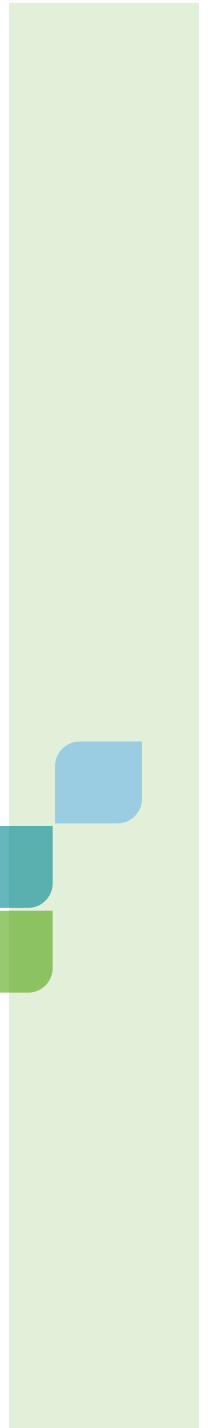




message  
systems™

**Gimli:**  
**Server Process Monitoring  
& Fault Analysis**





## Agenda

- What problem does Gimli solve?
- How does it work?
- How can I leverage Gimli?
- Where can I get Gimli?
- How can I contribute to Gimli?



## The Setting

- You have software deployed in Production
- Production:
  - has stability and/or downtime requirements
  - May prohibit gdb and interactive debugging tools for security purposes
  - May be at the other end of a VPN and/or be subject to draconian access policies



## The Problem

- Your software is manifesting a crash/hang problem in Production
- Can't use gdb:
  - Takes too long to poke around
  - May not be able to run gdb
- Getting a usable core file is hard:
  - Big server processes leave big cores
  - Can't debug the core on the box
  - Can't get the core back to your box
- How can you effectively get a bead on the issue?



## Gimli – a solution in 2 parts

- Monitor: supervises your target process
  - Optional watchdog support via heartbeat
  - If a fault is detected (crash, hang) invokes the analysis component, then re-spawns the target process
- Glider (the analyzer):
  - Pstack on steroids
  - DWARF-3 assisted stack unwinding
  - Full backtrace and deep structure printing
  - Optional extension modules to augment trace
- Runs on: Linux, Solaris, Darwin OS/X.
  - i386, x86\_64/amd64.
  - Incomplete support for Solaris Sparc (mostly there!)



## The Gimli Monitor

- A bit like “init”, but focused on a single process
- Detaches from the terminal and establishes a new session
- Spawns the child process and monitors:
  - Did the child exit abnormally (signalled)?
  - Did the child get STOP’d?
  - If the child opted in to heartbeat, did its heart stop beating?
- If one of these conditions triggers, and the process is still running, the Glider is invoked
- The process is then respawned (and throttled if respawning too fast)



## Monitor options

- All options can be command line, environmental or in a config file
- Watchdog interval – heart must beat at least once in this many seconds
- Detach, setstid: alter daemonizing behavior
- Trace-dir – where to record trace files
- Pidfile – where to store pidfile
- Uid, gid – setuid/setgid to the specified user before spawning the child
- Respawn-frequency – brake on respawning to avoid hammering the system



## Heart beat

- Monitor assumes that the child does not support heartbeat until it beats at least once
- Two mechanisms for heart beat:
  - Simple: `kill(getppid(), SIGUSR1)`
    - Easy for scripts to implement
  - Shared Memory via libgimli:
    - `hb = gimli_heartbeat_attach()`
    - `gimli_heartbeat_set(hb, GIMLI_HB_RUNNING);`



## Effective Heart beat

- Intended to issue one heart beat per iteration of the main loop in your app
- If it stops ticking over at least once every watchdog interval (60 seconds by default), it is deemed to have hanged
- Generally not good to heart beat inside utility functions; better to keep it at the higher level in the app to avoid false negatives
- Exception to this rule might be an infrequent but long running function



## Trapping Faults

- If you don't install a signal handler, the Monitor won't be able to trap a fault and generate a trace
- libgimli provides a `gimli_establish_signal_handlers()` to set up suitable fault trapping
- Handler:
  - Clears signal handler so a double fault will kill us
  - SIGSTOP's self; monitor is notified via SIGCHLD
  - When it resumes, calls an application supplied shutdown hook
  - Sends the faulting signal to itself so that it terminates abnormally



## Tracing

- When the Monitor decides that the child is dead, it initiates tracing
- Creates a file named `appname.pid.trc` in the specified trace dir (default: `/tmp`)
- Emits a header describing the reason for the fault and when it happened
- Spawns Glider and has its output append to the trace file



## Trace Header

This is a trace file generated by Gimli.

Process: pid=6704 /path/to/faulty-app

Traced because: fault detected

Time of trace: (1278702746) Fri Jul 9 15:12:26 2010

Invoking trace program: /opt/msys/gimli/bin/gliders



## The Gimli Glider

- Name comes from a combination of well known Dwarf and Elf fantasy literature and an incident in Manitoba where an aircraft ran out of fuel
- Fundamental purpose is to get a stack trace of all threads in the target process in a *human readable form*
- Uses DWARF-3 debugging information
  - `gcc -gdwarf-2 -g3`
- Extracts signinfo from signal trampolines to show more detail about the fault



