COMP8050 – Security for Software Systems

Lab 1 – Introduction to gdb (and some assembly language, C)

Firstly: Start the Linux VM on lab computers.

- Login as "studentid@mtu.ie"
- Alternatively, you can use your own laptops as long as you have some version of Linux on it. Free link for a recommended Linux VM is available on Canvas.

Write a simple C program which we will work with (do not include the line numbers). You can call it **hello.c**

```
1.  #include <stdio.h>
2.
3.  void main( int argc, char *argv[] ){
4.
5.    printf("Hello World!\n");
6.
7.  }
```

As you can probably guess, this is a simple program that outputs the string "Hello World!". Line 1 is telling the program to include the stdio.h library to allow us to use the "printf" function on line 5. Printf is used to output strings to standard output (i.e. the screen). Line 3 is a standard main function definition – don't worry too much about what it means for now.

In order to compile and run the file use these commands:

```
gcc hello.c –o hello.o
./hello.o
```

gcc is the c compiler, and here we tell it to compile "hello.c" and use the "-o" command line option to name the output file as "hello.o". This output is a binary executable compiled to run on our system. The second command is to run the output file, and should give us the output "Hello World!".

If you get errors trying to compile the code, it may mean that your version of gcc is not up to date. To fix this, run the following two commands:

```
sudo apt-get update
sudo apt-get install build-essential
```

GDB is the GNU debugger which we can use to debug compiled binaries, and also to disassemble them (i.e. get their assembly language code from the binary). In this way we can carefully observe how our programs interact with the stack and how attacks to manipulate the stack will operate. To debug our program, we use the following:

```
gdb hello.o
```

- To exit GDB at any time, enter the command “q” or “quit”.

While we are debugging a program, we can execute the program with the command:

```
run
```

We can run the program in GDB as often as we like until we quit GDB.

We can set breakpoints to halt program execution at certain points so we can examine the state of the system memory/stack/heap/etc… Firstly, we will disassemble the binary to see the commands in assembly language. GDB supports 2 types of assembly language, AT&T (the default for Unix) and Intel (the default for Windows). Since Intel is easier to read, we shall use that format:

```
show disassembly-flavor             #check the current lang
set disassembly-flavor intel        #set the lang to intel
```

Then to disassemble the input binary (hello.o) we simply type:

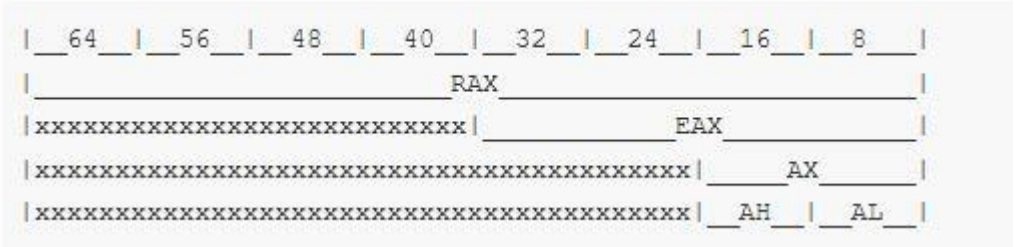
```
disassemble main
```

Instead of “main” you could give it other function names if they exist in the binary, or addresses (http://visualgdb.com/gdbreference/commands/disassemble). Since our program only contains the function “main” that is what we look at here.

This will give you output similar to this:(Note: your disassembled code may not be identical)

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000400526 <+0>:     push   rbp
   0x0000000000400527 <+1>:     mov    rbp,rsp
   0x000000000040052a <+4>:     sub    rsp,0x10
   0x000000000040052e <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x0000000000400531 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000400535 <+15>:    mov    edi,0x4005d4
   0x000000000040053a <+20>:    call   0x400400 <puts@plt>
   0x000000000040053f <+25>:    nop
   0x0000000000400540 <+26>:    leave
   0x0000000000400541 <+27>:    ret
End of assembler dump.
(gdb)
```

Note you may have different names for the registers: rbp, rsp, edi, etc… possibly: ebp, esp, edx, etc… if on a 32-bit machine instead of a 64-bit one. The below diagram shows the naming scheme for the AX general-purpose register with respect to bit sizes:



So the R-prefix is for 64bit registers, E-prefix is for 32bit, no prefix for standard 16bit registers and H/L suffixes for the 2 bytes composing the 16bit register.

The left of the disassembled code has the hexadecimal address of each instruction. e.g. The first address is 0x0000000000400526 or 0x400526 (we may exclude the leading 0s). We will need these addresses to reference particular lines of assembly code!

The first 3 lines (0x400526 to 0x40052a):

push rbp

mov rbp,rsp

sub rsp, 0x10

Are to update the values of the base/frame pointer (ebp/rbp) and stack pointer (esp/rsp) to take account of the new function call. “Push” is used to store the current value of rbp on the stack. “Mov” moves the current value of rsp into rbp (since what was previously the top of the stack is now the start of this new stack frame). “Sub” subtracts 0x10 from the current value of rsp to move it to the new top of the stack (0x10 being the size of the new stack frame for this function).

The next few lines are storing the string value “Hello World!” onto the stack from memory address 0x4005d4 where it is stored. The current values in edi and rsi, are stored on the stack at addresses relative to rbp. edi is then set to contain the address 0x4005d4 which is where the string is stored in memory.

Let’s set a breakpoint at the first line, address 0x400526:

```
break *0x400526
```

Note that the leading 0s can be dropped from the address (but not the “0x” which tells it is a hex value), the * is also necessary to indicate we are setting a break at an address.

To remove all break points, we can use the del command.

Now let’s run the program again after having set our breakpoint!

```
(gdb) break *0x400526
Breakpoint 1 at 0x400526
(gdb) run
Starting program: /home/osboxes/hello.o

Breakpoint 1, 0x0000000000400526 in main ()
(gdb)
```

We can see that when running the program, GDB will halt at the breakpoint(s). While at this break point, we can inspect the current contents of the registers:

```
info registers
```

```
(gdb) info registers
rax            0x400526 4195622
rbx            0x0      0
rcx            0x0      0
rdx            0x7fffffffe038   140737488347192
rsi            0x7fffffffe028   140737488347176
rdi            0x1      1
rbp            0x400550 0x400550 <__libc_csu_init>
rsp            0x7fffffffdf48   0x7fffffffdf48
r8             0x4005c0 4195776
r9             0x7ffff7de78e0   140737351940320
r10            0x846    2118
r11            0x7ffff7a2e740   140737348036416
r12            0x400430 4195376
r13            0x7fffffffe020   140737488347168
r14            0x0      0
r15            0x0      0
rip            0x400526 0x400526 <main>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
```

Here we can see the current contents of the registers, in hexadecimal and decimal. Note that if you run the “info registers” command when the program is not running, we will get an error message as all the registers are empty.

The rax,rbx, rcx, rdx, rsi, rdi registers are general purpose registers (more info on why rsi and rdi don’t follow the naming scheme of the first 4: https://stackoverflow.com/questions/23367624/intel-64-rsi-and-rdi-registers)

r8-r15 are also general purpose.

While rbp, rsp and rip are the base pointer, stack pointer and instruction pointer respectively.

To continue running from a breakpoint we use:

```
c
```

Or “continue” if you wish to type it in full. This will resume execution until the next breakpoint, or the end of the program.

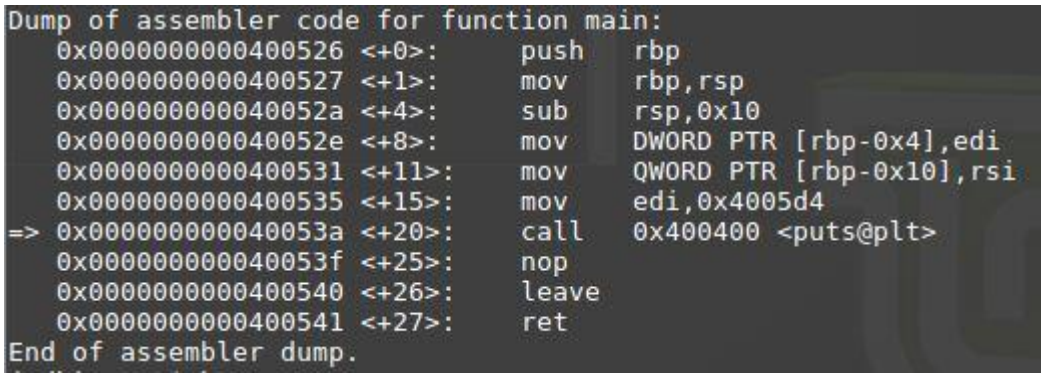
Alternatively, we can continue execution 1 instruction at a time:

```
si
```

Which will move us ahead a single assembly instruction. Combining this with examining the registers can allow us to see clearly how each instruction alters the contents of the registers!

If you type “disassemble” while the program is running, it will show you with an arrow where you currently are, if you get lost!

Either using a breakpoint, or si to get there incrementally... continue until the program has reached “call” instruction (address 0x40053a for my pictures).



This “call” instruction is calling a function (“puts”, because we did not need the complex “printf” for the output we wanted so the compiler optimized it to “puts” instead) which is part of a shared library (which we included at the start of our C code).

We can use the "x/" command to examine the contents of an address in memory. To see the contents of the memory holding the string for example we can enter:

