Machine-Level Programming: Basic Data Types

Fall 2012

Instructors: Aykut & Erkut Erdem

Acknowledgement: The course slides are adapted from the slides prepared by R.E. Bryant, D.R. O’Hallaron, G. Kesden and Markus Püschel of Carnegie-Mellon Univ.
Last time

- Control: Condition codes
- Conditional branches
- Loops
- Switch statements
- IA 32 Procedures
  - Stack Structure
  - Calling Conventions
  - Illustrations of Recursion & Pointers
Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structures**

- **Unions**

- **Memory Layout**

- **Buffer Overflow**

- **Floating Point**
Basic Data Types

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

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</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>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long int (x86-64)</td>
</tr>
</tbody>
</table>

- Floating Point
  - Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</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>10/12/16</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array Allocation

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

```
char string[12];  
```

```
int val[5];  
```

```
double a[3];  
```

```
char *p[3];  
```

IA32

x86-64
Array Access

- **Basic Principle**
  
  \( T \text{ A}[L] \);

  - Array of data type \( T \) and length \( L \)
  - Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

  ```
  int val[5];
  ```

- **Reference**

```plaintext
<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>x + 4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + 4 i</td>
</tr>
</tbody>
</table>
```
## Array Example

```c
#define ZLEN 5
typedef int zip_dig[ZLEN];

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

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

```c
int get_digit (zipDig z, int dig)
{
    return z[dig];
}
```

IA32

```assembly
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`
Referencing Examples

- **zip_dig cmu;**
  - 1 5 2 1 3
  - 16 20 24 28 32 36

- **zip_dig mit;**
  - 0 2 1 3 9
  - 36 40 44 48 52 56

- **zip_dig ucb;**
  - 9 4 7 2 0
  - 56 60 64 68 72 76

<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>No</td>
</tr>
</tbody>
</table>

- No bound checking
- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
Array Loop Example (IA32)

```c
void zincr(zip_digit z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```asm
# edx = z
movl  $0, %eax  # %eax = i
.L4:
    # loop:
    addl  $1, (%edx,%eax,4)  # z[i]++
    addl  $1, %eax          # i++
    cmpl  $5, %eax          # i:5
    jne   .L4              # if !==, goto loop
```
**Pointer Loop Example (IA32)**

```c
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

```c
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;
  } while (i != ISIZE*ZLEN);
}
```

```
# edx = z = vz
movl  $0, %eax  # i = 0
.L8:         # loop:
  addl  $1, (%edx,%eax)  # Increment vz+i
  addl  $4, %eax  # i += 4
  cmpl  $20, %eax  # Compare i:20
  jne   .L8  # if !=, goto loop
```
Nested Array Example

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},,
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

- “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
  - Variable pgh: array of 4 elements, allocated contiguously
  - Each element is an array of 5 int’s, allocated contiguously
- “Row-Major” ordering of all elements guaranteed
Multidimensional (Nested) Arrays

- **Declaration**
  
  \[ \begin{array}{c}
  \text{T } A[R][C] ; \\
  \text{2D array of data type } T \\
  \text{R rows, C columns} \\
  \text{Type } T \text{ element requires } K \text{ bytes}
  \end{array} \]

- **Array Size**
  
  \[ \begin{array}{c}
  \text{R } \times \text{C } \times \text{K bytes}
  \end{array} \]

- **Arrangement**
  
  - Row-Major Ordering

```c
int A[R][C];
```
Nested Array Row Access

**Row Vectors**
- \(A[i]\) is array of \(C\) elements
- Each element of type \(T\) requires \(K\) bytes
- Starting address \(A + i \times (C \times K)\)

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

![Diagram of nested array row access](attachment:image.png)
Nested Array Row Access Code

```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```

```c
#define PCOUNT 4
zipDig pgh[PCOUNT] = {
    {1, 5, 2, 0, 6},
    {1, 5, 2, 1, 3},
    {1, 5, 2, 1, 7},
    {1, 5, 2, 2, 1}};
```

```c
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(%eax,4),%eax # pgh + (20 * index)
```

