Lab 10: Introduction to x86 Assembly

Revisions: *Nov. 9* The `sos_03.s` file on p. 10 was incorrect and has been fixed.

Reading:
Hacking, 0x250, 0x270

Overview

We will now study low-level programming details that are essential for understanding software vulnerabilities like buffer overflow attacks and format string exploits. You will get exposure to the following:

- Understanding conventions used by compiler to translate high-level programs to low-level assembly code (in our case, using Gnu C Compiler (gcc) to compile C programs).
- The ability to read low-level assembly code (in our case, Intel x86).
- Understanding how assembly code instructions are represented as machine code.
- Being able to use `gdb` (the Gnu Debugger) to read the low-level code produced by `gcc` and understand its execution.

In tutorials based on this handout, we will learn about all of the above in the context of some simple examples.

Intel x86 Assembly Language

Since Intel x86 processors are ubiquitous, it is helpful to know how to read assembly code for these processors.

We will use the following terms: *byte* refers to 8-bit quantities; *short word* refers to 16-bit quantities; *word* refers to 32-bit quantities; and *long word* refers to 64-bit quantities.

There are many registers, but we mostly care about the following:

- EAX, EBX, ECX, EDX are 32-bit registers used for general storage.
- ESI and EDI are 32-bit indexing registers that are sometimes used for general storage.
- ESP is the 32-bit register for the *stack pointer*, which holds the address of the element currently at the top of the stack. The stack grows "up" from high addresses to low addresses. So pushing an element on the stack decrements the stack pointer, and popping an element increments the stack pointer.
- EBP is the 32-bit register for the *base pointer*, which is the address of the current activation frame on the stack (more on this below).
- EIP is the 32-bit register for the *instruction pointer*, which holds the address of the next instruction to execute.

At the end of this handout is a two-page "Code Table" summarizing Intel x86 instructions. The Code Table uses the standard Intel conventions for writing instructions. But the GNU assembler in Linux uses the so-called AT&T conventions, which are different. Some examples:

AT&T Format	Intel Format	Meaning
`movl $4, %eax`	`movl eax, 4`	Load 4 into EAX.
`addl %ebx, %eax`	`addl eax, ebx`	Put sum of EAX and EBX into EAX.
`pushl $X`	`pushl [X]`	Push the contents of memory location named X onto the stack.
`popl %ebp`	`popl ebp`	Pop the top element off the stack and put it in EBP.
`movl %ecx, -4(%esp)`	`movl [esp - 4] ecx`	Store contents of ECX into memory at an address that is 4 less than the contents of ESP.
`leal 12(%ebp), %eax`	`leal eax [ebp + 12]`	Load into EAX the address that is 12 more than the contents of EBP.
`movl (%ebx,%esi,4), %eax`	`movl eax [ebx + 4*esi]`	Load into EAX the contents of the memory location whose address is the sum of the contents of EBX and four times the contents of ESI.
`cmpl $0, 8(%ebp)`	`cmpl [ebp + 8] 0`	Compare the contents of memory at an address 8 more than the contents of EBP with 0. (This comparison sets flags in the machine that can be tested by later instructions.)
`jg L1`	`jg L1`	Jump to label `L1` if last comparison indicated "greater than".
`jmp L2`	`jmp L2`	Unconditional jump to label `L2`.
`call printf`	`call printf`	Call the `printf` subroutine.

We will focus on instructions that operate on 32-bit words (which have the `l` suffix), but there are ways to manipulate quantities of other sizes (the `b` suffix operates indicates byte operations and the `w` suffix indicates 16-bit-word operations).

Typical Calling Conventions for Compiled C Code

The stack is typically organized into a list of activation frames. Each frame has a base pointer that points to highest address in the frame; since stacks grow from high to low, this is at the bottom of the frame:[1]

```
                  <local vars for F>
                  ....
                  <local vars for F>
  <base pointer>:  <old base pointer (of previous frame)>
                   # ----Bottom of frame for F----
                   <return address for call to F>
                   <arg 1 for F>
                   <arg 2 for F>
                   ...
                   <arg n for F>
                   <local vars for caller of F>
                   ...
                   <local vars for caller of F>
<old base pointer>: <older base pointer>
                   # ----Bottom of frame for caller of F----
```

To maintain this layout, the calling convention is as follows:

1. The caller pushes the subroutine arguments on the stack from last to first.
2. The caller uses the `call` instruction to call the subroutine. This pushes the return address (address of the instruction after the `call` instruction) on the stack and jumps to the entry point of the called subroutine.
3. In order to create a new frame, the callee pushes the old base pointer and remembers the current stack address as the new base pointer via the following instructions:

```
pushl %ebp              # \ Standard callee entrance
movl %esp, %ebp         # /
```

4. The callee then allocates local variables and performs its computation.

When the callee is done, it does the following to return:

1. It stores the return value in the EAX register.
2. It pops the current activation frame off the stack via:

```
movl %ebp, %esp
popl %ebp
```

This pair of instructions is often written as the `leave` pseudo-instruction.