## Example partial trace

```
Thread 0 (LWP 6704)
#0  0x00000035ad69a1a1 /lib64/libc-2.5.so`nanosleep+41
#1  0x00000035ad699fc3 /lib64/libc-2.5.so`sleep+93
#2  0x00002b029434bde6 /opt/msys/3rdParty/lib/perl5/5.10.0/x86_64-linux-thread-
    multi/CORE/libperl.so`Perl_pp_sleep+56 (pp_sys.c:4525)
PerlInterpreter *my_perl = 0x6ff2010 (/opt/msys/3rdParty/bin/perl`0x6ff2010)
[deref'ing my_perl]
PerlInterpreter @ 0x6ff2010 = {
  SV **Istack_sp = 0x71938c8 (/opt/msys/3rdParty/bin/perl`0x71938c8)
  OP *Iop = 0x7cd8f30 (/opt/msys/3rdParty/bin/perl`0x7cd8f30)
  SV **Icurpad = 0x7019fc0 (/opt/msys/3rdParty/bin/perl`0x7019fc0)
  SV **Istack_base = 0x71938c0 (/opt/msys/3rdParty/bin/perl`0x71938c0)
  SV **Istack_max = 0x71958a8 (/opt/msys/3rdParty/bin/perl`0x71958a8)
  I32 *Iscopystack = 0x76331c0 (/opt/msys/3rdParty/bin/perl`0x76331c0)
  I32 Iscopystack_ix = 5 (0x5)
  I32 Iscopystack_max = 108 (0x6c)
```



## Glider vs pstack

- Pstack is terse, does not include file:line info
- Glider is verbose (full backtrace)
- Glider is extensible



## Extending Glider

- It is often necessary to get more than just a stack trace out of a faulting application
- Want a readable snapshot of important datastructures like:
  - Circular log buffer
  - Stats
  - State machine dumps
  - Memory utilization
- Glider provides an API for extracting this type of data from your process



## Gimli Modules

- Gimli modules are shared objects that are loaded and executed by the glider process, not the target, faulting process
- Glider examines the mapped objects in the target process and locates corresponding gimli modules.
- Example: if I link against libfoo.so, gimli looks for a gimli\_libfoo.so
- If a module in the target has a “gimli\_tracer\_module\_name” symbol, it is interpreted as the name of a module to load in the glider instead



## Glider Modules

- Gimli modules export a function named “gimli\_ana\_init” which is invoked when the module is loaded into the Glider
- It is expected to return a Gimli module structure that allows the module to hook into aspects of the tracing
- When all the modules have been loaded, the trace begins
- Can intercept each threads trace and each frame of the trace
  - Can either emit additional info or suppress the item
- After the stack trace, modules have their generic tracing hook invoked to allow arbitrary other data to be emitted



## Glider API: libgimli\_ana.h

- Provides functions that:
  - resolve symbols
  - Reverse map an address to a symbol
  - Copy memory from the target process
  - Copy a string from the target process
  - Get source (file:line) information for an address
  - Examine DWARF parameter information for a named parameter in a given stack frame



## Extracting data from globals

- Given the following global data in the target, how can I print it out in my Gimli module?

```
char *my_str_ptr = "hello";  
int my_int;  
char my_str[1024];  
struct foo {  
    int value;  
} my_foo;  
struct foo *my_foo_ptr = &my_foo;
```



```
static void perform_trace(const struct
    gimli_ana_api *api, const char *object)
{
    // First the string pointer
    char *s = api->get_string_symbol(object,
        "my_str_ptr");
    printf("%s\n", s ? s : "null");
}
```



```
// Now the int. The 0 below means don't deref
int ival;
if (api->copy_from_symbol(object, "my_int",
    0, &ival, sizeof(ival)) {
    printf("my_int: %d\n", ival);
}
```



```
// Now the string buffer
char buf[1024];
if (api->copy_from_symbol(object, "my_str",
    0, &buf, sizeof(buf)) {
    printf("my_str: %s\n", buf);
}
```



```
// Now the foo struct; you need to #include its
// definition, or re-declare it here
struct foo f;
if (api->copy_from_symbol(object, "my_foo",
    0, &f, sizeof(f)) {
    printf("foo.value: %d\n", f.value);
}
if (api->copy_from_symbol(object, "my_foo_ptr",
    1, &f, sizeof(f)) { // 1 = deref once
    printf("foo_ptr->value: %d\n", f.value);
}
```



## Making an address human readable

```
// Some address I plucked from the target
void *addr = ???;
// What is the closest symbol?
char buf[1024];
printf(“%p = %s\n”, addr, api->sym_name(addr, buf,
    sizeof(buf)));
// Gives:
0x00000035ad699fc3 = /lib64/libc-2.5.so`sleep+93
```



## File:line info

```
// if it is code, what's the file and line?  
int line;  
if (api->get_source_info(addr, buf, sizeof(buf),  
                        &line)) {  
    printf("%p => %s:%d\n", addr, buf, line);  
}
```



## Keep in mind

- Gimli modules are automated debuggers
- There is no direct access to pointers
- You need to copy the memory space from the target process before you can work with it naturally in C
- The target faulted; its pointers may be bogus and point to invalid memory
- The read routines will indicate an error or short read if the address is invalid or if the address range is only partially valid
- If you crash the glider, you may leave the target in limbo



## Future Directions

- Gimli modules for Lua, Perl etc. to render the script stack trace interspersed with the C stack trace
- Lua (and/or other script) bindings that leverage DWARF info and allow scripted analyzer modules
- Analysis across traces? Integration with a system like Mozilla Socorro?
- One monitor process for multiple children and/or “init” integration?



## Where can I get Gimli?

- Available from <https://bitbucket.org/wez/gimli/>
- OpenSource: 3-clause BSD license
- Interested in contributing?
  - Docs, code, testing, bug reports and test cases are all welcome!
  - Email me: [wez@messagesystems.com](mailto:wez@messagesystems.com)