GDB Tutorial

by: Hoa Nguyen

As programmers, we all make errors. Certainly, most of us at least have tried placing "printf" statements in our code hoping to catch the errors, however, we need to know more than that. Debugger is a good tool for tracing bugs. In this tutorial, we will show you how to use gdb -- a "GNU" debugger.

Compiling programs to run with gdb:

Below is a not-so-well written program (crash.c) which reads a number n from standard input, calculates the sum from 1 to n and prints out the result:

```
1     #include <stdio.h>
2     #include<string.h>
3     #include<stdlib.h>
4
5     char * buf;
6
7     int sum_to_n(int num)
8     {
9         int i,sum=0;
10         for(i=1;i<=num;i++)
11             sum+=i;
12       return sum;
13    }
14
15    void printSum()
16    {
17       char line[10];
18       printf("enter a number:\n");
19       fgets(line, 10, stdin);
20       if(line != null)
21           strtok(line, "\n");
22       sprintf(buf,"sum=%d",sum_to_n(atoi(line)));
23       printf("%s\n",buf);
24     }
25
26     int main()
27     {
28         printSum();
29         return 0;
30     }
```

In order to run crash.c with gdb, we must compile it with the -g option which tells the compiler to embed debugging information for the debugger to use. So, we compile crash.c as follows:

```
gcc -g ocrash crash.c
```

Now, let's run the program.

```
./crash
enter a number:
5
Segmentation fault
```

Looks familiar? The infamous "Segmentation fault" means there is some kind of invalid memory access. Unfortunately, that is all the compiler tells us. Now, let's see how we can use gdb to spot the problem(s).

Starting gdb:

To start gdb for our crash.c, on the command prompt type "gdb crash". You'll see the following:

```
$gdb crash
GNU gdb Red Hat Linux (6.1post-1.20040607.52rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb)
```

Good! We have successfully loaded gdb with crash. Let's run the program with command "**run**" to see what kind of information we will get.

run

The "**run**" command starts the program. If we do not set up any "**breakpoints**" (we'll see how to use this command later) the program will run until it terminates or core dumps.

```
(gdb) run
Starting program: /student/nguyen_h/csc408/contribuition/crash
enter a number:
10

Program received signal SIGSEGV, Segmentation fault.
0x0017fa24 in _IO_str_overflow_internal () from /lib/tls/libc.so.6
```

Ok, so it crashed. To get more information we use the "**backtrace**" command.

backtrace

The "***backtrace***" command tells gdb to list all the function calls (that leads to the crash) in the stack frame.

```
(gdb) backtrace
#0 0x0017fa24 in _IO_str_overflow_internal () from /lib/tls/libc.so.6
#1 0x0017e4a8 in _IO_default_xsputn_internal () from /lib/tls/libc.so.6
#2 0x001554e7 in vfprintf () from /lib/tls/libc.so.6
#3 0x001733dc in vsprintf () from /lib/tls/libc.so.6
#4 0x0015e03d in sprintf () from /lib/tls/libc.so.6
#5 0x08048487 in printSum () at crash.c:22
#6 0x080484b7 in main () at crash.c:28
(gdb)
```

Let's now have a careful look at the messages. As we can see, main() called printSum() which in turn called sprintf() which then went on to call a bunch of lower level functions which eventually led to the crash. Anything from sprintf() down is not in our control, so let's carefully examine what we passed to sprintf(). The output above tells us that we called sprintf() in line 20 inside function printSum().

```
22       sprintf(buf,"sum=%d",sum_to_n(atoi(line)));
```

We now show how to use break points to examine the values of variables we are interested in at the point we like to break.

Break Points

This sets a break point. Its basic functionality is to type break and a filename and line number. In our case we want to stop in crash.c line 22, we could do the following in gdb:

```
(gdb) break crash.c:22
Breakpoint 1 at 0x804845b: file crash.c, line 22.
(gdb)
```

Ok, we've set the break point, now let's re-run the program.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /student/nguyen_h/csc408/contribuition/crash
enter a number:
10

Breakpoint 1, printSum () at crash.c:22
22 sprintf(buf,"sum=%d",sum_to_n(atoi(line)));
```

print

We now can retrieve the values of all variables we're interested in. To do this we use the "**print**" command.

```
(gdb) print line
$1 = "10\000\000\000\000\000\000¨Á"
(gdb)
```

The line variable has the character values '1' followed by '0' and then a null terminator '\0', and then junk. So, this seems ok. Now, let's move on and examine what buf holds.

```
(gdb) print buf
$2 = 0x0
(gdb)
```

By now the error should be obvious. We're trying to copy stuff into a buffer pointed to by buf which hasn't been allocated resulting in a segmentation fault. Note that we were lucky in this case: because buf is a global variable and was automatically initialized to 0 (null pointer). If it were not, it might have contained an arbitrary value like 0xbffff580 then it would be no longer obvious that the address points to in memory is invalid. Bugs like this are a real pain to track down.

Conditional break points:

Sometimes we wish to set a break point under some condition. For example, we may want to break at line 10 of crash.c only when the value of num is 50:

```
(gdb) break crash.c:10
Breakpoint 1 at 0x8048441: file crash.c, line 10.
(gdb) condition 1 num==50
(gdb) run
Starting program: /student/nguyen_h/csc408/contribuition/crash
enter a number:
50

Breakpoint 1, sum_to_n (num=50) at crash.c:10
10 for(i=1;i<=num;i++)
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00689a24 in _IO_str_overflow_internal () from /lib/tls/libc.so.6
```

Note that you we resume execution with the "**continue**" command.

Some basic commands (used with break points):

Once our program has reached a break point, we can see the the execution by using the following commands:

n (for "next")
This executes the current command, and moves to the next command in the program.

s (for "step")
This steps through the next command. There are differences between step and next. If you are at a function call, and you hit next,
then the function will execute and return. But if you hit step, then you will

go to the first line of that function.
u (for "until")
This is like **n**, except that if we are in a loop, **u** will continue execution until the loop is exited.

```
(gdb) break crash.c:10
Breakpoint 1 at 0x8048441: file crash.c, line 10.
(gdb) condition 1 num==50
(gdb) run
Starting program: /student/nguyen_h/csc408/contribuition/crash
enter a number:
50

Breakpoint 1, sum_to_n (num=50) at crash.c:10
10 for(i=1;i<=num;i++)
(gdb) n
11 sum+=i;
(gdb) n
10 for(i=1;i<=num;i++)
(gdb) u
12 return sum;
(gdb)
```

Other commands (used with break points) of interest:

list [line#]
Prints lines from the source code around line#.
If we give it a function name as the argument function, it prints lines from the beginning of that function.
If we give it no argument, it prints lines around the break point

delete [n]
With no argument, deletes all breakpoints that we have set.
Deletes break point number n.

clear function_name
Deletes the breakpoint set in that function.

print var
Prints a variable located in the current scope.

x address
Prints the content at address:

```
(gdb) print &num
$1 = (int *) 0xbffff580
(gdb) x 0xbffff580
0xbffff580: 0x00000064
(gdb)
```

Useful links:

Check out this Top ten list

References:

http://oucsace.cs.ohiou.edu/~bhumphre/gdb.html