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

Machine-Level Programming V: Advanced Topics
Sept. 18, 2008

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

Memory Allocation Example

```c
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 ... */
}
```

IA32 Example Addresses

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

Address range ~2^32

&p2           0x18049760
**x86-64 Example Addresses**

- **$rsp**: 0x7ffffff8d1f8
- **p3**: 0x2aaabaadd010
- **p1**: 0x2aaaaaad010
- **p4**: 0x000011501120
- **p2**: 0x000011501010
- **beyond**: 0x0000000500a44
- **big_array**: 0x000010500a80
- **huge_array**: 0x0000000500a50
- **main()**: 0x0000000400510
- **useless()**: 0x000000040050
- **finalmalloc()**: 0x00386ae6a170

Address range ~2^47

- **&p2**: 0x000010500a60

---

**C operators**

**Operators**

- `()`  `[]`  `->`
- `!`  `~`  `++`  `--`  `+`  `-`
- `*`  `/`  `%`
- `+`  `-`
- `<<`  `>>`
- `<`  `<=`  `>`  `>=`  `==`  `!=`  `&`  `^`  `|`
- `&&`  `||`  `?:`
- `+=`  `-=`  `*=`  `/=`  `%=`  `&=`  `^=`  `|=`

**Associativity**

- left to right
- right to left

- `()`  `[]`  `->` has very high precedence
- `:` has very high precedence
- `monadic *` just below

---

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

---

**Avoiding Complex Declarations**

- Use `typedef` to build up the declaration
- Instead of `int (*(*x[3][]))[][5]:`
  
  ```c
  typedef int fiveints[5];
  typedef fiveints* p5i;
  typedef p5i (*f_of_p5i)();
  
  f_of_p5i 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

**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 = getchar();
    char * p = dest;
    while (c != EOF && c != '\n') {  
        * p++ = c;
        c = getchar();
    }  
    * 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**
- `/* Echo Line */`
```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

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

unix> ./bufdemo
Type a string: 123455678
Segmentation Fault

unix> ./bufdemo
Type a string: 1234556789ABC
Segmentation Fault
```

Buffer Overflow Disassembly

```
080484f0 <echo>:
push %ebp
80484f0: 55
80484f1: 89 e5
80484f3: push %ebp
80484f4: 8d 5d f8
lea 0xfffffffff8(%ebp),%ebx
80484f7: 83 ec 14
sub $0x14,%esp
80484f9: 89 1c 24
mov %ebx,(%esp)
80484fd: e8 ae ff ff ff
call 80484b0 <gets>
8048502: 89 1c 24
mov %ebx,(%esp)
8048505: e8 8a fe ff ff
call 8048394 <puts@plt>
804850a: 83 c4 14
add $0x14,%esp
804850d: 5b
pop %ebx
804850e: c9
leave
804850f: c3
ret
```

Buffer Overflow Stack

```
0xbfffc638
```

Buffer Overflow Stack Example

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

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp$1 = 0xffffc638
print /x *(unsigned *)$ebp
$2 = 0xffffc658
print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f7
```

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
Breakpoint 1, 0x8048583 in echo ()
(gdb) print /x $ebp
$1 = 0xfffffc638
(gdb) print /x *(unsigned *)&ebp
$2 = 0xfffffc658
(gdb) print /x *((unsigned *)&ebp + 1)
$3 = 0x80485f7
```
Buffer Overflow Example #1

Before Call to gets

Input = "1234567"

Overflow buf, but no problem

Buffer Overflow Stack Example #2

Input = "12345678"

Example #3 Failure

Input = "123456789ABC"

No longer pointing to desired return point
Example #2 Failure

Malicious Use of Buffer Overflow

Exploits Based on Buffer Overflows

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.

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.

Stack Frame for main

Input = "12345678"

echo code restores %ebp with corrupted value

Subsequent references based on %ebp invalid

Return from echo:

Stack after call to gets()

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

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

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

Stack

foo stack frame

bar stack frame

exploit code

pad

data written by gets()

A

return address

B

exploit code

pad

data written by gets()

A

return address

B

exploit code

pad

data written by gets()

A

return address

B

exploit code

pad

data written by gets()
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!
Avoiding Overflow Vulnerability

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
  - Add explicit “execute” permission

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 Data Register Stack

FPU register format (extended precision)

```
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>78</td>
<td>6463</td>
</tr>
</tbody>
</table>
```

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

```
stack grows down
%st(0) %st(1) %st(2) %st(3)
```

Instruction decoder and sequencer

Integer Unit

FPU

Memory
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>

Programming with SSE3

XMM Registers
- 16 total, each 16 bytes
- 16 single-byte integers
- 8 16-bit integers
- 4 32-bit integers
- 4 single-precision floats
- 2 double-precision floats
- 1 single-precision float
- 1 double-precision float

Scalar & SIMD Operations

- Scalar Operations: Single Precision
  - addss %xmm0, %xmm1
- SIMD Operations: Single Precision
  - addps %xmm0, %xmm1
- SIMD Operations: Double Precision
  - addpd %xmm0, %xmm1

x86-64 FP Code

Example

Compute Inner Product of Two Vectors
- Single precision arithmetic
- Common computation
- Uses SSE3 instructions

```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;
}
```
Final Observations

Memory Layout
- OS/machine dependent (including kernel version)
- Basic partitioning: stack/data/text/heap/shared-libs 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

Floating Point
- IA32: Strange “shallow stack” architecture
- x86-64: SSE3 permits more conventional, register-based approach

Final Observations (Cont.)

Assembly Language
- Very different than programming in C
- Architecture specific (IA-32, X86-64, Sparc, PPC, MIPS, ARM, 370, ...)
- No types, no data structures, no safety, just bits&bytes
- Rarely used to program
- Needed to access the full capabilities of a machine
- Important to understand for debugging and optimization