CSCI 2021: x86-64 Assembly Extras and Wrap

Chris Kauffman

Last Updated:
Thu 19 Mar 2020 07:56:04 PM CDT
Logistics

Reading Bryant/O’Hallaron

Skim the following on Assembly/C

- Ch 3.8-3.9: Arrays, Structs
- Ch 3.10: Pointers/Security
- Ch 3.11: Floating Point

Goals

- Data in Assembly
- Security Risks
- Floating Point Instr/Regs

### Date Event

<table>
<thead>
<tr>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 3/09</td>
<td>Spring Break</td>
</tr>
<tr>
<td>Wed 3/18</td>
<td>Asm Control</td>
</tr>
<tr>
<td>Fri 3/20</td>
<td>Asm Wrap-up</td>
</tr>
<tr>
<td>Mon 3/23</td>
<td>Practice Exam 2</td>
</tr>
<tr>
<td>Tue 3/24</td>
<td>P3 Due</td>
</tr>
<tr>
<td>Wed 3/25</td>
<td><strong>Exam 2</strong></td>
</tr>
</tbody>
</table>

Project 3

- Problem 1: Clock Assembly Functions (50%)
- Problem 2: Binary Bomb via debugger (50%)
Exercise: All Models are Wrong…

- Rule #1: The Doctor Lies
- Below is our original model for memory layout of C programs
- Describe what is **wrong** based on x86-assembly
- Will all variables have a position in the stack?
- What else is on the stack / control flow info?
- What registers are likely used?

```c
9: int main(...){
10:   int x = 19;
11:   int y = 31;
+<12:   swap(&x, &y);
| 13:     printf("%d %d\n",x,y);
| 14:     return 0;
V 15: }
```

STACK: Caller main(), prior to swap()

```
<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>#2048</td>
<td>x</td>
<td>19</td>
</tr>
</tbody>
</table>
```

```
|--+|
| line:12 | #2044 | y | 31 |
```

STACK: Callee swap() takes control

```
<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>#2048</td>
<td>x</td>
<td>19</td>
</tr>
</tbody>
</table>
```

```
|--+|
| line:12 | #2044 | y | 31 |
```

```
|--+|
| line:19 | #2036 | b | #2044 |
```

```
|--+|
| #2032 | tmp | ? |
```
Answers: All Models are Wrong, Some are Useful

9: int main(...){
10:   int x = 19;
11:   int y = 31;
+-<12:   swap(&x, &y);
| 13:   printf("%d %d\n",x,y);
| 14:   return 0;
V 15: }
|
18: void swap(int *a,int *b){
+->19:   int tmp = *a;
20:      *a = *b;
21:      *b = tmp;
22:      return;
23: }

STACK: Callee swap() takes control
| FRAME | ADDR | NAME | VALUE |
|-------+-------+------+-------|
|-------+-------+-------+-------|
| main() | #2048 | x | 19 |
| | #2044 | y | 31 |
|-------+-------+-------+-------|
| swap() | #2036 | rip | Line 13 |
|-------+-------+-------+-------|

REGS as swap() starts
| REG | VALUE | NOTE |
|-----+-------+------|
|-----+-------+------|
| rdi | #2048 | for *a |
| rsi | #2044 | for *b |
| rax | ? | for tmp |
| rip | L19 | line in swap |

▶ main() must have stack space for locals passed by address
▶ swap() needs no stack space for arguments: in registers
▶ Return address is next value of rip register in main()
▶ Mostly don’t need to think at this level of detail but can be useful in some situations
Data In Assembly

**Arrays**

Usually: base + index × size

```assembly
arr[i] = 12;
movl $12,(%rdi,%rsi,4)
```

```c
int x = arr[j];
movl (%rdi,%rcx,4),%r8d
```

- Array starting address often held in a register
- Index often in a register
- Compiler inserts appropriate size (1,2,4,8)

**Structs**

Usually base+offset

```c
typedef struct {
    int i; short s;
    char c[2];
} foo_t;
```

```c
foo_t *f = ...;
short sh = f->s;
movw 4(%rdi),%si
```

```c
f->c[i] = 'X';
movb $88, 6(%rdi,%rax)
```
Packed Structures as Procedure Arguments

- Passing pointers to structs is 'normal': registers contain addresses to main memory
- Passing actual structs may result in packed structs where several fields are in a single register
- Assembly must unpack these through shifts and masking

1 // packed_struct_main.c
2 typedef struct {
3   short first;
4   short second;
5 } twoshort_t;
6
7 short sub_struct(twoshort_t ti);
8
9 int main(){
10   twoshort_t ts = {.first=10,
11     .second=-2};
12   int sum = sub_struct(ts);
13   printf("%d - %d = %d\n",
14       ts.first, ts.second, sum);
15   return 0;
16 }

1 // packed_struct_main.c
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7   int sum;
8   printf("%d - %d = %d\n",
9       sub_struct(10), sub_struct(-2), sum);
10   return 0;
11 }

1 // packed_struct_main.c
2 typedef struct {
3   short first;
4   short second;
5 } twoshort_t;
6
7 int main()
8 {
9   int sum;
10   printf("%d - %d = %d\n",
11       sub_struct(10), sub_struct(-2), sum);
12   return 0;
13 }

