Debugging Linux C++

Greg Law

co-founder and CTO, Undo

What this talk is and isn’t.

It is specific tools (with some tips and tricks) to detect and root cause bugs.

What this talk isn’t:

- Generic advice.
- About testing or testing tools.
- How to avoid writing bugs in the first place.
- Performance profiling (though some tools can be used for both - e.g. perf, ftrace).
- Exhaustive - really just a random bunch of things I’ve picked up and found useful.
- A workshop - you won’t become expert in any of it, but at least you’ll know stuff exists.

It does contain a bit of a context - why I care and why you should do.

The history

I well remember [...] on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

Sir Maurice Wilkes, 1913-2010

Debugging is most underestimated part of programming.

But surely it’s better not to write the bugs in the first place?

Well.. duh! Of course, but there will always be bugs.

Everyone knows that debugging is twice as hard as writing a program in the first place.
So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

This has profound implications - it means debuggability is the limiting factor.

There are many tools, use them.

Two main classes of debugging tool today:

1. **Checkers** (static and dynamic)
   “Did my code do bad thing x, y, z?”
   Examples: Address Sanitizer, Valgrind, Coverity

2. **Debuggers**
   “What exactly did my code *do?*”
   Examples: GDB, LLDB, rr, UndoDB, Live Recorder.

Tools we'll look at today

GDB	- 15 mins
Valgrind & Sanitizers	- 10 mins
strace	- 5 mins
ltrace	- 5 mins
ftrace	- 10 mins
perf trace	- 10 mins
GDB + rr/Undo	- 10 mins
GDB + Valgrind	- 5 mins
GDB + Asan	- 10 mins

GDB - more than you knew

GDB may not be intuitive but it is very powerful

- Easy to use, just not so easy to learn

TUI: Text User Interface

- As useful as it is poorly named!

GDB TUI (Test User Interface) top tips

ctrl-x-a: toggle to/from TUI mode

ctrl-l: refresh the screen

ctrl-p / ctrl-n: prev, next, commands

ctrl-x-2: second window; cycle through

GDB has Python!

Full Python interpreter with access to standard modules

(Unless your gdb installation is messed up!)

The **`gdb`** python module gives most access to gdb

`(gdb) python gdb.execute()` to do gdb commands

`(gdb) python gdb.parse_and_eval()` to get data from inferior

`(gdb) python help('gdb')` to see online help

Python Pretty Printers

```
class MyPrinter(object):
    def __init__(self,val):
        self.val = val
    def to_string(self):
        return ( self.val['member'])


import gdb.printing
pp = gdb.printing.RegexpCollectionPrettyPrinter('mystruct')
pp.add_printer('mystruct', '^mystruct$', MyPrinter)
gdb.printing.register_pretty_printer( gdb.current_objfile(), pp)
```

In-built pretty printers for STL

GDB will (try to) pretty-print most STL container classes (std::vector, std::string, etc), e.g.

```
10          vec.push_back(5);
(gdb) next
12          return 0;
(gdb) print vec
$6 = std::vector of length 3, capacity 4 = {3, 4, 5}
(gdb)
```

Note that this relies on Python pretty printers installed on the target system.

Compiling/linking with a different version of libstdc++ (e.g. executable built on a different host than the one being used to debug), then pretty printing might give strange results.

There are many (list with `info pretty-printers`), including:

`std::string`, `std::bitset`, `std::list`, `std::multimap`, `std::queue`, `std::set`, `std::shared_ptr`, `std::stack`, `std::tuple`, `std::unique_ptr`, `std::vector`, `std::weak_ptr`, and iterators.

.gdbinit

My ~/.gdbinit is nice and simple:

```
set history save on
set print pretty on
set pagination off
set confirm off
```

If you’re funky, it’s easy for weird stuff to happen.

Hint: have a project gdbinit with lots of stuff in it, and source that.

GDB is built on ptrace and signals

GDB is built atop ptrace.

When a program being traced receives a signal, it is suspended and the tracer gets notified (via waitpid)

So when the inferior receives a signal, it stops and gdb gets control.

Usually gdb returns to the prompt, but what it will do depends on the signal and how it is configured.

Two signals are special:

- SIGINT is generated when you hit ^C
- SIGTRAP is generated when the inferior hits a breakpoint or is single stepped.

