Use of GNU Debugger (GDB) for Reverse Engineering of C Programs in a Linux Platform

Due: April 22, 2013, 7.30 PM
Maximum Points: 100

Demo and Hardcopy Submission: You need to show a demo of the working of Phase 2 and 3 of the project to me before the above deadline and submit a hardcopy of the report (for all the three phases) in class at the above date/time.

The purpose of the project is to familiarize students with the GNU Debugger (GDB) and its use for reverse engineering. This will be done in Ubuntu OS with the use of simple C programs. If you need to set up Ubuntu as a virtual machine, follow the instructions provided for the same at http://143.132.8.23/cms/tues/docs/CSC439-AIS-Spring2013/Project-2-Email-Security-PGP.pdf

This project is divided into three phases. The first phase will introduce you to the GNU debugger (GDB) and the next two phases will require you to explore things built on top of the basics that you learn in Phase 1.
Phase 1: Introduction to the GNU Debugger (GDB)
25 points

What to submit:
Screenshots of ALL the practice steps. Indicate the Step # in your screenshots.

1. Open a terminal and create the following C program MaxOfTen.c using an editor of your choice. I recommend using pico or vi. Type in the following code, save it and exit from the editor.
2. Now, compile the program using `gcc -g -o MaxOfTen MaxOfTen.c`. After compiling, run the program using `./MaxOfTen`. The program will require you to enter your first name. Do so and press **enter**. It will then require you to enter 5 integers in the terminal. Separate the integers with spaces. After typing in the integers, press **enter** and the program will sort the integers in ascending order and print the result to the screen.

```
ubuntu@ubuntu:~$ ./MaxOfTen
Enter your first name: Natarajan
Enter 5 numbers and separate them with spaces..
45 10 12 2 7
Hello Natarajan
The ascending order is:
  2
  7
 10
 12
 45
ubuntu@ubuntu:~$
```

3. Next, we will load the program into gdb so we can analyze through our code. Load the program using **gdb MaxOfTen**

```
ubuntu@ubuntu:~$ gdb MaxOfTen
GNU gdb (Ubuntu/Linux 7.3-0ubuntu2) 7.3-2011.08
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linux/>...
Reading symbols from /home/ubuntu/MaxOfTen...done.
(gdb)
```

4. Type in **list 1** and press **enter** to list the program. If all the lines do not show up at once, you can keep pressing the **enter** key to list all the code lines.
5. Set a breakpoint at main so that we can be able to step through the program execution. To set a breakpoint, type the command `break main` and hit `enter`.

6. Disassemble the main function to get the assembly code equivalence. We are not going to do any assembly programming, this is just to take a peek into what's going on in the CPU. Use `disas main` to do this. Keep pressing the enter key until it reaches the end of assembler dump.
7. Enter the **run** command to begin executing the program. Notice that the program will break at main. It shows the breakpoint at main, and below it is the line of code that will be executed next. Press `s` 3 times on your keyboard to step through the program. After pressing the third `s`, the pointer is going to hold and require you to type in your first name. Press **enter** after keying in your first name. The next instruction shown will print the string that requires you to enter the integers you are going to work with. Press `s` to step through this.
8. Press \texttt{s} again to continue stepping through the program. Here, we will monitor the variable \texttt{i} to see how it increments as the for loop is executed. Step again through the program by typing \texttt{s}, and then type in \texttt{print i} to see the beginning value of \texttt{i} which is 0.

\begin{verbatim}
(gdb) s
Enter 5 numbers and separate them with spaces.
12 for (i=0; i<5; i++)
(gdb) s
scanf("%d", &num[i]);
(gdb) print i
$1 = 0
(gdb) s
12 for (i=0; i<5; i++)
(gdb) display i
1: i = 0
(gdb) s
13 scanf("%d", &num[i]);
1: i = 1
(gdb) s
2
12 for (i=0; i<5; i++)
1: i = 1
(gdb)
\end{verbatim}

Press \texttt{s} again. The pointer is going to wait for you to type in an integer. Type in your first integer and press \texttt{enter}.

Now, you will have to show the value of \texttt{i} after every iteration of the for loop. You can do this using the \texttt{display i} command. This way, the value of \texttt{i} will be displayed after every iteration without you having to type the \texttt{print i} command.

9. Step through the program using \texttt{s} and notice that the value of \texttt{i} will be printed each time you step. Keep stepping through the program and entering the integers until it reads the 5th one. Remember to take the screenshot of each step. After entering the 5th integer, press \texttt{s} and the program will move to the next for loop because the value of \texttt{i} is now 5, and this causes the program to finish the first loop and set up the next loop for execution. Take screenshot covering all \texttt{i} values.

10. Type \texttt{info b} to print information about existing break points.
11. Disable the breakpoint using the command `disable 1`. The number 1 in the command represents the breakpoint number because you can set more than one breakpoint. Print the breakpoint information again and notice that the character under `Enb` has changed from `y` to `n`.

