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/](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.

   ```bash
   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`).

   ```bash
   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:

   ```bash
   gdb ./sample
   ```

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

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

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

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

   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
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](image)

   This command shows the value of registers and the decoded value. Here we just need to use the first number (hexadecimal 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.
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 \(0xbffff32c - 0x4 = 0xbfffff328\).

Then, the `mov %esp, %ebp` instruction will set EBP to the value of ESP, \(0xbfffff328\).

Finally, the stack pointer is moved down by \(0x28\) to make space for local variables. The new stack pointer ESP is \(0xbffff328 - 0x28 = 0xbfffff300\). Therefore, the local variables of `sample_function` should be in the range of \(0xbfffff300\) to \(0xbfffff328\).

10. Do a single step to enter `sample_function`

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

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

```
(gdb) info registers
eax  0x26  38
ecx  0x0   0
dx   0x0   0
ebx  0xb7fc5ff4  -1208197132
esp  0xbffff300  0xbffff300
ebp  0xbffff328  0xbffff328
esti 0x0    0
edx  0x0    0
ebx  0xbfffff328  0xbfffff328
flags 0x286  [PF SF IF ]
cx  0x73   115
ss  0x7b   123
ds  0x7b   123
es  0x7b   123
fs  0x0    0
```

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
or

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.