  Actually, not so special - these signals are treated no differently to any others

GDB - terminal problems

Breakpoints and watchpoints

`watch foo`	stop when foo is modified
`watch -l foo`	watch location
`rwatch foo`	stop when foo is read
`watch foo thread 3`	stop when thread 3 modifies foo
`watch foo if foo > 10`	stop when foo is > 10

thread apply

```
thread apply 1-4 print $sp

thread apply all backtrace

Thread apply all backtrace full
```

Dynamic Printf

Use dprintf to put printf's in your code without recompiling, e.g.

```
dprintf mutex_lock,"m is %p m->magic is %u\n",m,m->magic
```

control how the printfs happen:

```
set dprintf-style gdb|call|agent
set dprintf-function fprintf
set dprintf-channel mylog
```

Calling inferior functions

`call foo()` will call foo in your inferior

But beware, print may well do too, e.g.

```
print foo()
print foo+bar
print errno
```

(if C++)

And beware, below will call **`strcpy()`** *and* **`malloc()`**!

```
call strcpy( buffer, "Hello, world!\n")
```

Catchpoints

Catchpoints are like breakpoints but catch certain events, such as C++ exceptions

- e.g. **`catch catch`** to stop when C++ exceptions are caught
- e.g. **`catch syscall nanosleep`** to stop at nanosleep system call
- e.g. **`catch syscall 100`** to stop at system call number 100

Remote debugging

Debug over serial/sockets to a remote server

Start **`gdbserver localhost:2000 ./a.out`**

Then connect from a gdb with e.g. **`target remote localhost:2000`**

Multiprocess Debugging

Debug multiple ‘inferiors’ simultaneously

Add new inferiors

Follow fork/exec

Multiprocess Debugging

```
set follow-fork-mode child|parent

set detach-on-fork off

info inferiors

inferior N

set follow-exec-mode new|same

add-inferior <count> <name>  [ attach PID ]

remove-inferior N

clone-inferior

print $_inferior
```

More Python

Create your own commands

```
class my_command( gdb.Command):
    '''doc string'''
    def __init__( self):
        gdb.Command.__init__( self, 'my-command', gdb.COMMAND_NONE)
    def invoke( self, args, from_tty):
         do_bunch_of_python()
my_command()
```

Yet More Python

Hook certain kinds of events

```
def stop_handler( ev):
    print( 'stop event!')
    if isinstance( ev, gdb.SignalEvent):
        print( 'its a signal: ' + ev.stop_signal)

gdb.events.stop.connect( stop_handler)
```

Other cool things...

`tbreak`	temporary breakpoint
`rbreak`	reg-ex breakpoint
`command`	list of commands to be executed when breakpoint hit
`silent`	special command to suppress output on breakpoint hit
`save breakpoints`	save a list of breakpoints to a script
`save history`	save history of executed gdb commands
`info line foo.c:42`	show PC for line
`info line * $pc`	show line begin/end for current program counter

And finally...

- gcc's -g and -O are orthogonal; gcc -Og is optimised but doesn't mess up debug
- -ggdb3 is better than -g

Valgrind

Valgrind is actually a platform, on which many different checkers are built.

- memcheck is the default and the most used/known.
- Also there is: cachegrind, callgrind, helgrind, drd, massif, lackey, none

Can be a bit slow, but very easy to use -- it generally just works.

- Can be as simple as `valgrind ./a.out`

Can be used in conjunction with gdb

```
valgrind --vgdb --vgdb-error 0 ./a.out
```

Valgrind tools

Cachegrind: cache profiler, simulates I1, D1, L2 caches

Callgrind: like cachegrind, but also with call-graphs

Massif: heap profiler

Helgrind: find race conditions in multithreaded programs

DRD: Data Race Detector. Like Helgrind, but uses less memory.

Lackey / None: demo/unit test of valgrind itself.

Valgrind memcheck - pros + cons

Pros:

- No need to recompile or otherwise modify binaries
- Good at spotting uninitialised accesses, and heap problems (buffer overrun, uninit data, etc)
- Good at spotting (simple) memory leaks.

Cons:

- Slow
- Not good for static/local buffer overruns.

Sanitizers

AddressSanitizer, MemorySanitizer, ThreadSanitizer, LeakSanitizer

Requires a special build of your program using clang, but faster and more powerful.

```
clang -g -fsanitize=address foo.c
./a.out
```

Or with gcc, like:

```
gcc -g -fsanitize=address -static-libasan out_bounds.c
./a.out
```

Address sanitizer pros and cons

Pros:

- Fast
- Good at static and local buffer overruns (incl stack smashing)
- Compatible with other tools

Cons:

- Recompile
- Uses lots of memory
- False positives with ‘fortification’

See also: Memory Sanitizer, Thread Sanitizer

See also: `__attribute__((no_sanitize_address))`

_FORTIFY_SOURCE

Compile with `-D_FORTIFY_SOURCE={0,1,2}`

Adds checks to:

memcpy, mempcpy, memmove, memset,

strcpy, stpcpy, strncpy, strcat, strncat,

sprintf, vsprintf, snprintf, vsnprintf, gets.

