NCS lab 6
Understanding Assembly

Olga Chernukhina
BS-18-SB-01

Task 2 - Theory

1. What is the difference between mov and lea commands?

In short, `lea` acts like a **pointer** in C-type languages, while `mov` is like a regular **variable**

A more complex analysis:

criterium	mov	lea
abbreviation meaning	MOVe	Load Effective Address
semantics	copies the data in the source to the destination	loads the specified register with the offset of a memory location
format	MOV destination, source	`LEA` register, memory
can be indexed	no	yes
opcode	0xA0...0xA3	0x8D
INSTRUCTION reg, addr meaning	read a variable stored at address addr into register reg	read the address (not the variable stored at the address) into register reg

2. What is ASLR, and why do we need it?

Address Space Layout Randomization is an **algorithm** in operating systems for assigning **random memory addresses** to program components. ASLR **protects against** many **attacks**, particularly zero-day exploits, that are based on the hacker's ability to know or guess the **position** of processes and functions **in memory**.

3. How can the debugger insert a breakpoint in the debugged binary/application?

Usually, the requirement for setting a breakpoint without directly specifying the address is that the debugger knows where (at which address) program functions and lines of source code start. Moreover, the code had better not been optimized by the compiler.

Then there are **2 options** of inserting a breakpoint for debugger - **software** and **hardware** breakpoint. When inserting a **software breakpoint**, the debugger places a **special instruction** (usually some kind of interruption or exception) into **the copy of the binary** that is loaded to memory. To insert a **hardware breakpoint**, it is not necessary to change the binary. They are highly dependent on **CPU**, which compares **program counter** with **breakpoint addresses** and breaks the execution on a hit.

Task 3 - Reversing

1. Disable ASLR

```
cli2@cli2-VirtualBox:~/Downloads/ghidra_9.2.2_PUBLIC$ sudo sysctl -w kernel.ran
domize_va_space=0
kernel.randomize_va_space = 0
```

2. Load the binaries into a disassembler/debugger.

3. Does the function prologue and epilogue differ in 32bit and 64bit? What about calling conventions?

Prologue

x32

```
000105d3 8d 4c 24 04     LEA         ECX=>Stack[0x4],[ESP + 0x4]
000105d7 83 e4 f0        AND         ESP,0xfffffff0
000105da ff 71 fc        PUSH        dword ptr [ECX + local_res0]
000105dd 55              PUSH        EBP
000105de 89 e5           MOV         EBP,ESP
000105e0 53              PUSH        EBX
000105e1 51              PUSH        ECX
000105e2 83 ec 10        SUB         ESP,0x10
000105e5 e8 31 00        CALL        __x86.get_pc_thunk.ax
         00 00
```

x64

```
0010070f 55              PUSH        RBP
00100710 48 89 e5        MOV         RBP,RSP
00100713 48 83 ec 10     SUB         RSP,0x10
00100717 48 8d 45 fc     LEA         RAX=>x,[RBP + -0x4]
0010071b 48 89 c6        MOV         RSI,RAX
0010071e 48 8d 3d        LEA         RDI,[s_In_main(),_x_is_
         63 01 00 00
```

Having `__x86.get_pc_thunk.ax` and global pointer set at the beginning we understand x32 uses position-independent code, while x64 uses relative addressing

Epilogue

x32

```
00010611 8d 65 f8        LEA         ESP=>local_10,[EBP + -0x8]
00010614 59              POP         ECX
00010615 5b              POP         EBX
00010616 5d              POP         EBP
00010617 8d 61 fc        LEA         ESP,[ECX + -0x4]
0001061a c3              RET
```

x64

```
0010073e c9              LEAVE
0010073f c3              RET
```

In position-dependent code the epilogue usually looks like

```
mov   esp, ebp
pop   ebp
ret
```

Here the x32 semantics of the code is the same as x64, but it is performed with respect to position independence. x64 example has a regular position-dependent epilogue.

Calling conventions

Apart from using different-sized registers, having some mismatches in caller/callee-saved policy and passing arguments via stack+regs or stack only, they are the same.