3. It returns control to the caller via the `ret` instruction, which pops the return address off the stack and jumps there.
4. The caller is responsible for removing arguments to the call from the stack.

[1] We will follow the convention of displaying memory on the page increasing from low to high addresses.

Writing Assembly Code by Hand for the SOS Program

Following the above conventions, we can write assembly code by hand for the sum-of-squares program we studied last time:

```
    /* Contents of the file sos.c */

    #include <stdio.h>

    /* Calculates the square of integer x */
    int sq (int x) {
      return x*x;
    }

    /* Calculates the sum of squares of a integers y and z */
    int sos (int y, int z) {
      return sq(y) + sq(z);
    }

    /* Reads two integer inputs from command line
       and displays result of SOS program */
    int main (int argn, char** argv) {
      int a = atoi(argv[1]);
      int b = atoi(argv[2]);
      printf("sos(%i,%i)=%i\n", a, b, sos(a,b));
    }
```

```
# HANDWRITTEN ASSEMBLY CODE FOR THE SOS PROGRAM (in the file sos.s)

        .section .rodata        # Begin read-only data segment
        .align 32               # Address of following label will be a multiple of 32
.fmt:                           # Label of SOS format string
        .string "sos(%i,%i)=%i\n" # SOS format string
        .text                   # Begin text segment (where code is stored)
        .align 4                # Address of following label will be a multiple of 4
sq:                             # Label for sq() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /
        movl    8(%ebp), %eax   # result <- x
        imull   8(%ebp), %eax   # result <- x*result
        leave                   # \ Standard callee exit
        ret                     # /
       .align 4                 # Address of following label will be a multiple of 4
sos:                            # Label for sos() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /
        pushl   8(%ebp)         # push y as arg to sq()
        call    sq              # %eax <- sq(y)
        movl    %eax, %ebx      # save sq(y) in %ebx
        addl    $4, %esp        # pop y off stack (not really necessary)
        pushl   12(%ebp)        # push z as arg to sq()
        call    sq              # %eax <- sq(z)
        addl    $4, %esp        # pop z off stack (not really necessary)
        addl    %ebx, %eax      # %eax <- %eax + %ebx
        leave                   # \ Standard callee exit
        ret                     # /
        .align 4                # Address of following label will be a multiple of 4
```

```
.globl main                     # Main entry point is visible to outside world
main:                           # Label for main() function
        pushl   %ebp            # \ Standard callee entrance
        movl    %esp, %ebp      # /

        # int a = atoi(argv[1])
        subl    $8, %esp        # Allocate space for local variables a and b
        movl    12(%ebp), %eax  # %eax <- argv pointer
        addl    $4, %eax        # %eax <- pointer to argv[1]
        pushl   (%eax)          # push string pointer in argv[1] as arg to atoi()
        call    atoi            # %eax <- atoi(argv[1])
        movl    %eax, -4(%ebp)  # a <- %eax
        addl    $4, %esp        # pop arg to atoi off stack

        # int b = atoi(argv[2])
        movl    12(%ebp), %eax  # %eax <- argv pointer
        addl    $8, %eax        # %eax <- pointer to argv[2]
        pushl   (%eax)          # push string pointer in argv[2] as arg to atoi()
        call    atoi            # %eax <- atoi(argv[2])
        movl    %eax, -8(%ebp)  # b <- %eax
        addl    $4, %esp        # pop arg to atoi off stack

        # printf("sos(%i,%i)=%d\n", a, b, sos(a,b))#
        # First calculate sos(a,b) and push it on stack
        pushl   -8(%ebp)        # push b
        pushl   -4(%ebp)        # push a
        call    sos             # %eax <- sos(a,b)
        addl    $8, %esp        # pop args to sos off stack
        pushl   %eax            # push sos(a,b)
        # Push remaining args to printf
        pushl   -8(%ebp)        # push b
        pushl   -4(%ebp)        # push a
        pushl   $.fmt           # push format string for printf
        # Now call printf
        call    printf
        addl    $16, %esp       # pop args to printf off stack (not really necessary)
        leave                   # \ Standard callee exit
        ret                     # /
# END OF ASSEMBLY CODE FILE
```

