

## Lab2-Sample Shellcode Development Lab

In previous years, I installed the Virtual Machine (VM) and downloaded the *SEED-Ubuntu20.04.zip* from the SEED website on my personal computer. Then I had to extract this file before I upload it to the virtual machine. After I completed and ran this virtual machine, it been normal and there was no crash at this time. I had to create a password and username on this Ubuntu virtual machine **[Figure 1: Ubuntu Home Screen]**.

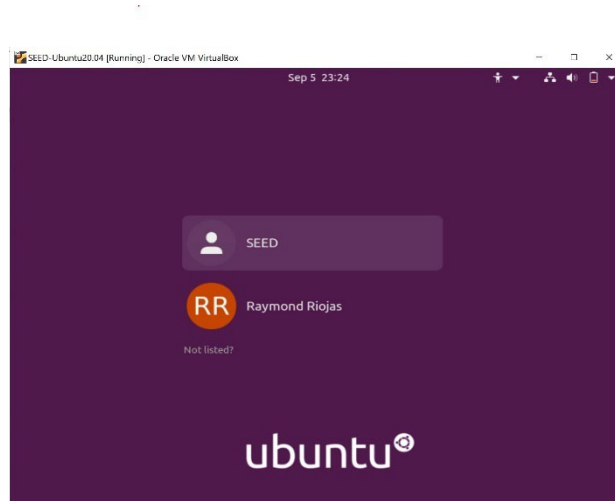


Figure 1: Ubuntu Home Screen

Once I login into this username and password, I had to download the *labsetup.zip* from the Shellcode lab site then I extracted this file into this virtual machine instead personal device **[Figure 2: Labsetup File]**.

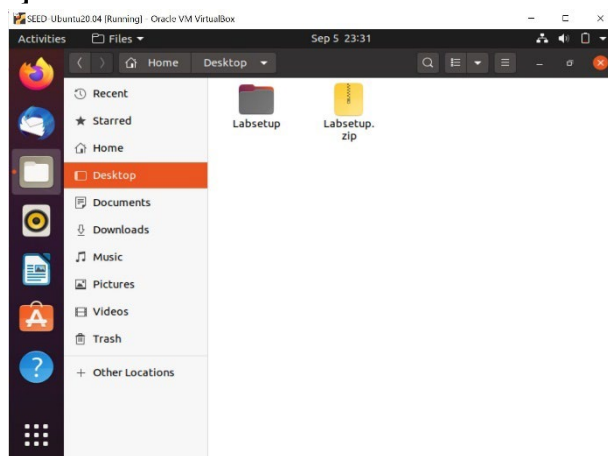


Figure 2: Labsetup File

In this lab, we're going to discuss what shellcode is in the real-world scenario. Shellcode is part of the injection and exploit vulnerabilities in a system. I had to write the code for compiling to object code using *nasm* command. The code is *nasm -f elf32 hello.s -o hello.o* on

the terminal before everything is ready. Once I get the object code *hello.o*, that way we can run the linker program which means that the last step in compilation, so the code is *ld hello.o -o hello*. After linker program, we had to test this file which I typed this code is *./hello* then it say “Hello, world!” [Figure 3: Hello World Output].

```
raymondriojas@VM:~/Desktop/Labsetup$ nasm -f elf64 hello.s -o hello.o
raymondriojas@VM:~/Desktop/Labsetup$ ld hello.o -o hello
raymondriojas@VM:~/Desktop/Labsetup$ ./hello
Hello, world!
raymondriojas@VM:~/Desktop/Labsetup$
```

Figure 3: Hello World Output

Once I completed the last step, I had to use the *objdump* command to disassemble the executable or object file. On the VM, I use the code is *objdump -Intel -d hello.o* then I had to see this hexadecimal numbers on this output [Figure 4: Hexadecimal].

```
raymondriojas@VM:~/Desktop/Labsetup$ objdump -Intel -d hello.o
hello.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <start>:
 0:  bf 01 00 00 00      mov     edi,0x1
 5:  48 be 00 00 00 00    movabs  rsi,0x0
 c:  00 00 00
 f:  ba 0e 00 00 00      mov     edx,0xe
14:  b8 01 00 00 00      mov     eax,0x1
19:  0f 05               syscall
1b:  bf 00 00 00 00      mov     edi,0x0
20:  b8 3c 00 00 00      mov     eax,0x3c
25:  0f 05               syscall
```

Figure 4: Hexadecimal

Next step, I had to use `xxd` command which means that print out the content of the binary file. Also, I should be able to find any hexadecimal from other output **[Figure 5: xxd Command]**.

