Bela Lugosi in Dracula

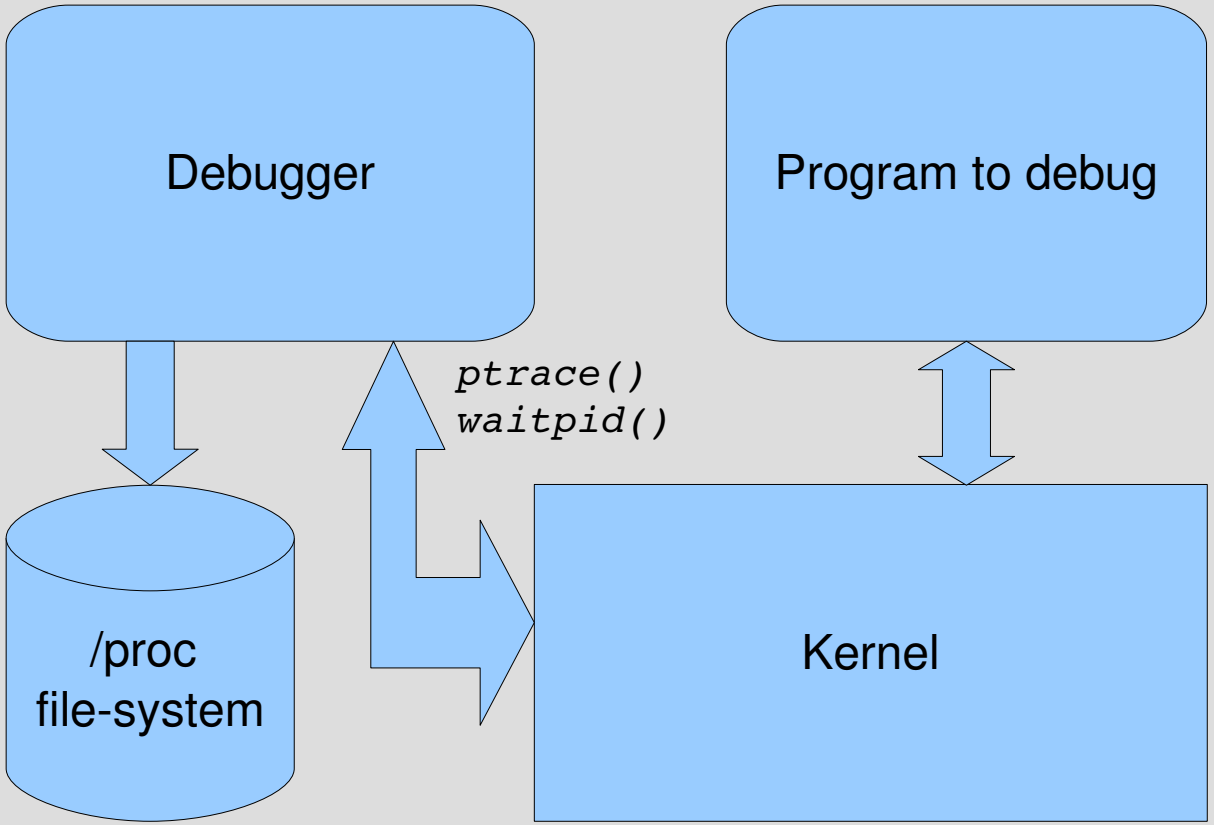
Anatomy of a Debugger

- Understanding how a debugger works and how it is put together increases the developer's efficiency in using tools
- Understanding debugger architecture allows developers to choose the right tool for the job
- Today's Linux debuggers have room for improvement

Agenda

- Overview of debugger / program interaction
- Remote & cross-debugging: out of scope
- Fine-tunning the debug information
- Multi-threaded programs
- Limitations
- Advanced debugger usage
- Possible performance improvements
- Q and A