Here's how to compile and run our hand-written code:

```
lynux@cs342-ubuntu-1:~/assembly-intro$ gcc -o sos-by-hand sos-by-hand.s
lynux@cs342-ubuntu-1:~/assembly-intro$ ./sos-by-hand 3 4
sos(3,4)=25
lynux@cs342-ubuntu-1:~/assembly-intro$ ./sos-by-hand 10 5
sos(10,5)=125
```

Exercise 1: Create and test an x86 assembly program `max-by-hand.s` that acts like the `max.c` program in figure 1. Some notes:

- The C programs in today's lab can be found on puma in `~cs342/download/assembly-intro`.
- When doing x86 assembly, work on your CS342 Ubuntu VM or the Linux clients (`finch`, `lark`, `jay`, etc.) rather than the CTF VM, `puma`, or `tempest`. These other machines have a different (64-bit rather than 32-bit) architecture and generate very different x86 assembly for C programs.
- Start with the template `sos-by-hand.s` from above and make little changes to massage it to have the right behavior for `max.c`.

```
#include <stdio.h>

int max (int x, int y) {
  if (x >= y) {
    return x;
  } else {
    return y;
  }
}

int main (int argn, char** argv) {
  if (argn == 3) {
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    printf("max(%i,%i)=%d\n", a, b, max(a,b));
  } else {
    printf("Usage: max <int1> <int2>\n");
  }
}
```

Figure 1: A C program `max.c` for finding the maximum of two command line integers.

Compiling `sos.c` to Assembly Code

Writing assembly code by hand is tedious and error prone. This is why compilers were invented! They automatically translate code that's written at a higher level than assembly[2] into assembly instructions. These instructions can be assembled into even lower level machine code – the bits that can actually be executed on a processor like an x86.

We can use `gcc` to compile `sos.c` into assembly code as follows:[3]

```
lynux@cs342-ubuntu-1:~/intro-to-c$ gcc -S sos.c
```

This creates the file `sos.s` shown below. Note that the code is a bit different than what we generated by hand.

```
# Contents of the assembly file sos.s created by gcc -S sos.c
        .file "sos.c"
        .text
        .globl sq
        .type sq, @function
sq:
.LFB0:
        .cfi_startproc
        pushl %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl %esp, %ebp
        .cfi_def_cfa_register 5
        movl 8(%ebp), %eax
        imull 8(%ebp), %eax
        popl %ebp
        .cfi_def_cfa 4, 4
        .cfi_restore 5
        ret
        .cfi_endproc
.LFE0:
        .size sq, .-sq
        .globl sos
        .type sos, @function
sos:
.LFB1:
        .cfi_startproc
        pushl %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl %esp, %ebp
        .cfi_def_cfa_register 5
        pushl %ebx
        subl $4, %esp
        movl 8(%ebp), %eax
        movl %eax, (%esp)
        .cfi_offset 3, -12
```

[2] Of course, we know that C is not at *that* much higher a level than assembly, but I digress ...

[3] These are the results we get if we compile the code on a 32-bit machine like those in the Linux microfocus cluster. We get very different results if we compile the code on a 64-bit machine like puma.

```
        call sq
        movl %eax, %ebx
        movl 12(%ebp), %eax
        movl %eax, (%esp)
        call sq
        addl %ebx, %eax
        addl $4, %esp
        popl %ebx
        .cfi_restore 3
        popl %ebp
        .cfi_def_cfa 4, 4
        .cfi_restore 5
        ret
        .cfi_endproc
.LFE1:
        .size sos, .-sos
        .section .rodata
.LC0:
        .string "sos(%i,%i)=%d\n"
        .text
        .globl main
        .type main, @function
main:
.LFB2:
        .cfi_startproc
        pushl %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl %esp, %ebp
        .cfi_def_cfa_register 5
        andl $-16, %esp
        subl $32, %esp
        movl 12(%ebp), %eax
        addl $4, %eax
        movl (%eax), %eax
        movl %eax, (%esp)
        call atoi
        movl %eax, 24(%esp)
        movl 12(%ebp), %eax
        addl $8, %eax
        movl (%eax), %eax
        movl %eax, (%esp)
        call atoi
        movl %eax, 28(%esp)
        movl 28(%esp), %eax
        movl %eax, 4(%esp)
        movl 24(%esp), %eax
        movl %eax, (%esp)
        call sos
        movl $.LC0, %edx
        movl %eax, 12(%esp)
        movl 28(%esp), %eax
        movl %eax, 8(%esp)
```

```
        movl 24(%esp), %eax
        movl %eax, 4(%esp)
        movl %edx, (%esp)
        call printf
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE2:
        .size main, .-main
        .ident "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section .note.GNU-stack,"",@progbits
```

