Machine-Level Programming V: Advanced Topics

15-213/18-213/14-513/15-513/18-613: Introduction to Computer Systems
9th Lecture, September 24, 2019
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: 00007FFFFFFFH (≈ 2**47**−1)

00007FFFFFFFO000000

Diagram not drawn to scale
Memory Allocation Example

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

int global = 0;

int useless() { return 0; }

int main () {
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);    /* 256 MB */
    psmall2 = malloc(1L << 8);    /* 256  B */
    phuge3 = malloc(1L << 32);   /*  4 GB */
    psmall4 = malloc(1L << 8);   /* 256  B */
    /* Some print statements ... */
}
```

Where does everything go?

x86-64 Example Addresses

address range \( \sim 2^{47} \)

local 0x00007ffe4d3be87c
phuge1 0x00007f7262a1e010
phuge3 0x00007f7162a1d010
psmall4 0x000000008359d120
psmall2 0x000000008359d010
big_array 0x0000000080601060
huge_array 0x0000000000601060
main() 0x000000000040060c
useless() 0x00000000000400590

(Exact values can vary)
Runaway Stack Example

```
int recurse(int x) {
    int a[1<<15];  // 4*2^15 = 128 KiB
    printf("x = %d.  a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```

- Functions store local data on in stack frame
- Recursive functions cause deep nesting of frames
Today

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

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) -> 3.1400000000
fun(1) -> 3.1400000000
fun(2) -> 3.1399998665
fun(3) -> 2.0000006104
fun(6) -> Stack smashing detected
fun(8) -> Segmentation fault

- Result is system specific
Memory Referencing Bug Example

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

fun(0) -> 3.1400000000
fun(1) -> 3.1400000000
fun(2) -> 3.1399998665
fun(3) -> 2.0000006104
fun(4) -> Segmentation fault
fun(8) -> 3.1400000000

Explanation:

Location accessed by fun(i)

struct_t

<table>
<thead>
<tr>
<th>???</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical State</td>
<td>7</td>
</tr>
<tr>
<td>Critical State</td>
<td>6</td>
</tr>
<tr>
<td>Critical State</td>
<td>5</td>
</tr>
<tr>
<td>Critical State</td>
<td>4</td>
</tr>
<tr>
<td>d7 ... d4</td>
<td>3</td>
</tr>
<tr>
<td>d3 ... d0</td>
<td>2</td>
</tr>
<tr>
<td>a[1]</td>
<td>1</td>
</tr>
<tr>
<td>a[0]</td>
<td>0</td>
</tr>
</tbody>
</table>
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
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}

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

btw, how big is big enough?

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

unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Segmentation Fault
Buffer Overflow Disassembly

echo:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>40069c:</td>
<td>48 83  ec</td>
<td>sub $0x18,%rsp</td>
<td>4006a0:</td>
<td>48 89  e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006a3:</td>
<td>e8 a5  ff</td>
<td>callq 40064d &lt;gets&gt;</td>
<td>4006a8:</td>
<td>48 89  e7</td>
<td>mov %rsp,%rdi</td>
</tr>
<tr>
<td>4006ab:</td>
<td>e8 50  fe</td>
<td>callq 400500 <a href="mailto:puts@plt">puts@plt</a></td>
<td>4006b0:</td>
<td>48 83  c4</td>
<td>add $0x18,%rsp</td>
</tr>
<tr>
<td>4006b4:</td>
<td>c3</td>
<td>retq</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

callEcho:

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>4006b5:</td>
<td>48 83  ec</td>
<td>sub $0x8,%rsp</td>
<td>4006b9:</td>
<td>b8 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>4006be:</td>
<td>e8 d9  ff</td>
<td>callq 40069c &lt;echo&gt;</td>
<td>4006c3:</td>
<td>48 83  c4</td>
<td>add $0x8,%rsp</td>
</tr>
<tr>
<td>4006c7:</td>
<td>c3</td>
<td>retq</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Buffer Overflow Stack Example

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 $0x18, %rsp
    movq %rsp, %rdi
    call gets
    ...

Buffer Overflow Stack Example

Before call to `gets`

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

echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    . . .

call_echo:
    . . .
    4006be: callq 4006cf <echo>
    4006c3: add $0x8, %rsp
    . . .

Stack Frame for `call_echo`

20 bytes unused

buf ← %rsp
Buffer Overflow Stack Example #1

After 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>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>c3</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

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

echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    ...

call_echo:
    ...
    4006be: callq 4006cf <echo>
    4006c3: 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`

<table>
<thead>
<tr>
<th>Stack Frame for <code>call_echo</code></th>
<th>void echo()</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00 00</td>
<td>{</td>
</tr>
<tr>
<td>00 40 06 00</td>
<td>char buf[4];</td>
</tr>
<tr>
<td>33 32 31 30</td>
<td>gets(buf);</td>
</tr>
<tr>
<td>39 38 37 36</td>
<td>. . .</td>
</tr>
<tr>
<td>35 34 33 32</td>
<td>}</td>
</tr>
<tr>
<td>31 30 39 38</td>
<td></td>
</tr>
<tr>
<td>37 36 35 34</td>
<td></td>
</tr>
<tr>
<td>33 32 31 30</td>
<td></td>
</tr>
</tbody>
</table>

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

echo:

```
subq $0x18, %rsp
movq %rsp, %rdi
call gets
. . .
```

call `_echo`:

```
. . .
4006be: callq 4006cf <echo>
4006c3: add $0x8,%rsp
. . .
```

