Linux C++ Quality & Debugging Tools - Under the covers

Greg Law, CTO, Undo
Dewang Li, Software Integrity Architect, Synopsys

undo

Debugging - our dirty secret

Most of a developer's time is spent debugging.

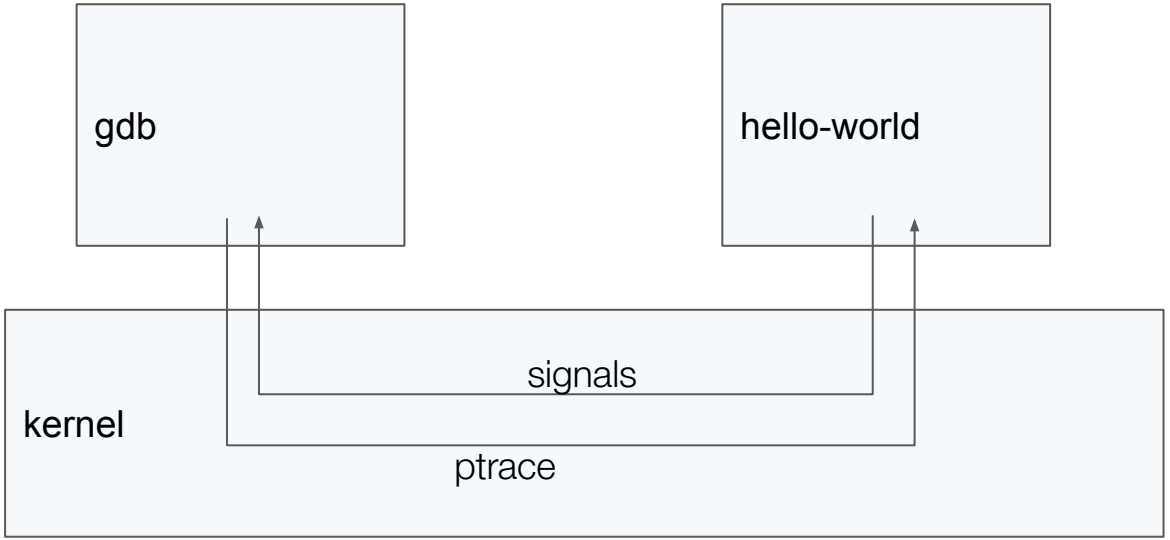
What happened?

Types of tools and some examples

- The debugger - GDB
- Record and replay - rr, Live Recorder
- Dynamic checkers - Valgrind, Sanitizers
- Static analysis - cppcheck, Coverity

- what is my program doing?
- what just happened?
- did thing X happen?
- could thing X happen?

GDB

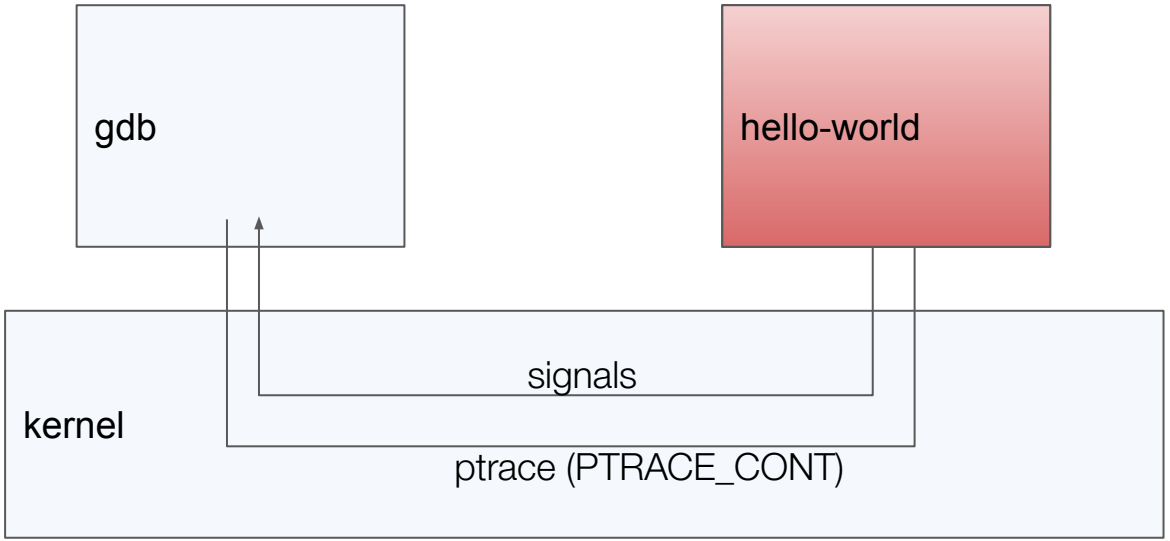


Signals and ptrace

On receipt of signal:

