Lab 6: GDB with Assembly

For this lab, create a new directory named `lab6` under your `cs449` directory and create your program there:

```
mkdir lab6
cd lab6
```

For this lab, you can get the starter file (`sample`) to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/sample .
```

For those who works on Ubuntu on your own computer, the starter file is also available on the CourseWeb. Note that the starter file `sample` is not a C source file. It is an executable file. So, run this program and enter a number when prompt:

```
./sample
Enter a number: 123
Boom!!!
```

If you encounter `"Permission Denied..."`, execute the following command:

```
chmod +x sample
```

and execute the program again. So, in this lab, we are going to try to find out what number we should enter.

1 Walkthrough

Let's use `gdb` to trace this program:

```
gdb sample
```

As usual, a good place to start is at the `main` function. So, let's set the breakpoint there:

```
(gdb) b main
Breakpoint 1 at 0x4005b8
```

Then run using the `r` command until it hits the breakpoint:

```
(gdb) r
...
Breakpoint 1, 0x00000000004005b8 in main ()
```

Now, let's try to see the source code using the command `list`:

```
(gdb) list
No symbol table is loaded.  Use the "file" command.
```

The above information says that this executable file does not have its source code attached to it. In other words, it was compiled without `-g` flag. So, we are out of luck, we need to trace this code in assembly. To do so, use `disas` command:

```
(gdb) disas
Dump of assembler code for function main:
   0x08048464 <+0>:     push   %ebp
   0x08048465 <+1>:     mov    %esp,%ebp
=> 0x08048467 <+3>:     and    $0xfffffff0,%esp
   0x0804846a <+6>:     sub    $0x20,%esp
   0x0804846d <+9>:     mov    $0x8048590,%eax
   0x08048472 <+14>:    mov    %eax,(%esp)
   0x08048475 <+17>:    call   0x8048360 <printf@plt>
   0x0804847a <+22>:    mov    $0x80485a1,%eax
   0x0804847f <+27>:    lea    0x1c(%esp),%edx
   0x08048483 <+31>:    mov    %edx,0x4(%esp)
   0x08048487 <+35>:    mov    %eax,(%esp)
   0x0804848a <+38>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x0804848f <+43>:    mov    0x1c(%esp),%eax
   0x08048493 <+47>:    cmp    $0x1c1,%eax
   0x08048498 <+52>:    jne    0x80484a8 <main+68>
   0x0804849a <+54>:    movl   $0x80485a4,(%esp)
   0x080484a1 <+61>:    call   0x8048370 <puts@plt>
   0x080484a6 <+66>:    jmp    0x80484b4 <main+80>
   0x080484a8 <+68>:    movl   $0x80485af,(%esp)
   0x080484af <+75>:    call   0x8048370 <puts@plt>
   0x080484b4 <+80>:    mov    $0x0,%eax
   0x080484b9 <+85>:    leave
   0x080484ba <+86>:    ret
End of assembler dump.
```

Do not be scared with what you see. First thing first, let's start with some notations:

- `eax`, `ebx`, `ecx`, `edx`, `esp` (stack pointer), and `ebp` (based pointer) are registers.
- Percent symbols (`%`) are used as a prefix to indicate registers (e.g., `%eax`)
- Dollar sign symbol (`$`) are used as a prefix to indicate constants (e.g., `$0x20`)
- A memory location (address) can be referred by effective addressing (e.g., `0x1c(%esp)` or `(%esp)`). Note that this is similar to MIPS.
- The expression `R[reg]` where `reg` is a register will be used to describe operations related to registers. For examples,
  - `R[eax] = x` means store the value `x` into the register `eax`, and
  - `R[eax] = R[edx]` means store the value that is in the register `edx` into the register `eax`.
- The expression `M[addr]` where `addr` is a memory location (address) will be used to describe operations related to memory. For examples,

- M[addr] = x means store the value x into the memory at the address addr, and
- R[eax] = M[addr] means store the value that is in the memory at the address addr into the register eax

The above code shows the assembly code very similar to the MIPS assembly code that you have learned. Each instruction consists of an operator follows by a number of operands or no operand. Note that these are call X86 assembly/GAS syntax. For this assembly syntax, the last operand usually indicates destination.

So, let's make some observation about the above code. There are four `call` instructions as shown below:

```
:
0x08048475 <+17>:        call   0x8048360 <printf@plt>
:
0x0804848a <+38>:        call   0x80483a0 <__isoc99_scanf@plt>
:
0x080484a1 <+61>:        call   0x8048370 <puts@plt>
:
0x080484af <+75>:        call   0x8048370 <puts@plt>
:
```

Notice something familiar? `printf` and `scanf` are functions that you use all the time. This observation gives you a hint that in X86 assembly, the `call` instruction is used to call a function. So, what are those numbers before the function names? You may be able to guess that those are the addresses of the first instruction of each function. Remember MIPS? Instructions `j` and `jal` need to know the address where to jump to. Same as in X86. Now, let's make some more observations about these function calls in the original code. Did you see something similar among those four function calls? Notice that before each `call` instruction, there are a set of `mov` instruction that move something to the address point by the register `esp` (stack pointer):.

```
:
0x08048472 <+14>:        mov    %eax,(%esp)
0x08048475 <+17>:        call   0x8048360 <printf@plt>
:
0x08048483 <+31>:        mov    %edx,0x4(%esp)
0x08048487 <+35>:        mov    %eax,(%esp)
0x0804848a <+38>:        call   0x80483a0 <__isoc99_scanf@plt>
:
0x0804849a <+54>:        movl   $0x80485a4,(%esp)
0x080484a1 <+61>:        call   0x8048370 <puts@plt>
:
0x080484a8 <+68>:        movl   $0x80485af,(%esp)
0x080484af <+75>:        call   0x8048370 <puts@plt>
:
```

This is how X86 makes a function call. Before making a function call, arguments must be put onto the stack. Unlike MIPS where we usually put argument into registers `$a0` to `$a3`. In X86, stack is used to pass arguments since there is no specific set of registers for this purpose. So, let's focus on a function that we are familiar with, `printf`.

```
0x0804846d <+9>:      mov    $0x8048590,%eax
0x08048472 <+14>:     mov    %eax,(%esp)
0x08048475 <+17>:     call   0x8048360 <printf@plt>
```

The instruction `mov $0x8048590, %eax` store the constant `0x8048590` into the register `eax` (`R[eax] = 0x8048590`). The next instruction is `mov %eax, (%esp)` which takes the value stored in the register `eax` and put it into the memory location point by the register `esp` (`M[R[esp]] = R[eax]`). The question is, what is the value `0x8048590`? Is it a huge unsigned number or an address? Recall that the first argument of the function `printf()` is always a formatting string. Note that a string in C is an array of character. To send a string as an argument to a function, we send the address (pointer) of the first character to the function. This information give you a hint that the constant `0x8048590` may be an address (pointer). If it is an address, what is in there? To see what is in an address we use the `x` command:

```
(gdb) x 0x8048590
0x8048590:      0x65746e45
```

Look like a large number. But from our understanding of the function `printf`, the first argument must be a formatting string. Let's format the output into a null-terminated string using the command `x/s`:

```
(gdb) x/s 0x8048590
0x8048590:       "Enter a number: "
```

Remember that prompt what you run this program for the first time? Now you may know that those three instruction possibly associated to the statement

```
printf("Enter a number: ");
```

Note that the command `x/s` changes how the command `x` reads and displays the output. If you use the command `x` again without `/s`, you will see that this time, it shows a null-terminated string.