Even though the code looks different, it behaves the same way, as demonstrated by compiling it to machine code:

```
lynux@cs342-ubuntu-1:~/assembly-intro$ gcc -o sos-from-assembly sos.s
lynux@cs342-ubuntu-1:~/assembly-intro$ ./sos-from-assembly 3 4
sos(3,4)=25
```

Optimizing `sos.c`

Invoking `gcc` with an optimization flag (`-O1, -O2, -O3`) can create more compact code by using clever optimizations.

```
lynux@cs342-ubuntu-1:~/intro-to-c$ gcc -S -O3 -o sos_03.s sos.c
```

```
# Part of the contents of sos_03.s created by gcc -S -O3 -o sos_03.s sos.c
sq:
.LFB22:
.cfi_startproc
movl 4(%esp), %eax
imull %eax, %eax
ret
.cfi_endproc
.LFE22:
.size sq, .-sq
.p2align 4,,15
.globl sos
.type sos, @function
sos:
.LFB23:
.cfi_startproc
movl 4(%esp), %edx
movl 8(%esp), %eax
imull %edx, %edx
imull %eax, %eax
addl %edx, %eax
ret
.cfi_endproc
.LFE23:
.size sos, .-sos
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "sos(%i,%i)=%d\n"
.section .text.startup,"ax",@progbits
.p2align 4,,15
.globl main
.type main, @function
main:
.LFB24:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
pushl %esi
pushl %ebx
andl $-16, %esp
subl $32, %esp
movl 12(%ebp), %esi
.cfi_offset 3, -16
```

```
.cfi_offset 6, -12
movl 4(%esi), %eax
movl %eax, (%esp)
call atoi
movl %eax, %ebx
movl 8(%esi), %eax
movl %eax, (%esp)
call atoi
movl %ebx, %ecx
imull %ebx, %ecx
movl %ebx, 8(%esp)
movl $.LC0, 4(%esp)
movl $1, (%esp)
movl %eax, %edx
imull %eax, %edx
movl %eax, 12(%esp)
addl %ecx, %edx
movl %edx, 16(%esp)
call __printf_chk
leal -8(%ebp), %esp
popl %ebx
.cfi_restore 3
popl %esi
.cfi_restore 6
popl %ebp
.cfi_def_cfa 4, 4
.cfi_restore 5
ret
.cfi_endproc
```

Using GDB to Disassemble Code

What if we don't have the source code to generate assembly code, but only the binary code? Then we can use the GNU Debugger (`gdb`) to disassemble the binary, as shown below:

```
lynux@cs342-ubuntu-1:~/assembly-intro$ gdb sos-from-assembly
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/lynux/assembly-intro/sos-from-assembly...(no debugging symbols found)...done.

(gdb) disassemble sq
Dump of assembler code for function sq:
   0x08048414 <+0>:     push   %ebp
   0x08048415 <+1>:     mov    %esp,%ebp
   0x08048417 <+3>:     mov    0x8(%ebp),%eax
   0x0804841a <+6>:     imul   0x8(%ebp),%eax
   0x0804841e <+10>:    pop    %ebp
   0x0804841f <+11>:    ret
End of assembler dump.

(gdb) disassemble 0x08048414
Dump of assembler code for function sq:
   0x08048414 <+0>:     push   %ebp
   0x08048415 <+1>:     mov    %esp,%ebp
   0x08048417 <+3>:     mov    0x8(%ebp),%eax
   0x0804841a <+6>:     imul   0x8(%ebp),%eax
   0x0804841e <+10>:    pop    %ebp
   0x0804841f <+11>:    ret
End of assembler dump.

(gdb) disassemble sos
Dump of assembler code for function sos:
   0x08048420 <+0>:     push   %ebp
   0x08048421 <+1>:     mov    %esp,%ebp
   0x08048423 <+3>:     push   %ebx
   0x08048424 <+4>:     sub    $0x4,%esp
   0x08048427 <+7>:     mov    0x8(%ebp),%eax
   0x0804842a <+10>:    mov    %eax,(%esp)
   0x0804842d <+13>:    call   0x8048414 <sq>
   0x08048432 <+18>:    mov    %eax,%ebx
   0x08048434 <+20>:    mov    0xc(%ebp),%eax
   0x08048437 <+23>:    mov    %eax,(%esp)
   0x0804843a <+26>:    call   0x8048414 <sq>
   0x0804843f <+31>:    add    %ebx,%eax
   0x08048441 <+33>:    add    $0x4,%esp
   0x08048444 <+36>:    pop    %ebx
   0x08048445 <+37>:    pop    %ebp
   0x08048446 <+38>:    ret
End of assembler dump.(gdb)
```