**Row Vector**
- `pgh[index]` is array of 5 int’s
- Starting address `pgh+20*index`

**IA32 Code**
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`
Nested Array Row Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$, which requires $K$ bytes
  - Address $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

```c
int A[R][C];
```

![Diagram of array access](image)
Nested Array Element Access Code

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

```assembly
movl 8(%ebp), %eax       # index
leal (%eax,%eax,4), %eax  # 5*index
addl 12(%ebp), %eax       # 5*index+dig
movl pgh(,%eax,4), %eax   # offset 4*(5*index+dig)
```

- **Array Elements**
  - `pgh[index][dig]` is int
  - Address: `pgh + 20*index + 4*dig`
    - `= pgh + 4*(5*index + dig)`

- **IA32 Code**
  - Computes address `pgh + 4*((index+4*index)+dig)`
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

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```
### Strange Referencing Examples

```c
zip_dig pgh[4];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Element Access in Multi-Level Array

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

- **Computation (IA32)**
  - Element access `Mem[Mem[univ+4*index]+4*dig]`
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

```assembly
movl  8(%ebp), %eax          # index
movl univ(,%eax,4), %edx    # p = univ[index]
movl  12(%ebp), %eax         # dig
movl  (%edx,%eax,4), %eax   # p[dig]
```
Array Element Accesses

Nested array

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

Multi-level array

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

Accesses looks similar in C, but addresses very different:

Mem[pgh+20*index+4*dig]  Mem[Mem[univ+4*index]+4*dig]
Strange Referencing Examples

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

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

- **Fixed dimensions**
  - Know value of N at compile time

- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays

- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```c
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
  (fix_matrix a, int i, int j)
{
    return a[i][j];
}

#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
  (int n, int *a, int i, int j)
{
    return a[IDX(n,i,j)];
}

/* Get element a[i][j] */
int var_ele
  (int n, int a[n][n], int i, int j)
{
    return a[i][j];
}
```
16 X 16 Matrix Access

- **Array Elements**
  - Address $A + i \times (C \times K) + j \times K$
  - $C = 16, K = 4$

```c
/* Get element $a[i][j]$ */
int fix_ele(fix_matrix a, int i, int j) {
    return a[i][j];
}
```

```assembly
movl 12(%ebp), %edx    # i
sall $6, %edx          # i*64
movl 16(%ebp), %eax    # j
sall $2, %eax          # j*4
addl 8(%ebp), %eax     # a + j*4
movl (%eax,%edx), %eax # *(a + j*4 + i*64)
```
n X n Matrix Access

- **Array Elements**
  - Address A + i * (C * K) + j * K
  - C = n, K = 4

```c
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
    return a[i][j];
}
```

```assembly
movl 8(%ebp), %eax      # n
sall $2, %eax           # n*4
movl %eax, %edx         # n*4
imull 16(%ebp), %edx    # i*n*4
movl 20(%ebp), %eax     # j
sall $2, %eax           # j*4
addl 12(%ebp), %eax     # a + j*4
movl (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```
## Optimizing Fixed Array Access

### Computation
- Step through all elements in column \( j \)

### Optimization
- Retrieving successive elements from single column

```c
#define N 16
typedef int fix_matrix[N][N];

/* Retrieve column \( j \) from array */
void fix_column
    (fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```
Optimizing Fixed Array Access

- **Optimization**
  - Compute \( \text{ajp} = \&a[i][j] \)
    - Initially = \( a + 4*j \)
    - Increment by \( 4*N \)

---

```c
/* Retrieve column j from array */
#define ajp (fix_matrix a, int j, int *dest)
{
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```

---

## Register Value Table

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>ajp</td>
</tr>
<tr>
<td>%ebx</td>
<td>dest</td>
</tr>
<tr>
<td>%edx</td>
<td>i</td>
</tr>
</tbody>
</table>

---

