Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>Type</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>

Array Allocation

Basic Principle
- \( T \ A[L] \);
  - Array of data type \( T \) and length \( L \)
  - Contiguously allocated region of \( L \cdot \text{sizeof}(T) \) bytes

<table>
<thead>
<tr>
<th>char string[12];</th>
<th>x</th>
<th>x+1</th>
<th>x+2</th>
<th>x+3</th>
<th>x+4</th>
<th>x+5</th>
<th>x+6</th>
<th>x+7</th>
<th>x+8</th>
<th>x+9</th>
<th>x+10</th>
<th>x+11</th>
<th>x+12</th>
</tr>
</thead>
<tbody>
<tr>
<td>int val[5];</td>
<td>x</td>
<td>x+1</td>
<td>x+2</td>
<td>x+3</td>
<td>x+4</td>
<td>x+5</td>
<td>x+6</td>
<td>x+7</td>
<td>x+8</td>
<td>x+9</td>
<td>x+10</td>
<td>x+11</td>
<td>x+12</td>
</tr>
<tr>
<td>double a[4];</td>
<td>x</td>
<td>x+4</td>
<td>x+8</td>
<td>x+12</td>
<td>x+16</td>
<td>x+20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char *p[3];</td>
<td>x</td>
<td>x+4</td>
<td>x+8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Array Access

Basic Principle
- \( T \ A[L] \);
  - Array of data type \( T \) and length \( L \)
  - Identifier \( A \) can be used as a pointer to array element 0

<table>
<thead>
<tr>
<th>int val[5];</th>
<th>x</th>
<th>x+4</th>
<th>x+8</th>
<th>x+12</th>
<th>x+16</th>
<th>x+20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Type</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>val</td>
<td>int</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>val+1</td>
<td>int</td>
<td>x+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int</td>
<td>x+8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>val + i</td>
<td>int</td>
<td>x+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page 1
Document Page 2

**Array Example**

```
typedef int zip_dig[5];
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };    
zip_dig ucb = { 9, 4, 7, 2, 0 };    
```

Notes
- Declaration "zip_dig cmu" equivalent to "int cmu[5]"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

**Referencing Examples**

```
zip_dig cmu;
1 5 2 1 3
```

```
zip_dig mit;
0 2 1 3 9
```

```
zip_dig ucb;
9 4 7 2 0
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4* 3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4* 5 = 56</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

**Nested Array Example**

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
(1, 5, 2, 0, 6),
(1, 5, 2, 1, 7),
(1, 5, 2, 2, 1));
```

Notes
- Declaration "zip_dig pgh[4]" equivalent to "int pgh[4][5]"
  - Variable pgh denotes an array of 4 elements
    - Allocated contiguously
    - Each element is an array of 5 int's
      - Allocated contiguously
  - "Row-Major" ordering of all elements guaranteed

**Nested Array Allocation**

Declaration
```
T A[R][C];
```

- Array of data type T
- R rows, C columns
- Type T element requires K bytes

Array Size
```
R * C * K bytes
```

Arrangement
- Row-Major Ordering
```
A[0][0] ... A[0][C-1]  
...  
A[R-1][0] ... A[R-1][C-1]
```

```
4*R*C Bytes
```
Nested Array Row Access

- **Row Vectors**
  - $A[i]$ is array of $C$ elements
  - Each element of type $T$
  - Starting address $A + i \cdot C \cdot K$

  \[
  \text{int } A[R][C];
  \]

  \[
  \begin{array}{cccc}
  A[0] & \cdots & A[i] & \cdots \\
  \hline
  0 & \cdots & 0 & \cdots \\
  C-1 & \cdots & C-1 & \cdots \\
  \end{array}
  \]

  \[
  A + i \cdot C \cdot 4 \quad A + (R-1) \cdot C \cdot 4
  \]

Nested Array Element Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$
  - Address $A + (i \cdot C + j) \cdot K$

  \[
  \text{int } A[R][C];
  \]

  \[
  \begin{array}{cccc}
  A[0] & \cdots & A[i] & \cdots \\
  \hline
  0 & \cdots & 0 & \cdots \\
  C-1 & \cdots & C-1 & \cdots \\
  \end{array}
  \]

  \[
  A + i \cdot C \cdot 4 \quad A + (i \cdot C + j) \cdot 4
  \]

