CS 161 – Computer Security
Instructor: Tygar
2 December 2014

Notes on reading assembly code for CS 161 Fall 2014

For Midterm 3, as we have mentioned in class, you will need to be able to read basic 86-64 assembly code produced by the **`gcc`** compiler. This handout will help you practice that skill.

First, some basic comments: when used without the optimizer, **`gcc`** produces a very restricted range of assembly code. Generally, **`gcc`**'s code tends to follow restricted patterns, and to very closely follow the C code as laid out, making it particularly easy to manually "de-compile" code.

Second, we will not be using any exotic features of the 86-64 assembly code – just standard features the way we have discussed in class.

Third, we won't be asking you to disassemble a large program or function on the midterm (for one thing, there would not be sufficient time). So you should focus your attention on small programs or functions.

Fourth, you can generate as many examples as you want. Just compile using **`gcc –S program.c`** to produce an assembly file **`program.s`**.

Bubblesort in C

Here is the classic bubblesort function in C. (Quick test: Shellsort was written by Mr. Shell (Donald Shell). So who wrote bubblesort?)

```
void bubblesort(long list[], long n)
{
        long i, j, temp;

        for (i = 0 ; i < ( n - 1 ); i++)
                for (j = 0 ; j < n - i - 1; j++)
                        if (list[j] > list[j+1])
                        {
                                /* Swapping */
                                temp      = list[j];
                                list[j]   = list[j+1];
                                list[j+1] = temp;
                        }
}
```

Let's compile it using **`gcc –S bubblesort.c`** on an 86-64 machine (such as a hive machine). Here is the assembly code that we get:

```
        .file   "bubblesort.c"
        .text
        .globl  bubblesort
```

```
        .type   bubblesort, @function
bubblesort:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movq    %rdi, -40(%rbp)
        movq    %rsi, -48(%rbp)
        movq    $0, -8(%rbp)
        jmp     .L2
.L6:
        movq    $0, -16(%rbp)
        jmp     .L3
.L5:
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rdx
        movq    -16(%rbp), %rax
        addq    $1, %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        cmpq    %rax, %rdx
        jle     .L4
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        movq    %rax, -24(%rbp)
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    -16(%rbp), %rdx
        addq    $1, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movq    (%rdx), %rdx
        movq    %rdx, (%rax)
        movq    -16(%rbp), %rax
        addq    $1, %rax
        salq    $3, %rax
```

```
        addq    -40(%rbp), %rax
        movq    -24(%rbp), %rdx
        movq    %rdx, (%rax)
.L4:
        addq    $1, -16(%rbp)
.L3:
        movq    -8(%rbp), %rax
        movq    -48(%rbp), %rdx
        movq    %rdx, %rcx
        subq    %rax, %rcx
        movq    %rcx, %rax
        subq    $1, %rax
        cmpq    -16(%rbp), %rax
        jg      .L5
        addq    $1, -8(%rbp)
.L2:
        movq    -48(%rbp), %rax
        subq    $1, %rax
        cmpq    -8(%rbp), %rax
        jg      .L6
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   bubblesort, .-bubblesort
        .ident  "GCC: (GNU) 4.6.3 20120306 (Red Hat 4.6.3-2)"
        .section        .note.GNU-stack,"",@progbits
```

