Machine-Level Programming V: Advanced Topics

15-213: Introduction to Computer Systems
9th Lecture, Feb. 9, 2016

Instructors:
Franz Franchetti & Seth Copen Goldstein, Ralf Brown, and Brian Raling
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions
x86-64 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 / Shared Libraries**
  - Executable machine instructions
  - Read-only

Hex Address

```
00007FFFFFFF
```

```
400000
```

```
000000
```
Memory Allocation Example

```c
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
    /* Some print statements ... */
}
```

Where does everything go?
x86-64 Example Addresses

address range $\sim 2^{47}$

- local
  - Address: 0x00007ffe4d3be87c
- p1
  - Address: 0x00007f7262a1e010
- p3
  - Address: 0x00007f7162a1d010
- p4
  - Address: 0x000000008359d120
- p2
  - Address: 0x000000008359d010
- big_array
  - Address: 0x000000000080601060
- huge_array
  - Address: 0x0000000000601060
- main()
  - Address: 0x000000000040060c
- useless()
  - Address: 0x0000000000400590

Diagram of address spaces: Text, Data, Stack, Heap.
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions
Recall: Memory Referencing Bug Example

```c
typedef struct {
    int a[2];
    double d;
} struct_t;

double fun(int i) {
    volatile struct_t s;
    s.d = 3.14;
    s.a[i] = 1073741824; /* Possibly out of bounds */
    return s.d;
}
```

- fun(0) -> 3.14
- fun(1) -> 3.14
- fun(2) -> 3.1399998664856
- fun(3) -> 2.00000061035156
- fun(4) -> 3.14
- fun(6) -> Segmentation fault

- Result is system specific
Memory Referencing Bug Example

typedef struct {
    int a[2];
    double d;
} struct_t;

fun(0)  ->  3.14
fun(1)  ->  3.14
fun(2)  ->  3.1399998664856
fun(3)  ->  2.00000061035156
fun(4)  ->  3.14
fun(6)  ->  Segmentation fault

Explanation:

<table>
<thead>
<tr>
<th>Critical State</th>
<th>Location accessed by fun(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3.14</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td></td>
</tr>
<tr>
<td>d7 ... d4</td>
<td></td>
</tr>
<tr>
<td>d3 ... d0</td>
<td></td>
</tr>
<tr>
<td>a[1]</td>
<td></td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
</tr>
</tbody>
</table>

struct_t
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()`

```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;
}
```

- 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

```c
typedef int     int;
typedef struct{
    int a;
    int b;
} struct1;

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

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

Unix>
 ./bufdemo-nsp
Type a string:
012345678901234567890123
012345678901234567890123

Unix> ./bufdemo-nsp
Type a string: 012345678901234567890123
Segmen… Fault

btw, how big is big enough?
## Buffer Overflow Disassembly

### echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000004006cf</td>
<td>48 83 ec 18</td>
<td>sub $0x18,%rsp</td>
</tr>
<tr>
<td>00000000004006d3</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>00000000004006d6</td>
<td>e8 a5 ff ff ff</td>
<td>callq 400680 &lt;gets&gt;</td>
</tr>
<tr>
<td>00000000004006db</td>
<td>48 89 e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>00000000004006de</td>
<td>e8 3d fe ff ff</td>
<td>callq 400520 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>00000000004006e3</td>
<td>48 83 c4 18</td>
<td>add $0x18,%rsp</td>
</tr>
<tr>
<td>00000000004006e7</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>

### call_echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000004006e8</td>
<td>48 83 ec 08</td>
<td>sub $0x8,%rsp</td>
</tr>
<tr>
<td>00000000004006ec</td>
<td>b8 00 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>00000000004006f1</td>
<td>e8 d9 ff ff ff</td>
<td>callq 4006cf &lt;echo&gt;</td>
</tr>
<tr>
<td>00000000004006f6</td>
<td>48 83 c4 08</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>00000000004006fa</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>
Buffer Overflow Stack

Before call to gets

Stack Frame for call_echo

Return Address
(8 bytes)

20 bytes unused

buf ← %rsp

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

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    . . .
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call_echo

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
<td></td>
</tr>
</tbody>
</table>
20 bytes unused

buf ← %rsp

call_echo:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>subq $24, %rsp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>movq %rsp, %rdi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call gets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp

...
Buffer Overflow Stack Example #1

After call to gets

Stack Frame for call_echo

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>f6</td>
</tr>
<tr>
<td>00</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
</tbody>
</table>

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8,%rsp
    ...

buf ← %rsp

unix> ./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012

“01234567890123456789012\0”

Overflowed buffer, but did not corrupt state
Buffer Overflow Stack Example #2

After call to gets

Stack Frame for call_echo

void echo()
{
    char buf[4];
    gets(buf);
    . . .
}

call_echo:

        . . .
    4006f1: callq 4006cf <echo>
    4006f6: add $0x8,%rsp
        . . .

buf ← %rsp

unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault

"0123456789012345678901234\0"

Overflowed buffer and corrupted return pointer
Buffer Overflow Stack Example #3

After call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 00</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

void echo()
{
    char buf[4];
    gets(buf);
    ...
}

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...

    4006f1: callq 4006cf <echo>
    4006f6: add $0x8,%rsp
    ...  

buf ← %rsp

unix> ./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123

“012345678901234567890123\0”

Overflowed buffer, corrupted return pointer, but program seems to work!
Buffer Overflow Stack Example #3 Explained

After call to `gets`

<table>
<thead>
<tr>
<th>Stack Frame for call echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 48 83 80</td>
</tr>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 40 06 00</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
<tr>
<td>39 38 37 36</td>
</tr>
<tr>
<td>35 34 33 32</td>
</tr>
<tr>
<td>31 30 39 38</td>
</tr>
<tr>
<td>37 36 35 34</td>
</tr>
<tr>
<td>33 32 31 30</td>
</tr>
</tbody>
</table>

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

“Returns” to unrelated code

Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main` – `call_echo` has no local variables
Code Injection Attacks

- Input string contains byte representation of executable code
- Overwrite return address $A$ with address of buffer $B$
- When $Q$ executes `ret`, will jump to exploit code

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