Multi-Level Array Example

- Variable `univ` denotes array of 3 elements
  - Each element is a pointer
  - 4 bytes
  - Each pointer points to array of int’s

  \[
  \text{zip} \_ \text{dig } \text{cmu} = \{1, 5, 2, 1, 3\};
  \text{zip} \_ \text{dig } \text{mit} = \{0, 2, 1, 3, 9\};
  \text{zip} \_ \text{dig } \text{ucb} = \{9, 4, 7, 2, 0\};
  \]

  \[
  \# \text{define UCOUNT } 3 \quad \text{int } *\text{univ}[\text{UCOUNT}] = (\text{mit}, \text{cmu}, \text{ucb});
  \]

Element Access in Multi-Level Array

- **Computation**
  - Element access
    - $\text{Mem}[\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig}]$
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

  \[
  \# \% \text{ecx} = \text{index} \quad \# \% \text{eax} = \text{dig}
  \text{leal } 0(,%\text{ecx},4),%\text{edx} \quad \# 4*\text{index}
  \text{movl } \text{univ}(%\text{edx}),%\text{eax} \quad \# \text{Mem}[\text{univ}+4*\text{index}]$

  \[
  \text{movl } (%\text{edx},%\text{eax},4),%\text{eax} \quad \# \% \text{Mem}[\ldots+4*\text{dig}]
  \]

  \[
  \text{int } \text{get_univ_digit}(\text{int } \text{index}, \text{int } \text{dig})
  \{
  \quad \text{return } \text{univ}[\text{index}][\text{dig}];
  \}
  \]
Array Element Accesses

- Similar C references
- Different address computation

**Nested Array**

int get_pgh_digit
(int index, int dig)
{
    return pgh[index][dig];
}

**Element at**

Mem[pgh+20*index+4*dig]

**Multi-Level Array**

int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}

**Element at**

Mem[Mem[univ+4*index]+4*dig]

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ[2][3]</td>
<td>56+4*3 = 68</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+4*5 = 36</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>univ[2][-1]</td>
<td>56+4*-1 = 52</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>univ[3][-1]</td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td>univ[1][12]</td>
<td>16+4*12 = 64</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

Structures

**Concept**

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

**Accessing Structure Member**

```
void set_i(struct rec *r, int val)
{
    r->i = val;
}
```

**Memory Layout**

```
  i  a  p
0  4  16  20
```

**Generating Pointer to Struct. Member**

```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

```
int *find_a (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

**Generating Pointer to Array Element**

- Offset of each structure member determined at compile time

```
int *find_a (struct rec *r, int idx)
{
    return &r->a[idx];
}
```

```
# %edx = r
leal 0(%eax,4),%eax # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```
Structure Referencing (Cont.)

C Code

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```

```c
void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

Alignment

Aligned Data
- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data
- Memory accessed by (aligned) double or quad-words
  - inefficient to load or store datum that spans quad word boundaries
  - virtual memory very tricky when datum spans 2 pages

Compiler
- inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment

Size of Primitive Data Type:
- 1 byte (e.g., char)
  - no restrictions on address
- 2 bytes (e.g., short)
  - lowest 1 bit of address must be 0
- 4 bytes (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00
- 8 bytes (e.g., double)
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type
- 12 bytes (long, double)
  - Windows:
    - lowest 2 bits of address must be 00
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type

Satisfying Alignment with Structures

Offsets Within Structure
- Must satisfy element’s alignment requirement

Overall Structure Placement
- Each structure has alignment requirement K
- Largest alignment of any element
- Initial address & structure length must be multiples of K

Example (under Windows):
- K = 8, due to double element

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

```
<table>
<thead>
<tr>
<th>C</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
</tr>
</tbody>
</table>
```

- Multiple of 4
- Multiple of 8
- Multiple of 8
Linux vs. Windows

Windows (including Cygwin):
- K = 8, due to double element

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
</tr>
</tbody>
</table>
  Multiple of 4   Multiple of 8   Multiple of 8

Linux:
- K = 4; double treated like a 4-byte data type

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
</tr>
</tbody>
</table>
  Multiple of 4   Multiple of 4   Multiple of 4

Overall Alignment Requirement

Windows: p+24
Linux: p+20

p must be multiple of:
- 8 for Windows
- 4 for Linux

Ordering Elements Within Structure

struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;

<table>
<thead>
<tr>
<th>c1</th>
<th>v</th>
<th>c2</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+8</td>
<td>p+16</td>
<td>p+20</td>
</tr>
</tbody>
</table>

10 bytes wasted space in Windows

struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;