Debugger / Program Interaction



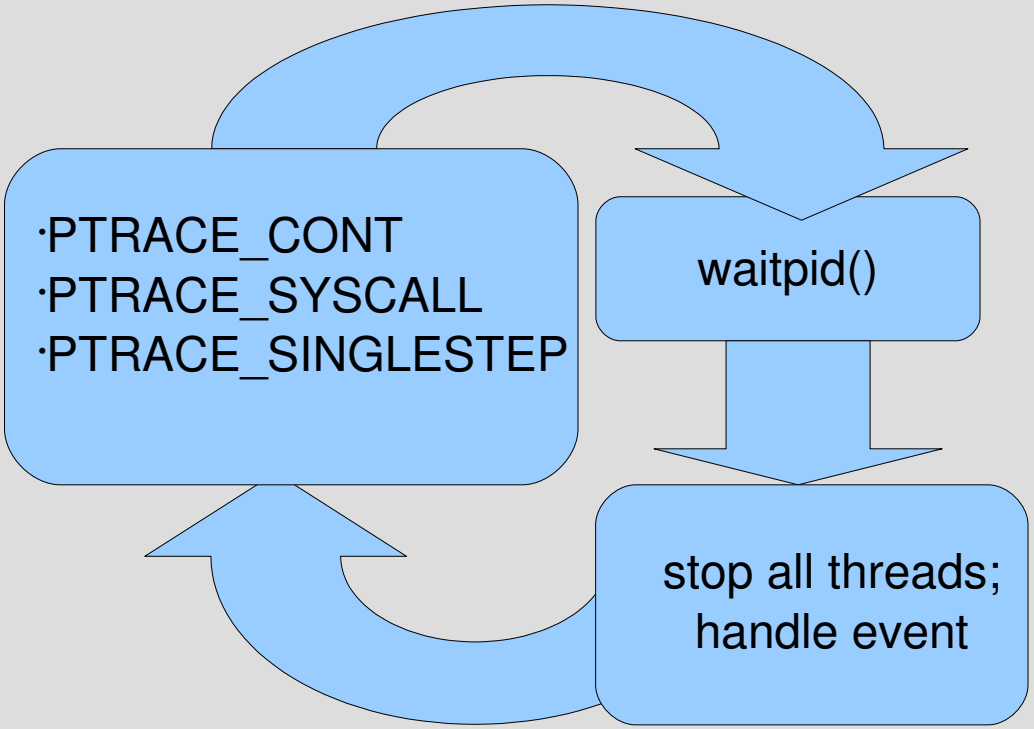
Program Interaction

- The debugger uses the ptrace() API to:
  - attach / detach to / from targets
  - read / write to / from debugged program memory
  - resume execution of program
- Debugger sends signals to program
  - most notably SIGSTOP
- The /proc file-system allows reading / writing of memory, and provides access to other information (command line args, etc)

Debug Events

- When signals occur in the program the debugger is notified (before signal is delivered)
- The status of the debugged program is collected with the waitpid() call
- **The debugger is a big event loop**
- Example of events: SIGTRAP (breakpoint hit, or single-stepping), SIGSEGV, SIGSTOP

Main Event Loop



Resuming the Program

- ptrace(PTRACE_CONT) resumes program
- PTRACE_SINGLESTEP: executes one machine instruction
- PTRACE_SYSCALL: continue up until a system call occurs (this is what strace uses)

Handling Events

- Most common case: display the state of the program to user
  - call stack trace
  - variables
  - CPU registers
  - current source code location
- Execute a user command
- Allow user to insert breakpoints
- Allow user to evaluate expressions

Symbolic Information

- There are two sources of information:
  - the ELF symbol tables used by the linker
  - debug info stored in a special ELF section
- Each shared object and executable has one or two ELF symbol tables (static, dynamic)
- It is easy to do an address lookup to find the current function
- The maps file under /proc may be consulted to determine which symbol table to use

