Machine-Level Programming: x86-64 Procedures, Basic Data Types

Spring 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.
Today

- Procedures (x86-64)
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- Floating point
  - x87 (available with IA32, becoming obsolete)
  - SSE3 (available with x86-64)
x86-64 Integer Registers

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits
### x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>%rax</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>Argument #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
x86-64 Registers

- Arguments passed to functions via registers
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well

- All references to stack frame via stack pointer
  - Eliminates need to update `%ebp`/%`rbp`

- Other Registers
  - 6 callee saved
  - 2 caller saved
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)
**x86-64 Long Swap**

```c
void swap_l(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- **Operands passed in registers**
  - First (`xp`) in `%rdi`, second (`yp`) in `%rsi`
  - 64-bit pointers
- **No stack operations required (except `ret`)**
- **Avoiding stack**
  - Can hold all local information in registers

**swap:**
```
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
```
x86-64 Locals in the Red Zone

/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}

Avoiding Stack Pointer Change
- Can hold all information within small window beyond stack pointer

swap_a:
    movq (%rdi), %rax
    movq %rax, -24(%rsp)
    movq (%rsi), %rax
    movq %rax, -16(%rsp)
    movq -16(%rsp), %rax
    movq %rax, (%rdi)
    movq -24(%rsp), %rax
    movq %rax, (%rsi)
    ret

rtn Ptr

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>unused</td>
</tr>
<tr>
<td>-16</td>
<td>loc[1]</td>
</tr>
<tr>
<td>-24</td>
<td>loc[0]</td>
</tr>
</tbody>
</table>

%rsp
x86-64 NonLeaf without Stack Frame

```c
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i) {
    swap(&a[i], &a[i+1]);
}
```

- No values held while swap being invoked
- No callee save registers needed
- `rep` instruction inserted as no-op
  - Based on recommendation from AMD

```
swap_ele:
    movslq %esi,%rsi       # Sign extend i
    leaq 8(%rdi,%rsi,8), %rax  # &a[i+1]
    leaq (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)
    movq %rax, %rsi         # (2nd arg)
    call swap
    rep
    ret
```
x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
    (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of &a[i] and &a[i+1] in callee save registers
- Must set up stack frame to save these registers

```
swap_ele_su:
    movq   %rbx, -16(%rsp)
    movq   %rbp, -8(%rsp)
    subq   $16, %rsp
    movslq  %esi,%rax
    leaq   8(%rdi,%rax,8), %rbx
    leaq   (%rdi,%rax,8), %rbp
    movq   %rbx, %rsi
    movq   %rbp, %rdi
    call   swap
    movq   (%rbx), %rax
    imulq  (%rbp), %rax
    addq   %rax, sum(%rip)
    movq   (%rsp), %rbx
    movq   8(%rsp), %rbp
    addq   $16, %rsp
    ret
```
Understanding x86-64 Stack Frame

swap_ele_su:

```
movq   %rbx, -16(%rsp)          # Save %rbx
movq   %rbp, -8(%rsp)           # Save %rbp
subq   $16, %rsp               # Allocate stack frame
movslq %esi,%rax               # Extend i
leaq   8(%rdi,%rax,8), %rbx    # &a[i+1] (callee save)
leaq   (%rdi,%rax,8), %rbp     # &a[i]   (callee save)
movq   %rbx, %rsi              # 2nd argument
movq   %rbp, %rdi              # 1st argument
call   swap                    # Get a[i+1]
imulq (%rbp), %rax            # Multiply by a[i]
addq   %rax, sum(%rip)         # Add to sum
movq   (%rsp), %rbx            # Restore %rbx
movq   8(%rsp), %rbp           # Restore %rbp
addq   $16, %rsp               # Deallocate frame
ret
```
Understanding x86-64 Stack Frame

