C for Science - 2011 - A Linux Debugging Example - Steven Capper

We now consider a worked example which will motivate us to use the tools `gdb` and `valgrind`. A small C project with a UNIX makefile can be found on the course home page; once it is downloaded it needs to be unpacked using the following command line:

```
$ tar xvzf debugdemo.tar.gz
```

we build the example as follows:

```
$ cd debugdemo
$ make
gcc -pedantic -ggdb -Wall -ansi    -c -o crypticc.o crypticc.c
gcc -pedantic -ggdb -Wall -ansi    -c -o matrixfunctions.o matrixfunctions.c
gcc crypticc.o matrixfunctions.o -lm -o cryptic
```

If we try to run the program we obtain:

```
$ ./cryptic
cryptic: matrixfunctions.c:10: allocMatrix: Assertion ‘(rows > 0) && (cols > 0)’ failed.
Aborted
```

We've run into an assertion failure, the program has terminated printing out an error message to `stderr`. We could look at line 10 in the `matrixfunctions.c` file and try and work out where the problem is. For a small program this is easy to do; but for large programs, especially ones written by someone else, this becomes non-trivial.

It is apparent from the error output that either `rows == 0` or `cols == 0` (or both). A couple of `printf` statements would clarify this, but instead we will us a debugger, `gdb`, to investigate this. We invoke the debugger as follows:

```
$ gdb ./cryptic
GNU gdb Red Hat Linux (6.5-25.el5_1.1rh)
Copyright (C) 2006 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/libthread_db.so.1".

(gdb)
```

The debugger is text based, commands are entered via the keyboard and output is printed to the text console. Let's try and run our program through the debugger and see what happens. To run the program we can type in `run` (or `r` as a shortcut):

```
(gdb) run
Starting program: /home/sdc99/clectures/debugdemo/cryptic
cryptic: matrixfunctions.c:10: allocMatrix: Assertion ‘(rows > 0) && (cols > 0)’ failed.

Program received signal SIGABRT, Aborted.
0x00f34402 in __kernel_vsyscall ()
(gdb)
```

This is the same output as received before but with one subtle difference, the program hasn't finished running yet. We can examine variables in an attempt to deduce the root cause of the assertion failure.

Below the standard assertion failure text above, we see the text `__kernel_vsyscall`, this is where our program is running currently. It is in fact the final function called before our program terminates, the debugger has caught it and suspended the program. As we are interested in the assertion failure rather than the program termination function, we need to somehow navigate to the point of program failure. If we issue the command `where` we receive the following output:

```
(gdb) where
#0  0x00f34402 in __kernel_vsyscall ()
#1  0x00129ba0 in raise () from /lib/libc.so.6
#2  0x0012b4b1 in abort () from /lib/libc.so.6
#3  0x001231db in __assert_fail () from /lib/libc.so.6
#4  0x080487c2 in allocMatrix (rows=0, cols=0) at matrixfunctions.c:10
#5  0x08048645 in main () at crypticc.c:35
(gdb)
```

This tells us that `__kernel_vsyscall` was called by `raise`, which was called by `abort`, which was called by `__assert_fail`, which was called by `allocMatrix` (the function we are interested in). We can also see that `allocMatrix` received zero as arguments for *both* `rows` and `cols`. We can examine the `allocMatrix` function in more detail by navigating to its stack frame, this is achieved using the `up` command:

```
(gdb) up
#1  0x00129ba0 in raise () from /lib/libc.so.6
(gdb) up
#2  0x0012b4b1 in abort () from /lib/libc.so.6
(gdb) up
#3  0x001231db in __assert_fail () from /lib/libc.so.6
(gdb) up
#4  0x080487c2 in allocMatrix (rows=0, cols=0) at matrixfunctions.c:10
10              assert((rows > 0) && (cols > 0));
(gdb)
```

Rather than fire up another window to view `matrixfunctions.c`, we can ask `gdb` to display some lines of source code by using the `list` command:

```
(gdb) list
5       double ** allocMatrix(unsigned int rows, unsigned int cols)
6       {
7               double ** matrix;
8               unsigned int i;
9
10              assert((rows > 0) && (cols > 0));
11
12              matrix = (double **)malloc(rows*sizeof(double *));
13              if (!matrix) return NULL; /* failed */
14
(gdb)
```

We can confirm that both `rows` and `cols` are zero by asking `gdb` to print out the values with the `print` command:

```
(gdb) print rows
$1 = 0
(gdb) print cols
$2 = 0
```

