Lab: Hello, World (of RE)

Environment Needed:

- Linux Virtual Machine (recommend Ubuntu)
- Nasm (`sudo apt-get install nasm`)

1. Open a Terminal Window

2. Type the command: `cd ~/<path where you saved the lab downloads>`
3. In this folder you should two files (check with the command `ls`):
    a. helloworld.asm
    b. Makefile
4. Check out the content of helloworld.asm:
    a. Command: `gedit helloworld.asm`
5. We're using int 0x80 to request the OS write a string:

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

```
global _start

section .text
_start:
    mov eax, 4 ; write
    mov ebx, 1 ; stdout
    mov ecx, msg
    mov edx, msg.len
    int 0x80

    mov eax, 1 ; exit
    mov ebx, 0
    int 0x80

section .data
msg:    db  "Hello, world!", 10
.len:   equ $ - msg
```

6. We're storing the string "Hello World!" in the data section.
7. First, we will build helloworld.asm manually. Run the following 2 commands:
   a. `nasm -f elf32 -g helloworld.asm`
   b. `ld -melf_i386 -g helloworld.o -o helloworld.out`
8. You should now see two new files (check with the command `ls`):
   a. helloworld.o
   b. helloworld.out
9. Now let's run our helloworld application:
   a. Command: `./helloworld.out`
10. Success!!
11. Building manually is tedious… let's use a Makefile to automate it!
    a. Makefiles are not a programming thing, they are a Linux way of grouping commands.
    b. Check out the content of Makefile.
       i. Command: `gedit Makefile`

```
all: helloworld.o

helloworld.o: helloworld.asm
    nasm -f elf32 -g helloworld.asm
    ld -melf_i386 -g helloworld.o -o helloworld.out

clean:
    rm helloworld.o helloworld.out
```

    c. The Makefile automates the build process.
    d. Running the command `make` will build and link our assembly.
    e. `make clean` will remove the output files.
12. Run the command `make clean` to remove the manually built files.
    a. Use `ls` to verify they are gone.

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**

13. Run the command `make` to rebuild our application.
14. Execute the new helloworld:
    a. Command: `./helloworld.out`
15. Let's finish by examining the contents of the executable that we created.
    a. First, try opening the *executable* in your favorite text editor; for example:
       ```
       gedit helloworld.out
       ```
    b. What happens? The contents of the file look like mostly gibberish. That is because the executable file does not contain the *assembly instructions* that we first typed – after building our assembly program into an executable, the executable now only contains *machine code* (and some symbols to help the OS navigate the machine code).
    c. To examine the file in a human-readable way, we need to translate the machine code back into assembly. We can do this with the tool objdump. Objdump the executable to translate its machine code into (somewhat) human readable x86:
       ```
       objdump -d -Mintel helloworld.out
       ```
16. Carefully compare the disassembled code to the code that we originally built. What are the similarities? What are the differences?
    a. Takeaway: building a program into machine code is a *lossy* process – some of the original information and programmer intent is lost. When we try to go the reverse direction – recovering assembly code from machine code – we can get good results, but not perfect. Without the original programmer annotations, symbol names, and other metadata, it becomes much harder to read and interpret the assembly. This is where *reverse engineering* comes in.

X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES
ISBN: **1394199880**