```assembly
movq  %rbx, -16(%rsp)  # Save %rbx
movq  %rbp, -8(%rsp)   # Save %rbp

subq  $16, %rsp        # Allocate stack frame

movq  (%rsp), %rbx    # Restore %rbx
movq  8(%rsp), %rbp   # Restore %rbp
addq  $16, %rsp       # Deallocate frame
```

Interesting Features of Stack Frame

- **Allocate entire frame at once**
  - All stack accesses can be relative to %rsp
  - Do by decrementing stack pointer
  - Can delay allocation, since safe to temporarily use red zone

- **Simple deallocation**
  - Increment stack pointer
  - No base/frame pointer needed
x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers

- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block

- Many tricky optimizations
  - What kind of stack frame to use
  - Various allocation techniques
Today

- Procedures (x86-64)
- 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**

\[ T \ A[L]; \]

- 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

IA32

x86-64

x86-64
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: Type \( T^* \)

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

<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 zipDig[ZLEN];

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

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

zip_dig cmu;

16 20 24 28 32 36

int get_digit
    (zip_dig z, int dig)
{
    return z[dig];
}

IA32

# %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

<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_dig z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# edx = z
movl $0, %eax
.L4:
    # %eax = i
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);
}

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

```assembly
# edx = z = vz
movl $0, %eax
.L8:
  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**
  
  - 2D array of data type $T$
  - $R$ rows, $C$ columns
  - Type $T$ element requires $K$ bytes

- **Array Size**
  - $R \times C \times K$ bytes

- **Arrangement**
  - Row-Major Ordering

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

Arrays are stored as follows:

- Row-Major Ordering:
  
  $A[0][0]$ \ldots $A[0][C-1]$
  
  $\vdots$
  
  $A[R-1][0]$ \ldots $A[R-1][C-1]$

Total size: $4 \times R \times C$ Bytes
Nested Array Row Access

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

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

![Diagram showing nested array row access](image)
Nested Array Row Access Code

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

```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 }};
```

```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 nested array access](image)
Nested Array Element Access Code

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

```
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

```
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};
```

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

```
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]
```

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

<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><code>56+4*3</code></td>
<td><code>68</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td><code>16+4*5</code></td>
<td><code>36</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td><code>56+4*-1</code></td>
<td><code>52</code></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><code>16+4*12</code></td>
<td><code>64</code></td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
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];
}
```

```
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 $A + i \times (C \times K) + j \times K$
  - $C = n, K = 4$

```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 \( ajp = a[i][j] \)
    - Initially = \( a + 4*j \)
    - Increment by \( 4*N \)

```c
/* 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];
}
```

<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:
```
    movl (%ecx), %eax
    movl %eax, (%edi,%edx,4)  # Save in dest[i]
    addl $1, %edx
    addl $ebx, %ecx
    cmpl $edx, %esi
    jg .L18
```

# loop:
    # Read *ajp
    # i++
    # ajp += 4*n
    # n:i
    # if >, goto loop
Today

- Procedures (x86-64)
- 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](image)
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
```asm
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```
Generating Pointer to Structure Member

```c
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`

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

```assembly
movl 12(%ebp), %eax   # Get idx
sall $2, %eax         # idx*4
addl 8(%ebp), %eax    # r+idx*4
```
Following Linked List

**C Code**

```c
void set_val (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->n;
    }
}
```

**Diagram**

- **struct rec**
  - int a[3];
  - int i;
  - struct rec *n;

**Table**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>r</td>
</tr>
<tr>
<td>%ecx</td>
<td>val</td>
</tr>
</tbody>
</table>

**Code**

```
.L17:
    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
```
Alignment

- **Unaligned Data**
  
<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
</tr>
</tbody>
</table>

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

```c
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 \(0_2\)

■ 4 bytes: `int`, `float`, ...
  ▪ lowest 2 bits of address must be \(00_2\)

■ 8 bytes: `double`, `char *`, ...
  ▪ Windows & Linux:
    ▪ lowest 3 bits of address must be \(000_2\)

■ 16 bytes: `long double`
  ▪ Linux:
    ▪ lowest 3 bits of address must be \(000_2\)
    ▪ 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

```
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

```
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

- Procedures (x86-64)
- 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;
```

```
+---+---+---+---+---+---+
| c | i[0]| i[1]|     | v   |     |
+---+---+---+---+---+---+
| sp+0| sp+4| sp+8| sp+16| sp+24|
```
Using Union to Access Bit Patterns

typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}

Same as (float) u? 

Same as (unsigned) f?
Byte Ordering Revisited

■ Idea
  ▪ Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  ▪ Which 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;
```
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>

