Practice: Observing Function Call and Return using GDB

The goal of this practice 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.

Here is a document about the memory layout of programs in Linux: http://www.thegeekstuff.com/2012/03/linux-processes-memory-layout/

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

   ```
   sudo sysctl -w kernel.randomize_va_space=0
   ```

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

   ```
   wget http://www.comp.nus.edu.sg/~cs5231/demo/sample.c
   gcc -fno-stack-protector -g -o sample sample.c
   ```

3. Start the GDB debugger:

   ```
   gdb ./sample
   ```

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

   (Under the `gdb` prompt) `break main`

   ```
   (gdb) break main
   Breakpoint 1 at 0x8048487: file sample.c, line 21.
   (gdb)
   ```

5. Before we run the program under the debugger, disassemble the main function to note down an important value from the program.

   (Under the `gdb` prompt) `disassemble main`

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804847e <+0>:     push   %ebp
   0x0804847f <+1>:     mov    %esp,%ebp
   0x08048481 <+3>:     and    $0xfffffff0,%esp
   0x08048484 <+6>:     sub    $0x20,%esp
   0x08048487 <+9>:     mov    $0x804863c,%eax
   0x0804848c <+14>:    lea    0x1c(%esp),%edx
   0x08048490 <+18>:    mov    %edx,0x4(%esp)
   0x08048494 <+22>:    mov    %eax,(%esp)
   0x08048497 <+25>:    call   0x8048320 <printf@plt>
   0x0804849c <+30>:    call   0x8048414 <sample_function>
   0x080484a1 <+35>:    leave
   0x080484a2 <+36>:    ret
End of assembler dump.
(gdb)
```

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 *0x0804849c* (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 *0x080484a1* (the address in the red rectangle). Note down this address.

6. Now we can start to execute the program:

   (Under the `gdb` prompt) `run ./sample`
   Or simply `run`

```
(gdb) run
Starting program: /home/cs4238/teaching/demo/overflowsample/sample

Breakpoint 1, main () at sample.c:21
21              printf("In main(), x is stored at 0x%08x.\n", (unsigned int)&x);
(gdb)
```

   Now the program stops in `main()`, before calling the `printf()` function.

7. Do a single step, executing the `printf()` functions. From the output, you can see the memory address of the variable x.

   (Under the `gdb` prompt) `step`

```
(gdb) step
In main(), x is stored at 0xbffff33c.
22              sample_function();
(gdb)
```

Now the program is about to call the function `sample_function`.

8. Let's inspect the register values

   (Under the `gdb` prompt) `info registers`

```
(gdb) info registers
eax            0x26     38
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc5ff4       -1208197132
esp            0xbffff330       0xbffff330
ebp            0xbffff358       0xbffff358
esi            0x0      0
edi            0x0      0
eip            0x804849c        0x804849c <main+30>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

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

   We can see: the stack pointer ESP is at *0xbffff330*. The base pointer EBP is at *0xbffff358*. The instruction pointer EIP is at *0x0804849c*. Can you check from the disassembly of `main()`, which instruction will be executed next?

