Reverse Engineering March, 2023

Lab session 0x02

In this lab session, we will see some assembly programming and disassembly.

1 Lab files

The files for this lab session are available at https://pwnthybytes.ro/unibuc_re/02-lab-files.zip and the password for the zip file is *infected*.

2 Tools we use (Linux)

Today, all the work will be done in the Linux environment. Make sure you have *python3* and *pwntools* installed on your VM:

```
$ apt-get update
$ apt-get install python3-pycryptodome xinetd libffi-dev python3-wheel
    ↪ gcc gdb python3-setuptools python3-dev libssl-dev git libc6-dbg
    ↪ python3-pip make gcc-multilib socat
$ pip3 install pwntools
```

2.1 Tasks: assembly analysis

The tasks today will make use of the compiler explorer Godbolt[1]. Using the gcc compiler write short sequences of code and check the resulting assembly code for:

1. Write a C function that subtracts two integers. Observe the calling convention (RDI/RSI) and the return value (RAX).

2. Write a C function that adds two integers. What assembly instruction did the compiler use?

3. Write a C function that adds three integers. What assembly instructions do we have now?

4. Write a C function that adds the first n positive integers. Observe the loops. Also try:

   - Try also the *clang* compiler. Try using optimization flags O1 and O3. What happens now?
   - 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. Try using flags O1 and O3.

6. Define a *struct* like

   ```
   #include <stdint.h>

   typedef struct mystruct_t{
           uint64_t v1;
           uint64_t v2;
   } mystruct_t;
   ```

   and access v1 and v2 by writing these values in the console (use *printf*). Observe the pointer arithmetic (change the data types of v1 and v2) and the first string reference given to *printf.*

[1] https://godbolt.org/

7. Consider and analyze the following code that traverses a linked list:

```
#include <stdint.h>

typedef struct mystruct_t{
        uint64_t v1;
        struct mystruct_t *next;
} mystruct_t;

mystruct_t *get_last(mystruct_t *head){
        mystruct_t *cur = head;

        while (cur->next != 0)
                cur = cur->next;

        return cur;
}
```

8. Write a C function that divides an integer by constants 4, 5, 32. Do the same for multiplication by the same constants. Division is the bane of computer performance and the compiler will go to extreme lengths to avoid it.

9. Check out the following simple *password checking* code:

```
#include <stdio.h>
#include <stdint.h>

int main()
{
        uint64_t secret_value = 0xdeadc0de;
        uint64_t user_input;

        scanf("%lld", &user_input);

        user_input ^= 0x1337cafe;

        if (user_input == secret_value)
                puts("Correct!");
        else
                puts("Wrong");

        return 0;
}
```

Understand how this code works and what the corresponding assembly code is doing.

3 Lab tasks: assembly to C code conversion

3.1 Assembly source code 1 (2p)

```
myst2:
        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
```

3.2 Assembly source code 2 (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
```

3.3 Assembly source code 3 (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 Bonus task 1 (2p)

Find out and explain what the following code is doing:

```
my_function:
        movabs  rdx, -1085102592571150095
        mov     rax, rdi
        mul     rdx
        mov     rax, rdx
        shr     rax, 4
        ret
```

5 Bonus task 2 (2p)

Take the last piece of code presented in Section 2, write the C program on your computer and compile it with *gcc*. Edit the binary file (not the source code!) to make it print *Correct!* when the wrong secret value is given and vice-versa.