README

RE Lab 0x02 - x86_64 assembly

updated by S. Radu - march 2025
based on previous work by R. Caragea and C. Rusu

1. Setup

In this lab session, we will see some assembly programming and disassembly. We will be working only on Linux! Checkout [1] when you need a refresher for any of the ASM instruction.

Ensure you have python3 and pwntools installed:

```
$ apt update
$ apt install python3 python3-pwntools
```

or follow the instructions here

Optionally install `nasm` as well:

```
$ apt install nasm
```

Get your lab files from moodle: 02-lab-files.zip. The password for the zip is `infected`.

2. Practical Examples: Assembly analysis

The tasks today will make use of the compiler explorer Godbolt[2]. Write short sequence of C code and check the resulting output for the following:

1. Write a C function that subtracts two integers. Observe:
    - the calling convention (RDI/RSI). Read more at [3]
    - the return value (RAX)
2. Write a C function that adds two integers.
    - What instructions were used?
    - Observe what happened: the compiler found a shorter way to do the operations
3. Write a C function that adds three integers.
    - What assembly instructions do we have now?
    - Observe the limitations of the previous optimization
4. Write a C function that adds the first n positive integers.

    - Observe the loops
    - Try another compiler (e.g. clang).
    - Try using optimization flags (O1, O2, O3). What happens now?
    - Observe the initial TEST
    - Which registers were allocated to i and sum ?
    - At the beginning of the function fix the number n to a constant value.
5. Write a C function that adds the elements in a vector of integers.
6. Using structures
    - Observe the pointer arithmetic
    - Change the data types of v1 and v2
    - What is the first parameter given to `printf` (the string reference)?
7. Traversing a linked list
8. Write a C function that divides an integer by constants (e.g. 2, 5, etc.). Do the same for multiplication, using the same constants.
    - What do you notice?
    - *Division is the bane of computer performance and the compiler will go to extreme lengths to avoid it.*
9. Simple crackme
    - Understand how this code works and what the corresponding assembly code is doing.

3. Tasks - Assembly analysis

For each of the following, write an equivalent C line for each assembly instruction (if possible), then try to simplify until you understand what the code does.

At the end you can check Godbold[2:1] to see if you get similar assembly generated from your C code (perfect matches might not be possible though).

You will need to give each function a descriptive name after you understand what it does.

3.1 Write the equivalent in C for this ASM snippet (1p)

```
myst1:
        test    rdi, rdi
        je      .L4
        mov     edx, 0
        mov     eax, 0
.L3:
        mov     rcx, rdx
        imul    rcx, rdx
        add     rax, rcx
        add     rdx, 1
        cmp     rdi, rdx
        jne     .L3
        ret
.L4:
```

```
        mov     rax, rdi
        ret
```

3.2 Assembly source code 2 (1p)

```
myst2:
        sub     r8, r9
        add     r8, rdx
        sub     r8, rcx
        lea     rax, [r8+rdi]
        sub     rax, rsi
        ret
```

3.3 Write the equivalent in C for this ASM snippet (2p)

```
myst3:
        cmp     BYTE PTR [rdi], 0
        je      .L4
        mov     eax, 0
.L3:
        add     rax, 1
        cmp     BYTE PTR [rdi+rax], 0
        jne     .L3
        ret
.L4:
        mov     eax, 0
        ret
```

3.4 Write the equivalent in C for this ASM snippet (3p)

```
myst4:
        push    rbp
        push    rbx
        sub     rsp, 8
        mov     rbx, rdi
        cmp     rdi, 1
        ja      .L4
.L2:
        mov     rax, rbx
        add     rsp, 8
        pop     rbx
        pop     rbp
        ret
.L4:
        lea     rdi, [rdi-1]
        call    myst4
        mov     rbp, rax
        lea     rdi, [rbx-2]
        call    myst4
```

```
        lea     rbx, [rbp+0+rax]
        jmp     .L2
```

3.5 Write the equivalent in C for this ASM snippet (5p)

```
myst5:
        xor     eax, eax
        cmp     rdi, 1
        jbe     .L1
        cmp     rdi, 3
        jbe     .L6
        test    dil, 1
        je      .L1
        mov     ecx, 2
        jmp     .L3
.L4:
        mov     rax, rdi
        xor     edx, edx
        div     rcx
        test    rdx, rdx
        je      .L8
.L3:
        add     rcx, 1
        mov     rax, rcx
        imul    rax, rcx
        cmp     rax, rdi
        jbe     .L4
.L6:
        mov     eax, 1
        ret
.L8:
        xor     eax, eax
.L1:
        ret
```

4. Writing some assembly (2p)

Starting from the `.py` template in the lab archive, write an assembly snippet to call the `sys_time` syscall (index 201) in order to obtain the current unix timestamp. Also checkout[4] for the full syscall reference.

Alternatively, use the `nasm` assembler, starting from the `.asm` template. Take the following “hello world” as an example:

```
section .data
    hello: db 'hello world',10 ; 10 = '\n'

section .text
    global _start
```

```
_start:
    ; write syscall
    mov rax, 1
    mov rdi, 1
    mov rsi, hello
    mov rdx, 12
    syscall
```

Compile with `nasm -f elf64 task.asm -o task.o` Link with `ld task.o task` to get an executable.

Note: the syntax for nasm and the syntax used in pwntools DIFFER

Since it is not immediately possible to call higher-level print functions such as printf, intercept the result using the tools from the previous class (strace or ltrace -S) in order to verify the results. Check against the Unix time at Epoch Converter

Note: if the crash at the end bothers you, also call `sys_exit` to close the process gracefully.

5. Bonus 1 - Compiler Party Tricks (2p)

Find out what the original code was. Explain what it does. You can try writing the equivalent C/ Python and play around with some inputs.

```
my_function:
        shr     rdi
        movabs  rdx, -4392081922311798003
        mov     rax, rdi
        mul     rdx
        mov     rax, rdx
        shr     rax, 4
        ret
```

6. Bonus 2 - Binary Patching (2p)

Take the code from Section 2.9, write the C program on your computer and compile it with gcc (*you can do this through Godbolt as well*). Edit the binary file (**not the source code!**) to make it print Correct! when the wrong secret value is given and vice-versa.

Resources

---

1. https://www.felixcloutier.com/x86/ ↩

2. https://godbolt.org ↩ ↩

3. https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI ↩

4. https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/ ↩