Computer Science Department

CSCI 247 Computer Systems I
Laboratory Exercise 5

Objectives

Practice reverse engineering an executable software

Submitting Your Work

Save your C program files as Exercise1.c, Exercise2.c and Exercise3.c and submit it via the **Lab Exercise 5 Submission** item on the course web site. You must submit your program by 11:59pm on the Friday following your scheduled lab session.

How to Reverse Engineer an Executable file

We start by writing and compiling a simple C program. We compile this program with `gcc` and then run the program within the `gdb` debug environment. In `gdb`, we use the `disassemble` command to convert the machine language of the program back to assembly language. This will show us where the variables in the program are stored and how the statements of the C program are implemented in assembly language.

1. Save the following C program as `sample.c`

```
int main()
{
        int x, y;
        x = 10;
        y = 15;
        x = y-x;
        y = x+y;
}
```

2. Compile the program:

```
gcc –g –o sample sample.c
```

3. Run the program in gdb

```
gdb ./sample
```

4. At the (gdb) prompt, set a breakpoint at the start of the main() function of the program by using the b command. This will make the program pause when it starts executing the main() function.

```
(gdb) b main
```

You will see the following output:

```
Breakpoint 1 at 0x4004da
```

5. A the (gdb) prompt, run the program by using the r command:

```
(gdb) r
```

You will see the following output:

```
Starting program: /home/gopalas/CSCI247/Lab5/Example

Breakpoint 1, 0x00000000004004da in main ()
(gdb)
```

6. Execution of the program has paused at the breakpoint. We can now disassemble the program by entering the disassemble command at the (gdb) prompt:

```
(gdb) disassemble
```

You will see the following output (or something similar):

```
Dump of assembler code for function main:
          0x00000000004004d6 <+0>: push %rbp
          0x00000000004004d7 <+1>: mov %rsp,%rbp
=>        0x00000000004004da <+4>: movl $0xa,-0x8(%rbp)
          0x00000000004004e1 <+11>: movl $0xf,-0x4(%rbp)
          0x00000000004004e8 <+18>: mov -0x4(%rbp),%eax
          0x00000000004004eb <+21>: sub -0x8(%rbp),%eax
          0x00000000004004ee <+24>: mov %eax,-0x8(%rbp)
          0x00000000004004f1 <+27>: mov -0x8(%rbp),%eax
          0x00000000004004f4 <+30>: add %eax,-0x4(%rbp)
          0x00000000004004f7 <+33>: mov $0x0,%eax
          0x00000000004004fc <+38>: pop %rbp
          0x00000000004004fd <+39>: retq
End of assembler dump.
```

```
(gdb)
```

Note that %rbp (the base pointer) is set to the top of the stack. The local variables x and y are stored on the stack at positions beyond the base pointer. Since x and y are both int, each requires 4 bytes so their values are stored at addresses -0x8(%rbp) and -0x4(%rbp), respectively.

Here's the assembly instructions again, with annotations:

```
push %rbp                          # save the old value of %rbp
mov %rsp,%rbp                      # set %rbp to the value of %rsp
movl $0xa,-0x8(%rbp)               # x = 10
movl $0xf,-0x4(%rbp)               # y = 15
mov -0x4(%rbp),%eax                # load y into %eax
sub -0x8(%rbp),%eax                # y - x is now in %eax
mov %eax,-0x8(%rbp)                # x = y - x
mov -0x8(%rbp),%eax                # load x into %eax
add %eax,-0x4(%rbp)                # y = x + y
mov $0x0,%eax
pop %rbp                           # restore the old value of %rbp
retq                               # return from main()
```

Now see if you can reverse engineer the executable programs provided on the course web site: Example1, Example2 and Example3.