GDB Debugger How To

Computer Security
School of Informatics
University of Edinburgh

Debuggers are powerful tools with the main purpose to aid programmers in understanding the exact behavior of a programme, for example when trying to fix bugs. Things that can be done include tracing the execution steps of a programme and pause the execution at a desired line in order to interactively experiment with its current state. GDB is the most commonly used debugger for programmes written in C and other common compiled languages, as well as x86 machine code. We will use GDB in Coursework 3 to exploit a buffer overflow vulnerability. The purpose of this walkthrough is to get to know the basic use of this tool.

1 Basics

As always when starting to experiment with something, we open a terminal and move to a new empty directory:

```
$mkdir gdb-practice
$cd gdb-practice
```

Then we create a file named `sample.c` that contains the following C code using your favorite editor:

```c
#include <stdio.h>

int gcd(int a, int b) {
    if(b == 0) {
            return a;
    }
    else {
        return gcd(b, a % b);
    }
}

int main() {
    int a, b, res;

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    res = gcd(a, b);

    printf("The GCD of %d and %d is %d.\n", a, b, res);
    return 0;
}
```

We now compile this file to `sample` using the `-g` option to enable debugging support:

```
$gcc -g sample.c -o sample
```

Now we can start using `gdb`:

```
$gdb sample
```

Now we are in the interactive environment. `gdb` has loaded the compiled programme and waits for a command. The simplest thing we can do is just run the programme by issuing the command `run` (pretty intuitive). Incidentally this is the same as just executing the programme normally, without `gdb` (i.e. issuing `./sample` in the terminal). After doing the interaction with the programme (and learning the gcd of two numbers), `gdb` will inform us that the process exited normally. We can now issue `quit` to quit from `gdb` (what a surprise). The whole thing should look something like this:

```
Reading symbols from sample...done.
(gdb) run
Starting program: sample
Enter two integers: 42 60
The GCD of 42 and 60 is 6.
[Inferior 1 (process 7158) exited normally]
(gdb) quit
```

The `run` command can be shortened to `r`; likewise `quit` is the same as `q`.

Now for the `next` step, which is to see the execution of the programme line by line. We start `gdb` just like before:

```
$gdb sample
```

We first issue the command `start` which will start the execution of the programme but will not actually run it. It will instead wait at the beginning of the `main()` function for another command:

```
(gdb) start
Temporary breakpoint 1 at 0x40060c: file sample.c, line 12.
Starting program: sample

Temporary breakpoint 1, main () at sample.c:12
12      int main() {
(gdb)
```

Now we will issue the `next` (a.k.a., you guessed it, `n`) command, which executes one line of code at a time. If you invoke it repeatedly you can see the execution of the programme line by line until `main() returns`:

```
(gdb) start
Temporary breakpoint 1 at 0x40060c: file sample.c, line 12.
Starting program: sample

Temporary breakpoint 1, main () at sample.c:12
12      int main() {
(gdb) n
15          printf("Enter two integers: ");
(gdb)
16          scanf("%d %d", &a, &b);
(gdb)
Enter two integers: 60 32
18          res = gcd(a, b);
(gdb)
20          printf("The GCD of %d and %d is %d.\n", a, b, res);
(gdb)
The GCD of 60 and 32 is 4.
21          return 0;
(gdb)
22      }
(gdb)
0x00007ffff7a60640 in __libc_start_main () from /lib64/libc.so.6
(gdb)
Single stepping until exit from function __libc_start_main,
which has no line number information.
[Inferior 1 (process 7354) exited normally]
```

Another shortcut that will prove useful: pressing Enter with no command will issue again the last command given.

