Tools for Testing and Debugging

Note: these examples often invoke **undefined behaviour** so the results can vary depending on platform, optimisation compiler etc.

Actual commands are in (), {} mark arguments you should fill in with relevant values. See e.g. https://sourceware.org/gdb/onlinedocs/gdb/ for command details or use (help {command})

For these first exercises, try the debugger BEFORE looking at the source code. The answers are quite easy to spot, but in real code they're less obvious, so you want to practice using the tool. ALWAYS start at the first error!!

- GDB_eg1

- Demonstrates: array bounds violation

Info:

Try without and with -g. Doesn't change the output but gives line-numbers in errors

In this example, we can easily see where the fault is, and this helps us to fix it

Try:

Compile, the code, start GDB (gdb ./{program_name}) and run (run)

Add a breakpoint on the print line (line 9) and run the code. Note it stops BEFORE running the line

Print *i* (p i, print i) and *a*. Note *i* is 0 and *a* is garbage

Continue (continue) repeatedly until *i*=3

Check *a*, note we have successfully set each element to 0

Continue again, and should get a SIGTERM

All that continuing is silly

Remove the breakpoint you set (clear {linenum}) and instead try using a condition so that it runs only when we want it to (break {linenum} if i==3)

Run again (If asked "The program being debugged has been started already. Start it from the beginning? (y or n)" say y).

Now quit the debugger (quit)

- GDB_eg2

- Demonstrates: array bounds violation for heap array (malloc)

Info:

Sometimes bugs go unnoticed because they don’t error during normal running. On my test machine, an out-of-bounds read runs normally in this example, but failing to allocate at all crashes

Notice that gdb helps us locate WHERE the error is, but doesn't help us tell where it came from. Other tools (later) help more.

Try:

Run the code: expect ``Program received signal SIGSEGV, Segmentation fault.''

We can then get a backtrace (bt or backtrace)

Print the value of *i*: for me this crashes on the first iteration (i=0)

Print *a*. I see 0x0 which is clearly a bad pointer

Putting the facts together we can infer the error is when we access *a* at all

Add back the allocation line 9 and rerun

On my machine this runs without issue, even though there is still a bug!

- GDB_eg3

- Demonstrates: named breakpoints

Info:

In a real program, we would use the debugger to gain info to help us to work out WHY our input is going negative. In the example we can simply examine inside get_x

You may need to use -lm to link the maths library

Try:

run the code, note that we get -ve arguments hence NaN

set a conditional breakpoint on entering my_sqrt (break my_sqrt if input < 0)

run the code: it stops and we see that input was -3 BUT can't see the values in main that got us here

Get a backtrace (bt) - we came straight from main

We have to return to the correct “frame” (see glossary) (frame 1 or up)

Can see all local variables and values now! (info locals or p i, x)

See that this was i=0 iteration. Continue until next bad value, for me -1 at i=3

- GDB_eg4

- Demonstrates: a simple real program with bugs

Info:

The program is a simple binary search in ordered list of integers. It uses fixed size arrays, randomly filled with increasing values. There are two compile options: NORAND which uses fixed data, and DEBUG which does this and adds extra prints. Examine these first

Try:

run the code a few times. Note that we get a random array, although always increasing. By default the target value is size/2. Supply a number at command line to search for it (>> ./eg4 6) In GDB supply the number to the run command

If your compiler hasn't already warned you of one bug, turn on all warnings (-Wall in gcc)

Try changing the target value: check the lowest, highest, middle etc

The array size is set on line 6. Experiment with even numbers, powers of 2 etc since we know this code is repeatedly splitting into two. Try size 1 which is often strange

A list of the intentional bugs is at the end of this document

- Val_eg1

- Demonstrates: a heap array bounds violation

Info:

This is the same as GDB_eg2, but the array is allocated so we just have the bounds problem and I have added a second error, where we take a pointer to an array element and forget we have freed it. On my machine this mostly runs fine, but sometimes it can crash, so we always want to check for things like this.