4. Does function calls differ in 32bit and 64bit? What about argument passing?

Function calls:

x64

```
00100734 e8 51 ff        CALL        sample_function
         ff ff
```

x32

```
00010607 e8 41 ff        CALL        sample_function
         ff ff
```

x64

```
00100717 48 8d 45 fc     LEA         RAX=>x,[RBP + -0x4]
0010071b 48 89 c6        MOV         RSI,RAX
0010071e 48 8d 3d        LEA         RDI,[s_In_main(),_x_
         63 01 00 00
00100725 b8 00 00        MOV         EAX,0x0
         00 00
0010072a e8 21 fe        CALL        printf
         ff ff
```

x32

```
000105ef 8d 55 f4        LEA         EDX=>local_14,[EBP + -0xc]
000105f2 83 ec 08        SUB         ESP,0x8
000105f5 52              PUSH        EDX
000105f6 8d 90 88        LEA         EDX,[EAX + 0xffffe788]=>s_In
         e7 ff ff
000105fc 52              PUSH        EDX=>s_In_main(),_x_is_store
000105fd 89 c3           MOV         EBX,EAX
000105ff e8 cc fd        CALL        printf
         ff ff
```

Function calls are identical (as it is shown in the examples of `printf` and `sample_function`), arguments passing are different in the sense that x64 convention prescribes to pass first four parameters via registers, while x32 uses stack only.

5. What does the command ldd do? “ ldd BINARY-NAME ”.

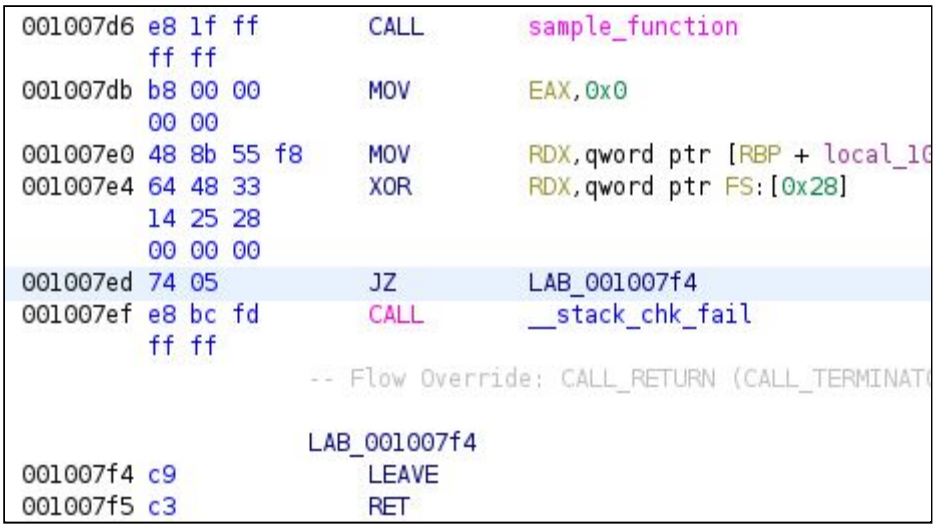
It lists the libraries required for `BINARY-NAME` execution

6. Why in the “sample64-2“ binary, the value of i didn’t change even if our input was very long?

x64

```
00100734 e8 51 ff        CALL        sample_function
         ff ff
00100739 b8 00 00        MOV         EAX,0x0
         00 00
0010073e c9              LEAVE
0010073f c3              RET
```

x64-2



The interface `__stack_chk_fail` stops the function execution if it was called with a message of stack overflow detection. Usually, gcc automatically includes a stack smashing protector starting from a certain variable size.

Bonus

Describe the difference between PE and ELF executable files. Try to show it by a practical example.

The primary difference is that PE is used in Windows and ELF in Unix systems. What is more, ELF files do not have a special extension and are available in a 64-bit version. ELF files are linked in such a way that the boundaries and sizes of the sections fall on 4-kilobyte blocks of the file. In PE format, despite the fact that the format itself allows you to align sections by 512 bytes, the alignment of sections by 4k is used, a smaller alignment in Windows is not considered correct.