You may notice that at line 18 the debugger did not show the contents of the `gcd()` function. If we want to do this, the `step` command will prove useful. Let's invoke `gdb` once more and issue `start` as the first command. We can now issue `next` until we see line 18 and then issuing `step` (or `s`) will take us into the `gcd()` function code. We can also try the `list` (or `l`, so predictable) command to see the 10 lines of code around our current position. Issuing this command one more time will print the next ten lines. Furthermore, if we're interested in seeing the value stored in, say, variable `a`, we can issue `p a` (exercise for the reader: guess what the shorthand we just used stands for). We can even do `p a = 4` to change the value of the variable if we wish to. When we've had enough `step`s and we want to just execute the rest of the programme, we can use `c` (which stands for `continue`).

```
(gdb) start
Temporary breakpoint 1 at 0x40060c: file sample.c, line 12.
Starting program: sample

Temporary breakpoint 1, main () at sample.c:12
12      int main() {
(gdb) n
15          printf("Enter two integers: ");
(gdb)
16          scanf("%d %d", &a, &b);
(gdb)
Enter two integers: 101 51
18          res = gcd(a, b);
(gdb) s
gcd (a=101, b=51) at sample.c:4
4           if(b == 0) {
(gdb) l
1       #include <stdio.h>
2
3       int gcd(int a, int b) {
4           if(b == 0) {
5                   return a;
6           }
7           else {
8               return gcd(b, a % b);
9           }
10      }
(gdb)
11
12      int main() {
13          int a, b, res;
14
15          printf("Enter two integers: ");
16          scanf("%d %d", &a, &b);
17
18          res = gcd(a, b);
19
20          printf("The GCD of %d and %d is %d.\n", a, b, res);
(gdb) p a
$1 = 101
(gdb) p a = 606
$2 = 606
(gdb) p a
$3 = 606
(gdb) cont
Continuing.
The GCD of 101 and 51 is 3.
[Inferior 1 (process 7700) exited normally]
```

Notice that in this example we messed with the variable `a` *inside* the function `gcd()` and as a result the `printf()` at the end of `main()` printed the original value of `a` (that is 101), as we haven't changed that variable (since C passes arguments to functions by value, not by reference). The result returned however is the gcd of 606 and 51, because the whole calculation was done inside `gcd()` with `a = 606`. We have managed to trick the programme into printing something wrong.

One last command and we will have covered all the basics. We start `gdb` once more. Suppose that we want to see what is going on in `gcd()` without having to `next` our way there every time. For this we use a very important tool in debuggers, breakpoints. The first command, before even starting, will be `break gcd` (a.k.a. `b gcd`), which will set a breakpoint inside the first line of the function. We can now directly `run` our programme (which is the same as `start` and `continue`) and it will automatically stop just inside the function. We can use the previous commands to mess around as we please. Breakpoints work with line numbers, too.

Notice however that `gcd()` calls itself (jargon: `gcd()` is *recursive*), thus issuing `continue` again will reach the same breakpoint repeatedly until the function isn't called anymore. To avoid smashing Enter for a potentially long time, we can remove the breakpoint with `clear gcd`:

```
(gdb) b gcd
Breakpoint 1 at 0x4005e4: file sample.c, line 4.
(gdb) run
Starting program: sample
Enter two integers: 61 15

Breakpoint 1, gcd (a=61, b=15) at sample.c:4
4            if(b == 0) {
(gdb) cont
Continuing.

Breakpoint 1, gcd (a=15, b=1) at sample.c:4
4            if(b == 0) {
(gdb)
Continuing.

Breakpoint 1, gcd (a=1, b=0) at sample.c:4
4            if(b == 0) {
(gdb) clear gcd
Deleted breakpoint 1
(gdb) b 21
Breakpoint 2 at 0x400670: file sample.c, line 21.
(gdb) cont
Continuing.
The GCD of 61 and 15 is 1.

Breakpoint 2, main () at sample.c:21
21           return 0;
(gdb)
Continuing.
[Inferior 1 (process 8012) exited normally]
```

OK, fine, it was two commands. If you master these tools though you can debug effectively programmes written in C and many other languages.

2 Return address of a function

The debugger can also be used to interact with the assembly of a progamme. In order to exploit the buffer overflow in Coursework 3, we will have to learn the return address of a function, which can be retrieved from the assembly.

In order to see the assembly of the `main()` function of our executable, we will issue the command `disassemble main` (or `disas main`):