9. Before we enter `sample_function`, disassemble the function.

```
(gdb) disassemble sample_function
Dump of assembler code for function sample_function:
   0x08048414 <+0>:     push   %ebp
   0x08048415 <+1>:     mov    %esp,%ebp
   0x08048417 <+3>:     sub    $0x28,%esp
   0x0804841a <+6>:     movl   $0x0,-0xc(%ebp)
   0x08048421 <+13>:    mov    $0x8048580,%eax
   0x08048426 <+18>:    lea    -0xc(%ebp),%edx
   0x08048429 <+21>:    mov    %edx,0x4(%esp)
   0x0804842d <+25>:    mov    %eax,(%esp)
   0x08048430 <+28>:    call   0x8048320 <printf@plt>
   0x08048435 <+33>:    mov    $0x80485b0,%eax
   0x0804843a <+38>:    lea    -0x16(%ebp),%edx
   0x0804843d <+41>:    mov    %edx,0x4(%esp)
   0x08048441 <+45>:    mov    %eax,(%esp)
   0x08048444 <+48>:    call   0x8048320 <printf@plt>
   0x08048449 <+53>:    mov    -0xc(%ebp),%edx
   0x0804844c <+56>:    mov    $0x80485e4,%eax
   0x08048451 <+61>:    mov    %edx,0x4(%esp)
   0x08048455 <+65>:    mov    %eax,(%esp)
   0x08048458 <+68>:    call   0x8048320 <printf@plt>
   0x0804845d <+73>:    lea    -0x16(%ebp),%eax
   0x08048460 <+76>:    mov    %eax,(%esp)
   0x08048463 <+79>:    call   0x8048330 <gets@plt>
   0x08048468 <+84>:    mov    -0xc(%ebp),%edx
   0x0804846b <+87>:    mov    $0x8048610,%eax
   0x08048470 <+92>:    mov    %edx,0x4(%esp)
   0x08048474 <+96>:    mov    %eax,(%esp)
   0x08048477 <+99>:    call   0x8048320 <printf@plt>
   0x0804847c <+104>:   leave
   0x0804847d <+105>:   ret
End of_assembler dump.
```

The first three instructions of this function is common across most of the functions generated by the gcc compiler. It saves the base pointer on the stack (`push %ebp`), point the base pointer to the current stack top (`mov %esp, %ebp`), and move down the stack pointer to allocate space for local variables (`sub $0x28, %esp`). The rest of the instructions is generated from the C code of `sample_function`.

Let's see what will happen to the stack when the program enters `sample_function`. The stack pointer is originally at *0xbffff330*, shown in the previous "`info registers`" command.

First, a return address will be pushed on the stack by the call instruction. A return address is 4 bytes on a 32-bit computer. Therefore, the stack pointer will be at *0xbffff330 – 0x4 = 0xbffff32c*. This is the location of the return address of this activation of `sample_function`.

Next, the `push %ebp` instruction will push a 4-byte EBP on to the stack. The stack pointer will be moved down by 4, resulting in a new value *0xbffff32c – 0x4 = 0xbffff328*.

Then, the `mov %esp, %ebp` instruction will set EBP to the value of ESP, *0xbffff328*.

Finally, the stack pointer is moved down by 0x28 to make space for local variables. The new stack pointer ESP is *0xbffff328 – 0x28 = 0xbffff300*. Therefore, the local variables of `sample_function` should be in the range of *0xbffff300* to *0xbffff328*.

10. Do a single step to enter sample_function

```
(gdb) step
sample_function () at sample.c:5
5               int i = 0xFFFFFFFF;
(gdb)
```

11. Check the register values to see whether they match our analysis

```
(gdb) info registers
eax            0x26     38
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc5ff4       -1208197132
esp            0xbffff300       0xbffff300
ebp            0xbffff328       0xbffff328
esi            0x0      0
edi            0x0      0
eip            0x804841a        0x804841a <sample_function+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```

12. Where will this program go after this function finishes? Let's check the return address. It is at location *0xbffff32c*. It can also be found by EBP+4, why?

(Under the `gdb` prompt) `x/xw 0xbffff32c`
or (Under the `gdb` prompt) `x/xw $ebp+4`

```
(gdb) x /xw 0xbffff32c
0xbffff32c:     0x080484a1
(gdb)
```

or

```
(gdb) x /xw $ebp+4
0xbffff32c:     0x080484a1
```

You can also check the return address byte-by-byte (Under the `gdb` prompt) `x/4xb 0xbffff32c`

Task:

Use a figure to illustrate the stack layout when the program is (1) right before sample_function is called; (2) in sample_function; (3) right after sample_function returns. Mark the location of the stack pointer, the base pointer, and return address. Also describe the role of the stack pointer (esp), the base pointer (ebp), and the instruction pointer (eip) in a program.