[illegible]

### Figure 5: xxd Command

In actual attacks, we had to find the shellcode in our attacking code, which may be written in a language like Python. This python code is going to the terminal, the code is `./converrt.py`. I looked at this output code on terminal, but it said there are 15 from the length of the shellcode **[Figure 6: Attack code]**.

```
6d736700000000000000000000000000000000000000000000000  
000000000100000003000000000000000000000000000000000000  
000000000000000000  
raymondrl@VM: ~/Desktop/Labsetup$ ./convert.py  
Length of the shellcode: 15  
shellcode= (  
    "\xc0\xpy\x a\xnd|x p\xas|xt|x m|\xac|xhi|x e |xco|xde|x h|xer"  
).encode('latin-1')  
raymondrl@VM: ~/Desktop/Labsetup$
```

Figure 6: Attack code

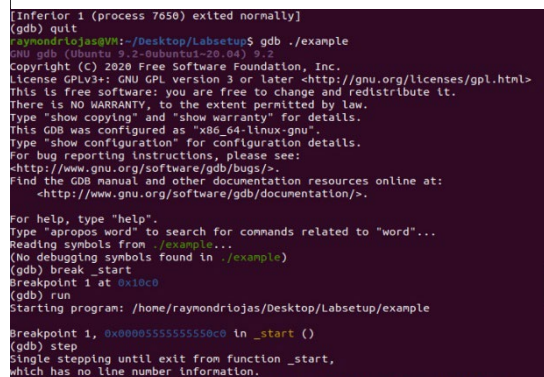
## Task 2: Writing Shellcode (Approach 1)

I had to create the code for `execve()` before I uploaded it to the terminal. When I type the command *touch* which means that I can open the program. I have to store three different ways are string, array, address [Figure 7: `Execve()`]. I had to compile and run the code with enables the debugging information. There is no error on this code, before I type this command *gdb* which means that to debug the program and show how the program gets the address of the shell string [Figure 8: `gdb` commands].



```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7     char *bin_sh = "/bin/sh";
8     char *argv[] = { "bin/sh", NULL };
9     char *envp[] = { NULL };
10
11     if(execve(bin_sh, argv, envp) == -1)
12     {
13         perror("execve");
14         exit(EXIT_FAILURE);
15     }
16
17     return 0;
18 }
```

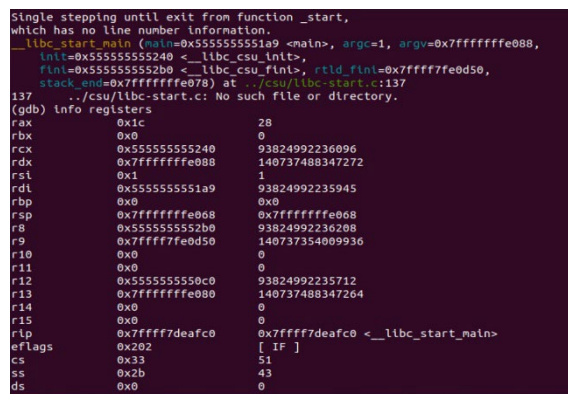
Figure 7: `Execve()`



```
[Inferior 1 (process 7650) exited normally]
(gdb) quit
raymondriojas@VM:~/Desktop/Labsetup$ gdb ./example
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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 "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./example...
(gdb) break _start
Breakpoint 1 at 0x00c0
(gdb) run
Starting program: /home/raymondriojas/Desktop/Labsetup/example

Breakpoint 1, 0x000055555550c0 in _start ()
(gdb) step
Single stepping until exit from function _start,
which has no line number information.
```



```
Single stepping until exit from function _start,
which has no line number information.
__libc_start_main (main=0x555555551a9 <main>, argc=1, argv=0x7fffffffe088,
                 init=0x55555555240 <__libc_csu_init>,
                 fini=0x555555552b0 <__libc_csu_fini>, rtld_fini=0x7fffffffe0d0,
                 stack_end=0x7fffffffe078) at ../csu/libc-start.c:137
137 ../csu/libc-start.c: No such file or directory.
(gdb) info registers
rax             0x1c             28
rbx             0x0              0
rcx             0x55555555240      93824992236096
rdx             0x7fffffffe088    140737488347272
rsi             0x1              1
rdi             0x555555551a9      93824992235945
rbp             0x0              0
rsp             0x7fffffffe068    0x7fffffffe068
r8              0x555555552b0      93824992236208
r9              0x7fffffffe0d0    140737354009936
r10             0x0              0
r11             0x0              0
r12             0x555555550c0      93824992235712
r13             0x7fffffffe080    140737488347264
r14             0x0              0
r15             0x0              0
rip             0x7fffffffe0fc0    0x7fffffffe0fc0 <__libc_start_main>
eflags          0x202             [ IF ]
cs              0x33             51
ss              0x2b             43
ds              0x0              0
```

```

rsp      0x7fffffff068      0x7fffffff068
r8       0x555555552b0      93824992236208
r9       0x7ffff7fe0d50     140737354009936
r10      0x0                0
r11      0x0                0
r12      0x555555550c0      93824992235712
r13      0x7fffffff080      140737488347264
r14      0x0                0
r15      0x0                0
rip      0x7ffff7deafc0     0x7ffff7deafc0 <_libc_start_main>
eflags   0x202              [ IF ]
cs       0x33               51
ss       0x2b               43
ds       0x0                0
es       0x0                0
fs       0x0                0
gs       0x0                0
(gdb) x/16bx 0x555555551a9
0x555555551a9 <main>: 0xf3 0x0f 0x1e 0xfa 0x55 0x48 0x89 0
xe5
0x555555551b1 <main+8>: 0x48 0x83 0xec 0x30 0x64 0x48 0
x8b 0x04

```

Figure 8:gdb commands

I must see different options in *gdb* commands including *break \_start*, *run*, *info registers*, and *x/16bx 0x55....1a9*.

## Summary

On the first quiz or laboratory, .....