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400604 <+0>:     push   %rbp
   0x0000000000400605 <+1>:     mov    %rsp,%rbp
   0x0000000000400608 <+4>:     sub    $0x20,%rsp
   0x000000000040060c <+8>:     mov    %fs:0x28,%rax
   0x0000000000400615 <+17>:    mov    %rax,-0x8(%rbp)
   0x0000000000400619 <+21>:    xor    %eax,%eax
   0x000000000040061b <+23>:    mov    $0x400714,%edi
   0x0000000000400620 <+28>:    mov    $0x0,%eax
   0x0000000000400625 <+33>:    callq  0x4004a0 <printf@plt>
   0x000000000040062a <+38>:    lea    -0x10(%rbp),%rdx
   0x000000000040062e <+42>:    lea    -0x14(%rbp),%rax
   0x0000000000400632 <+46>:    mov    %rax,%rsi
   0x0000000000400635 <+49>:    mov    $0x400729,%edi
   0x000000000040063a <+54>:    mov    $0x0,%eax
   0x000000000040063f <+59>:    callq  0x4004c0 <__isoc99_scanf@plt>
   0x0000000000400644 <+64>:    mov    -0x10(%rbp),%edx
   0x0000000000400647 <+67>:    mov    -0x14(%rbp),%eax
   0x000000000040064a <+70>:    mov    %edx,%esi
   0x000000000040064c <+72>:    mov    %eax,%edi
   0x000000000040064e <+74>:    callq  0x4005d6 <gcd>
   0x0000000000400653 <+79>:    mov    %eax,-0xc(%rbp)
   0x0000000000400656 <+82>:    mov    -0x10(%rbp),%edx
   0x0000000000400659 <+85>:    mov    -0x14(%rbp),%eax
   0x000000000040065c <+88>:    mov    -0xc(%rbp),%ecx
   0x000000000040065f <+91>:    mov    %eax,%esi
   0x0000000000400661 <+93>:    mov    $0x40072f,%edi
   0x0000000000400666 <+98>:    mov    $0x0,%eax
   0x000000000040066b <+103>:   callq  0x4004a0 <printf@plt>
   0x0000000000400670 <+108>:   mov    $0x0,%eax
   0x0000000000400675 <+113>:   mov    -0x8(%rbp),%rcx
   0x0000000000400679 <+117>:   xor    %fs:0x28,%rcx
   0x0000000000400682 <+126>:   je     0x400689 <main+133>
   0x0000000000400684 <+128>:   callq  0x400490 <__stack_chk_fail@plt>
   0x0000000000400689 <+133>:   leaveq
   0x000000000040068a <+134>:   retq
End of assembler dump.
```

By looking closely we can see that the call to `gcd()` happens at this instruction:

```
0x000000000040064e <+74>:  callq 0x4005d6 <gcd>
```

We have a lot of information here: First of all, this instruction resides at the position `0x000000000040064e` in memory, or 74 positions after the beginning of `main()`, so it will be executed when the programme counter points to this position. Second, the instruction is of type `callq` and calls the function whose first line is at the position `0x4005d6` (the preceeding 0s are implied) and whose name is `gcd`, so after this instruction is executed, the programme counter will point to `0x00000000004005d6`. What the `callq` function does before jumping to `0x00000000004005d6` is to push onto the stack the current value of the programme counter so that `gcd()` knows the address to return to when it completes its execution, in order for `main()` to continue executing from where it handed over to `gcd()`. The variables defined within the function are also pushed onto the stack, thus if the programme allows us to place in a variable more data than it is designed to hold, the overflowing data may overwrite the spot of the stack where the return address is saved. When we design our exploit, we will have to change this return address to point to our malicious piece of code. Please note that the addresses shown when viewing the assembly of a programme in `gdb` are slightly offset comparing to those that the programme actually uses when executed, so some trial and error may be needed to find this exact offset and calculate the correct return address.

Some more `gdb` commands that we will use are the following: `layout split`, which creates a frame containing the assembly and a frame with the source code. This will greatly help us to keep context when debugging. If we want to see only the source code, we can instead use `layout src` and for the assembly `layout asm`. To leave the `layout` environment (a.k.a. TUI) we can type `<Ctrl> + x a`. To execute line by line the assembly code we can use the variants `nexti` and `stepi` (next instruction and step instruction respectively). They work just like `next` and `step` but they consider assembly lines, not source code lines. The last useful command is `x/bx <address>`, which returns the machine code of the instruction in the given address:

```
(gdb) start
Temporary breakpoint 1 at 0x40060c: file sample.c, line 12.
Starting program: sample

Temporary breakpoint 1, main () at sample.c:12
12      int main() {
(gdb) stepi
0x0000000000400615      12      int main() {
(gdb)
0x0000000000400619      12      int main() {
(gdb)
15          printf("Enter two integers: ");
(gdb) nexti
0x0000000000400620      15          printf("Enter two integers: ");
(gdb)
0x0000000000400625      15          printf("Enter two integers: ");
(gdb)
16          scanf("%d %d", &a, &b);
(gdb) x/bx 0x40064e
0x40064e <main+74>:     0xe8
```

Finally, if we want to compile the source code to assembly language, we can run

```
gcc -S sample.c -o sample.s
```

to generate the file `sample.s`.