Using gdb Review Lab

Background

gdb is an open source debugger for applications that are written for the x86 family of processors (32-bit or 64-bit). It is typically built into *nix systems by default. gdb helps in analysing software problems by; allowing a review of the microprocessor code, allowing access to microprocessor register information and by allowing breakpoints to be set in the code so that runtime errors can be investigated. gdb uses the command line interface rather than a graphical user interface, which makes it lightweight and easy to use over remote shell connections.

gdb is run in Linux by typing "gdb" followed by the compiled binary that you wish to investigate (an example is shown below):

```
fly@VM:~/vde/week1$ gdb ./password_cracker
```

Disassembling Code

Once gdb is open you can access a simple help menu by typing help. For all of the binaries that we are going to investigate (i.e. small programs), the starting point will be to show a disassembly of the main function (this will then give examples of the names of other functions). The full command for this is:

```
(gdb) disassemble main
```

The command line interface is able to auto-complete commands in the same way as the terminal; in addition to this, gdb can also use the shortened version of a command (as long as the number of characters provided indicates only one function). An example of a shortened command is given below:

```
(gdb) disas main
```

The syntax that is provided by default is AT&T syntax. The other syntax for assembly code is Intel syntax. gdb can be set to either use Intel or AT&T by using the following:

```
(gdb) set disassembly-flavor att
(gdb) set disassembly-flavor intel
```

The main difference between AT&T and Intel syntax is the direction of the operands. In Intel syntax the command is:

instruction destination, source

In AT&T syntax the command is:

instruction source, destination

AT&T syntax also prefixes registers with "%" and prefixes hexadecimal values with "$0x". In Intel syntax hex values have "0" as a prefix and "h" as a suffix.

The disassembled code will look different depending on whether it was originally compiled for a 32-bit or 64-bit system. Firstly the memory address of the code in RAM (as mapped by the OS) will look different.

For a 32-bit processor we will see something like the following:

```
0x000011db <+14>:    push   ebp
0x000011dc <+15>:    mov    ebp,esp
0x000011de <+17>:    push   ebx
0x000011df <+18>:    push   ecx
0x000011e0 <+19>:    sub    esp,0x10
```

For a 64-bit processor we will something like the following:

```
0x000000000000116d <+4>:     push   rbp
0x000000000000116e <+5>:     mov    rbp,rsp
0x0000000000001171 <+8>:     sub    rsp,0x20
```

What we have seen here is that the memory address for the 32-bit program is 4 bytes in length and the memory address for the 64-bit program is 8 bytes in length. We also have an idea of where the program is mapped in memory. We also see references to registers; e.g. the base pointer register. In the 32-bit program this is the "ebp" - the extended base pointer. In the 64-bit program this is "rbp" - the r base pointer. All 32-bit registers have an equivalent in the 64-bit processor (to enable a 64-bit processor to run 32-bit programs). The following are the registers that we may have an interest in;

32-bit Register	64-bit register	Purpose
EAX	RAX	The accumulator; used to store data and operands.
EBX	RBX	The base register and pointer to data.
ECX	RCX	The counter (used for looping etc).
EDX	RDX	The data register; used to store the I/O pointer.
ESI/EDI	RSI/RDI	The index registers; used to hold data pointers when performing memory operations.
ESP	RSP	The stack pointer register; used to hold the current position on the stack (the "top" of the stack).
EBP	RBP	The base pointer; used to hold the base position on the stack of the current code being executed (the location of the "bottom" of the stack).
EIP	RIP	The instruction pointer; used to hold the address of the next instruction to be executed.

When disassembling the main function it is possible to find calls to other functions which can then also be disassembled.

Breakpoints

Breakpoints can be added to the program code to halt execution at various points and allow interrogation of either memory or registers at run time. This is a particularly important aspect for this module as we will want to look at the contents of memory and also the contents of the registers at run-time.