Program “returned” to 0x0400600, and then crashed.

unix>./bufdemo-nsp
Type a string: 012345678901234567890123
012345678901234567890123
Segmentation fault
Stack Smashing Attacks

- Overwrite normal return address A with address of some other code S
- When Q executes `ret`, will jump to other code

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

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

void S() {
    /* Something unexpected */
    ...
}
```
Crafting Smashing String

Stack Frame for `call_echo`

```
%rsp
00 00 00 00
00 40 06 c3
```

Target Code

```c
void smash() {
    printf("I've been smashed!\n");
    exit(0);
}
```

```
00000000004006c8 <smash>:
4006c8:  48 83 ec 08
```

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```
Smashing String Effect

Stack Frame for `call_echo`

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>c8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>39</td>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>32</td>
<td>31</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

%rsp: `00000000004006c8`

Target Code

```c
void smash() {
    printf("I've been smashed!\n");
    exit(0);
}
```

```
00000000004006c8 <smash>:
4006c8:  48 83 ec 08
```

Attack String (Hex)

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
```

```
c8 06 40 00 00 00 00 00
```
Performing Stack Smash

- Put hex sequence in file smash-hex.txt
- Use hexify program to convert hex digits to characters
  - Some of them are non-printing
- Provide as input to vulnerable program

```c
void smash() {
    printf("I've been smashed!\n");
    exit(0);
}
```

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 c8 06 40 00 00 00 00 00
```
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
void P()
{
    Q();
    ...
}

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

Stack after call to `gets()`

- `P` stack frame
- `Q` stack frame
- Data written by `gets()`
- Pad
- Exploit code
- Stack after call to `gets()`

return address A
How Does The Attack Code Execute?

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

int Q()
{
    char buf[64];
    gets(buf); // A->B
    ...
    return ...;
}
```
What To Do About Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Let's talk about each...
1. Avoid Overflow Vulnerabilities in Code (!)

/* Echo Line */
void echo()
{
    char buf[4];
    fgets(buf, 4, stdin);
    puts(buf);
}

- 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
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

Stack base

Random allocation

main

Application Code

B?

pad

exploit code

B?
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

Stack after call to `gets()`

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
```

```shell
unix> ./bufdemo-sp
Type a string: 012345678
*** 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
400758: xor %fs:0x28,%rax
400761: je 400768 <echo+0x39>
400763: callq 400580 <__stack_chk_fail@plt>
400768: add $0x18,%rsp
40076c: retq
```

**Aside: \%fs:0x28**

- Read from memory using segmented addressing
- Segment is read-only
- Value generated randomly every time program runs
Setting Up Canary

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

Canary (8 bytes)

[3][2][1][0]

buf ← %rsp

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

Return Address (8 bytes)

Canary (8 bytes)

buf ← %rsp

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

After call to gets

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

Input: 0123456

Some systems:
LSB of canary is 0x00
Allows input 01234567

buf ← %rsp

echo:

```assembly
    ...  
    mov  0x8(%rsp),%rax    # Retrieve from stack
    xor %fs:0x28,%rax      # Compare to canary
    je    .L6             # If same, OK
    call __stack_chk_fail  # FAIL
```
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/10968
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;
}
```

Gadget address = 0x4004d4

- Use tail end of existing functions
Gadget Example #2

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

Gadget address = 0x4004dc

- Repurpose byte codes

Encodes `movq %rax, %rdi`

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

`rdi ← rax`

ROP Execution

- **Trigger with `ret` instruction**
  - Will start executing Gadget 1

- **Final `ret` in each gadget will start next one**
  - `ret`: pop address from stack and jump to that address
Crafting an ROP Attack String

<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 40 04 dc</td>
</tr>
<tr>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>00 40 04 d4</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>

%rsp

- **Gadget #1**
  - 0x4004d4 rax ← rdi + rdx

- **Gadget #2**
  - 0x4004dc rdi ← rax

- **Combination**
  - rdi ← rdi + rdx

---

**Attack String (Hex)**

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00 dc 04 40 00 00 00 00 00 00
```

Multiple gadgets will corrupt stack upwards
What Happens when `echo` returns?

<table>
<thead>
<tr>
<th>Stack Frame for call echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00 00 00 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>

1. **Echo executes `ret`**
   - Starts Gadget #1

2. **Gadget #1 executes `ret`**
   - Starts Gadget #2

3. **Gadget #2 executes `ret`**
   - Goes off somewhere ...

Multiple gadgets will corrupt stack upwards
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;
```

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

```
sp+0  sp+4  sp+8  sp+16  sp+24
```

- `c` occupies 3 bytes
- `i[0]` and `i[1]` each occupy 4 bytes
- `v` occupies 8 bytes
Using Union to Access Bit Patterns

```c
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?

### Memory addresses growing

#### 32-bit

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>

#### 64-bit

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
Byte Ordering Example (Cont).

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

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

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

printf("Long 0 == [0x%lx]\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>
</tbody>
</table>

| l[0] |

### Output on Sun:

- **Characters** 0–7 == `[0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]`
- **Shorts** 0–3 == `[0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]`
- **Ints** 0–1 == `[0xf0f1f2f3, 0xf4f5f6f7]`
- **Long** 0 == `[0xf0f1f2f3]`
Byte Ordering on x86-64

Little Endian

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

Output on x86-64:

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 `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.

- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet 😊)
    - see June 1989 article in *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.
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
- 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**