**LSB** → **MSB**

### 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

\[\begin{array}{cccccccc}
\text{f0} & \text{f1} & \text{f2} & \text{f3} & \text{f4} & \text{f5} & \text{f6} & \text{f7} \\
i[0] & & i[1] \\
l[0] & & & & & & & \\
\end{array}\]

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></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>
</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

- Procedures (x86-64)
- Arrays
- Structures
- Unions
- Memory Layout
- Buffer Overflow
  - Vulnerability
  - Protection
- 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

```c
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 \( \sim 2^{32} \)

\[
\begin{align*}
\$\text{esp} & : 0xfffffbcd0 \\
p3 & : 0x65586008 \\
p1 & : 0x55585008 \\
p4 & : 0x1904a110 \\
p2 & : 0x1904a008 \\
&p2 & : 0x18049760 \\
&\text{beyond} & : 0x08049744 \\
\text{big_array} & : 0x18049780 \\
\text{huge_array} & : 0x08049760 \\
\text{main()} & : 0x080483c6 \\
\text{useless()} & : 0x08049744 \\
\text{final malloc()} & : 0x006be166
\end{align*}
\]

\textit{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>0x00002aaaaad010</td>
</tr>
<tr>
<td>p4</td>
<td>0x00000011501120</td>
</tr>
<tr>
<td>p2</td>
<td>0x00000011501010</td>
</tr>
<tr>
<td>&amp;p2</td>
<td>0x000000010500a60</td>
</tr>
<tr>
<td>&amp;beyond</td>
<td>0x00000000500a44</td>
</tr>
<tr>
<td>big_array</td>
<td>0x00000010500a80</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0000000500a50</td>
</tr>
<tr>
<td>main()</td>
<td>0x0000000400510</td>
</tr>
<tr>
<td>useless()</td>
<td>0x0000000400500</td>
</tr>
<tr>
<td>final malloc()</td>
<td>0x000000386ae6a170</td>
</tr>
</tbody>
</table>

`malloc()` is dynamically linked

address determined at runtime
Today

- Procedures (x86-64)
- 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?
ABSTRACT

On the evening of 2 November 1988, someone infected the Internet with a "worm" program. That program exploited flaws in utility programs in systems based on BSD-derived versions of UNIX. The flaws allowed the program to break into those machines and copy itself, thus "infecting" those systems. This program eventually spread to thousands of machines, and disrupted normal activities and Internet connectivity for many days.

This report gives a detailed description of the components of the worm program—data and functions. It is based on study of two completely independent reverse-compilations of the worm and a version disassembled to VAX assembly language. Almost no source code is given in the paper because of current concerns about the state of the "immune system" of Internet hosts, but the description should be detailed enough to allow the reader to understand the behavior of the program.

The paper contains a review of the security flaws exploited by the worm program, and gives some recommendations on how to eliminate or mitigate their future use. The report also includes an analysis of the coding style and methods used by the author(s) of the worm, and draws some conclusions about his abilities and intent.

Copyright © 1988 by Eugene H. Spafford. All rights reserved. Permission is hereby granted to make copies of this work, without charge, solely for the purposes of instruction and research. Any such copies must include a copy of this title page and copyright notice. Any other reproduction, publication, or use is strictly prohibited without express written permission.

November 29, 1988; revised December 8, 1988

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();
}
```