```assembly
.L8:  
    movl (%ecx), %eax          # Read *ajp
    movl %eax, (%ebx,%edx,4)   # Save in dest[i]
    addl $1, %edx              # i++
    addl $64, %ecx             # ajp += 4*N
    cmpl $16, %edx             # i:N
    jne .L8                   # if !=, goto loop
```
Optimizing Variable Array Access

- Compute \( ajp = &a[i][j] \)
  - Initially = \( a + 4*j \)
  - Increment by \( 4*n \)

```c
/* Retrieve column j from array */
void var_column
    (int n, int a[n][n],
     int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>ajp</td>
</tr>
<tr>
<td>%edi</td>
<td>dest</td>
</tr>
<tr>
<td>%edx</td>
<td>i</td>
</tr>
<tr>
<td>%ebx</td>
<td>4*n</td>
</tr>
<tr>
<td>%esi</td>
<td>n</td>
</tr>
</tbody>
</table>

```
.L18:     # loop:
    movl (%ecx), %eax    # Read *ajp
    movl %eax, (%edi,%edx,4)    # Save in dest[i]
    addl $1, %edx    # i++
    addl $ebx, %ecx    # ajp += 4*n
    cmpl $edx, %esi    # n:i
    jg .L18    # if >, goto loop
```
Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structures**
  - Allocation
  - Access
  - Alignment

- **Unions**
- **Memory Layout**
- **Buffer Overflow**
- **Floating Point**
Structure Allocation

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

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

![Memory Layout Diagram]
## Structure Access

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

### Accessing Structure Member
- Pointer indicates first byte of structure
- Access elements with offsets

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

### IA32 Assembly
```
# %edx = val
# %eax = r
movl %edx, 12(%eax) # Mem[r+12] = val
```
Generating Pointer to Structure Member

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

- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
  - Arguments
    - Mem[%ebp+8]: r
    - Mem[%ebp+12]: idx

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

```
movl  12(%ebp), %eax    # Get idx
sal1  $2, %eax          # idx*4
addl  8(%ebp), %eax     # r+idx*4
```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->n;
  }
}

.L17:
  # loop:
  movl 12(%edx), %eax  # r->i
  movl %ecx, (%edx,%eax,4) # r->a[i] = val
  movl 16(%edx), %edx  # r = r->n
  testl %edx, %edx  # Test r
  jne .L17  # If != 0 goto loop

struct rec {
  int a[3];
  int i;
  struct rec *n;
};
Alignment

- Unaligned Data

```
c  i[0]  i[1]  v
p   p+1  p+5  p+9  p+17
```

- Aligned Data
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

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

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - 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 (IA32)

- **1 byte: char, ...**
  - no restrictions on address

- **2 bytes: short, ...**
  - lowest 1 bit of address must be $0_2$

- **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be $00_2$

- **8 bytes: double, ...**
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be $000_2$
  - Linux:
    - lowest 2 bits of address must be $00_2$
    - i.e., treated the same as a 4-byte primitive data type

- **12 bytes: long double**
  - Windows, Linux:
    - lowest 2 bits of address must be $00_2$
    - i.e., treated the same as a 4-byte primitive data type
Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address

- **2 bytes: short, ...**
  - lowest 1 bit of address must be 0₂

- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be 00₂

- **8 bytes: double, char *, ...**
  - Windows & Linux:
    - lowest 3 bits of address must be 000₂

- **16 bytes: long double**
  - Linux:
    - lowest 3 bits of address must be 000₂
    - i.e., treated the same as a 8-byte primitive data type
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K

- **Example (under Windows or x86-64):**
  - K = 8, due to double element

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

- **x86-64 or IA32 Windows:**
  - $K = 8$, due to `double` element

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

- **IA32 Linux**
  - $K = 4$; `double` treated like a 4-byte data type
Meeting Overall Alignment Requirement

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

- **Compute array offset 12i**
  - sizeof(S3), including alignment spacers

- **Element j is at offset 8 within structure**

- **Assembler gives offset a+8**
  - Resolved during linking

```c
short get_j(int idx) {
    return a[idx].j;
}
```

```assembly
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(%eax,4),%eax
```
**Saving Space**

