CSCI-UA.0201-003

Computer Systems Organization

Lecture 11-12: Machine-Level Programming IV: Advanced Topics

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
IA32 Linux Memory Layout

- **Stack**
  - Local variables

- **Heap**
  - Dynamically allocated memory
  - When calling `malloc`, `new`

- **Data**
  - Statically allocated variables declared in code
    - E.g. Global variables

- **Text**
  - Executable machine instructions
  - Read-only
Procedures in x86-64
A Quick Glimpse
x86-64 Registers

• Arguments passed to functions via registers
  – If more than 6 integral 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`
# x86-64 Integer Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<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>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<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 Long Swap

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

swap:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret

• Operands passed in registers
  – First (xp) in %rdi, second (yp) in %rsi
  – 64-bit pointers

• No stack operations required (except ret)
  – Can hold all local information in registers
x86-64 Procedure Summary

- Many functions do not require a stack frame. Only functions that cannot keep all local variables in registers need to allocate space on the stack.

- There is no frame pointer. Instead, references to stack locations are made relative to the stack pointer

- Minimal use of stack
  - Sometimes none except for return addresses
  - Allocate/de-allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - No base/frame pointer needed
Arrays
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

- \textbf{char string}[12];
- \textbf{int val[5];}
- \textbf{double a[3];}
- \textbf{char *p[3];}
Array Access

• Basic Principle

```
T A[L];
int val[5];
```

• Reference Type

<table>
<thead>
<tr>
<th>Reference Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
</tr>
<tr>
<td>val + 1</td>
<td>int *</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
</tr>
</tbody>
</table>

- val[4]: int
- val: int *
- val + 1: int *
- &val[2]: int *
- val[5]: int
- *(val+1): int
- val + i: int *
Array Example

int nyu[5] = { 1, 0, 0, 0, 3 };  
int mit[5] = { 0, 2, 1, 3, 9 };  

• Note: 2 arrays allocated in successive 20 byte blocks
  – Not guaranteed to happen in general
Array Accessing Example

```c
int nyu[5];

int get_digit(int 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)`
2D Array: Example

```c
int pgh[4][5] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3},
     {1, 5, 2, 1, 7},
     {1, 5, 2, 2, 1}};
```

- 2D array: “Row-Major” ordering of all elements

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

- A[i][j] == *(A+i*C+j)
```
2D Array Element Access Code

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

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

```assembler
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)
```
Structures
Structure Allocation

- **Struct members laid out contiguously in memory**
  - Offset of each struct member determined at compile time
### Structure Access

#### 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
```
Following Linked List

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

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

```assembly
.L17:  # loop:

    movl 12(%edx), %eax # i = 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
```

# %edx = r, %ecx = val, %eax = i

Element i
```

<table>
<thead>
<tr>
<th>a</th>
<th>i</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Alignment
Structures & Alignment

• Unaligned Data

- For a primitive data type of $K$ bytes, address is multiple of $K$
- Inefficient to load or store data that spans word boundaries

\begin{verbatim}
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
\end{verbatim}

• Aligned Data

- For a primitive data type of $K$ bytes, address is multiple of $K$
Satisfying Alignment with Structures

• Alignment requirement:
  1. Must align each element of a struct
  2. Initial address & structure length must be multiples of the biggest alignment of a struct’s elements

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

- Define a struct to put large data types first

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

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

- c: 3 bytes
- i: 4 bytes
- d: 3 bytes
- i: 4 bytes
- c: 1 byte
- d: 1 byte

Total:
- 3 bytes
- 2 bytes
About Security!
### 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`
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: 1234567
1234567
```

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

```
unix>./bufdemo
Type a string: 123456789ABC
Segmentation Fault
```
Buffer Overflow Stack

Before call to gets

Stack Frame for call_echo

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
...
Buffer Overflow Stack Example

Before call to gets

Stack Frame for call_echo

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

Stack Frame for echo

Before call to gets

Stack Frame for call_echo

- 08 04 85 f0
- ff ff d6 88
- Saved %ebx
- xx xx xx xx

Stack Frame for echo

- 0xfffff688
- 0xfffff678
- buf

80485eb: call 80485c5 <echo>
80485f0: leave
Buffer Overflow Example #1

Before call to gets

Stack Frame for call_echo

08 04 85 f0
ff ff d6 88
Saved %ebx

buf

Stack Frame for echo

Input 1234567

Stack Frame for call_echo

08 04 85 f0
ff ff d6 88
00 37 36 35
34 33 32 31

Saved %ebx

buf

Overflow buf, and corrupt %ebx

Note: ASCII of 1→31, 2→32, ..., 9→39
Before call to gets

Input 12345678

Note: leave is equivalent to movl %ebp, %esp and pop %ebp
Buffer Overflow Example #3

Before call to gets

Stack Frame for call_echo

08 04 85 f0
ff ff d6 88
Saved %ebx
xx xx xx xx

buf

Stack Frame for echo

Input 123456789

Stack Frame for call_echo

08 04 85 00
43 42 41 39
38 37 36 35
34 33 32 31
buf

Stack Frame for echo

0xffffd688

0xffffd688

echo:
... ...
call 8048575 <gets>
leave # Set %ebp to corrupted value
ret # pop and return to corrupted return address
Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When \texttt{bar()} executes \texttt{ret}, will jump to exploit code
Avoiding Overflow Vulnerability

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

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

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

- Non-executable code segments
  - In old x86, memory is marked as either “read-only” or “writeable”
    • Can execute anything readable
  - X86-64 added explicit “execute” permission
    • Mark stack as non-executable

- Stack Canaries
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
Conclusions

• We have looked at the main characteristics of x86 assembly (i.e. IA 32)
• We took a glimpse at x86_64
• It is now very useful that you write some simple C code, compile it with gcc -S -m32 and compare it to assembly version