Breakpoints can only be added if the binary has been compiled with the "-g" flag enabled (thereby providing symbol tables that gdb can use). If the binary has been compiled with the symbol tables then it is (usually) possible to list the contents of the program. This can be achieved by typing "list".

```
(gdb) list
```

Each time you type "list" you will see the next 10 lines of code. If you want to rewind to the start of the program then type "list 0".

```
(gdb) list 0
```

If you want to see some help on using “list” (and other functions) then type “help list”.

```
(gdb) help list
```

Once you see where you want to have a breakpoint you can add the breakpoint by typing “break” followed by the line number.

```
(gdb) break 17
Breakpoint 1 at 0x124a: file pointers.c, line 17.
```

You can add multiple breakpoints by repeating this process. Should you wish to delete a breakpoint you need to type “delete breakpoints” followed by the breakpoint number. Alternatively you can type “delete” to delete all breakpoints.

```
(gdb) delete breakpoints 3
(gdb) delete
Delete all breakpoints? (y or n) y
```

Once you have some breakpoints in place you can run the program by typing “run”.

```
(gdb) break 17
Breakpoint 4 at 0x124a: file pointers.c, line 17.
(gdb) run
```

The program will run until the first break point. To get the program to continue to execute after the first break point (and until the next breakpoint or the end of the code) you need to type “continue”.

```
(gdb) continue
Continuing.
```

After any breakpoint (but before the end of the program) the registers and information in memory can be inspected.

Register Inspection

In order to look at the registers the program must be halted in some way so that the values are still present in the registers. It is not possible to look at the value of registers after the program has finished executing (as nothing is currently running in the debugging environment). To look at the registers you need to type “info register” - the shortened version is shown below:

```
(gdb) info reg
rax            0xb                 11
rbx            0x555555555280      93824992236160
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x7fffffffdac0      140737488345792
rbp            0x7fffffffe020      0x7fffffffe020
rsp            0x7fffffffe000      0x7fffffffe000
r8             0xa                 10
r9             0x0                 0
r10            0x7ffff7f61ac0      140737353489088
r11            0x0                 0
r12            0x5555555550c0      93824992235712
r13            0x7fffffffe110      140737488347408
r14            0x0                 0
r15            0x0                 0
rip            0x55555555524a      0x55555555524a <main+142>
```

Looking into Memory

While the program is still being debugged (i.e. execution is halted) we can look at the values of information that is stored in memory. In order to do this we can print out the memory address of a variable. Let's say that I have a variable called “b” in my program, I can print out its location in memory by typing “print &b”, as shown below:

```
(gdb) print &b
$1 = (int *) 0x7fffffffe014
```

I now have the memory address where the integer b is stored. I can then inspect the memory contents by typing “x/x” followed by the memory address. I can see multiple segments of memory by typing “x/4x” or “x/8x” depending on how many memory segments I want to see.

```
(gdb) x/x 0x7fffffffe014
0x7fffffffe014: 0x00000005
(gdb) x/4x 0x7fffffffe014
0x7fffffffe014: 0x00000005      0x1d9f0200      0xa7591be2      0x00000000
(gdb) x/8x 0x7fffffffe014
0x7fffffffe014: 0x00000005      0x1d9f0200      0xa7591be2      0x00000000
0x7fffffffe024: 0x00000000      0xf7dea0b3      0x00007fff      0xf7ffc620
```

In fact, I don't need to explicitly refer to a memory address, I can look at the memory location by directly referring to the variable, as shown below:

```
(gdb) x/x &b
0x7fffffffe014: 0x00000005
(gdb) x/4x &b
0x7fffffffe014: 0x00000005      0x1d9f0200      0xa7591be2      0x00000000
```

Task 1

1. Write some code to declare a variable ‘x’ with the following values and ensure that you verify that the correct value is in memory;
    a. 3987
    b. 10000
    c. 678,890,876,987,876,456
2. Attempt to use arrays of characters using the declaration “char *y = “an array”;” and verify the information is stored in memory.