```c
void P() {
    Q();
    ...
}
```

Stack after call to `gets()`

- $P$ stack frame
- $A$ return address
- $B$ data written by `gets()`
- Pad
- Exploit code

Stack after call to `gets()`

- $Q$ stack frame
- $B$
How Does The Attack Code Execute?

```c
void P()
{
    Q();
    ...
}

int Q()
{
    char buf[64];
    gets(buf); // A->B
    ...
    return ...;
}
```
Crafting an Injection Attack String

Stack Frame
for call echo

---

Injected Code

```
B8 00 00 00 00   mov    $0x0,%eax
C3               retq
```

Attack: `int echo()` always returns 0

```
int echo() {
    char buf[4];
    gets(buf);
    ...
    return ...;
}
```

buf = 00007FFFFFFFAB80

---

**Attack String**

```
"\xB8\0\0\0\0\xC3678901234567890123\x80\xAB\xFF\xFF\xFF\xFF\x07\0\0"
```

Cannot have \n or EOF in attack string
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 (!)

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

```c
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```
2. System-Level Protections can help

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code
    - Stack repositioned each time program executes

![Diagram showing stack base and allocation](image)
2. System-Level Protections can help

- **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
  - Stack marked as non-executable

Any attempt to execute this code will fail
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`
  - Now the default (disabled earlier)

```shell
unix> ./bufdemo-sp
Type a string: 0123456
  0123456

unix> ./bufdemo-sp
Type a string: 01234567
*** stack smashing detected ***
```
Protected Buffer Disassembly

echo:

```assembly
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: mov    0x8(%rsp),%rax
400761: je     400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq
```
## Setting Up Canary

### Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for <code>call_echo</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address (8 bytes)</td>
</tr>
<tr>
<td>Canary (8 bytes)</td>
</tr>
<tr>
<td>[3] [2] [1] [0]</td>
</tr>
</tbody>
</table>

Before call to `gets`, the stack frame looks like this:

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

### Canary

Before the call to `gets`, the `buf` pointer points to the stack frame:

```
buf ← %rsp
```

### Canary Setup

- `movq %fs:40, %rax`  # Get canary
- `movq %rax, 8(%rsp)`  # Place on stack
- `xorl %eax, %eax`  # Erase canary

```
echo:
    . . .
    movq %fs:40, %rax  # Get canary
    movq %rax, 8(%rsp)  # Place on stack
    xorl %eax, %eax     # Erase canary
    . . .
```
Checking Canary

After call to gets

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

Input: 0123456

buf ← %rsp

echo:

... 

