**GDB & Assembly**

## 1 Introduction

This handout introduces GDB (the GNU symbolic debugger), as well as the basics of assembly. Portions of this activity will be useful for your future labs, and it will be absolutely necessary for you to know this information to succeed.

## 2 Getting Started

To begin this activity, login to a shark machine. Then download the handout archive by typing

```
wget http://www.cs.cmu.edu/~213/activities/gdb-and-assembly.tar
```

Extract the activity with `tar xf gdb-and-assembly.tar`, then `cd gdb-and-assembly`.

Run the program by typing `./act1` and follow its instructions for rerunning it inside GDB. For the remainder of this exercise, follow the instructions on your screen, writing the answers to any questions in the blanks on this sheet. Stuck? This sheet gives extra help and additional explanations.

## 3 Activity 1, Step 1

This first activity deals with stepping through functions, examining registers, assembly syntax, disassembling, and basic use of GDB. Once you run it inside GDB, it will prompt you with the commands to enter to continue through the activity. It will show you how to use GDB's `i`, `r`, and `help` commands (with more details available on the reference at the end of this writeup).

We can use the `r` command to run a program in GDB. Use the following commands to run `act1` in GDB.

```
shark> gdb ./act1
(gdb) r 1
```

Registers are basically local "variables" in assembly. They are not located in memory, but instead directly on the CPU core. Rather than referring to them by addresses, each register has a specific name. [1]

We can use the following command to view the contents of each register in GDB.

[1] Different architectures use different names, most commonly some version of r1, r2, etc.

```
(gdb) info registers
```

Given the information that GDB just stopped the program at the beginning of a function taking one or more arguments, write down your best guess at the purposes of the %rsp, %rdi, and %rax registers.

`%rsp` Holds stack pointer, points to top of stack.

`%rdi` Holds first argument. Historically, destination index.

`%rax` Holds function return value. Historically, accumulator.

## 4 Activity 1, Steps 2 and 3

Now continue to step 2 with the following command:

```
(gdb) r 2
```

We can disassemble a function to view the assembly translation of a specific function. This is useful to better understand the execution of a program.

When we type `(gdb) disassemble squareInt` as per the activity instructions, we see the following output.

```
Dump of assembler code for function squareInt:
   0x00000000004009c3 +0:     mov    %edi,%eax
   0x00000000004009c5 +2:     imul   %edi,%eax
   0x00000000004009c8 +5:     retq
End of assembler dump.
```

It is easiest to understand this "dump" by reading each line from right to left. Take the first line, for example:

- `mov %edi,%eax`: The assembly-language decoding of a machine instruction.
- <+0> : The offset, in bytes, of this instruction from the beginning of the function.
- `0x00000000004009c3` : The memory address of this instruction. (You will probably see a different number than we did when we wrote this handout.)

We see the disassembled output uses the `MOV` instruction, a powerful instruction which copies values between registers or load or store with memory. In the above line, `%edi` holds the argument to the function and `%eax` the return value. Fill in the pseudo C code for this function.

```
int squareInt(int x) {

    return ____________ x * x ____________

}
}
```

Type `(gdb) disassemble squareLInt`.

How do the names of the registers differ from `squareInt`?

The registers of `squareLInt` begin with an 'r' instead of an 'e', indicating that they use the full 64-bit x86-64 registers instead of their 32-bit IA32 counterparts (which are now the lower 32 bits of the 64-bit registers).

Now type `(gdb) disassemble squareFloat`. Did `squareFloat` use the same registers from before?

No, `squareFloat` uses the %xmm0 register, a special purpose register for floating-point numbers.

From the questions above, we see that different registers correspond to different types of data. `squareLInt` used long ints, hence it accessed `%rdi`, which is 64 bits. `squareInt` used ints, hence it accessed `%edi` which is the lower 32 bits of `%rdi`. Smaller types will access other sections of the registers. [2]

Now continue to step 3 by typing the following command into GDB

```
(gdb) r 3
```

Knowing that `%rdi` is the first argument and `%rsi` is the second. And knowing when a register is in (), then it is serving as a memory location. What do you think the function `whatIsThis` is doing?

The function `whatIsThis` swaps the values at the memory locations pointed to by its two arguments.

## 5 Activity 2, Step 0

Launch activity 2 with `shark> ./act2`.

Did `whatIsThis`: **C**ompare, **S**wap, **A**dd or **M**ultiply two numbers? Enter the character corresponding to the correct operation in GDB as the argument to run.

## 6 Activity 2, Step 1

The following function introduces us to the concept of array accesses. These memory references are of the following form:

```
D(B, I, S)
```

This can be interpreted as `D + B + I * S`. For example, the instruction `(%rdi,%rsi,4)` can be interpreted as `*(%rdi + 4 * %rsi)`.

If the components are missing from the assembly, they have reasonable defaults (usually 0).

With this knowledge, examine the assembly for the function `viewThis`.

What are the function's argument(s)? A memory address.