- Terminate (potentially dumping core)
- Ignore
- Stop (suspend)
- Run signal handler
- Stop (tracing stop)

Running a program under ptrace



The debugger and signals

Signals only reach the tracee via PTRACE_CONT

e.g. ptrace(PTRACE_CONT, pid, NULL, (void*)SIGALRM)

if blocked by tracee the handler will run when the signal becomes unblocked

if ignored by tracee the signal is discarded

Breakpoints and single-step are SIGTRAPs

^C is SIGINT

Signal algorithm

if sig == SIGKILL: kill process

else if sig == SIGSTOP: suspend process

else if traced: suspend process (tracing stop)

else if blocked(sig): mark pending

else if ignored: ignore

else if handler: run handler

else: terminate process

ptrace

PTRACE_CONT

PTRACE_SINGLESTEP

PTRACE_SYSCALL

PTRACE_GETREGS

PTRACE_SETREGS

PTRACE_PEEKTEXT/PTRACE_PEEKDATA

PTRACE_POKETEXT/PTRACE_POKEDATA

PTRACE_PEEKUSER

PTRACE_POKEUSER

ptrace continued

PTRACE_GETSIGINFO / PTRACE_SETSIGINFO

PTRACE_GETSIGMASK / PTRACE_SETSIGMASK

PTRACE_PEEKSIGINFO

PTRACE_GETFPREGS / PTRACE_SETFPREGS

PTRACE_GETREGSET / PTRACE_SETREGSET

PTRACE_ATTACH / PTRACE_DETACH

PTRACE_SETOPTIONS

PTRACE_GETEVENTMSG

PTRACE_SEIZE

strace uses PTRACE_SYSCALL

Two SIGTRAPs per syscall - on syscall entry, on syscall exit.

On x86 distinguish entry/exit as AX contains -38 (-ENOSYS) on entry.

Syscall restart

A stopped process is stopped in userspace.

A PTRACE_CONT transparently (mostly) restarts the syscall.

If you see e.g. ERESTARTSYS this has probably gone wrong.

Breakpoints and watchpoints

Breakpoints - on x86, instruction “int $3” is written (single-byte 0xcc).
Results in SIGTRAP sent to tracee (but of course intercepted by debugger)

Hardware watchpoints on x86 via the ‘debug registers’ - db0..db7

Set from the debugger by PTRACE_POKEUSER

The debugger - DWARF info

Debugging With Attributed Record Formats

- Map a PC to a source file:line
- Stack layout
- Type information
- Function prototypes
- Macros, classes, templates, …

```
g++ -g hello.cpp
g++ -Og -g hello.cpp
g++ -O3 -g3 hello.cpp
```

Debugging optimized code

(gdb) print foo

$1 = <optimized out>

-ggdb3 -gdwarf-4 -fvar-tracking-assignments -ginline-points -gstatement-frontiers

Or just

-g3

inline, macros, templates

Exploring DWARF info

readelf --debug-dump

readelf --debug-dump=loc

Stacks and CFA / CFI

Frame pointers often omitted

(e.g. with -O on x86-64, with -fomit-frame-pointer on i386)

Call Frame Address (CFA) to the rescue

Map PC -> offset from current SP to stack frame

Exception handling code uses CFA and CFI (Call Frame Information) to unwind the stack. (And a whole bunch of other stuff.)

Catchpoints

catch throw / catch catch uses breakpoints

catch syscall uses PTRACE_SYSCALL

libthreaddb

Thread library provides a library that the debugger can call.