```
(gdb) x 0x8048590
0x8048590:       "Enter a number: "
```

The syntax of the `x` (examine) command is `x/nfu addr` or `x addr` where

- `addr` is a memory address
- `n` is the repeat count (in decimal integer). It specifies how much memory (counting by untis `u`) to display. The default is 1.
- `f` is the display format; `s` for null-terminated string, `i` for machine instruction, and `x` for hexadecimal (default).
- `u` is the unit size; `b` for bytes, `h` for halfwords (two bytes), `w` for words (four bytes), and `g` for giant words (eight bytes). The default is `w`.

So, we you want to change the way it displays the result back to its default, you have to use the command `x/1xw`. Now, let's focus on the `scanf` function:

```
   0x0804847a <+22>:        mov    $0x80485a1,%eax
   0x0804847f <+27>:        lea    0x1c(%esp),%edx
   0x08048483 <+31>:        mov    %edx,0x4(%esp)
   0x08048487 <+35>:        mov    %eax,(%esp)
   0x0804848a <+38>:        call   0x80483a0 <__isoc99_scanf@plt>
```

From the above code fragment, the first and forth instruction put the constant `0x80485a1` onto the stack. Let's see what is in that address:

```
(gdb) x 0x80485a1
0x80485a1:         "%i"
```

Look familiar to you? Possibly `scanf("%i",...);`. What about the second and third instruction? The instruction `lea` stands for Load Effective Address. `lea 0x1c(%esp), %dex` simply put the effective address `0x1c(%esp)` to the register `edx` (`R[edx] = 0x1c + R[esp]`). The third instruction moves that value onto the stack. Recall that we usually use `scanf` as `scanf("%i", &x);` for a variable `x`. So, the value `0x1c + R[esp]` must be the address of an unknown variable. So, now you know that whatever number you put in after the prompt, it will be stored at the memory location `0x1c + R[esp]`. So, let's try this by setting the breakpoint right after calling the function `scanf()` at the address `0x0804848f` and continue running the program:

```
(gdb) b *0x0804848f
Breakpoint 2 at 0x804848f
(gdb) c
Continuing.
Enter a number: 45

Breakpoint 2, 0x0804848f in main ()
```

Note that when we set the breakpoint, we have to use `*` to indicate that this is an address. Otherwise, `gdb` will try to set the breakpoint at line `0x08048493`. In the above example, I enter 45 at the prompt. Now let's see what is in the address `0x1c + R[esp]` by using the following command:

```
(gdb) x 0x1c + $esp
0xffffd0ac:        0x0000002d
```

If you see `"-"`, you may need to use the command `x/1xw 0x1c + $esp` to change the display back to one word hexadecimal. The value shown above is 0x2d which is 45 in decimal as we expected. Note that in the above command, I use `0x1c + $esp` as an address for the `x` command which imitate effective addressing.

We are almost done. Right now you know that the number that you entered will be stored at the memory location `0x1c + R[esp]` (`0x1c(%esp)`. Now we need see how that number is used by the program. Let's examine the rest of the program:

```
   0x0804848f <+43>:        mov    0x1c(%esp),%eax
   0x08048493 <+47>:        cmp    $0x1c1,%eax
   0x08048498 <+52>:        jne    0x80484a8 <main+68>
   0x0804849a <+54>:        movl   $0x80485a4,(%esp)
   0x080484a1 <+61>:        call   0x8048370 <puts@plt>
   0x080484a6 <+66>:        jmp    0x80484b4 <main+80>
   0x080484a8 <+68>:        movl   $0x80485af,(%esp)
   0x080484af <+75>:        call   0x8048370 <puts@plt>
```

The first instruction move the content from the memory at the address `0x1c + R[esp]` to the register `eax`. Remember? That is the number that you entered. So, after the first instruction, the register `eax` contains the number that you just entered. So, to verify, let's put a breakpoint at the address `0x08048493` which is the second instruction and continue:

```
(gdb) b *0x08048493
Breakpoint 3 at 0x8048493
(gdb) c
Continuing.