- **Put large data types first**

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

- **Effect (K=4)**

```c
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

| i | c | d | 2 bytes |
Today

- Arrays
- Structures
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Floating Point
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 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
## Byte Ordering Example

```c
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></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>
<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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></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>
<td></td>
</tr>
</tbody>
</table>
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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]`

**Print**

LSB → MSB → LSB → MSB
# 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></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>

### Print

#### 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>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 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

- **Arrays in C**
  - 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
Today

- Arrays
- Structures
- Unions
- Memory Layout
  - Vulnerability
  - Protection
- Buffer Overflow
- Floating Point
IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables

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

- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code

- **Text**
  - Executable machine instructions
  - Read-only

Upper 2 hex digits = 8 bits of address

Not drawn to scale
Memory Allocation Example

char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28); /* 256 MB */
p2 = malloc(1 << 8); /* 256 B */
p3 = malloc(1 <<28); /* 256 MB */
p4 = malloc(1 << 8); /* 256 B */
/* Some print statements ... */
}

Where does everything go?
IA32 Example Addresses

address range \( \sim 2^{32} \)

\[
\begin{align*}
\text{$esp$} & : 0xffffbcd0 \\
p3 & : 0x65586008 \\
p1 & : 0x55585008 \\
p4 & : 0x1904a110 \\
p2 & : 0x1904a008 \\
&p2 & : 0x18049760 \\
&beyond & : 0x08049744 \\
big\_array & : 0x18049780 \\
huge\_array & : 0x08049760 \\
main() & : 0x080483c6 \\
useless() & : 0x08049744 \\
final\ malloc() & : 0x006be166
\end{align*}
\]

`malloc()` is dynamically linked
address determined at runtime
x86-64 Example Addresses

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rsp</td>
<td>0x00007fffffff8d1f8</td>
</tr>
<tr>
<td>p3</td>
<td>0x00002aaabaadd010</td>
</tr>
<tr>
<td>p1</td>
<td>0x00002aaaaaadc010</td>
</tr>
<tr>
<td>p4</td>
<td>0x0000000011501120</td>
</tr>
<tr>
<td>p2</td>
<td>0x0000000011501010</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x00000000010500a60</td>
</tr>
<tr>
<td>&amp;beyond</td>
<td>0x00000000000500a44</td>
</tr>
<tr>
<td>big_array</td>
<td>0x0000000010500a80</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0000000000500a50</td>
</tr>
<tr>
<td>main()</td>
<td>0x00000000000400510</td>
</tr>
<tr>
<td>useless()</td>
<td>0x00000000000400500</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x000000386ae6a170</td>
</tr>
</tbody>
</table>

malloc() is dynamically linked
address determined at runtime
Today

- Arrays
- Structures
- Structures
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Floating Point
Internet Worm and IM War

- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
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
Internet Worm and 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.
  - How did it happen?

- The Internet Worm and AOL/Microsoft War were both based on *stack buffer overflow* exploits!
  - many library functions do not check argument sizes.
  - allows target buffers to overflow.
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();
}
```

unix> ./bufdemo
Type a string: 123
1234567

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

unix> ./bufdemo
Type a string: 12345678
Segmentation Fault
Buffer Overflow Disassembly

echo:

```
80485c5:  55       push   %ebp
80485c6:  89 e5    mov    %esp,%ebp
80485c8:  53       push   %ebx
80485c9:  83 ec 14  sub    $0x14,%esp
80485cc:  8d 5d f8  lea    0xfffffff8(%ebp),%ebx
80485cf:  89 1c 24  mov    %ebx,(%esp)
80485d2:  e8 9e ff ff ff  call   8048575 <gets>
80485d7:  89 1c 24  mov    %ebx,(%esp)
80485da:  e8 05 fe ff ff  call   80483e4 <puts@plt>
80485df:  83 c4 14  add    $0x14,%esp
80485e2:  5b       pop    %ebx
80485e3:  5d       pop    %ebp
80485e4:  c3       ret
```

call_echo:

```
80485eb:  e8 d5 ff ff ff  call   80485c5 <echo>
80485f0:  c9       leave
80485f1:  c3       ret
```
Buffer Overflow Stack

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx

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

Stack Frame for echo

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

echo:
pushl %ebp       # Save %ebp on stack
movl %esp, %ebp  # Save %esp
pushl %ebx       # Save %ebx
subl $20, %esp   # Allocate stack space
leal -8(%ebp),%ebx # Compute buf as %ebp-8
movl %ebx, (%esp) # Push buf on stack
call gets        # Call gets
...
Buffer Overflow
Stack Example

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
[3][2][1][0]

Stack Frame for echo

Before call to gets

Stack Frame for main

08 04 85 f0
ff ff d6 88

Saved %ebx

Stack Frame for echo

xx xx xx xx

buf 0xfffffd688

buf 0xfffffd678

unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xfffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xfffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0

80485eb: e8 d5 ff ff ff
80485f0: c9
Buffer Overflow Example #1

Before call to `gets`

Stack Frame for `main`

<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>

Saved `%ebx`

| xx | xx | xx | xx |

Stack Frame for `echo`

0xffffd688

Input 1234567

Stack Frame for `main`

<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00</th>
<th>37</th>
<th>36</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

Saved `%ebx`

Stack Frame for `echo`

0xffffd678

Overflow buf, and corrupt `%ebx`, but no problem
Buffer Overflow Example #2

Before call to gets

Stack Frame for main

08 04 85 f0
ff ff d6 88

Saved %ebx

Stack Frame for echo

xx xx xx xx

Input 12345678

Stack Frame for main

08 04 85 f0
ff ff d6 00

0xffffd678

Stack Frame for echo

38 37 36 35
34 33 32 31

buf

Base pointer corrupted

80485eb: e8 d5 ff ff ff call 80485c5 <echo>
80485f0: c9 leave # Set %ebp to corrupted value
80485f1: c3 ret
Buffer Overflow Example #3

Before call to gets

Stack Frame for main

```
<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>ff</td>
<td>d6</td>
<td>88</td>
</tr>
</tbody>
</table>
```

Saved %ebx

```
| xx | xx | xx | xx |
```

Stack Frame for echo

```
buf
```

Input 123456789

Stack Frame for main

```
<table>
<thead>
<tr>
<th>08</th>
<th>04</th>
<th>85</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>42</td>
<td>41</td>
<td>39</td>
</tr>
<tr>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
</tr>
</tbody>
</table>

Stack Frame for echo

```
| 34 | 33 | 32 | 31 |
```

Return address corrupted

```
80485eb:   e8 d5 ff ff ff   call   80485c5 <echo>
80485f0:   c9               leave   # Desired return point
```
Malicious Use of Buffer Overflow

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When \texttt{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.
Exploits Based on Buffer Overflows

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

- **IM War**
  - AOL exploited existing buffer overflow bug in AIM clients
  - exploit code: returned 4-byte signature (the bytes at some location in the AIM client) to server.
  - When Microsoft changed code to match signature, AOL changed signature location.
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!**
Code Red Exploit Code

- Starts 100 threads running
- Spread self
  - Generate random IP addresses & send attack string
  - Between 1st & 19th of month
- Attack www.whitehouse.gov
  - Send 98,304 packets; sleep for 4-1/2 hours; repeat
    - Denial of service attack
  - Between 21st & 27th of month
- Deface server’s home page
  - After waiting 2 hours

Welcome to http://www.worm.com!

