GDB debugger Introduction
FIT5003 Software Security week 1

Faculty of Information Technology, Monash University

1 Getting started with GDB

Whenever you want to find out more information about GDB commands feel free to search for it inside the documentation or by using the help command followed by your area of interest.

1.1 opening a program with GDB

A program can be opened for debugging in a number of ways. We can run GDB directly attaching it to a program:

```
gdb [executable-file]
```

Or we can open up GDB and then specify the program we are trying to attach to using the file or file-exec command:

```
 gdb
(gdb) file [executable-file]
```

Furthermore we can attach GDB to a running service if we know it's process id:

```
 gdb --pid [pid_number]
```

2 Experimentation - Task

Let's assume that you have the following program:

```
    #include <stdio.h>

int main(int arg_count, char *arg_list[]) {
   int i;
   char *hello = "Hello World";
   printf("Variable i is at %p and contains %d\n", &i, i);
   printf("Variable hello is at %p, contains %s\n", hello, hello);
   printf("There were %d arguments provided:\n", arg_count);
   for(i=0; i < arg_count; i++)
      printf("argument #%d\t-\t%s\n", i, arg_list[i]);
}
```

Copy the program in some document inside the provided VM and name it `test_code.c`

Then, compile the program by executing:

```
gcc -g test_code.c -o test_code
```

This will create an executable file called `test_code`

The Goal of this brief task is to find the following:

1. What is the exact functionality of the above program?
2. Disassemble the code and see what is the assembly behind the c file
3. Place a breakpoint just before the for loop of the code
4. At the breakpoint, find the following:
    - what is the current address that is been executed (where does the eip points to)?
    - where is the ebp pointing to?
    - Which is the address and what are the content of register ebx?
    - Which is the address of esp ?

In the following Sections-Subsections we give examples on how to perform the above task activities.

2.1 Disassembling

GDB allows disassembling of binary code using the disassemble command (it may be shortened to disas). The command can be issued either on a memory address or using labels.

In the following, you can see how to execute the command and an example result.

Command Line

```
(gdb) disas *main
Dump of assembler code for function main:
=> 0x080484c4 <+0>:,  push   ebp
   0x080484c5 <+1>:,  mov    ebp,esp
   0x080484c7 <+3>:,  and    esp,0xfffffff0
   0x080484ca <+6>:,  sub    esp,0x30
   0x080484cd <+9>:,  mov    DWORD PTR [esp+0x12],0x24243470
   0x080484d5 <+17>:,mov    DWORD PTR [esp+0x16],0x64723077
   0x080484dd <+25>:,mov    WORD PTR [esp+0x1a],0x21
....Output ommited.....
(gdb) disas 0x080484c4
Dump of assembler code for function main:
=> 0x080484c4 <+0>:,  push   ebp
   0x080484c5 <+1>:,  mov    ebp,esp
   0x080484c7 <+3>:,  and    esp,0xfffffff0
   0x080484ca <+6>:,  sub    esp,0x30
   0x080484cd <+9>:,  mov    DWORD PTR [esp+0x12],0x24243470
   0x080484d5 <+17>:,mov    DWORD PTR [esp+0x16],0x64723077
   0x080484dd <+25>:,mov    WORD PTR [esp+0x1a],0x21
```

2.2 Adding Breakpoints

Breakpoints are important to suspend the execution of the program being debugged in a certain place. Adding breakpoints is done with the break command. A good idea is to place a breakpoint at the main function of the program you are trying to exploit. Given the fact that you have already run objdump and disassembled the program you know the address for the start of the main function. This means that we can set a breakpoint for the start of our program in two ways:

```
(gdb) break *main
(gdb) break *0x[main_address_obtained_with_objdump]
```

The general format for setting breakpoints in GDB is as follows:

```
(gdb) break [LOCATION] [thread THREADNUM] [if CONDITION]
```

Issuing the break command with no parameters will place a breakpoint at the current address.

GDB allows using abbreviated forms for all the commands it supports. Learning these abbreviations comes with time and will greatly improve you work output. Always be on the lookout for using abbreviated commands.

The abbreviated command for setting breakpoints is simply b.

2.3 Listing Breakpoints

At any given time all the breakpoints in the program can be displayed using the info breakpoints command:

```
(gdb) info breakpoints
```

You can also issue the abbreviated form of the command

```
(gdb) i b
```

2.4 Deleting Breakpoints

Breakpoints can be removed by issuing the delete breakpoints command followed by the breakpoints number, as it is listed in the output of the info breakpoints command.

```
(gdb) delete breakpoints [breakpoint_number]
```

You can also delete all active breakpoints by issuing the following the delete breakpoints command with no parameters:

```
(gdb) delete breakpoints
```

Once a breakpoint is set you would normally want to launch the program into execution. You can do this by issuing the run command. The program will start executing and stop at the first breakpoint you have set.

```
(gdb) run
```

2.5 Execution flow

Execution flow can be controlled in GDB using the continue, stepi, nexti as follows:

Command Line

```
(gdb) help continue
#Continue program being debugged, after signal or breakpoint.
#If proceeding from breakpoint, a number N may be used as an argument,
#which means to set the ignore count of that breakpoint to N - 1 (so that
#the breakpoint won't break until the Nth time it is reached).
(gdb) help stepi
#Step one instruction exactly.
#Argument N means do this N times (or till program stops for another reason).
(gdb) help nexti
#Step one instruction, but proceed through subroutine calls.
#Argument N means do this N times (or till program stops for another reason).
```

You can also use the abbreviated format of the commands: c (continue), si (stepi), ni (nexti).