1 // packed_struct_main.c
2 typedef struct {
3   short first;
4   short second;
5 } twoshort_t;
6
7 short sub_struct(twoshort_t ti);
8
9 int main()
10 {
11   twoshort_t ts = {first=10,
12     second=-2};
13   int sum = sub_struct(ts);
14   printf("%d - %d = %d\n",
15       ts.first, ts.second, sum);
16   return 0;
17 }
General Cautions on Structs

Struct Layout by Compilers

- Compiler honors order of source code fields in struct
- BUT compiler may add padding between/after fields for alignment
- Compiler knows total struct size

Struct Layout Algorithms

- Baked into compiler
- May change from compiler to compiler
- May change through history of compiler

Structs in Mem/Regs

- Stack structs spread across several registers
- Don’t need a struct on the stack at all in some cases (just like don’t need local variables on stack)
- Struct arguments packed into 1+ registers

Stay Insulated

- Programming in C insulates you from all of this
- Feel the warmth of gcc’s abstraction blanket
Security Risks in C

Buffer Overflow Attacks

- No default bounds checking in C: Performance favored over safety
- Allows classic security flaws:
  ```c
  char buf[1024];
  printf("Enter you name:\n");
  fscanf(file,"%s",buf); // BAD
  // or
  gets(buf); // BAD
  // my name is 1500 chars
  // long, what happens?
  ```
- For data larger than `buf`, begin overwriting other parts of the stack
  - Clobber return addresses
  - Insert executable code and run it

Counter-measures

- **Stack protection** is default in gcc in the modern era
- Inserts “canary” values on the stack near return address
- Prior to function return, checks that canaries are unchanged
- **Stack / Text Section Start randomized** by kernel, return address and function addresses difficult to predict ahead of time
- Kernel may also vary virtual memory address as well
- Disabling protections is risky
Sample Buffer Overflow Code

```c
#include <stdio.h> // compiled with gcc will likely result
void never(){ // only in 'stack smashing'
    printf("This should never happen\n");
    return;
}

int main(){
    union {long addr; char str[9];} never_info;
    never_info.addr = (long) never;
    never_info.str[8] = '\0';

    printf("Address of never: %0p\n",never_info.addr);
    printf("Address as string: %s\n",never_info.str);

    printf("Enter a string: ");
    char buf[4];
    fscanf(stdin,"%s",buf);
    // By entering the correct length of string followed by the ASCII
    // representation of the address of never(), one might be able to
    // get that function to run (on windows...)

    printf("You entered: %s\n",buf);
    return 0;
}
```
Accessing Global Variables in Assembly

Global data can be set up in assembly in `.data` sections with labels and assembler directives like `.int` and `.short`

```
.data
an_int:     # single int
   .int 17
some_shorts:  # array of shorts
   .short 10  # some_shorts[0]
   .short 12  # some_shorts[1]
   .short 14  # some_shorts[2]
```

Modern Access to Globals

```
movl an_int(%rip), %eax
leaq some_shorts(%rip), %rdi
```

➤ Uses `%rip` relative addressing
➤ Default in `gcc` as it plays nice with OS security features
➤ May discuss again later during Linking/ELF coverage

Traditional Access to Globals

```
movl an_int, %eax     # ERROR
leaq (some_shorts), %rdi  # ERROR
```

➤ Not accepted by `gcc` by default
➤ Yields compile/link errors

```
/usr/bin/ld: /tmp/ccocSiw5.o:
  relocation R_X86_64_32S against `.data'
  can not be used when making a PIE object;
  recompile with `-fPIE`
```
Floating Point Operations

- The original Intel Chips 8086 didn’t have floating point ops
- Had to buy a co-processor, Intel 8087, to add FP ops
- Modern CPUs ALL have FP ops but they feel separate from the integer ops: FP Unit versus AL Unit

FP “Media” Registers

<table>
<thead>
<tr>
<th>256-bits</th>
<th>128-bits</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ymm0</td>
<td>%xmm0</td>
<td>FP Arg 1/ Ret</td>
</tr>
<tr>
<td>%ymm1</td>
<td>%xmm1</td>
<td>FP Arg 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>%ymm7</td>
<td>%xmm7</td>
<td>FP Arg 8</td>
</tr>
<tr>
<td>%ymm8</td>
<td>%xmm8</td>
<td>Caller Save</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>%ymm15</td>
<td>%xmm15</td>
<td>Caller Save</td>
</tr>
</tbody>
</table>

- Can be used as “scalars” - single values but...
- %xmmI is 128 bits big holding
  - 2 64-bit FP values OR
  - 4 32-bit FP values
- %ymmI doubles this

Instructions

- Usually 3 operands:
  \[ C = B \text{ op } A \]
- Ex: Subtraction vsubsd, with d for 64-bit double
  \[ # \text{xmm0} = \text{xmm2} - \text{xmm4} \]
  \[ \text{vsubsd} \ %\text{xmm2, xmm4, xmm0} \]
- 3-operands common in modern assembly
- Can operate on single values or “vectors” of packed values
Floating Point and ALU Conversions

- Recall that bit layout of Integers and Floating Point numbers are quite different (how?)
- Leads to a series of assembly instructions to interconvert between types
  
  ```
  # int eax = ...;
  # double xmm0 = (double) eax;
  vcvtsi2sd %eax,%xmm0,%xmm0
  
  # double xmm1 = ...
  # long rcx = (int) xmm1;
  vcvttsd2siq %xmm1,%rcx
  ```

- These are non-trivial conversions: 5-cycle latency (delay) before completion, can have a performance impact on code which does conversions