/proc/<pid>/maps

```
[cristiv@orcas:~]$ cat /proc/3454/maps
08048000-08061000 r-xp 00000000 08:01 568685                             /usr/bin/vmnet-dhcpd
08061000-08063000 rw-p 00019000 08:01 568685                             /usr/bin/vmnet-dhcpd
08063000-0808e000 rw-p 08063000 00:00 0                                  [heap]
f7db7000-f7dc0000 r-xp 00000000 08:01 90135                              /lib/libnss_files-2.4.so
f7dc0000-f7dc2000 rw-p 00008000 08:01 90135                              /lib/libnss_files-2.4.so
f7dc2000-f7dc3000 rw-p f7dc2000 00:00 0
f7dc3000-f7eea000 r-xp 00000000 08:01 90121                              /lib/libc-2.4.so
f7eea000-f7eeb000 r--p 00127000 08:01 90121                              /lib/libc-2.4.so
f7eeb000-f7eed000 rw-p 00128000 08:01 90121                              /lib/libc-2.4.so
f7eed000-f7ef0000 rw-p f7eed000 00:00 0
f7f0d000-f7f0f000 rw-p f7f0d000 00:00 0
f7f0f000-f7f27000 r-xp 00000000 08:01 90114                              /lib/ld-2.4.so
f7f27000-f7f28000 r--p 00017000 08:01 90114                              /lib/ld-2.4.so
f7f28000-f7f29000 rw-p 00018000 08:01 90114                              /lib/ld-2.4.so
ffce9000-ffcec000 rwxp ffce9000 00:00 0                                  [stack]
ffffe000-fffff000 r-xp ffffe000 00:00 0
[cristiv@orcas:~]$
`
```

Tracking Symbol Tables

- Symbol tables are read from ELF binary files (not from debugged program's memory, I.E. no debugging API is needed)
- The debugger needs to track the loading and unloading of dynamic library
- The dynamic loader provides a special location where the debugger needs to insert a breakpoint (see ***/usr/include/link.h***)

Debug Information

- Source file and line number
- Variables' names and data types
- Frame unwinding information
- The format is not specified by ELF
- The format is language-independent (kind of)
- Most common formats: DWARF, STAB
- The amount of debug info is proportional to the complexity of your code

Fine Tunning the Debug Info

- Prefer DWARF over STAB debug format
- STAB is linear by design, DWARF allows “accelerated” lookups
- In DWARF, there is one “handle” per shared object: modular programs are easier to debug
- Lookups work faster with smaller modules

Know Your Compiler

- Older GCC (2.95) used STAB by default
  - Hints:
    - -gstabs+ generates more info
    - -gdwarf-2 is even better
- Eliminate unused info
  - if you are not interested in casting when evaluating expressions in the debugger, you may instruct the compiler to omit some type info

C++ Language Specifics

- Be aware of name-space aliases and macros when evaluating expressions
  - example: std::string may actually be _STL::string
- Sometimes, forward declared types cause the debugger to work harder when looking up type information

Debugger Architectural Blocks

- Main Event Loop
- Thread Management (detects and attaches to new threads)
- Symbol Table Management
- Breakpoint Management
- Debug Info Readers
- Expression Evaluators (Interpreters)
- Scripting support, for automating tasks

Design Lends Itself to C++

- Polymorphism. Examples: software and hardware breakpoints are handled uniformly, ditto for live processes and core dumps
- The entire debugger system is designed around an object model, expressed as pure abstract classes.

Poor Man's COM

- Most interfaces derive from ZObject, which provides `add_ref`, `release`, and `query_interface`
- Unlike in MS COM, add_ref and release are not public
- Can throw exceptions between components (*if you carefully read the fine print...*)

Example: The RefCounted Class

```
DECLARE_INTERFACE_(RefCounted, Unknown)
{
    template<typename T> friend class RefPtr;
    template<typename T> friend class WeakPtr;
public:
    virtual long ref_count() const volatile = 0;
    virtual void self_check() const volatile = 0;

protected:
    virtual void inc_ref() = 0;
    virtual bool dec_ref_and_test() = 0;
//...

};
```

