address

Saved %ebp

buf

Stack Frame for echo

%ebp

08 04 86 4d

buf

Stack Frame for echo

Before Call to gets

Input = "123"

Stack Frame for main

Return Address

Saved %ebp

buf

Stack Frame for echo

0xbffff8f8

buf

No Problem

Buffer Overflow Example #1

Input = "12345"

Stack Frame for main

Return Address

Saved %ebp

buf

Stack Frame for echo

%ebp

0xbffff8f8

buf

Saved value of %ebp set to 0xbff0035

Bad news when later attempt to restore %ebp

echo code:
8048592: push %ebx
8048593: call 80483ed <_init+0x50> # gets
8048598: mov 0xbffff8f8(%ebp),%ebx
804859b: mov %ebp,%eax
804859d: pop %ebp # %ebp gets set to invalid value
804859f: ret

Buffer Overflow Stack Example #2

Input = "12345678"

Stack Frame for main

Return Address

Saved %ebp

buf

%ebp

38 37 36 35

buf

%ebp and return address corrupted

Invalid address

No longer pointing to desired return point

Buffer Overflow Stack Example #3

Copyright (c) Carnegie Mellon, 2003
Malicious Use of Buffer Overflow

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

void bar() {
  char buf[64];
  gets(buf);
  ...
}

- Input string contains byte representation of executable code
- Overwrite return address with address of buffer
- 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.

The Internet Worm

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/2</td>
<td>18:24</td>
<td>First west coast computer infected</td>
</tr>
<tr>
<td></td>
<td>19:04</td>
<td>UCB gateway infected</td>
</tr>
<tr>
<td></td>
<td>20:00</td>
<td>MIT attacked</td>
</tr>
<tr>
<td></td>
<td>20:49</td>
<td>CSUtah.edu infected</td>
</tr>
<tr>
<td></td>
<td>21:21</td>
<td>Load avg reaches 5 on CSUtah.edu</td>
</tr>
<tr>
<td></td>
<td>21:41</td>
<td>Load avg reaches 7</td>
</tr>
<tr>
<td></td>
<td>22:01</td>
<td>Load avg reaches 16</td>
</tr>
<tr>
<td></td>
<td>22:20</td>
<td>Worm killed on CSUtah.edu</td>
</tr>
<tr>
<td></td>
<td>22:41</td>
<td>CSUtah.edu reinfected, load avg 27</td>
</tr>
<tr>
<td></td>
<td>22:49</td>
<td>CSUtah.edu shut down</td>
</tr>
<tr>
<td></td>
<td>23:31</td>
<td>Reinfected, load reaches 37</td>
</tr>
</tbody>
</table>

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.
Code Red Worm

History
- June 18, 2001. Microsoft announces buffer overflow vulnerability in IIS Internet server
- July 19, 2001. Over 250,000 machines infected by new virus in 9 hours
- White house must change its IP address. Pentagon shut down public WWW servers for day

When We Set Up CS:APP Web Site
- Received strings of form
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN....
HTTP/1.0" 400 325 "-" "-

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

Code Red Effects
Later Version Even More Malicious
- Code Red II
- As of April, 2002, over 18,000 machines infected
- Still spreading

Paved Way for NIMDA
- Variety of propagation methods
- One was to exploit vulnerabilities left behind by Code Red II

ASIDE (security flaws start at home)
- .rhosts used by Internet Worm
- Attachments used by MyDoom (1 in 6 emails Monday morning!)
Avoiding Overflow Vulnerability

```c
/* 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

FPU Data Register Stack

FPU register format (extended precision)

```
+-------------------+---------------+
| 79 78 6463 0      |
| s  exp            |
| frac             |
```

FPU registers

- 8 registers
- Logically forms shallow stack
- Top called `%st(0)`
- When push too many, bottom values disappear

IA32 Floating Point

History

- 8086: first computer to implement IEEE FP
- Separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip

Summary

- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats

- Single precision (C `float`): 32 bits
- Double precision (C `double`): 64 bits
- Extended precision (C `long double`): 80 bits

FPU instructions

Large number of fp instructions and formats

- ~50 basic instruction types
- Load, store, add, multiply
- Sin, cos, tan, arctan, and log!

Sample instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldz</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flds Addr</td>
<td>push M[Addr]</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>fmuls Addr</td>
<td>%st(0) ← %st(0) * M[Addr]</td>
<td>Multiply</td>
</tr>
<tr>
<td>faddp</td>
<td>%st(1) ← %st(0) + %st(1); pop</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>
**Floating Point Code Example**

**Compute Inner Product of Two Vectors**
- Single precision arithmetic
- Common computation

```c
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i]*y[i];
    }
    return result;
}
```

**Floating Point Inner Product Stack Trace**

### Initialization

1. **fldz**
   - Push +0.0
   - **st(0)** = 0.0

### Iteration 0

2. **flds** (%ebx, %eax, 4)
   - Push x[0]
   - **st(1)** = x[0]

3. **fmuls** (%ecx, %eax, 4)
   - Multiply x[0] by y[0]
   - **st(1)** = st(0) + st(0)

4. **faddp**
   - **st(0)** = st(0) + st(1)

### Iteration 1

5. **flds** (%ebx, %eax, 4)
   - Push x[1]
   - **st(1)** = x[1]

6. **fmuls** (%ecx, %eax, 4)
   - Multiply x[1] by y[0]
   - **st(1)** = st(0) + st(0)

7. **faddp**
   - **st(0)** = st(0) + st(1)

Copyright (c) Carnegie Mellon, 2003
Final Observations

Memory Layout
- OS/machine dependent (including kernel version)
- Basic partitioning: stack/data/text/heap/DLL found in most machines

Type Declarations in C
- Notation obscure, but very systematic

Working with Strange Code
- Important to analyze nonstandard cases
  - E.g., what happens when stack corrupted due to buffer overflow
  - Helps to step through with GDB

IA32 Floating Point
- Strange "shallow stack" architecture

Final Recommendation

Sign your mail.

Google "pgp"