Breakpoint 3, 0x08048493 in main ()
```

Now let's see what is in the register `eax`. The command `info register reg` is used to see the content in the register `reg`. We can also use the command `info registers` to see contents of all register. For now, let's check the content of the register `eax`:

```
(gdb) info register eax
eax            0x7b        45
```

It says that the register `eax` contains the value 45, the value that you just entered. The next instruction is `cmp` which stands for CoMPare. X86 assembly uses instruction `cmp` to compare values. Flags will be set or reset based on the result of comparison. Instruction `jne` (jump if not equal) or `je` (jump if equal) will read the flags and jump accordingly. In the above code, the instruction `cmp` is `cmp $0x1c1, %eax`. Note that the dollar sign symbol indicates a constant. So, it compares your number with the constant `0x1c1` which is 449 in decimal. So, together with the next instruction (`jne`), you may be able to guess that, if the number that you entered is **NOT** equal to 449, it will jump to the address `0x80484a8`. At that address are these two instructions:

```
   0x080484a8 <+68>:        movl   $0x80485af,(%esp)
   0x080484af <+75>:        call   0x8048370 <puts@plt>
```

The function `puts` is a function that print a given string on the console screen. The string that it is going to print will be its argument which is again on the stack. So, let's examine what string is stored at the address `0x80455af`:

```
(gdb) x 0x80485af
0x80485af:       "Boom!!!"
```

Look familiar? Now you know that, if the number that you entered is not 449, the program will show the message `Boom!!!` on the console screen. Now, you already crack the program. Quit the `gdb` by pressing `Ctrl-D` or enter the command `quit`. Then run this program again but this time enter `449` when prompt.

2 Your Turn

For this section, you can get the starter file (`what`) to your directory using the following command:

```
cp /afs/cs.pitt.edu/usr0/tkosiyat/public/cs0449/what .
```

Again, this file is an executable file without C source embedded. When you run this program, it will ask you to enter three 1-digit numbers separated by space. Then it will ask for another number as shown below:

```
./what
Enter three 1-digit numbers separated by space: 1 4 3
Enter a number: 7
BOOM!!!
```

Again, if you encounter `"Permission Denied..."`, execute the following commnad:

```
chmod +x what
```

This time, by yourself, use `gdb` to find out what number do you need to enter after you enter the three 1-digit numbers. Is there any relationship among those numbers?

3 What to Hand In

For this lab, create a file named `lab6.txt` using your preferred text editor. In the file, explain the relationship between three 1-digit numbers and the last number that you have to enter so that you will not get the message `Boom!!!`.

To submit this lab, we are going to do the usual steps. First, let us go back up to our cs449 directory:

```
cd ..
```

Now, let us first make the archive. Type your username for the USERNAME part of the filename:

```
tar cvf USERNAME_lab6.tar lab6
```

And then we can compress it:

```
gzip USERNAME_lab6.tar
```

Which will produce a `USERNAME_lab6.tar.gz` file.

If you work on `cs449.cs.pitt.edu` (`thoth`) you can skip to the next section. **If you use Ubuntu your own machine, you need to transfer the file to `cs449.cs.pitt.edu` first**. This can simply be done by a command line. For example, assume that your username is `abc123` and you are in the same directory as the file `abc123_lab6.tar.gz`. To transfer the file to `cs449.cs.pitt.edu` use the following command:

```
scp abc123_lab6.tar.gz abc123@cs449.cs.pitt.edu:.
```

The above command will copy the file to your home directory in `cs449.cs.pitt.edu`. If you want to copy it to your `private` directory, use the following command:

```
scp abc123_lab6.tar.gz abc123@cs449.cs.pitt.edu:./private/.
```

Copy File to Submission Directory

We will then submit that file to the submission directory:

```
cp USERNAME_lab6.tar.gz /afs/cs.pitt.edu/public/incoming/CS0449/tkosiyat/sec1
```

Once a file is copied into that directory, you cannot change it, rename it, or delete it. If you make a mistake, resubmit a new file with slightly different name, being sure to include your username. For example `USERNAME_lab6_2.tar.gz`. **Check the due date of this lab in our CourseWeb under Labs/Recitations**.