As we look at this code we can immediately remove lines beginning with a dot – those are assembly directives. Also, we can remove addresses of the form **.LF** which are also used as assembly directives. Removing that, we get a slimmer version of the file:

```
bubblesort:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -40(%rbp)
        movq    %rsi, -48(%rbp)
        movq    $0, -8(%rbp)
        jmp     .L2
.L6:
        movq    $0, -16(%rbp)
        jmp     .L3
.L5:
        movq    -16(%rbp), %rax
```

```
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rdx
        movq    -16(%rbp), %rax
        addq    $1, %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        cmpq    %rax, %rdx
        jle     .L4
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        movq    %rax, -24(%rbp)
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    -16(%rbp), %rdx
        addq    $1, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movq    (%rdx), %rdx
        movq    %rdx, (%rax)
        movq    -16(%rbp), %rax
        addq    $1, %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    -24(%rbp), %rdx
        movq    %rdx, (%rax)
.L4:
        addq    $1, -16(%rbp)
.L3:
        movq    -8(%rbp), %rax
        movq    -48(%rbp), %rdx
        movq    %rdx, %rcx
        subq    %rax, %rcx
        movq    %rcx, %rax
        subq    $1, %rax
        cmpq    -16(%rbp), %rax
        jg      .L5
        addq    $1, -8(%rbp)
.L2:
        movq    -48(%rbp), %rax
        subq    $1, %rax
```

```
cmpq    -8(%rbp), %rax
jg      .L6
popq    %rbp
ret
```

Disassembling the code

Now, let's work on understanding the assembly code. In fact, we'll assume that we have just the assembly code and no source code. The first thing to notice is that many opcodes end with a "q". That indicates that they are "quadword" (a word is 16 bits, so 64 bit) values. Some other suffixes to know are "b" (for byte), "w" (for word – 2 bytes), and "l" (for doubleword (long) – 4 byte values). So, all the values we are dealing with here 64 bit values – or what C (on 86-64 architectures) calls long. (On 32 bit architectures, a "long" was 32 bits – so that's why assembly code uses the "l" suffix for 32 bit values.)

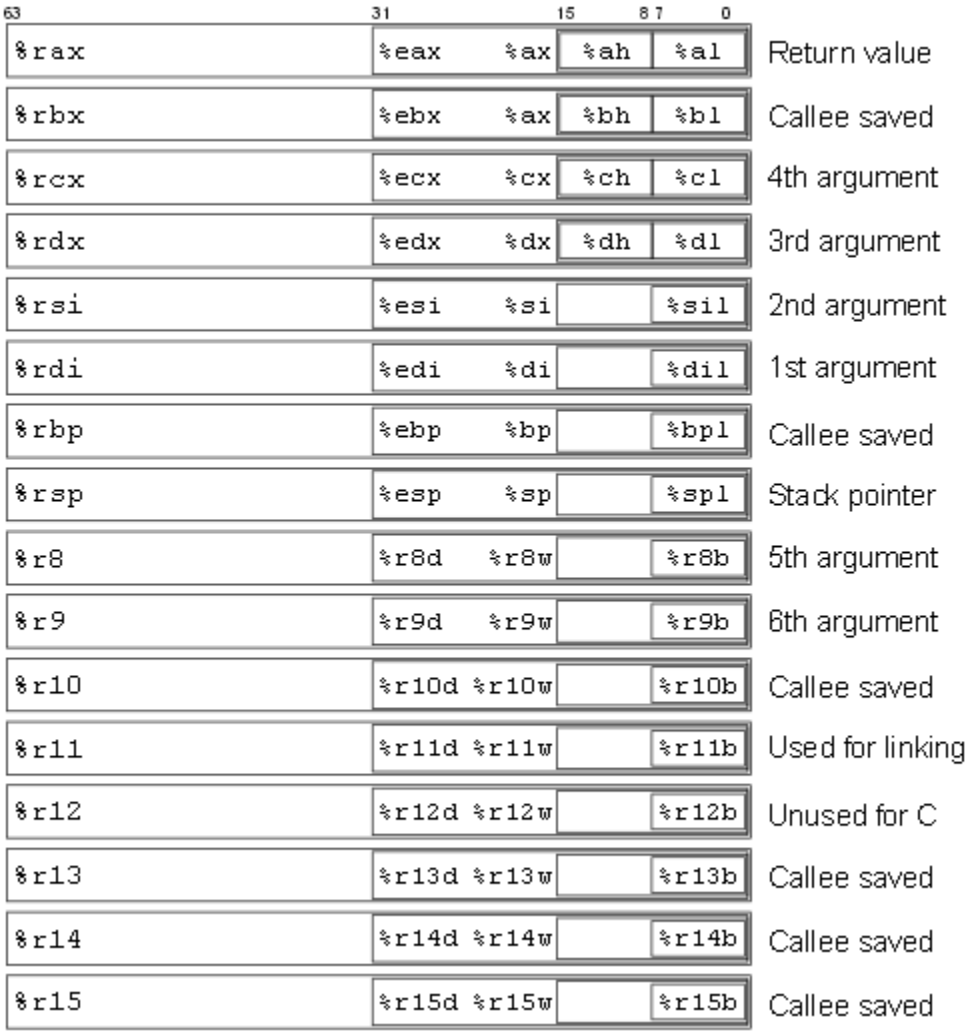
Now, let's start with the first line of assembly code. I'll write the assembly code fragments in red to make it easy to distinguish from other code:

```
bubblesort:
```