Also used for TLS, including errno

GDB RESOURCE PORTAL

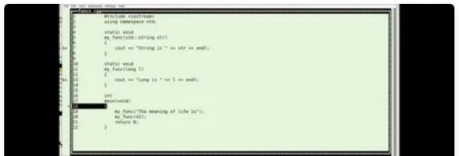
gdbWatchPoint

Search Resources Search

All types (5)

gdbWatchPoint (5)

All types 5 items



Address Sanitizer + Valgrind

"Shadow memory" tells us whether or not memory is accessible.

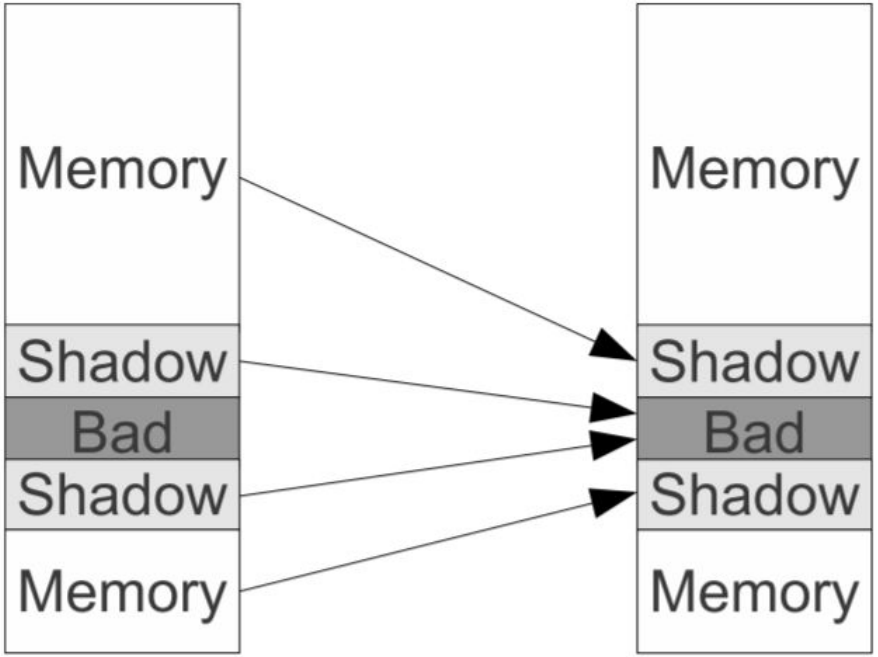
malloc and free are intercepted, and redzones appended/prepended

Valgrind: binary JIT instrumentation

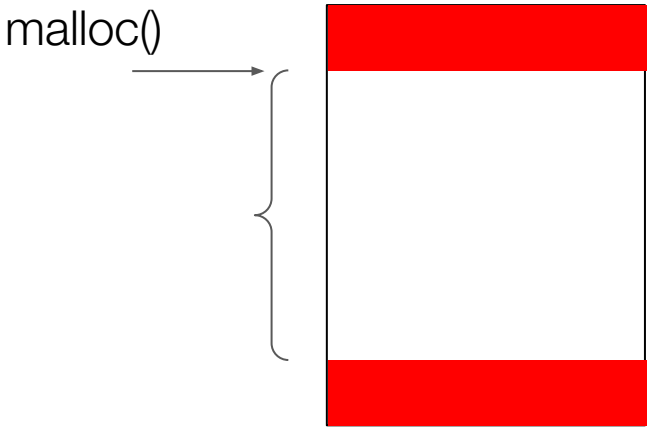
- No changes to the application

Asan: done by the the compiler

- Some runtime overhead
- Detects stack buffer overrun



Red-zones



Record and replay - rr and Live Recorder

Go to any instruction that executed and see any piece of state

Use deterministic re-execution to compute prior states

Record and replay non-deterministic stimuli:

- System calls
- Thread switches
- Signals
- Non-deterministic instructions
- Shared memory

Need a precise notion of where in the program's execution we are

rr - performance counters

CPU performance counters (“retired branches”)

- Fast
- Simple
- Not always available
- Can’t do everything

Use PEBS (Precise Event Based Sampling) to interrupt for replay.

Live Recorder - JIT instrumentation

JIT to count branches

- Imposes runtime overhead
- Works everywhere
- Covers all cases (including shared memory)

undo™