CSCI 2021: x86-64 Assembly Extras and Wrap

Chris Kauffman

Last Updated:
Mon Mar 11 12:04:39 CDT 2019

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

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

Date	Event
Mon 3/11	Assembly Wrap
Tue/Wed	Review Lab
Wed 3/13	Review
Thu 3/14	A3 Due
Fri 3/15	**Exam 2**
3/18-3/24	Spring Break

Exam 2 Review Wed

Expect a practice exam

Assignment 3: Questions?

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?

```
    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: Caller main(), prior to swap()
| FRAME   | ADDR  | NAME | VALUE |
|---------+-------+------+-------|
| main()  | #2048 | x    |    19 |
| line:12 | #2044 | y    |    31 |
|---------+-------+------+-------|

STACK: Callee swap() takes control
| FRAME   | ADDR  | NAME | VALUE |
|---------+-------+------+-------|
| main()  | #2048 | x    |    19 |<-+
| line:12 | #2044 | y    |    31 |<-|+
|---------+-------+------+-------|  ||
| swap()  | #2040 | a    | #2048 |--+|
| 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  |   L12 |
|----------+-------+------+-------|
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 old value of `rip` register
- 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 $\times$ size

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

int x = arr[j];
movl (%rdi,%rcx,4),%r8
```

- 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

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

short sh = f->s;
movw 4(%rdi),%si

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

General Cautions on Structs

Struct Layout: Compiler calculates

- Ordering of fields struct in memory
- Padding between/after fields for alignment
- Total struct size

Struct Layout Algorimths

- 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:

```
char buf[1024];
printf("Enter you name:");
fscanf(file,"%s",buf); // BAD
// or
gets(buf);              // BAD
// my name is 1500 chars
// long, what happens?
```

- For data larger than buffer, 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 (start) randomized** by kernel making address of functions difficult to predict ahead of time
- Kernel may also vary virtual memory address as well
- Disable protections at your own risk

Sample Buffer Overflow Code

```
#include <stdio.h>
void never(){
  printf("This should never happen\b");
  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;
}
```

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

256-bits	128-bits	Use
`%ymm0`	`%xmm0`	FP Arg 1/ Ret
`%ymm1`	`%xmm2`	FP Arg 2
...	...	...
`%ymm7`	`%xmm7`	FP Arg 8
`%ymm8`	`%xmm8`	Caller Save
...	...	...
`%ymm15`	`%xmm15`	Caller Save

- 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 op A`

- Ex: Subtraction `vsubsd`, with `d` for 64-bit `double`

```
# xmm0 = xmm2 - xmm4
vsubsd %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