12. Show the content of the array variable `num` using `print num`.

13. Examine the name variable as a string using `x/s name`. This command will print your first name in string format. The 4-byte hexadecimal value to the left is the virtual memory address where the string is stored.

14. Examine a character using `x/c name`. This prints the first character in the name variable.

Increase the number of characters you want to print using `x/nc name` where `n` is an integer that represents how many characters you want to print. In the screenshot below, the integer 6 was used and this printed out 6 characters and their ASCII values.
Remember that the **MaxOfTen** program is still running. But we are able to pause the program and examine its execution.

15. Examine the processor registers using the command `info registers`. This will print the CPU registers and their contents.

```
(gdb) info registers
eax  0xc  12
ecx  0x1   1
edx  0x2   2
ebx  0x2a8ff4  2789364
esp  0xbffff280  0xbffff280
ebp  0xbffff2e8  0xbffff2e8
esi  0x0   0
edi  0x0   0
eip  0x80485b1  0x80485b1 <main+205>
eflags  0x297  [ CF PF AF SF IF ]
cs   0x73  115
ss   0x7b  123
ds   0x7b  123
es   0x7b  123
fs   0x0   0
gs   0x33  51
(gdb)
```

16. Press `c` to continue and finish up the execution of the program. Type `quit` and press `enter` to quit gdb.

```
(gdb) c
Continuing.
Hello Natarajan
The ascending order is:
2
8
12
56
87
[Inferior 1 (process 5061) exited normally]
(gdb)
```

17. Next, you will set a conditional breakpoint i.e. there will be a breakpoint if some condition is met. You will set a condition that will break the execution once the value of `j > 4`. Then you will modify the `name` string in which the name you entered was stored. To do this, load the program into gdb again. Use `list 1` to list the code. Keep pressing `enter` till the entire code has been listed.
18. Break at the line that you have the `printf("Hello %s
\n\n", name);` statement. From the screenshot above, the statement is at line 27. Use the command `break 27 if (j > 4)` and hit enter.

```
(gdb) break 27 if (j>4)
Breakpoint 1 at 0x80485d5: file MaxOfTen.c, line 27.
```

19. Run the program, enter you first name as required and press `enter`. Next, enter any 5 integers and press `enter`.

```
(gdb) break 27 if (j>4)
Breakpoint 1 at 0x80485d5: file MaxOfTen.c, line 27.
(gdb) run
Starting program: /home/ubuntu/MaxOfTen
Enter your first name:
Natarajan
Enter 5 numbers and separate them with spaces.
45 78 12 10 5
```

```
Breakpoint 1, main () at MaxOfTen.c:27
27    printf("Hello %s\n\n", name);
(gdb)
```
20. Now at the break point, enter `x/s name` to print the current value of the variable `name`. Change the value of the variable `name` from your first name to your last name. To do this, type in the command `set var name = "Meghanathan"` where Meghanathan should be switched with your last name. Examine the content of the variable `name` again and see that it has been changed.

Type `c` to continue and observe that your last name will be printed instead of your first name.

```
Breakpoint 1, main () at MaxOfTen.c:27
27    printf("Hello %s
\n", name);
    (gdb) x/s name
0xbffffff2c3:    "Natarajan"
    (gdb) set var name = "Meghanathan"
    (gdb) c
Continuing.
Hello Meghanathan
The ascending order is:
5
10
12
45
78
[Inferior 1 (process 5066) exited normally]
(gdb)
```

Type `quit` and hit `enter` to quit gdb.
Phase 2: Exploring GDB on Your Own Tasks
35 points

What to Submit (Phase 2): Screenshots of **ALL** the tasks. Indicate the task # in your screenshots.

1. Type and compile the simple code in the screenshot below. Name the program whatever you want.

```c
#include <stdio.h>

void add()
{
    int a, b, c, sum, mul;
    printf("Enter the three integers you want to add\n");
    scanf("%d%d%d", &a, &b, &c);
    sum = a + b + c;
    printf("The sum of the three numbers is: %d\n", sum);
    printf("The product of the three numbers is: %d\n", mul);

    return;
}

void main()
{
    add();
    return;
}
```

2. Load the program into gdb. You need the executable to load with gd. So, make sure to compile your C program.
3. List all the code in gdb.
4. Disassemble the main function.
5. Disassemble the add function.
6. Set a breakpoint at the add function.
7. Run the program.
8. In the execution of the program, modify and double the value of a, b, and c. So for example, if your initial values of a, b, and c were 2, 4, and 8, you should double and make them 4, 8, and 16. Make sure you do this after the values of a, b, and c have been added and stored in sum.
9. Set the value of mul to be the product of the doubled values of a, b, and c.
10. Finish the execution of the program.

**Hint:** If you run the program and enter 2, 4, and 8 as the input values of a, b, and c. The value of `sum` printed in the program should be 14 which is 2 + 4 + 8, and the value of `mul` printed in the program should be 512 which is 4 * 8 * 16.
Phase 3: Using gdb to obtain a password read into memory from a protected file

The objective of this phase of the project is to gain access to a password-protected program. This will be achieved by loading the program into gdb and breaking the execution process in order to peek into the stack and copy the password as soon as it is loaded from a file. Your goal will be to find out the correct password value stored in the PasswordFile (i.e., mimic as if you are not looking at the PasswordFile; but through the execution stack using gdb), enter it when prompted for the user password and have the program print that you entered the correct password. You can create the PasswordFile as a simple text file in your present Linux working directory.