This is a label, and allows us to know we are defining a function "**`bubblesort`**." So far, our reconstruction is:

```
bubblesort()
{
}
```

Now, let's remember how registers are used. Notice that each register can be accessed as an eight-byte (quadword) value (e.g., **`%rax`**), a four-byte (doubleword) value (e.g., **`%eax`**), a two-byte (word) value (e.g., **`%ax`**) or a byte (e.g., **`%al`**) value. Notice that **`%rbp`** is reserved for use as the frame pointer and **`%rsp`** is reserved for use as the stack pointer.



Looking through the assembly code, we can the see the following registers being used: **`%rbp, %rsp, %rdi, %rsi, %rax, %rdx,`** and **`%rcx`**.

```
pushq   %rbp
movq    %rsp, %rbp
```

This code begins every function. It saves the old frame pointer on the stack, and creates a new frame pointer by copying the stack pointer to the frame pointer register (**`%rbp`**).

```
movq    %rdi, -40(%rbp)
movq    %rsi, -48(%rbp)
```

Now we are saving values on the stack. The stack grows down. We save the first argument at -40 from the frame pointer. We save the second argument at -48 from the frame pointer. This means that we have two long arguments (**`x, y`**) to the function. So far our reconstruction is

```
bubblesort(long x, long y)
{
}
```

```
        movq    $0, -8(%rbp)
```

Now we are saving a value 0 to some variable on the stack. Thus we must have some long variable. So far our reconstruction is:

```
bubblesort(long x, long y)
{
        long a;
        a=0;
}
```

```
        jmp     .L2
```

This jump instruction is typically used in some sort of control structure. Since there is not a test here, we can infer that it is a loop structure (rather than an if structure). Let's use the most general loop structure – the while statement. So far our reconstruction is:

```
bubblesort(long x, long y)
{
        long a;
        a=0;
        while ()
        {
        }
}
```

```
.L6:
        movq    $0, -16(%rbp)
        jmp     .L3
```

This saves another (long) variable value, and starts another loop. So far our reconstruction is

```
bubblesort(long x, long y)
{
        long a, b;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                }
```

```
        }
}
```

```
.L5:
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rdx
```

So now we are using the variable stored at **-16(%rbp)**, multiplying it by 8 (using a shift-arithmetic-left by 3 bits instruction), adding the address at **x** to it, and fetching the value at the resulting address into **%rdx.** This implies that **x** is an array, and since we multiplied it by 8, it must be an array of long values. So far our reconstruction is

```
bubblesort(long x[], long y)
{
        long a, b;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        /* calculate x[b] */
                }
        }
}
```

```
        addq    $1, %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
```

This is very similar except we are now computing **x[b+1]** and saving the result in **%rax**. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        /* calculate x[b] and x[b+1] */
```

```
                }
        }
}
```

```
        cmpq    %rax, %rdx
        jle     .L4
```

Now we compare **x[b]** and **x[b+1]**, and if **x[b] > x[b+1]**, we execute code (otherwise we skip ahead). This is a classic if statement. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        if (x[b] > x[b+1])
                        {
                        }
                }
        }
}
```

```
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        movq    %rax, -24(%rbp)
```