Try:

Compile and run the code inside valgrind using (valgrind {name}). Don't forget the debugging symbols, and to turn off optimisation

Start at the first error

You should see something like

Invalid write of size 4

This says we're writing 4 bytes (an int) into memory we didn't allocate

Notice that it also says

Address 0xabcabcabc is 0 bytes after a block of size 40 alloc'd

This tells us the value we're accessing is right after a block of size 40 we **did** allocate

Next is

Invalid read of size 4

which tells us we're now reading this value back

Finally I get one more invalid read, for 3 total errors

On my machine, when I turn on optimisation (O1) Valgrind finds only 2 errors: the invalid read of a[10] is no longer noticed!

Fix the cause of the first error(s), the invalid access on *a*

Now examine that last message

Invalid read of size 4

....

Address 0x1007ff994 is 20 bytes inside a block of size 40 free'd

This tells us that we are accessing into some memory we already freed. Specifically we are 20 bytes (5 ints) into it.

The cause is easy to spot, rand_ptr = &a[5]; after freeing *a* this is orphaned

Fix the free and the program works as intended

- Val_eg2

- Demonstrates: uninitialised value

Info:

This uses the sum functions from yesterday: generate some random numbers and then sum them. This time there is a bug

Try:

Run the program a few times with (at least) O0 and O3. Any differences?

Sometimes Valgrind can be "fooled": try adding back the print in line 32, and rerun. I see a lot of errors,

Syscall param write(buf) points to uninitialised byte(s)

all traced to printing sum_a

Note that the report has hundreds of errors for a single bad print here. If you have a bad access like this, it can be tempting to remove it. Try removing both this line and the k++ line. What happens? This is because valgrind only tells you when a bad value is used, to avoid hundreds of false positives

Once again, valgrind tries to help, suggesting:

Use --track-origins=yes to see where uninitialised values come from

When we do this, we see that the bad value was created by the sum function. Sadly I don't get anything more useful than that, but I can go to the source code and see the problem

- Val_eg3

  - Demonstrates: a memory leak

Info:

Here we create a memory leak and some other issues. In C/C++ this is perhaps the commonest bug you will encounter.

Try:

Compile and run the program. You probably don't get any crashes or errors. What it *should* do is (slowly) print 5 different arrays.

Neither valgrind nor GDB will show you why all 5 arrays are the same. You'll have to look into the code

Try running the program inside valgrind (remember to turn off optimisation etc). You wont see an error but should see something like

LEAK SUMMARY:

==45574==   definitely lost: 200 bytes in 5 blocks

You should also have a line

Rerun with --leak-check=full to see details of leaked memory

Obey the line. It points us to where the leaking memory is allocated (not where it gets lost)

If you solved the random elements problem, it should be fairly obvious why we have a leak. (Hint: what happens to the array from the previous iteration?)

Fix the leak, and check it's fixed

- Prof_eg

  - Demonstrates: the basics of profiling

Info:

This is a very boring program that just calls a few levels of functions. Using either or both callgrind and a suitable system profiler, work out where the code spends most of its time.

Try:

Change optimisation levels. You'll probably find that O1 or more will inline the add, divide etc functions and remove the overhead of function call.

Try swapping the division operation (line 89) for the multiply. Does this help? Hinder?

Try adding file output (write_array function): how does this compare to the compute?

- Prof_branch

- Demonstrates: branch prediction

Info:

This code has 4 states, with different patterns of branching. Compile as normal, or use -DBR_{N} with N = 1, 2, 3. Ideally you would see consistently larger run times for some of these. It may or may not work, since some architectures manage to predict the branches. If it does work, try the things below

Try:

No option, and BR_1 should be quite similar, which proves that it is not the additional sum calculation changing the timing, but branch-prediction

If you are on a machine with a tool like perf you can run a basic stats using

perf stats {program_name}

IF you saw speed differences, you will probably see varying numbers under
``branch-misses   #   0.00% of all branches ''

Search online about branch prediction for more. The code here is adapted from https://stackoverflow.com/questions/11227809/why-is-it-faster-to-process-a-sorted-array-than-an-unsorted-array

BUGS in GDB_4:

Array definition has size-1 not size

index is unsigned, but fill_array returns -1 for not found. Bonus that the check fails, but the print shows -1 because it's a signed format code

Loop on line 80 is fine unless array_size is 1, in which case we get no iterations

Bug that debugger wont spot: atoi to set target wont care if you don't enter an int

Wont find last element (until first bug fixed, this element is out of range anyway)