<table>
<thead>
<tr>
<th>v</th>
<th>c1</th>
<th>c2</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+8</td>
<td>p+12</td>
<td>p+16</td>
</tr>
</tbody>
</table>

2 bytes wasted space

Arrays of Structures

Principle
- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

struct S6 {
    short i;
    float v;
    short j;
} a[10];

<table>
<thead>
<tr>
<th>i</th>
<th>v</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>a[1]</td>
<td>a[2]</td>
</tr>
</tbody>
</table>

• • •

allocated by repeating allocation for array type
In general, may nest arrays & structures to arbitrary depth
Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be a multiple of worst-case alignment for any element
  - `a` must be multiple of 4
- Offset of element within structure must be a multiple of element’s alignment requirement
  - `v`’s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```

Union Allocation

Principles

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

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

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;
}
```

Byte Ordering Revisited

Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Most significant byte has lowest address
- PowerPC, Sparc

Little Endian

- Least significant byte has lowest address
- Intel x86, Alpha
Byte Ordering Example

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

c[3]  s[1]  i[0]
c[2]  c[1]  s[0]  c[0]

Byte Ordering Example (Cont).

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 x86

Little Endian

f0  f1  f2  f3  f4  f5  f6  f7
  MSB  MSB  MSB  MSB  LSB  LSB  LSB  LSB
  i[0]  i[1]
  MSB  LSB

Output on Pentium:
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints 0-1 == [0xf3f2f1f0,0xf6f5f4f3]
Long 0 == [f3f2f1f0]

Byte Ordering on Sun

Big Endian

f0  f1  f2  f3  f4  f5  f6  f7
  LSB  LSB  LSB  LSB  MSB  MSB  MSB  MSB
  i[0]  i[1]
  MSB  LSB

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 == [f0f1f2f3]
Byte Ordering on Alpha

Little Endian

\[
\begin{array}{ccccccc}
\text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} \\
\text{c}[0] & \text{c}[1] & \text{c}[2] & \text{c}[3] & \text{c}[4] & \text{c}[5] & \text{c}[6] & \text{c}[7] \\
\end{array}
\]

LSB MSB LSB MSB LSB MSB LSB MSB

\[
\begin{array}{cccc}
\text{s} & \text{s} & \text{s} & \text{s} \\
\text{s}[0] & \text{s}[1] & \text{s}[2] & \text{s}[3] \\
\end{array}
\]

Print

Output on Alpha:

- Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
- Ints 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
- Longs 0 == [0xf7f6f5f4f3f2f1f0]

Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations

- Compiler often turns array code into pointer code (zd2int)
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions

- Overlay declarations
- Way to circumvent type system

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
- 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?

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

```
/* 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

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

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

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

Buffer Overflow Executions

```
unix> ./bufdemo
Type a string:123
123
```

```
unix> ./bufdemo
Type a string:12345
Segmentation Fault
```

```
unix> ./bufdemo
Type a string:12345678
Segmentation Fault
```

Buffer Overflow Stack

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

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

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

```
echo:
    pushl %ebp # Save %ebp on stack
    movl %esp,%ebp # Allocate space on stack
    subl $20,%esp # Allocate space on stack
    pushl %ebx
    addl $-12,%esp # Allocate space on stack
    leal -4(%ebp),%ebx # Compute buf as %ebp-4
    pushl %ebx
    call gets # Call gets
    ...
Buffer Overflow Stack Example

Before call to `gets`

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x8048583
(gdb) run
```

```
Breakpoint 1, 0x8048583 in echo ()
```

```
print /x *(unsigned *)$ebp
$1 = 0xbffff8f8
```

```
print /x *((unsigned *)$ebp + 1)
$3 = 0x804864d
```

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

### Buffer Overflow Example #1

**Input = “123”**

```
echo code:
```

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

**Bad news when later attempt to restore %ebp**

- Saved value of %ebp set to 0xbfff0035
- %ebp and return address corrupted

---

**Input = “12453”**

```
echo code:
```

```
8048592: push %ebx
8048593: call 80483e4 <_init+0x50> # gets
8048598: mov 0xffffffe8(%ebp),%ebx
804859b: mov %ebp,%esp
804859d: pop %ebp
804859e: ret
```

---

**Input = “12345678”**

```
echo code:
```

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

**Invalid address**

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

---

---

---
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.

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.

Disclaimer

Parts (most in this lecture) of the slides were developed by the course text authors: Dave O'Hallaron and Randy Bryant. The slides are intended for the sole purpose of instruction of computer organization at the University of Rochester. All copyrighted materials belong to their original owner(s).