The arguments to the `allocMatrix` function are indeed incorrect. Thus we need to focus our attention to the function that called `allocMatrix` in the first place. We can navigate to this function by using the `up` command once more:

```
(gdb) up
#5  0x08048645 in main () at crypticc.c:35
35              matrixA = allocMatrix(SZ, SZ);
```

So the problem is actually in our `main` function. If we look at the top of the file we see:

```
(gdb) list 1
1       #include <stdio.h>
2       #include <assert.h>
3       #include "matrixfunctions.h"
4
5       #define SZ 0
6
7       void whatdoIdo(double *A, double * B, double *C,
8                                      unsigned int L, unsigned int M,
9                                      unsigned int N)
10      {
(gdb)
```

We have defined `SZ` to be zero, which is completely wrong as `SZ` is the size of our matrices! As we know where the problem is we can quit from `gdb` by using the `quit` command:

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

We modify `SZ` to be 3, save the file and run make:

```
$ make
gcc -pedantic -ggdb -Wall -ansi    -c -o crypticc.o crypticc.c
gcc crypticc.o matrixfunctions.o -lm -o cryptic
```

Let's run the program to see what happens:

```
$ ./cryptic
 1.00000  0.00000  1.00000
 0.00000  1.00000  0.00000
 1.00000  0.00000  1.00000



 2.00000  0.00000  2.00000
 0.00000  2.00000  0.00000
 2.00000  0.00000  2.00000



 4.00000  0.00000  4.00000
 0.00000  2.00000  0.00000
 4.00000  0.00000  4.00000
```

It runs! The program runs, displays the expected output (two matrices and the result of a matrix multiply) then exits gracefully.

We have not finished debugging the program! Even though it gives the expected output, there may still be problems which have not yet manifested themselves. Most scientific computing programs perform linear algebra on large arrays of data. These arrays are typically malloced from the heap, and sometimes not freeed properly. Another common problem surrounds array indices, it is quite common to "loop too far" over an index essentially overwriting memory that doesn't belong to the function.

A program valgrind has been developed to check operations on the heap. It will keep track of all the mallocs and frees, and can also identify heap corruption caused by looping too far. We run our program through valgrind as follows:

```
$ valgrind ./cryptic
==1599== Memcheck, a memory error detector.
==1599== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==1599== Using LibVEX rev 1658, a library for dynamic binary translation.
==1599== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==1599== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==1599== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==1599== For more details, rerun with: -v
==1599==
 1.00000  0.00000  1.00000
 0.00000  1.00000  0.00000
 1.00000  0.00000  1.00000


 2.00000  0.00000  2.00000
 0.00000  2.00000  0.00000
 2.00000  0.00000  2.00000
==1599== Invalid read of size 8
==1599==    at 0x80485E9: whatdoIdo (crypticc.c:25)
==1599==    by 0x80487EF: main (crypticc.c:59)
==1599==  Address 0x4017168 is 0 bytes after a block of size 72 alloc'd
==1599==    at 0x40053C0: malloc (vg_replace_malloc.c:149)
==1599==    by 0x80488A7: allocMatrix (matrixfunctions.c:15)
==1599==    by 0x804865B: main (crypticc.c:36)
==1599==
==1599== Invalid read of size 8
==1599==    at 0x80485A7: whatdoIdo (crypticc.c:20)
==1599==    by 0x80487EF: main (crypticc.c:59)
==1599==  Address 0x40170B0 is 0 bytes after a block of size 72 alloc'd
==1599==    at 0x40053C0: malloc (vg_replace_malloc.c:149)
==1599==    by 0x80488A7: allocMatrix (matrixfunctions.c:15)
==1599==    by 0x8048644: main (crypticc.c:35)


 4.00000  0.00000  4.00000
 0.00000  2.00000  0.00000
 4.00000  0.00000  4.00000
==1599==
==1599== ERROR SUMMARY: 10 errors from 2 contexts (suppressed: 13 from 1)
==1599== malloc/free: in use at exit: 84 bytes in 2 blocks.
==1599== malloc/free: 6 allocs, 4 frees, 252 bytes allocated.
```

```
==1599== For counts of detected errors, rerun with: -v
==1599== searching for pointers to 2 not-freed blocks.
==1599== checked 51,180 bytes.
==1599==
==1599== LEAK SUMMARY:
==1599==    definitely lost: 84 bytes in 2 blocks.
==1599==      possibly lost: 0 bytes in 0 blocks.
==1599==    still reachable: 0 bytes in 0 blocks.
==1599==         suppressed: 0 bytes in 0 blocks.
==1599== Use --leak-check=full to see details of leaked memory.
```