-D_FORTIFY_SOURCE=2 is the default on many modern distros.

```
gcc -D_FORTIFY_SOURCE=2 -Wall -g -O2 fortify_test.c && ./a.out $(python -c 'print "\x41" * 360')
```

ftrace

“Function tracer” - a fast way to trace various kernel functions.

- Lots of predefined *events* (i.e. ‘trace-points’)
- Controlled through `/sys/kernel/debug/tracing`
- Or, use the `trace-cmd` utility

ftrace: a case study

Trying to get Live Recorder embedded into a large US software vendor's test-suite.

Customer: Live Recorder keeps dying with SIGKILL
Undo: Are you sure you don't have some kind of proces killer in your test-suite?
Customer: Totally sure, we have no such thing.
Undo: Hmm, ok, we'll take a look.
….
Undo: Are you really sure you don't have some kind of process killer?
Customer: Yep, 100%
Undo: Hmm, would you mind running this script for us and sending us the output?
Customer: Sure hang on…. Here you go!
Undo: Ummm, what's this process you have called "watchdog"?
Customer: I don't know, let me ask around….
Customer: That's some kind of process killer!

ftrace: tracking all the signals

```
root@tommy:-# echo "sig>=0" > /sys/kernel/debug/tracing/events/signal/filter
root@tommy:-# echo 1 > /sys/kernel/debug/tracing/events/signal/enable
root@tommy:-# echo 1 > /sys/kernel/debug/tracing/tracing_on
...
root@tommy:-# cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
Chrome_ChildIOT-3774  [001] d.h. 94826.246904: signal_generate: sig=14 errno=0 code=128 comm
           Xorg-2565  [001] d... 94826.246949: signal_deliver: sig=14 errno=0 code=128 sa_ha
            cat-30948 [003] d... 94834.119664: signal_generate: sig=17 errno=0 code=1 comm=b
           bash-30873 [001] d... 94834.119824: signal_deliver: sig=17 errno=0 code=1 sa_hand
           Xorg-2565  [000] d.h. 94834.130078: signal_generate: sig=14 errno=0 code=128 comm
           Xorg-2565  [000] d... 94834.130105: signal_deliver: sig=14 errno=0 code=128 sa_ha
```

ftrace: trace-cmd

```
root@tommy:~# less /sys/kernel/debug/tracing/available_events
root@tommy:~# trace-cmd record -e tcp:tcp_destroy_sock
root@tommy:~# trace-cmd report
```

strace

Trace all the system calls of a process.

`strace cmd`	Print all system calls issued by cmd
`strace -k cmd`	Show backtrace for each syscall
`strace -t cmd`	Prefix each syscall with wallclock time
`strace -o file cmd`	Write traced syscalls to file
`strace -D cmd`	strace process runs as detached grandchild of cmd
`strace -f cmd`	Follow forks
`strace -ff -o f cmd`	Follow forks, write output to f.%pid
`strace -p 1234`	Strace process-id 1234 (note: only the thread, not all threads in that process)
`strace -p 1234 -f`	Attach to all threads in process-group 1234

ltrace

Trace all the dynamic library calls of a process.

`ltrace cmd`	Print all library calls issued by cmd
`ltrace -w=4 cmd`	Show backtrace (up to 4 frames) (*but only if ltrace compiled with libunwind*)
`ltrace -t cmd`	Prefix each library call with wallclock time
`ltrace -l libc.so* c`	Trace calls to libc library only
`ltrace -e malloc+free-@libc.so* cmd`	only trace malloc and free symbols from libc

perf trace

Like strace but better (and worse).

`perf trace`	Trace every syscall by every process
`perf trace cmd`	Trace all syscalls issued by cmd
`perf trace -p 1234`	Trace all syscalls from process 1234
`perf trace -e read*`	Trace all syscalls that start with ‘read’ (e.g. read, readlink, readdir)
`perf trace -D 500`	Wait 500ms before tracing (skip all the startup gumf)
`perf trace record`	Record into perf.data (this is really for profiling, so won’t talk about it more!)

Better than strace: Much faster; more flexible.
Worse than strace: Needs privileges, doesn’t do as much decoding (e.g. strings look like pointers)

perf PT (Processor Trace)

```
perf record -e intel_pt//u ./a.out
perf script
```

Approx 1 bit per branch, can generate 100’s MB’s per second per core.

See also intel_bts (but requires no kpti boot).

Reversible Debugging - how did that happen?

GDB inbuilt reversible debugging: Works well, but is **very** slow

GDB in-build ‘record btrace’: Uses Intel branch trace.

- Not really reversible, no data
- Quite slow

rr: Very good at what it does, though somewhat limited features/platform support

UndoDB and Live Recorder: perfect!

- But expensive :)

Other cool things

CLion

backtrace.io

Thank you

Undo

22 Station Road,
Cambridge, CB1 2JD, UK