Machine-Level Representation: Advanced Topics

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

With Slides from Bryant, O’Hallaron
Announcement 2/24/2016

• Bomb Lab due 11:55pm Friday 2/26/2016

• First Mid-term Exam next Monday 2/29/2016 in class
  • Coverage Chapt 1 – Chapt 3
  • Open book, open notes
  • No e-devices

• How to prepare for the exams?
  • Go over all lecture slides, and make sure you understand all of the materials covered there.
  • UNITE has recorded lecture videos for review
    https://www.unite.umn.edu/
Outline

• Memory Layout
• Buffer Overflow
  • Vulnerability
  • Protection
Review: x86-64/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:"
    - Parameters for function about to call
  - Local variables
    - If can’t keep in registers
  - Saved register context
  - Old frame pointer (optional)

- Caller Stack Frame (ABI – Application Binary Interface)
  - Return address
    - Pushed by call instruction
  - Arguments for this call
x86-64 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)

- **Heap**
  - Dynamically allocated as needed
  - When call `malloc()`, `calloc()`, `new()`

- **Data**
  - **Statically** allocated data
  - E.g., `global` variables, `static` variables, string constants

- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only

Hex Address 400000 000000
Memory Allocation Example

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

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 $\sim 2^{47}$

<table>
<thead>
<tr>
<th>Local</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>local</code></td>
<td>0x00007ffe4d3be87c</td>
</tr>
<tr>
<td><code>p1</code></td>
<td>0x00007f7262a1e010</td>
</tr>
<tr>
<td><code>p3</code></td>
<td>0x00007f7162a1d010</td>
</tr>
<tr>
<td><code>p4</code></td>
<td>0x000000008359d120</td>
</tr>
<tr>
<td><code>p2</code></td>
<td>0x000000008359d010</td>
</tr>
<tr>
<td><code>big_array</code></td>
<td>0x0000000080601060</td>
</tr>
<tr>
<td><code>huge_array</code></td>
<td>0x0000000000601060</td>
</tr>
<tr>
<td><code>main()</code></td>
<td>0x000000000040060c</td>
</tr>
<tr>
<td><code>useless()</code></td>
<td>0x0000000000400590</td>
</tr>
</tbody>
</table>

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

```
Outline

• Memory Layout
• Buffer Overflow
  • Vulnerability
  • Protection
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;
    return s.d;
}

<table>
<thead>
<tr>
<th>fun(i)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(0)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(1)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(2)</td>
<td>3.1399998664856</td>
</tr>
<tr>
<td>fun(3)</td>
<td>2.00000061035156</td>
</tr>
<tr>
<td>fun(4)</td>
<td>3.14</td>
</tr>
<tr>
<td>fun(6)</td>
<td>Segmentation fault</td>
</tr>
</tbody>
</table>

Result is system specific
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;
    return s.d;
}

Warning:
C does not check array bound

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

Critical State

6
5
4
3
2
1
0

struct_t

Location accessed by fun(i)
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: 012345678901234567890123
012345678901234567890123
unix>./bufdemo-nsp
Type a string: 0123456789012345678901234
Segmentation Fault
```
Buffer Overflow Disassembly

echo:

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

00000000004006cf <echo>:
4006cf:  48 83 ec 18  subq $0x18,%rsp  # rsp-$24
4006d3:  48 89 e7  movq %rsp,%rdi
4006d6:  e8 a5 ff ff ff  callq 400680 <gets>
4006db:  48 89 e7  movq %rsp,%rdi
4006de:  e8 3d fe ff ff  callq 400520 <puts@plt>
4006e3:  48 83 c4 18  add $0x18,%rsp
4006e7:  c3  retq
```

call_echo:

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

00000000004006e8:  48 83 ec 08  subq $0x8,%rsp  # rsp-$8
00000000004006ec:  b8 00 00 00 00  movq $0x0,%eax
00000000004006f1:  e8 d9 ff ff ff  callq 4006cf <echo>
00000000004006f6:  48 83 c4 08  add $0x8,%rsp
00000000004006fa:  c3  retq
```
Buffer Overflow Stack

Before call to `gets`

Stack Frame for `call_echo`

Return Address (8 bytes)

20 bytes unused

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

void call_echo() {
    echo();
}

00000000004006cf <echo>:
4006cf:  subq $0x18,%rsp  # rsp-$24
4006d3:  movq %rsp,%rdi
4006d6:  callq 400680 <gets>
4006db:  movq %rsp,%rdi
4006de:  callq 400520 <puts@plt>
4006e3:  add $0x18,%rsp
4006e7:  retq
Buffer Overflow Stack Example

Before call to `gets`

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

call_echo:

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

stack frame for call_echo

00 00 00 00
00 40 06 f6

20 bytes unused

buf ↷ %rsp

4006e8: subq $0x8,%rsp # rsp-$8
4006ec: movq $0x0,%eax
4006f1: callq 4006cf <echo>
4006f6: add $0x8,%rsp
4006fa: retq
# Buffer Overflow Stack Example #1

After call to `gets`

<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 06 f6</td>
</tr>
<tr>
<td>00 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>

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

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

call_{echo}:
```asm
    add $0x8,%rsp
    ...
```

24 chars

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

buf ← %rsp

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

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

After call to gets

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:0123456789012345678901234
Segmentation Fault

