Binary Exploitation

This section talks about exploiting information at a register level. We will talk about debugging programs, how to hack into programs to make them do something different from their intended use, how to safeguard against such attacks and much more.

Debugging is essential for any serious programmer. It is unlikely that you will write code that works flawlessly the first time. However, there are many tips and tricks that can help the sometimes painful process of debugging go more smoothly.

There are many things that you can do but the easiest way to debug is print information that you think is useful and try to identify the location and the source of the bug. Start by identifying where the execution of the program halts, then try removing parts of the code until you know exactly what causes the crash or the error to occur. Now, is the “easy” part of understanding why the error is happening, what you intended to happen and what you should do to fix the error. This was just the first step to debugging and there are many more techniques and tools that can help you out. Let’s discuss one of the most important - GDB.

GDB - The GNU Project Debugger

GDB or The GNU Project Debugger is a very widely used software that can help programmers catch bugs. It can help start and stop the execution of the program at any stage, look at memory and register values, change things around in the program and many more useful tools to help in debugging. This is an essential tool in any Systems programmers arsenal so we will cover the essential part of GDB. *Note: You should have a working knowledge of C/C++, registers and memory before you continue. We recommend checking out our section on Reversing.*

Now, it isn’t important to know the inner workings of GDB to use it but we will cover just the basics to get you to appreciate the tool. GDB consists of two major parts -

1. The Symbol Side: This is concerned with the “symbolic side” of the program, which means the metadata about functions, variables, the source code and essentially the program in the way it was written.

2. The Target Side: This part deals with all the manipulations that can be done to the execution of the program like reading registers, accepting signals and starting or stopping the execution. In UNIX, it uses the system call called "ptrace" to accomplish this.

Both the parts are largely separate, however, the command interpreter and main control loop tie these two parts together. Here is an excellent article on all the inner workings of the GDB and talks about all the data structures used, how the different parts work and how it came about. It is a very interesting read but might be too complicated if you are a new programmer.

Here is a tutorial on almost everything you could want to know about GDB but we will cover some of the more fundamental concepts now.

Starting up the environment:

To compile the code (in C), type

```
$ gcc -g filename.c -o executableName
```

Here, gcc is the GNU C Compiler, the *-g* flag tells the compiler that you intend to use GDB, filename.c is the name of the file you wish to compile and *-o* is the optional flag that tells the compiler what you want the name of the executable file to be (default is a.out).

Now to start up GDB, type

```
$ gdb executableName
```

This just says to open up the executable file with gdb. You should see a bunch of information talking about the gdb (this mostly not important). There will be a new interface starting with *(gdb),* and this is GDB interface where you can use many different commands. Here are some important ones -

1. 'break' or 'b': This command is always associated with a line in the code and tells the compiler to stop program execution when it reaches this line. It is usually followed by a line in order to put a 'breakpoint' at that line. You can obtain information about all breakpoints by typing 'info b'.
2. 'run' or 'r': This command runs the file that was loaded into gdb. This program will run the file normally if no issues but will crash if there is an error or break and give useful information about why it stopped.

3. ‘step’ or ‘s’: Runs the next line of the program and essentially steps through the code. It will enter each function it encounters as opposed to ‘next’, which just runs sub routines executing them line by line.
4. ‘Print’ or ‘p’: Prints the value of an argument (variable or registers) given to it.

These commands can help you run, stop and step through your program and debug it by using printing and thus make up the most important commands. This is a great tutorial explaining the use of some more basic commands. Execute ‘*layout regs*’ before running your program as this will show you the assembly code and the register and memory address values while you run your code. This is really helpful while debugging.