We see the error message “Invalid read of size 8”, this occurs in the `whatdoIdo` function on line 25 of `crypticc.c`. It is possible to attach the debugger `gdb` to this program and inspect `whatdoIdo` in more detail. On my machine I had to use the command (this might vary on your installation of `valgrind`, please refer to the man pages for more details):

```
$ valgrind --db-attach=yes --db-command=" gdb -nw %f %p" ./cryptic
==1649== Memcheck, a memory error detector.
==1649== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==1649== Using LibVEX rev 1658, a library for dynamic binary translation.
==1649== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==1649== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==1649== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==1649== For more details, rerun with: -v
==1649==
 1.00000  0.00000  1.00000
 0.00000  1.00000  0.00000
 1.00000  0.00000  1.00000


 2.00000  0.00000  2.00000
 0.00000  2.00000  0.00000
 2.00000  0.00000  2.00000
==1649== Invalid read of size 8
==1649==    at 0x80485E9: whatdoIdo (crypticc.c:25)
==1649==    by 0x80487EF: main (crypticc.c:59)
==1649==  Address 0x4017168 is 0 bytes after a block of size 72 alloc'd
==1649==    at 0x40053C0: malloc (vg_replace_malloc.c:149)
==1649==    by 0x80488A7: allocMatrix (matrixfunctions.c:15)
==1649==    by 0x804865B: main (crypticc.c:36)
==1649==
==1649== ---- Attach to debugger ? --- [Return/N/n/Y/y/C/c] ----
```

Now we are presented with the option to attach a debugger. If we answer `y`, we obtain:

```
starting debugger
==1649== starting debugger with cmd:  gdb -nw /proc/1653/fd/1014 1653
GNU gdb Red Hat Linux (6.5-25.el5_1.1rh)
Copyright (C) 2006 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/libthread_db.so.1".

Attaching to program: /proc/1653/fd/1014, process 1653
0x080485e9 in whatdoIdo (A=0x4017068, B=0x4017120, C=0x40171d8, L=3, M=3, N=3)
    at crypticc.c:25
25                                      *cptr++ += temp*(*bptr++);
(gdb)
```

The error is being caused by the innermost loop of the matrix multiply function. We can expand the source code by using the `list` command, if we do `list` on its own it will centre the display around line 25. Instead we centre our display around line 21 (to see more of the preceding code):

```
(gdb) list 21
16              for(i=0; i < L; i++)
17              {
18                      for (k=0; k <= M; k++)
19                      {
20                              temp = A[i*M+k];
21                              cptr = &C[i*N];
22                              bptr = &B[k*N];
23                              bend = &B[(k+1)*N];
24                              while (bptr < bend)
25                                      *cptr++ += temp*(*bptr++);
```

Let's print out what the loop variables (`i` and `k`) are:

```
(gdb) print i
$1 = 0
(gdb) print k
$2 = 3
```

We have found the problem! Remember that in C, arrays are zero indexed, we expect the loop variables to go from 0 to 2 (as `L`, `M` and `N` are all 3, we can see this from the `gdb` output above). On line 18, we have `k <= M`, this should be replaced with `k < M`.

Now we have isolated one problem, we should quit from `gdb` (using the `quit`) command and just hit enter for the rest of the `valgrind` prompts. The file `crypticc.c` needs to be modified, and the program rebuilt:

```
$ make
gcc -pedantic -ggdb -Wall -ansi    -c -o crypticc.o crypticc.c
gcc crypticc.o matrixfunctions.o -lm -o cryptic
```

We can run the modified program through `valgrind` again:

```
$ valgrind ./cryptic
==1689== Memcheck, a memory error detector.
==1689== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==1689== Using LibVEX rev 1658, a library for dynamic binary translation.
==1689== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==1689== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==1689== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==1689== For more details, rerun with: -v
==1689==
 1.00000  0.00000  1.00000
 0.00000  1.00000  0.00000
 1.00000  0.00000  1.00000


 2.00000  0.00000  2.00000
 0.00000  2.00000  0.00000
 2.00000  0.00000  2.00000


 4.00000  0.00000  4.00000
 0.00000  2.00000  0.00000
 4.00000  0.00000  4.00000
==1689==
==1689== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
==1689== malloc/free: in use at exit: 84 bytes in 2 blocks.
==1689== malloc/free: 6 allocs, 4 frees, 252 bytes allocated.
==1689== For counts of detected errors, rerun with: -v
==1689== searching for pointers to 2 not-freed blocks.
==1689== checked 51,180 bytes.
==1689==
==1689== LEAK SUMMARY:
==1689==    definitely lost: 84 bytes in 2 blocks.
==1689==      possibly lost: 0 bytes in 0 blocks.
==1689==    still reachable: 0 bytes in 0 blocks.
==1689==         suppressed: 0 bytes in 0 blocks.
==1689== Use --leak-check=full to see details of leaked memory.
```