movq 8(%rsp), %rax       # Retrieve from stack
xorq %fs:40, %rax        # Compare to canary
je .L6                   # If same, OK
call __stack_chk_fail    # FAIL
Return-Oriented Programming Attacks

- **Challenge (for hackers)**
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code

- **Alternative Strategy**
  - Use existing code
    - E.g., library code from stdlib
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*

- **Construct program from gadgets**
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable
Gadget Example #1

```c
long ab_plus_c
    (long a, long b, long c)
{
    return a*b + c;
}
```

The gadget address is at 0x4004d4:

```
00000000004004d0 <ab_plus_c>:
    4004d0:  48 0f af fe  imul %rsi,%rdi
    4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
    4004d8:  c3           retq
```

rax ← rdi + rdx

Use tail end of existing functions
Gadget Example #2

```c
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

**<setval>:**

```
4004d9:  c7 07 d4 48 89 c7  movl $0xc78948d4,(%rdi)
4004df:  c3                 retq
```

- **Encodes** `movq %rax, %rdi`
- **rdi ← rax**
- **Gadget address = 0x4004dc**

- **Repurpose byte codes**
ROP Execution

- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**
Crafting an ROB Attack String

### Stack Frame for `call echo`

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>48</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>39</td>
<td>38</td>
</tr>
<tr>
<td>35</td>
<td>34</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>33</td>
<td>32</td>
</tr>
</tbody>
</table>

- `buf`:
  - Address: 33 32 31 30
  - Value: "012345678901234567890123\xd4\x04\x40\x00\x00\x00\x00"

### Gadget

```
00000000004004d0 <ab_plus_c>:
  4004d0: 48 0f af fe  imul %rsi,%rdi
  4004d4: 48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8: c3           retq
```

- `rax ← rdi + rdx`

### Attack

```c
int echo() {
    char buf[4];
    gets(buf);
    ...
    return ...;
}
```

Multiple gadgets will corrupt stack upwards
Craft Your Own Attack

<table>
<thead>
<tr>
<th>Stack Frame for call echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>00 48 83 80</td>
</tr>
<tr>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>00 40 04 d4</td>
</tr>
</tbody>
</table>

**Gadget**

\[
\text{movl } \$$0x7c8948d4,\text{(\%rdi)}
\]

\[
\text{retq}
\]

**Gadget address = 0x4004dc**

**Attack String**

```
```

**Attack: invoke**

\[
\text{movq } \%rax, \%rdi
\]
Craft Your Own Attack

Stack Frame for `call echo`

| 00 | 00 | 00 | 00 |
| 00 | 48 | 83 | 80 |
| 00 | 00 | 00 | 00 |
| 00 | 40 | 04 | dc |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

Gadget

Gadget address = 0x4004dc

Encoded `movq %rax, %rdi`

Attack String

```
"012345678901234567890123\xdc\x04\x40\0\0\0\0\0\0"
```
Today

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Unions
Union Allocation

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

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

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

■ 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**

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

How are the bytes inside short/int/long stored?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

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

printf("Characters 0-7 ==
[0x%02x,0x%02x,0x%02x,0x%02x,0x%02x,0x%02x,0x%02x,0x%02x]\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%02x,0x%02x,0x%02x,0x%02x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%02x,0x%02x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%01x]\n",
    dw.l[0]);
```
Byte Ordering on IA32

Little Endian

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Output on Sun:

#### Characters

\[0-7 == [\text{0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7}]\]

#### Shorts

\[0-3 == [\text{0xf0f1,0xf2f3,0xf4f5,0xf6f7}]\]

#### Ints

\[0-1 == [\text{0xf0f1f2f3,0xf4f5f6f7}]\]

#### Long

\[0 == [\text{0xf0f1f2f3}]\]
Byte Ordering on x86-64

Little Endian

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 of Compound Types in C

- **Arrays**
  - 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
Summary

- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
  - Code Injection Attack
  - Return Oriented Programming
- Unions
Exploits Based on Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**

- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult

- **Examples across the decades**
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more

- **You will learn some of the tricks in attacklab**
  - 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 \texttt{gets()} to read the argument sent by the client:
    - \texttt{finger droh@cs.cmu.edu}
  - Worm attacked fingerd server by sending phony argument:
    - \texttt{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 \textasciitilde 6000 computers in hours (10\% of the Internet 😊)
    - see June 1989 article in \textit{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
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