**Examples**

```bash
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 Disassembly

**echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80485c5:</td>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>80485c6:</td>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>80485c8:</td>
<td>53</td>
<td>push %ebx</td>
</tr>
<tr>
<td>80485c9:</td>
<td>83 ec 14</td>
<td>sub $0x14,%esp</td>
</tr>
<tr>
<td>8048cc:</td>
<td>8d 5d f8</td>
<td>lea 0xffffffffffff8(%ebp),%ebx</td>
</tr>
<tr>
<td>8048cf:</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048d2:</td>
<td>e8 9e ff ff ff</td>
<td>call 8048575 &lt;gets&gt;</td>
</tr>
<tr>
<td>8048d7:</td>
<td>89 1c 24</td>
<td>mov %ebx,(%esp)</td>
</tr>
<tr>
<td>8048da:</td>
<td>e8 05 fe ff ff</td>
<td>call 80483e4 <a href="mailto:puts@plt">puts@plt</a></td>
</tr>
<tr>
<td>8048df:</td>
<td>83 c4 14</td>
<td>add $0x14,%esp</td>
</tr>
<tr>
<td>8048e2:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048e3:</td>
<td>5d</td>
<td>pop %ebp</td>
</tr>
<tr>
<td>8048e4:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

**call_echo:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048eb:</td>
<td>e8 d5 ff ff ff</td>
<td>call 80485c5 &lt;echo&gt;</td>
</tr>
<tr>
<td>8048f0:</td>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>8048f1:</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>
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
    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
    ...

    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    subl $20, %esp
    leal -8(%ebp),%ebx
    movl %ebx, (%esp)
    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
xx xx xx xx
Stack Frame for echo

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

80485eb: e8 d5 ff ff ff
call 80485c5 <echo>
80485f0: c9
leave
Buffer Overflow Example #1

Before call to gets

Input 1234567

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
xx xx xx xx

Stack Frame for echo

Input 12345678

Stack Frame for main

08 04 85 f0
ff ff d6 00

Stack Frame for echo

Saved %ebx
xx xx xx xx

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

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>
<tr>
<td>34</td>
<td>33</td>
<td>32</td>
<td>31</td>
</tr>
</tbody>
</table>

Saved %ebx

buf

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 `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
Avoiding Overflow Vulnerability

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 = 0xffffc638
(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08
(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```
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`
Protected Buffer Disassembly

```assembly
804864d: 55
804864e: 89 e5
8048650: 53
8048651: 83 ec 14
8048654: 65 a1 14 00 00 00
804865a: 89 45 f8
804865d: 31 c0
804865f: 8d 5d f4
8048662: 89 1c 24
8048665: e8 77 ff ff ff
804866a: 89 1c 24
804866d: e8 ca fd ff ff
8048672: 8b 45 f8
8048675: 65 33 05 14 00 00 00
804867c: 74 05
804867f: e8 a9 fd ff ff
8048683: 83 c4 14
8048686: 5b
8048687: 5d
8048688: c3

push %ebp
mov %esp,%ebp
push %ebx
sub $0x14,%esp
mov %gs:0x14,%eax
mov %eax,0xfffffffff8(%ebp)
xor %eax,%eax
lea 0xfffffffff4(%ebp),%ebx
mov %ebx,(%esp)
call 80485e1 <gets>
call 804843c <puts@plt>
mov 0xfffffffff8(%ebp),%eax
xor %gs:0x14,%eax
je 8048683 <echo+0x36>
call 804842c <FAIL>
add $0x14,%esp
pop %ebx
pop %ebp
ret
```
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:
  . . .
  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

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

Echo:

    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

Benign corruption!
(allowed programmers to make silent off-by-one errors)

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

- Procedures (x86-64)
- Arrays
- Structures
- Structures
- Unions
- Memory Layout
- Buffer Overflow

**Floating Point**
- x87 (available with IA32, becoming obsolete)
- SSE3 (available with x86-64)
IA32 Floating Point (x87)

- **History**
  - 8086: first computer to implement IEEE FP
    - separate 8087 FPU (floating point unit)
  - 486: merged FPU and Integer Unit onto one chip
  - Becoming obsolete with x86-64

- **Summary**
  - Hardware to add, multiply, and divide
  - Floating point data registers
  - Various control & status registers

- **Floating Point Formats**
  - single precision (C `float`): 32 bits
  - double precision (C `double`): 64 bits
  - extended precision (C `long double`): 80 bits
FPU Data Register Stack (x87)

- FPU register format (80 bit extended precision)

```
 79 78 64 63 0
 s exp frac
```

- FPU registers
  - 8 registers %st(0) - %st(7)
  - Logically form stack
  - Top: %st(0)
  - Bottom disappears (drops out) after too many pushes
FPU instructions (x87)

- **Large number of floating point instructions and formats**
  - ~50 basic instruction types
  - load, store, add, multiply
  - sin, cos, tan, arctan, and log
    - Often slower than math lib

- **Sample instructions:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldz</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flds Addr</td>
<td>push Mem[Addr]</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>fmuls Addr</td>
<td>%st(0) ← %st(0)*M[Addr]</td>
<td>Multiply</td>
</tr>
<tr>
<td>faddp</td>
<td>%st(1) ← %st(0)+%st(1);pop</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>
**FP Code Example (x87)**

- **Compute inner product of two vectors**
  - Single precision arithmetic
  - Common computation