As above, we find the value at **x[b]**, and now we store it in a new location (indicating a new variable), **-24(%rbp)**. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        if (x[b] > x[b+1])
                        {
```

```
                                c = x[b];
                        }
                }
        }
}
```

```
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    -16(%rbp), %rdx
        addq    $1, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movq    (%rdx), %rdx
        movq    %rdx, (%rax)
```

Using similar reasoning to above, we see that we are looking up the value at **x[b+1]** and put it in **x[b]**. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                        }
                }
        }
}
```

```
        movq    -16(%rbp), %rax
        addq    $1, %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    -24(%rbp), %rdx
        movq    %rdx, (%rax)
.L4:
```

Using similar reasoning to above, we see that we are storing **c** at **x[b+1]**. We also see that the label **.L4:**, indicating that this the end of the if statement. (If there were an else clause, we would expect a **jmp** opcode here. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                }
        }
}
```

```
        addq    $1, -16(%rbp)
.L3:
```

We add 1 to **b**. Then we see the label **.L3:** indicating that we are now in the condition phase of the inner while loop. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while ()
        {
                b=0;
                while()
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                        b++;
                }
```

```
        }
}
```

```
        movq    -8(%rbp), %rax
        movq    -48(%rbp), %rdx
        movq    %rdx, %rcx
        subq    %rax, %rcx
        movq    %rcx, %rax
        subq    $1, %rax
        cmpq    -16(%rbp), %rax
        jg      .L5
```

We move load **a** into **%rax** and **y** into **%rdx** (and then **%rcx**), and compute **y-a-1**, and see if it is greater than **b**. If so, we proceed with the while loop. Our reconstruction so far is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while ()
        {
                b=0;
                while(b < y-a-1)
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                        b++;
                }
        }
}
```

```
        addq    $1, -8(%rbp)
.L2:
```

We add 1 to **a**. Then we see the label **.L2:** indicating that we are now in the condition phase of the outer while loop. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while ()
```

```
        {
                b=0;
                while(b < y-a-1)
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                        b++;
                }
                a++;
        }
}
```

```
        movq    -48(%rbp), %rax
        subq    $1, %rax
        cmpq    -8(%rbp), %rax
        jg      .L6
```

Now we are figuring out the condition in the outer while loop. We calculate n-1, and see if it is greater than a. If it is, we go through the loop. So far our reconstruction is:

```
bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while (a < n-1)
        {
                b=0;
                while(b < y-a-1)
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                        b++;
                }
                a++;
        }
}
```

```
        popq    %rbp
        ret
```

This is standard return code from a function. We pop the old values to the frame pointer, and return from the call.

One final question – does this code return a value? If so, it will be stored in %rax. This value is n-1. Now, we cannot tell from just this code whether the function returns n-1 or whether this just junk and the function returns nothing. Thus, we have two final possibilities:

```
void bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while (a < n-1)
        {
                b=0;
                while(b < y-a-1)
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                        b++;
                }
                a++;
        }
}
```

or

```
long bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while (a < n-1)
        {
                b=0;
                while(b < y-a-1)
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
```

```
                        x[b+1] = c;
                    }
                    b++;
                }
                a++;
            }
            return n-1;
}
```

Now, in this example we want through a lot of explanation and repetition of our reconstructed code, which is how we ended up with a fifteen-page long note. In the midterm, we would not ask you to disassemble such a long stretch of assembly code. However, you should be sure you are comfortable looking at assembly code and understanding what is going on.

Why is disassembly important? Because it is almost certain that in some point in your careers as working software engineers that you will need to examine machine code or assembly code and figure out what is going on. That may be because of a security reason (as we have done in this class), but it may be for another reason too: maybe you will be optimizing code in an embedded system, or chasing down a bug in a compiler, or trying to debug some subtle race condition or parallel computing situation. In any case, the ability to read assembly code is absolutely vital to being a successful security or systems software engineer.