```
(gdb) x/s 0x4005d4
0x4005d4:       "Hello World!"
```

How did we know 0x4005d4 was the location of the string?

We can see that the value of that address was copied to register edi (32bit of 64bit register rdi) in instruction at address 0x400535. Also no other memory address was copied into a register, and the address of the string had to be stored somewhere… So by elimination we could deduce that it was there. It is okay to not understand 100% of why all the assembly instructions are there. We want to focus on being able to identify the ones relevant to the major operations in the program code, e.g. function calls, variable assignments, etc..

The options for examining when using x/ are

Display format

x displays in hexadecimal

d displays in decimal

u displays in unsigned decimal

o displays in octal

t displays in binary (t for two)

a displays address both in hexadecimal and as an offset from the nearest preceding symbol

c displays as a character

s displays as a null-terminated string

t displays as a floating-point number

i displays as a machine instruction

Initial default is x. The default changes each time x is used.

It is also possible to show the contents of multiple addresses starting from the given address by including a number as follows:

```
x/24x <address>
```

This will output the contents of 24 addresses in hexadecimal (due to the x/x) format from the given address. We may also use registers or relative addresses to specify the starting point. e.g.

```
x/24x $rsp
x/12x $rbp-0x4
```

Which would output 24 addresses contents from the address of the value in the rsp register (i.e. the top of the stack, so we can view what's on the stack in this way).

The second command would show the contents of 12 addresses in hex from the address at the value of the register rbp minus 0x4.

We can also define what actions to take when we reach a breakpoint, so that we don't manually have to repeatedly give the same commands over and over again. We do this using what are called "Hook Stops".

```
define hook-stop
info registers
x/24x $rsp                  #$esp on 32-bit systems
x/2i $rip                   #$eip on 32-bit systems
end
```

The above commands will instruct GDB to run the 3 commands between "define hook-stop" and "end" every time a breakpoint is reached.

- In this Hook-stop, we check the registers in the first command.
- Then we examine 24 64-bit addresses from the address in the stack pointer (esp/rsp), displaying them as hexadecimal values.
- Lastly, we examine 2 64-bit addresses from the address of the instruction pointer (which shows the next instruction to be run) and display them as instructions (since /i instead of /x).

Set some break points in the program and run it, checking to ensure that your hook-stop works and outputs the 3 types of information from the 3 given commands.

When at a breakpoint, we can use the command:

```
info proc mappings
```

to show us the state of the mapped memory while the program is running. In particular, we can see the start and end addresses of the [stack]. If Address Space Layout Randomization (ASLR) is enabled, the address of the stack will vary slightly each time we run the program.

Here is another c program, **add.c**

```
1. #include <stdio.h>
2.
3.  void main( int argc, char *argv[] ){
4.    int a = 5;
5.    int b = 10;
6.    int c = a + b;
7.
8.    printf("c = %d", c);
9. }
```

Which simply adds two numbers and outputs the result.

- Compile it, and then debug the program with GDB.
- Create a hookstop which shows you the contents of the registers and observe how they change as you move through the code (set multiple break points).
- Identify which instructions are performing the assignments to “a” and “b”.
- Identify the instructions adding the values of “a” and “b” and assigning to “c”.
- Make sure you understand what it is you’re looking at in GDB, if you have questions ask!

Here is another program, **userinput.c** ,which prompts a user for input, stores it in a buffer of size 6 bytes (i.e. 6 characters), and then prints the buffer to the screen.

```
1. #include <stdio.h>
2. #include <string.h>
3.
4.  void main( int argc, char *argv[] ){
5.    char buffer[6];
6.
7.    gets( buffer );
8.    printf("%s", buffer);
9. }
```

You will get some warnings when compiling this program, relating to "gets", but you can ignore them.

- Compile **userinput.c**, and then debug the program with GDB.

- Create a hook-stop which shows the contents of the stack (say 24 addresses from the top of the stack, rsp) and observe how they change as you move through the code (set multiple break points). Use a hook-stop which shows the stack in hexadecimal values.

- The "gets" function attempts to read in user input. This string input you provide to the "gets" call is stored on the stack. Use input you can recognise to identify where in the stack your input is stored. The hex values of string values can be found here: http://www.asciitable.com/

- Test with different lengths of input, observe what happens to the stack, and what happens when you provide an input which is too long.

- Think about the reasons for this:
    - What length of input is acceptable?

    - If you look at the source code, what would you think is the largest valid input length? Is this the same as the largest input the program accepts without errors?

    - If the length of the valid input is not what you expected (which may or may not happen depending on the machine it is running on..): looking at the stack may help make it clear why it is a different size to what you might expect from the source code.