```c
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```assembly
pushl %ebp          # setup
movl %esp,%ebp
pushl %ebx

movl 8(%ebp),%ebx   # %ebx=&x
movl 12(%ebp),%ecx  # %ecx=&y
movl 16(%ebp),%edx  # %edx=n
fldz                 # push +0.0
xorl %eax,%eax      # i=0
cmpl %edx,%eax      # if i>=n done
jge .L3

flds (%ebx,%eax,4)  # push x[i]
fmuls (%ecx,%eax,4) # st(0)*=y[i]
faddp                # st(1)+=st(0); pop
incl %eax            # i++
cmpl %edx,%eax      # if i<n repeat
jl .L5

.L3:
movl -4(%ebp),%ebx  # finish
movl %ebp, %esp
popl %ebp
ret                 # st(0) = result
```
Inner Product Stack Trace

Initialization

1. fldz
   0.0 %st(0)

Iteration 0

2. flds (%ebx,%eax,4)
   0.0 %st(1)
   x[0] %st(0)

3. fmuls (%ecx,%eax,4)
   0.0 %st(1)
   x[0]*y[0] %st(0)

4. faddp
   0.0+x[0]*y[0] %st(0)

Iteration 1

5. flds (%ebx,%eax,4)
   x[0]*y[0] %st(1)
   x[1] %st(0)

6. fmuls (%ecx,%eax,4)
   x[0]*y[0] %st(1)
   x[1]*y[1] %st(0)

7. faddp
   x[0]*y[0]+x[1]*y[1] %st(0)

eax = i
ebx = *x
ecx = *y
Vector Instructions: SSE Family

- **SIMD (single-instruction, multiple data) vector instructions**
  - New data types, registers, operations
  - Parallel operation on small (length 2-8) vectors of integers or floats
  - Example:
    
    ```
    RGB + RGB  RGB x RGB  “4-way”
    ```

- **Floating point vector instructions**
  - Available with Intel’s SSE (streaming SIMD extensions) family
  - SSE starting with Pentium III: 4-way single precision
  - SSE2 starting with Pentium 4: 2-way double precision
  - All x86-64 have SSE3 (superset of SSE2, SSE)
SSE3 Registers

- All caller saved
- $\%xmm0$ for floating point return value

128 bit = 2 doubles = 4 singles

<table>
<thead>
<tr>
<th>$%xmm0$</th>
<th>Argument #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$%xmm1$</td>
<td>Argument #2</td>
</tr>
<tr>
<td>$%xmm2$</td>
<td>Argument #3</td>
</tr>
<tr>
<td>$%xmm3$</td>
<td>Argument #4</td>
</tr>
<tr>
<td>$%xmm4$</td>
<td>Argument #5</td>
</tr>
<tr>
<td>$%xmm5$</td>
<td>Argument #6</td>
</tr>
<tr>
<td>$%xmm6$</td>
<td>Argument #7</td>
</tr>
<tr>
<td>$%xmm7$</td>
<td>Argument #8</td>
</tr>
<tr>
<td>$%xmm8$</td>
<td></td>
</tr>
<tr>
<td>$%xmm9$</td>
<td></td>
</tr>
<tr>
<td>$%xmm10$</td>
<td></td>
</tr>
<tr>
<td>$%xmm11$</td>
<td></td>
</tr>
<tr>
<td>$%xmm12$</td>
<td></td>
</tr>
<tr>
<td>$%xmm13$</td>
<td></td>
</tr>
<tr>
<td>$%xmm14$</td>
<td></td>
</tr>
<tr>
<td>$%xmm15$</td>
<td></td>
</tr>
</tbody>
</table>
SSE3 Registers

- **Different data types and associated instructions**
- **Integer vectors:**
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes

- **Floating point vectors:**
  - 4-way single
  - 2-way double

- **Floating point scalars:**
  - single
  - double
SSE3 Instructions: Examples

- Single precision **4-way vector add**: `addps %xmm0 %xmm1`

- Single precision **scalar add**: `addss %xmm0 %xmm1`