What is the return register of the function? %eax or, equivalently, %rax.

Which instruction(s) initialize the return register? The first one, mov 0x4(%rdi),%eax.

What does the function do? The function adds the first four intss of a given array.

[2] A useful thing to put on your sheet is a table of these mappings.

## 7 Activity 2, Step 2

Next, type `r a`. What does the function `viewThisNext` do? Follow the same sequence of steps as you did previously.

The function `viewThisNext` accesses the element of the array given by the first argument at the index given by the second argument.

This section introduces different ways to view the addresses and contents of variables in GDB. We have the following two useful commands: `p` and `x`.

> `p` lets you examine values of arbitrary C code within GDB. For example, `p &arr` shows the address where the array `arr` is located.

> `x` lets you print the memory contents of a specific address, i.e. dereferences a specific address. For example, `x/8d &arr` prints the first 8 entries of the array located at the address specified by `arr`. The `d` formatter specifies the type we want to view the array contents as, where `d` corresponds to decimal integers.

Continue following the directions in the program and understanding its interaction with memory.

## 8 Activity 2, Step 3

Lastly, begin step 3 by typing the following command into GDB.

```
(gdb) r L
```

This section introduces the instruction `lea`. `lea` or load effective address is a variant of the `mov` instruction except, unlike `mov`, `lea` doesn't reference memory at all. It computes the address for the specified location, otherwise using the memory reference form.

This instruction effectively computes a constant added to a register, plus another register that is multiplied by a factor of 2.

View the assembly for the function `mx`.

> Write down the four parts of `lea`'s address: D__0__ + B__%rdi__ + I__%rdi__ * S__2__
>
> After accounting for the left shift, what value does `mx` multiply its argument by? ____`mx` multiplies its argument by 12.____

## 9 GDB Command Reference

This page is merely a reference for you to have on hand, pleas skip ahead to the actual activity if you have not yet completed it. Commands discussed in this lecture:

1. '`c`' ('`continue`') resumes program execution. Not to be used lightly.
2. '`disassemble`' outputs the assembly translation of the function currently being executed, or the translation of a target function if one is supplied.
3. '`help`' Provides a wealth of information on the following topic. Like '`i`,' enter it without a topic to see the list of possible topics.
4. '`i`' ('`info`') Provides information about the following subcommand. While '`registers`' is a common subcommand (shows the contents of primary registers), just type '`i`' for a list of all of them.
5. '`p`' ('`print`') is an extremely flexible command that lets you examine values and execute essentially arbitrary C code from within GDB, using references defined within the current scope (only works with certain compiler flags). Example usage: '`print instance.field == NULL`'
6. '`q`' ('`quit`') quits GDB session after confirmation.
7. '`r`' (or '`run`') Restarts the program currently being debugged. By default, it will use the same arguments supplied to the program during its last execution. However, you can instead pass a space-separated list of custom command-line arguments.
8. '`x`' A very powerful command that prints the memory contents of the supplied address[3]. Can be used with the format 'x/length/format address-expression. ' Good formats to know include: `s` (string), `x` (hex), and `d` (decimal). Example usage: '`x/3x 0x42`' to dump 3 hexadecimal integers starting from memory address `0x42`

While not strictly a command, in many contexts, ^C (Ctrl+C) issues an interrupt signal to the foreground. In GDB, this returns you to a `(gdb)` prompt if the program was busy.
Assorted tools that will be more useful later on in the course:

1. '`gdb -tui ./something`' will let you display the C code being stepped through alongside the standard GDB console. You may occasionally have to hit ^L to redraw. Some students have reported issues with tui and bomblab.
2. '`n`' ('`next`') executes the next line of C code, then stops the program again. '`ni`' performs the equivalent for just the next assembly instruction.

[3]This is an actual expression, so `0x10 + (1 * (7 - 5))` will work, for instance.

3. '`b`' ('`break`') tells GDB to stop the program whenever it reaches a location, which may be a function (e.g. `foo`), source line (e.g. `file.c:80`), or address (e.g. `*0xdeadbeef`).
4. '`explore`' lets you follow chains of pointers between structs. For example, if you have a struct instance you want to inspect that is 4 levels of inheritance removed from the reference you have, you can '`explore reference`' to examine the fields of each parent struct along the pointer chain from great-grandparent to child.

## 10 FAQ

1. When the program starts, GDB reports: `(no debugging symbols found)...done` Is this a problem?

   No. While helpful for debugging, the symbols are not required. Similarly for bomblab, you will be working on a bomb that does not have any symbols provided.

2. GDB is printing out: `Program received signal SIGTRAP, Trace/breakpoint trap.` What is happening?!

   For this activity, we have hard-coded breakpoints into the program, so that you can work with GDB. This message is informing you that one of those hard-coded points was reached.

3. When I try to run the program in GDB again, GDB asks me if I want to start the program from the beginning (y or n)?

   You are currently debugging, so GDB wants to make sure you want to stop this execution and start again. For these exercises, press y then enter.