Hacked By Chinese!
Avoiding Overflow Vulnerability

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

- Use library routines that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - Don’t use `scanf` with `%s` conversion specification
    - Use `fgets` to read the string
    - Or use `%ns` where `n` is a suitable integer
System-Level Protections

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code

- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either “read-only” or “writeable”
    - Can execute anything readable
  - X86-64 added explicit “execute” permission

```
unix> gdb bufdemo (gdb) break echo (gdb) run (gdb) print /x $ebp $1 = 0xfffffc638 (gdb) run (gdb) print /x $ebp $2 = 0xfffffbb08 (gdb) run (gdb) print /x $ebp $3 = 0xfffffc6a8
```
Stack Canaries

■ Idea
  ▪ Place special value (“canary”) on stack just beyond buffer
  ▪ Check for corruption before exiting function

■ GCC Implementation
  ▪ `-fstack-protector`
  ▪ `-fstack-protector-all`

```bash
unix>./bufdemo-protected
Type a string:1234
1234
```

```bash
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```
### Protected Buffer Disassembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Instructions</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>804864d:</td>
<td>push %ebp</td>
<td></td>
</tr>
<tr>
<td>804864e:</td>
<td>mov %esp,%ebp</td>
<td></td>
</tr>
<tr>
<td>8048650:</td>
<td>push %ebx</td>
<td></td>
</tr>
<tr>
<td>8048651:</td>
<td>sub $0x14,%esp</td>
<td></td>
</tr>
<tr>
<td>8048654:</td>
<td>mov %gs:0x14,%eax</td>
<td></td>
</tr>
<tr>
<td>804865a:</td>
<td>mov %eax,0xfffffffffffffff8(%ebp)</td>
<td></td>
</tr>
<tr>
<td>804865d:</td>
<td>xor %eax,%eax</td>
<td></td>
</tr>
<tr>
<td>804865f:</td>
<td>lea 0xfffffffffffffff4(%ebp),%ebx</td>
<td></td>
</tr>
<tr>
<td>8048662:</td>
<td>mov %ebx,(%esp)</td>
<td></td>
</tr>
<tr>
<td>8048665:</td>
<td>call 80485e1 &lt;gets&gt;</td>
<td></td>
</tr>
<tr>
<td>804866a:</td>
<td>call 804843c <a href="mailto:puts@plt">puts@plt</a></td>
<td></td>
</tr>
<tr>
<td>8048672:</td>
<td>mov 0xfffffffffffffff8(%ebp),%eax</td>
<td></td>
</tr>
<tr>
<td>8048675:</td>
<td>xor %gs:0x14,%eax</td>
<td></td>
</tr>
<tr>
<td>804867c:</td>
<td>je 8048683 &lt;echo+0x36&gt;</td>
<td></td>
</tr>
<tr>
<td>804867e:</td>
<td>call 804842c &lt;FAIL&gt;</td>
<td></td>
</tr>
<tr>
<td>8048683:</td>
<td>add $0x14,%esp</td>
<td></td>
</tr>
<tr>
<td>8048686:</td>
<td>pop %ebx</td>
<td></td>
</tr>
<tr>
<td>8048687:</td>
<td>pop %ebp</td>
<td></td>
</tr>
<tr>
<td>8048688:</td>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Setting Up Canary

Before call to gets

Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
Canary
[3][2][1][0]
Stack Frame for echo

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

echo:
....
movl %gs:20, %eax  # Get canary
movl %eax, -8(%ebp)  # Put on stack
xorl %eax, %eax   # Erase canary
....
Checking Canary

**Before call to gets**

Stack Frame for **main**

Return Address

Saved `%ebp`

Saved `%ebx`

Canary

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

Stack Frame for **echo**

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

**echo:**

...  

```asm
    movl -8(%ebp), %eax   # Retrieve from stack
    xorl %gs:20, %eax     # Compare with Canary
    je .L24               # Same: skip ahead
    call __stack_chk_fail # ERROR
.L24:
    ...
```
Canary Example

**Before call to gets**

```
Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
03 e3 7d 00
[3] [2] [1] [0]
```

```
Stack Frame for echo
```

**Input 1234**

```
Stack Frame for main

Return Address
Saved %ebp
Saved %ebx
03 e3 7d 00
34 33 32 31
```

```
Stack Frame for echo
```

```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

Benign corruption!
(allow programs to make silent off-by-one errors)
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**
  - Add itself to other programs
  - Cannot run independently

- Both are (usually) designed to spread among computers and to wreak havoc
Reading Assignment

Smashing the Stack for Fun and Profit, Aleph One