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
   ```

2. 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.

3. 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.

4. Start the GDB debugger:

```
gdb ./sample
```

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

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

```
(gdb) break main
Breakpoint 1 at 0x11ec: file sample.c, line 18.
(gdb)
```

   Note: The command can be shortened to "`b main`".

6. Now we can start to execute the program:

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

   Or simply `run`

```
(gdb) run
Starting program: /home/kaihang/CS4238/overflowsample/sample

Breakpoint 1, main () at sample.c:18
18	{
(gdb)
```

   Now the program stops at `main()`.

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

```
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x00005555555551ec <+0>:	endbr64
   0x00005555555551f0 <+4>:	push   %rbp
   0x00005555555551f1 <+5>:	mov    %rsp,%rbp
   0x00005555555551f4 <+8>:	sub    $0x10,%rsp
   0x00005555555551f8 <+12>:	lea    -0x4(%rbp),%rax
   0x00005555555551fc <+16>:	mov    %rax,%rsi
   0x00005555555551ff <+19>:	lea    0xeca(%rip),%rdi        # 0x5555555560d0
   0x0000555555555206 <+26>:	mov    $0x0,%eax
   0x000055555555520b <+31>:	callq  0x555555555060 <printf@plt>
   0x0000555555555210 <+36>:	mov    $0x0,%eax
   0x0000555555555215 <+41>:	callq  0x555555555169 <sample_function>
   0x000055555555521a <+46>:	mov    $0x0,%eax
   0x000055555555521f <+51>:	leaveq
   0x0000555555555220 <+52>:	retq
End of assembler dump.
(gdb)
```

(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`

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
=> 0x00005555555551ec <+0>:     endbr64
   0x00005555555551f0 <+4>:     push   rbp
   0x00005555555551f1 <+5>:     mov    rbp,rsp
   0x00005555555551f4 <+8>:     sub    rsp,0x10
   0x00005555555551f8 <+12>:    lea    rax,[rbp-0x4]
   0x00005555555551fc <+16>:    mov    rsi,rax
   0x00005555555551ff <+19>:    lea    rdi,[rip+0xec2]        # 0x5555555560c8
   0x0000555555555206 <+26>:    mov    eax,0x0
   0x000055555555520b <+31>:    call   0x555555555060 <printf@plt>
   0x0000555555555210 <+36>:    mov    eax,0x0
   0x0000555555555215 <+41>:    call   0x555555555169 <sample_function>
   0x000055555555521a <+46>:    mov    eax,0x0
   0x000055555555521f <+51>:    leave
   0x0000555555555220 <+52>:    ret
End of assembler dump.
```

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`.

8. 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`

```
(gdb) next
21              printf("In main(), x is stored at %p.\n", &x);
(gdb) next
In main(), x is stored at 0x7fffffffdebc.
22              sample_function();
```

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

9. Let's inspect the register values:

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

```
(gdb) info register
rax            0x2a                42
rbx            0x555555555230      93824992236080
rcx            0x0                 0
rdx            0x0                 0
rsi            0x5555555592a0      93824992252576
rdi            0x7ffff7fac4c0      140737353794752
rbp            0x7fffffffdec0      0x7fffffffdec0
rsp            0x7fffffffdeb0      0x7fffffffdeb0
r8             0x0                 0
r9             0x2a                42
r10            0x5555555560e4      93824992239844
r11            0x246               582
r12            0x555555555080      93824992235648
r13            0x7fffffffdfb0      140737488347056
r14            0x0                 0
r15            0x0                 0
rip            0x555555555210      0x555555555210 <main+36>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) |
```

   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?