If at any point you want to start the program execution from the beginning you can always reissue the run command.

Another technique that can be used for setting breakpoints is using offsets.

As you already know, each assembly instruction takes a certain number of bytes inside the executable file. This means that whenever you are setting breakpoints using offsets you must always set them at instruction boundaries.

In the following, you can see how to execute the command and an example result

Command Line

```
(gdb) break *main
Breakpoint 1 at 0x80484c4
(gdb) run
Starting program: bash_login

Breakpoint 1, 0x080484c4 in main ()
(gdb) disas main
Dump of assembler code for function main:
=> 0x080484c4 <+0>:,  push   ebp
   0x080484c5 <+1>: ,mov    ebp,esp
   0x080484c7 <+3>: ,and    esp,0xfffffff0
   0x080484ca <+6>: ,sub    esp,0x30
   0x080484cd <+9>: ,mov    DWORD PTR [esp+0x12],0x24243470
   0x080484d5 <+17>:,mov    DWORD PTR [esp+0x16],0x64723077
   0x080484dd <+25>:,mov    WORD PTR [esp+0x1a],0x21

.....Output ommited.....
(gdb) break *main+6
Breakpoint 2 at 0x80484ca
```

3 Examine and Print

GDB allows examining of memory locations be them specified as addresses or stored in registers. The x command (for examine) is arguably one of the most powerful tool in your arsenal and the most common command you are going to run when exploiting.

The format for the examine command is as follows:

Command Line

```
(gdb) x/nfu [address]
        n:  How many units to print
        f:  Format character
              a Pointer
              c Read as integer, print as character
              d Integer, signed decimal
              f Floating point number
              o Integer, print as octal
              s Treat as C string (read all successive memory addresses
                           until null character and print as characters)
              t Integer, print as binary (t="two")
              u Integer, unsigned decimal
              x Integer, print as hexadecimal
        u:  Unit
              b: Byte
              h: Half-word (2 bytes)
              w: Word (4 bytes)
              g: Giant word (8 bytes)
              i: Instruction (read n assembly instructions from
                                    the specified memory address)
```

In contrast with the examine command, which reads data at a memory location the print command (shorthand p) prints out values stored in registers and variables.

The format for the print command is as follows:

Command Line

```
(gdb) p/f [what]
        f:  Format character
              a Pointer
              c Read as integer, print as character
              d Integer, signed decimal
              f Floating point number
              o Integer, print as octal
              s Treat as C string (read all successive memory addresses
                           until null character and print as characters)
              t Integer, print as binary (t="two")
              u Integer, unsigned decimal
              x Integer, print as hexadecimal
              i Instruction (read n assembly instructions from
                                    the specified memory address)
```

3.1 An example

For a better explanation please follow through with the following example:

A breakpoint has been set inside a program and the program has been run with the appropriate commands to reach the breakpoint at this point we want to see which are the following 10 instructions

Command Line

```
(gdb) x/10i 0x080484c7
   0x80484c7 <main+3>:,and    esp,0xfffffff0
   0x80484ca <main+6>:,sub    esp,0x30
   0x80484cd <main+9>:,mov    DWORD PTR [esp+0x12],0x24243470
   0x80484d5 <main+17>:,mov    DWORD PTR [esp+0x16],0x64723077
   0x80484dd <main+25>:,mov    WORD PTR [esp+0x1a],0x21
   0x80484e4 <main+32>:,mov    eax,0x8048630
   0x80484e9 <main+37>:,mov    DWORD PTR [esp],eax
   0x80484ec <main+40>:,call   0x80483b0 <printf@plt>
   0x80484f1 <main+45>:,mov    eax,0x804864a
   0x80484f6 <main+50>:,lea    edx,[esp+0x1c]
```

Let's examine the memory at 0x80486a0 because we have a hint that the eax register holds a parameter as it is then placed on the stack

Command Line

```
(gdb) x/s 0x80486a0
0x80486a0:, "\nPlease provide password:"
# we now set a breakpoint for main+49
(gdb) break *0x80484e9
Breakpoint 3 at 0x80484e9
(gdb) continue
Continuing.
```

Let's examine the eax register (it should hold the address for the beginning of the string so let's interpret it as appropriately) take note that in GDB registers are preceded by the "$" character very much like variables

Command Line

```
Breakpoint 3, 0x080484e9 in main ()


(gdb) x/s $eax
0x8048630:, "\nPlease provide password:"
#now let's print the contents of the eax register as hexadecimal
(gdb) p/x $eax
$1 = 0x8048630
```

As you can see the eax register hold the memory for the beginning of the string this shows you how "x" interprets data from memory while "p" merely prints out the contents in the required format you can think of it as "x" dereferencing while "p" not dereferencing.