We see no read or write errors, and the program output is as expected. There is one problem though, the number of memory allocations (6) does not match the number of frees (4). Each matrix requires two allocations and we have allocated three matrices, but only `free`ed two. To quickly identify which matrix has not been `free`ed, we can re-run `valgrind` with extended leak checking:

```
$ valgrind --leak-check=full ./cryptic
==1695== Memcheck, a memory error detector.
==1695== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==1695== Using LibVEX rev 1658, a library for dynamic binary translation.
==1695== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==1695== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==1695== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==1695== For more details, rerun with: -v
==1695==
 1.00000  0.00000  1.00000
 0.00000  1.00000  0.00000
 1.00000  0.00000  1.00000


 2.00000  0.00000  2.00000
 0.00000  2.00000  0.00000
 2.00000  0.00000  2.00000


 4.00000  0.00000  4.00000
 0.00000  2.00000  0.00000
 4.00000  0.00000  4.00000
```

```
==1695==
==1695== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
==1695== malloc/free: in use at exit: 84 bytes in 2 blocks.
==1695== malloc/free: 6 allocs, 4 frees, 252 bytes allocated.
==1695== For counts of detected errors, rerun with: -v
==1695== searching for pointers to 2 not-freed blocks.
==1695== checked 51,180 bytes.
==1695==
==1695== 84 (12 direct, 72 indirect) bytes in 1 blocks are definitely lost in loss record 1
==1695==    at 0x40203C0: malloc (vg_replace_malloc.c:149)
==1695==    by 0x8048883: allocMatrix (matrixfunctions.c:12)
==1695==    by 0x8048672: main (crypticc.c:37)
==1695==
==1695== LEAK SUMMARY:
==1695==    definitely lost: 12 bytes in 1 blocks.
==1695==    indirectly lost: 72 bytes in 1 blocks.
==1695==      possibly lost: 0 bytes in 0 blocks.
==1695==    still reachable: 0 bytes in 0 blocks.
==1695==         suppressed: 0 bytes in 0 blocks.
==1695== Reachable blocks (those to which a pointer was found) are not shown.
==1695== To see them, rerun with: --show-reachable=yes
```

The offending memory block has been allocated in allocMatrix at line 12 (all memory allocation takes place in this function), which was called by line 37 of main:

```
        matrixC = allocMatrix(SZ, SZ);
```

At the end of main we have:

```
        freeMatrix(matrixA);
        freeMatrix(matrixB);
        return 0;
}
```

If we add code to free matrixC, rebuild and rerun the program through valgrind we obtain:

```
$ valgrind ./cryptic
==1716== Memcheck, a memory error detector.
==1716== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==1716== Using LibVEX rev 1658, a library for dynamic binary translation.
==1716== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==1716== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==1716== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==1716== For more details, rerun with: -v
==1716==
 1.00000  0.00000  1.00000
 0.00000  1.00000  0.00000
 1.00000  0.00000  1.00000



 2.00000  0.00000  2.00000
 0.00000  2.00000  0.00000
 2.00000  0.00000  2.00000
 4.00000  0.00000  4.00000
 0.00000  2.00000  0.00000
 4.00000  0.00000  4.00000
==1716==
==1716== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13 from 1)
==1716== malloc/free: in use at exit: 0 bytes in 0 blocks.
==1716== malloc/free: 6 allocs, 6 frees, 252 bytes allocated.
==1716== For counts of detected errors, rerun with: -v
==1716== All heap blocks were freed -- no leaks are possible.
```

No errors found by `valgrind`! We can be reasonably confident that the program is free from memory errors.

Summary

To summarise the following steps have helped us isolate the problems in the code:

- The command line option `-ggdb` for `gcc`, builds debug information into our program. (exercise: try removing this option from the Makefile, and follow the steps in this guide).
- No optimisation has been enabled in the Makefile, in fact optimisation should be *disabled* in all debug builds.
- The `assert` statements have also helped a great deal, they have stopped the program running before even more obscure problems were encountered (exercise: try to disable them all with `-DNDEBUG` in the Makefile and following this guide).
- We have seen how to navigate stack frames (`up` command) in `gdb`, and how to print out variables (`print` command). Other useful commands can be found online.
- Once the program has been found to be running as expected, we have performed some memory analysis using `valgrind`. Although the output is quite long-winded and intimidating at first, it has allowed us to clean up some very subtle errors in the code.
- Different versions of `valgrind` have slightly different command line parameters, I used version `valgrind-3.2.1` for this guide. You may need to refer to your documentation if you encounter any discrepancies.