DECLARE_INTERFACE?

```
#if defined(__GNUC__) && (__GNUC__ < 3) && !
defined(__INTEL_COMPILER)
 #define ZDK_VTABLE __attribute__((com_interface))
#else
// GCC 3.x and above generate COM-compatible vtables by
default
 #define ZDK_VTABLE
#endif

#define DECLARE_INTERFACE(i) struct ZDK_VTABLE i
#define DECLARE_INTERFACE_(i,b) struct i : public b
```

interface_cast<>

```
template<typename T> T inline
interface_cast(Unknown2& u)
{
    typedef typename boost::remove_reference<T>::type V;
    V* ptr = 0;

    if (!u.query_interface(V::_uuid(), (void**)&ptr))
    {
        throw bad_interface_cast(V);
    }
    assert(ptr);
    return *ptr;
}
```

interface_cast<> cont'd

```
template<typename T> T inline interface_cast(Unknown2* u)
{
    typedef typename boost::remove_pointer<T>::type V;
    T ptr = 0;
    if (u)
    {
        if (u->query_interface(V::_uuid(), (void**)&ptr))
        {
            assert(ptr);
        }
    }
    return ptr;
}
```

Support for D Language

- Demangling (generously contributed by Thomas Kühne)
- Associative arrays (we had to re-purpose DW_containing_type as the key type)
- Dynamic arrays
- ... and hopefully more to come

Limitations and Challenges

- Stopping threads with SIGSTOP is kludgey
- In a multi-threaded debugger, only the thread that has initially attached to the target with ptrace(PTRACE_ATTACH) can subsequently use ptrace and /proc on the same target
- ptrace calls (other than PTRACE_ATTACH) fail when invoked with the ID of a running process

Multi-Threaded Programs

- The easiest way of handling debug events in multi-threaded programs is to stop all threads upon events
- Stopping all threads with SIGSTOP: kill the group or use tkill() where available
- The thread ID as returned by pthread_create is NOT the same as the task ID

Multi-Threaded, Part 2

- On Linux, the relationship between user-space threads and kernel threads is 1-to-1
- Not necessarily true on other UNIX derivatives
  - Example: In FreeBSD the mapping is dynamic
- ptrace() operates on task ID, not thread ID
- libthread_db.so provides thread information

Affinity Workaround

- In ZeroBUGS, the UI and the main debugger engine run on separate threads
- They each are a big event loop
- The UI needs to be responsive
- The UI thread **cannot call ptrace()**
- Calls need to be marshaled between threads

Debugging Memory

- Using custom breakpoints to track heap
- Break on malloc / calloc / realloc / free and record the addresses and sizes of block
- Very slow, because of **numerous context switches**
- Remember? Signals go through the kernel
- Valgrind works better for memory debugging

Advanced Usage

- Use the debugger to test a patch without rebuilding your code.
- Example: you suspect that an uninitialized variable is the cause of a bug
- Set a conditional breakpoint at the line where you want to initialize the variable, and make the condition something like **((x = -1) && false)**

Advanced, Part 2

- Turn the debugger into a unit-test harness

```
 1 import zero
 2 #
 3 # user-defined action, verifies that the value
 4 # of 'x' is 42 when the breakpoint is hit
 5 #
 6 def verify_post_cond(thread, breakpoint):
 7     for sym in thread.eval('x'):
 8         assert sym.value() == '42'
 9     return True # don't discard the breakpoint
10
11 # the debugger calls this when attaching to a process
12 def on_process(process, thread):
13     for sym in process.symbols().lookup('some_fun'):
14         thread.set_breakpoint(sym.addr(), verify_post_cond)
```

Potential Improvements

- Store debug information into database (such as SQLite http://www.sqlite.org/)
- Multi-threaded design (take advantage of emerging multi-core architectures).
- Take advantage of the new UTRACE facility (http://lwn.net/Articles/224772/)

Q&A, Contact Info

- www.zerobugs.org
- www.zero-bugs.com/
- Cristian Vlasceanu cristian@zerobugs.org