**Observing Function Call and Return using GDB**

The goal of this group assignment is to get familiar with the GDB debugger, and use it to understand the low-level function call and return mechanism used by Intel CPUs.

**Lab setup**

You will work on your Ubuntu machine or VM.

**Practice steps**

1. To mitigate memory exploits, including buffer overflow, Ubuntu has its address-space randomization turned on by default. We need to turn it off for easily observing the low-level mechanisms for call and return. Use the following command to disable address-space randomization:

sudo sysctl -w kernel.randomize\_va\_space=0

Note: also look in /proc/sys/kernel

GDB by default disables randomization. To turn it off:

set disable-randomization off

1. Download the sample program from the following address:

<http://www.comp.nus.edu.sg/~liangzk/cs4238/overflowsample.tar.gz>

Find the downloaded file in Downloads directory in your home folder, and extract the file sample.c from the compressed file.

1. Compile the source file sample.c with stack-protector disabled   
   ( -fno-stack-protector), debugging information ( -g ), and generate an executable file named sample ( -o sample ):

gcc -fno-stack-protector -g -o sample sample.c

Ignore any warnings.

1. Start the GDB debugger:

gdb ./sample

1. Set a breakpoint at the beginning of the main() function:

(Under the gdb prompt) break main



Note: The command can be shortened to “b main”.

1. Now we can start to execute the program:

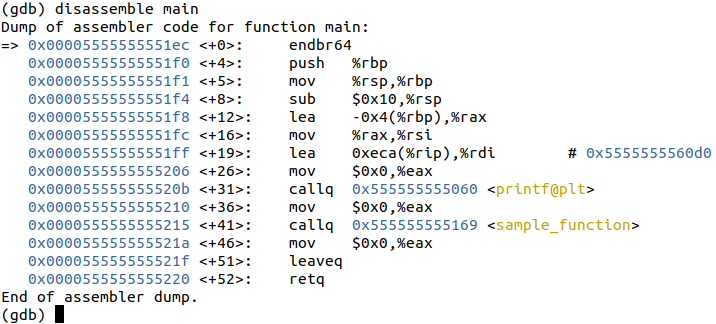
(Under the gdb prompt) run ./sample

Or simply run



Now the program stops at main().

1. Before we continue the program execution, disassemble the main function to note down an important value from the program:



(Under the gdb prompt) disassemble main

If you are more comfortable with intel style of disassembly, you can change it via command set disassembly-flavor intel

Table

Description automatically generated

This is the assembly code of the main() function. Each instruction line starts with the memory address of that instruction, followed by the disassembled instruction. Note that the instruction at the address 0x 555555555215 (the instruction above the red line) is the call to sample\_function(). Therefore, when the function returns, it should continue to execute the next instruction, whose address is 0x55555555521a. Note down this address for a comparison later.

**Note: the listings above are illustrative. It is possible for the exact details to vary with different compiler flags, compiler versions, and other compilers.**

You can also use gcc with –S to only compile sample.c into the assembly code in sample.s.

1. Do “next” for twice to step over executing the printf() functions. From the output, you can see the memory address of the variable x.

(Under the gdb prompt) next

Text

Description automatically generated with medium confidence

Now the program is about to call the function sample\_function().

1. Let’s inspect the register values:

(Under the gdb prompt) info registers

Table

Description automatically generated

This command shows the value of integer registers and the decoded value. Here we just need to use the first number (hexadecimal value of the register).

We can see that the stack pointer RSP is at 0x7fffffffdeb0. The frame (base) pointer RBP is at 0x7fffffffdec0. The instruction pointer RIP is at 0x555555555210. Can you check, from the disassembly of main(), which instruction will be executed next?

1. Before we enter the sample\_function(), do a disassemble of the sample function.

Table

Description automatically generated

The first instruction is an indirect branch terminating instruction. You may think of it as a nop instruction which does nothing.

The following three instructions of this function is common across most of the functions generated by the gcc compiler without optimization. It saves the base pointer on the stack (push rbp), point the base pointer to the current stack top (mov rbp,rsp), and move down the stack pointer to allocate space for local variables (sub rsp,0x10). The rest of the instructions is generated from the C code of sample\_function().

1. Let’s see what will happen to the stack when the program enters sample\_function(). The stack pointer is originally at 0x7fffffffdeb0, shown in the previous “info registers” command.

Turn on disassemble by “set disassemble-next-line on” and use “stepi” to step into sample\_function.

Calendar

Description automatically generated

First, a return address will be pushed on the stack by the call instruction. A return address is 8 bytes on a 64-bit computer. Therefore, the stack pointer should be at 0x7fffffffdeb0 – 0x8 = 0x7fffffffdea8.

We can check by “info registers” command.

Table

Description automatically generated

And check the saved return address on the stack, and it should be the value we recorded in STEP 7:

A picture containing text

Description automatically generated

(Please note the endianness)

1. Next, the push rbp instruction will push an 8-byte RBP on to the stack. The stack pointer will be moved down by 8, resulting in a new value 0x7fffffffdea8– 0x8 = 0x7fffffffdea0.

Then, the mov rbp,rsp instruction will set RBP to the value of RSP, 0x7fffffffdea0.

Finally, the stack pointer is moved down by 0x10 to make space for local variables. The new stack pointer ESP is 0x7fffffffdea0– 0x10 = 0x7fffffffde90. Therefore, the local variables of sample\_function() should be in the range of 0x7fffffffde90 to 0x7fffffffdea0.

We can check whether the registers match our analysis or not by stepping over the sub rsp,0x10 instruction and print the registers.

Table

Description automatically generated

13. Now where is the saved return address?

Text

Description automatically generated

**Task:**

Use a figure to illustrate the stack layout when the program is:

(1) right before sample\_function() is called;

(2) inside sample\_function() after stack memory allocation;

(3) right after sample\_function()returns.

Mark the location of the stack pointer, the base pointer, and return address.

Work out the role of the stack pointer (RSP), the base pointer (RBP), and the instruction pointer (RIP) in the program.

You can also modify the code to see how things change, e.g. adding more variables to see how the stack frame changes, change the compiler options (e.g. –O2), etc.

Take note that there are many GDB extensions which provide enhanced feature on GDB, you can explore them on your own. Just to list out the most common three:

peda: <https://github.com/longld/peda>

gef: <https://github.com/hugsy/gef>

pwndbg: <https://github.com/pwndbg/pwndbg>