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

After call to gets

Stack Frame for call_echo

<table>
<thead>
<tr>
<th>Buffer</th>
<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>00</td>
<td></td>
</tr>
<tr>
<td>33</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 $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

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

After call to gets

Stack Frame for 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>
<tr>
<td>00</td>
<td>40</td>
<td>06</td>
<td>00</td>
</tr>
<tr>
<td>33</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>

buf ← %rsp

register_tm_clones:

```
RO 00 00 00 00
RO 00 40 06 00
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30
```

“Returns” to unrelated code
Lots of things happen, without modifying critical state
Eventually executes retq back to main
Announcement 2/26/2016

• Bomb Lab due today 2/26/2016 at 11:55pm

• First Mid-term Exam next Monday 2/29/2016 in class
  • Coverage Chaps 1 – 3
  • Open book, open notes
  • No e-devices

• **How to prepare for the exams?**
  • Go over all lecture slides, and make sure you understand all of the materials covered there.
  • Go over Practice Problems as many as you can.
  • UNITE has recorded lecture videos for review
    
    https://www.unite.umn.edu/
Review: 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 ...;
}

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

Stack after call to `gets()`

- `P` stack frame
- `Q` stack frame
- `B` stack frame
- Data written by `gets()`
- Pad
- Exploit code

Return address

Return address A
Review: Buffer Overflow Stack

### After call to gets

<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 00 34</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>

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

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

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

Overflowed buffer and **corrupted return pointer**
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
OK, 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 (!)

```c
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    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
Review: char *Gets(*dest)

- 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
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
x86-64 Example Addresses

address range $\sim 2^{47}$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>0x00007ffe4d3be87c</td>
</tr>
<tr>
<td>p1</td>
<td>0x00007f7262a1e010</td>
</tr>
<tr>
<td>p3</td>
<td>0x00007f7162a1d010</td>
</tr>
<tr>
<td>p4</td>
<td>0x000000008359d120</td>
</tr>
<tr>
<td>p2</td>
<td>0x000000008359d010</td>
</tr>
<tr>
<td>big_array</td>
<td>0x0000000080601060</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0000000000601060</td>
</tr>
<tr>
<td>main()</td>
<td>0x000000000040060c</td>
</tr>
<tr>
<td>useless()</td>
<td>0x0000000000400590</td>
</tr>
</tbody>
</table>

```c
char big_array[1L<<24]; /* 16 MB global var */
char huge_array[1L<<31]; /* 2 GB global var */
int global = 0;
inuseless() { 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 ... */
```
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)

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

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

**Before call to gets**

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return Address (8 bytes)</td>
</tr>
<tr>
<td>Canary (8 bytes)</td>
</tr>
</tbody>
</table>

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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>40072f</td>
<td>sub $0x18,%rsp</td>
<td></td>
</tr>
<tr>
<td>400733</td>
<td>mov %fs:0x28,%rax</td>
<td></td>
</tr>
<tr>
<td>40073c</td>
<td>mov %rax,0x8(%rsp)</td>
<td></td>
</tr>
<tr>
<td>400741</td>
<td>xor %eax,%eax</td>
<td></td>
</tr>
<tr>
<td>400743</td>
<td>mov %rsp,%rdi</td>
<td></td>
</tr>
<tr>
<td>400746</td>
<td>callq 4006e0 &lt;gets&gt;</td>
<td></td>
</tr>
<tr>
<td>40074b</td>
<td>mov %rsp,%rdi</td>
<td></td>
</tr>
<tr>
<td>40074e</td>
<td>callq 400570 <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
</tr>
<tr>
<td>400753</td>
<td>mov 0x8(%rsp),%rax</td>
<td></td>
</tr>
<tr>
<td>400758</td>
<td>xor %fs:0x28,%rax</td>
<td></td>
</tr>
<tr>
<td>400761</td>
<td>je 400768 &lt;echo+0x39&gt;</td>
<td></td>
</tr>
<tr>
<td>400763</td>
<td>callq 400580 __stack_chk_fail@plt</td>
<td></td>
</tr>
<tr>
<td>400768</td>
<td>add $0x18,%rsp</td>
<td></td>
</tr>
<tr>
<td>40076c</td>
<td>retq</td>
<td></td>
</tr>
</tbody>
</table>
```

%fs, %gs, %cs, %ss, %ds, %es are segment registers.
Setting Up Canary

Before call to gets

<table>
<thead>
<tr>
<th>Stack Frame for call_echo</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>

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

echo:

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

buf ← %rsp
Checking Canary

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

Input: 0123456

stack frame for main

Input: 0123456

Stack Frame
for call_echo

Return Address
(8 bytes)

Canary
(8 bytes)

buf ← %rsp
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

Gadget Example #1

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

```
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

Gadget address = 0x4004d4

- Use tail end of existing functions
Gadget Example #2

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

Encodes `movq %rax, %rdi`

Gadget address = 0x4004dc

- Repurpose byte codes
The `ret` instruction does two things:
- It pops the entry pointed by `%rsp` to `%rip` (the next instruction to execute)
- It increments `%rsp` to the next stack top, which stores the address of next Gadget code

Final `ret` in each gadget will start next one. It can create a function to do almost anything

Outline

• Memory Layout

• Buffer Overflow
  • Vulnerability
  • Protection

• Union
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;
```
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

• 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

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

### 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></td>
<td>i[1]</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 == [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>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></td>
<td></td>
<td></td>
<td>i[1]</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 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