Tasks
40 points

What to Submit (Phase 3): Screenshots of ALL the tasks. Indicate the task # in your screenshots.

Password to be stored in PasswordFile for each student: Your password will be the name of your home city and a 5-digit zip code appended to it. For example, if your home city is Clinton and the zip code is 39056, your password will be Clinton39056.
1. Type the following C program (save it as ReadPass.c), compile it using `gcc -g -o ReadPass ReadPass.c` to generate the executable.

```c
#include <stdio.h>
#include <stdlib.h>

main(){
    char a[25];
    char b[25];
    FILE *file;

    file = fopen("PasswordFile", "r");
    if (file == NULL){
        perror("Error while opening...");
        exit(EXIT_FAILURE);
    }

    while (fscanf(file, "%s", a) != EOF);
    fclose(file);

    printf("Enter the secret password: ");
    scanf("%s", b);
    printf("\n");
    printf("The password you entered is: %s\n", b);

    int compare = strcmp(b, a);
    if (compare == 0)
        printf("It is correct.\n\n");
    else
        printf("It is incorrect.\n\n");

    return 0;
}
```

2. Create a password file (named `PasswordFile`) using an editor (e.g., pico) of your choice. Type in the password assigned to each of you in the beginning of the project description. In this description, I entered the password `Security2013` into the file. Note that the S in the text is in uppercase. Save the password in the file `PasswordFile` and exit. Use the `cat` command to print the contents of the `PasswordFile` and capture it as a screenshot.

```
$ cat PasswordFile
Security2013
```

3. Change the permission of the ReadPass file using `sudo chmod 777 ReadPass`. Run the program using `./ReadPass` This program will ask you to enter the secret password. When it
requires the password, type in **Jackson**. Jackson is not the correct password. The purpose of
typing it is to see what the program does when the password is incorrect. Now run the program
again and this time enter the password you saved in the PasswordFile (in my case, it is
Security2013).

4. Load the program into gdb using **gdb ReadPass**. Break the program at the main function
using the command **break main**.

5. Use the **list** command to see the contents of the source code. Find out which variable in the
program is used to store the password retrieved from the PasswordFile. What is the variable
name?

6. Run the program using the command **run**. Keep pressing **s** to run the program line by line
after the breakpoint. Once the execution is past the line where the password is retrieved from the
file, use the appropriate gdb command to find out the value of the variable that stores the value
of the password retrieved from the file. Keep pressing **s** and when it comes to the point where the
program asks the user to enter a password, enter the password that you believe you have figured
out (i.e., retrieved from the file). Finish the execution by typing **c** and pressing **enter**. The
program will validate the password you entered and print out whether your entry is correct or not correct. Quit gdb using the **quit** command.

7. Load the program again into gdb using **gdb ReadPass**. Examine the stack using the command **x/8wx $esp** and hit **enter**. Notice the result "No registers". This means that the program has not been run and therefore nothing has been loaded into the registers. Break the program at the main function using the command **break main**.

```
ubunut@ubuntu:~$ gdb ReadPass
GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.htm
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/ubuntu/ReadPass...done.
(gdb) x/8wx $esp
No registers.
(gdb) break main
Breakpoint 1 at 0x804863d: file ReadPass.c, line 3.
```

8. Now that you have figured out the name of the variable that stores the password retrieved from the file, find its virtual memory address and take a screenshot of the command used along with the memory address value displayed.

9. Examine the contents of the stack using the **x/8wx $esp** command and with different integer values (like 8, 12, 16, etc) until you can obtain a good range of virtual memory addresses that encompass the virtual memory address you found in task 8. Now, through these virtual memory addresses and the hexadecimal values of the contents stored in these addresses, extract the value of the password retrieved from the file. Next page, I show a screenshot of the strategy I use to retrieve the contents Security2013.

In my case, the starting virtual memory address of the variable that stores the password retrieved from the file is **0xbffff2aa**. As I indicate in the following screenshot, the values in the stack are the hexadecimal ASCII values of the characters in the string 'Security2013’ read from the reverse order. Repeat the same for your password (note that the starting virtual memory address for your password variable may be different or same - this is what you have to figure out from Task 8) and provide the appropriate screenshots.
53 at 0xbffff2aa is the hexadecimal ASCII value of 'S' [decimal ASCII value is 83]
65 at 0xbffff2ab is the hexadecimal ASCII value of 'e' [decimal ASCII value is 101]
63 at 0xbffff2ac is the hexadecimal ASCII value of 'c' [decimal ASCII value is 99]
75 at 0xbffff2ad is the hexadecimal ASCII value of 'u' [decimal ASCII value is 117]

and so on.........