10. Before we enter the `sample_function()`, do a disassemble of the sample function.

```
(gdb) disassemble sample_function
Dump of assembler code for function sample_function:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   rbp
   0x000055555555516e <+5>:     mov    rbp,rsp
   0x0000555555555171 <+8>:     sub    rsp,0x10
   0x0000555555555175 <+12>:    mov    DWORD PTR [rbp-0x4],0x0
   0x000055555555517c <+19>:    lea    rax,[rbp-0x4]
   0x0000555555555180 <+23>:    mov    rsi,rax
   0x0000555555555183 <+26>:    lea    rdi,[rip+0xe7e]        # 0x555555556008
   0x000055555555518a <+33>:    mov    eax,0x0
   0x000055555555518f <+38>:    call   0x555555555060 <printf@plt>
   0x0000555555555194 <+43>:    lea    rax,[rbp-0xe]
   0x0000555555555198 <+47>:    mov    rsi,rax
   0x000055555555519b <+50>:    lea    rdi,[rip+0xe96]        # 0x555555556038
   0x00005555555551a2 <+57>:    mov    eax,0x0
   0x00005555555551a7 <+62>:    call   0x555555555060 <printf@plt>
   0x00005555555551ac <+67>:    mov    eax,DWORD PTR [rbp-0x4]
   0x00005555555551af <+70>:    mov    esi,eax
   0x00005555555551b1 <+72>:    lea    rdi,[rip+0xeb0]        # 0x555555556068
   0x00005555555551b8 <+79>:    mov    eax,0x0
   0x00005555555551bd <+84>:    call   0x555555555060 <printf@plt>
   0x00005555555551c2 <+89>:    lea    rax,[rbp-0xe]
   0x00005555555551c6 <+93>:    mov    rdi,rax
   0x00005555555551c9 <+96>:    mov    eax,0x0
   0x00005555555551ce <+101>:   call   0x555555555070 <gets@plt>
   0x00005555555551d3 <+106>:   mov    eax,DWORD PTR [rbp-0x4]
   0x00005555555551d6 <+109>:   mov    esi,eax
   0x00005555555551d8 <+111>:   lea    rdi,[rip+0xeb9]        # 0x555555556098
   0x00005555555551df <+118>:   mov    eax,0x0
   0x00005555555551e4 <+123>:   call   0x555555555060 <printf@plt>
   0x00005555555551e9 <+128>:   nop
   0x00005555555551ea <+129>:   leave
   0x00005555555551eb <+130>:   ret
End of assembler dump.
(gdb) |
```

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()`.

11. 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.

```
(gdb) set disassemble-next-line on
(gdb) stepi
0x0000555555555215      22          sample_function();
   0x0000555555555210 <main+36>:    b8 00 00 00 00  mov    eax,0x0
=> 0x0000555555555215 <main+41>:    e8 4f ff ff ff  call   0x555555555169 <sample_function>
   0x000055555555521a <main+46>:    b8 00 00 00 00  mov    eax,0x0
(gdb) si
sample_function () at sample.c:4
4       {
=> 0x0000555555555169 <sample_function+0>:     f3 0f 1e fa     endbr64
   0x000055555555516d <sample_function+4>:     55      push   rbp
   0x000055555555516e <sample_function+5>:     48 89 e5        mov    rbp,rsp
   0x0000555555555171 <sample_function+8>:     48 83 ec 10     sub    rsp,0x10
(gdb)
```

    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.

```
(gdb) info registers
rax            0x0                 0
rbx            0x555555555230      93824992236080
rcx            0x0                 0
rdx            0x0                 0
rsi            0x5555555592a0      93824992252576
rdi            0x7ffff7fac4c0      140737353794752
rbp            0x7fffffffdec0      0x7fffffffdec0
rsp            0x7fffffffdea8      0x7fffffffdea8
r8             0x0                 0
r9             0x2a                42
r10            0x5555555560e4      93824992239844
r11            0x246               582
r12            0x555555555080      93824992235648
r13            0x7fffffffdfb0      140737488347056
r14            0x0                 0
r15            0x0                 0
rip            0x555555555169      0x555555555169 <sample_function>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb)
```

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

```
(gdb) x/xg $rsp
0x7fffffffdea8: 0x000055555555521a
(gdb)
```

(Please note the endianness)

12. 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.

```
(gdb) ni
0x0000555555555171      4       {
   0x0000555555555169 <sample_function+0>:     f3 0f 1e fa     endbr64
   0x000055555555516d <sample_function+4>:     55      push   rbp
   0x000055555555516e <sample_function+5>:     48 89 e5        mov    rbp,rsp
=> 0x0000555555555171 <sample_function+8>:     48 83 ec 10     sub    rsp,0x10
(gdb) ni
5               int i = 0;
=> 0x0000555555555175 <sample_function+12>:    c7 45 fc 00 00 00 00    mov    DWORD PTR [rbp-0x4],0x0
(gdb) info registers
rax            0x0                 0
rbx            0x555555555230      93824992236080
rcx            0x0                 0
rdx            0x0                 0
rsi            0x5555555592a0      93824992252576
rdi            0x7ffff7fac4c0      140737353794752
rbp            0x7fffffffdea0      0x7fffffffdea0
rsp            0x7fffffffde90      0x7fffffffde90
r8             0x0                 0
r9             0x2a                42
r10            0x5555555560e4      93824992239844
r11            0x246               582
r12            0x555555555080      93824992235648
r13            0x7fffffffdfb0      140737488347056
r14            0x0                 0
r15            0x0                 0
rip            0x555555555175      0x555555555175 <sample_function+12>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) |
```

13. Now where is the saved return address?

```
(gdb) x/xg $rbp+0x8
0x7fffffffdea8: 0x000055555555521a
(gdb) x/xg $rsp+0x18
0x7fffffffdea8: 0x000055555555521a
(gdb) |
```

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