More Modern Kernel Debugging Tools

BY TOM JONES

Panic is a truly wonderful word! It succinctly describes an incredibly complex emotional event. We can say "the soldiers panicked," and we know how a battle went. We can use it to give weight to a small oversight, and it explains what exactly went though our mind when we stepped onto a plane and discovered doubt about the current state of the oven.

Surely I turned it off.

It might just be my favourite term to see on the cover of a book: *"Panic! Unix System Crash Dump Analysis"*—wow! I've gotta have that. Any author using a title like that will have written a great book, even if it is an old Sun OS/Solaris technical manual.

Great book titles are more timeless than technical treatises, and the first material to expire is vendor-level documentation for existing systems. In 2024, I find it remarkably hard to find anything written recently about debugging approaches for operating systems. Looking at the works published, you wouldn't be blamed if you thought we'd perfected operating systems in 2004. I haven't ever used either SunOS or Solaris, but *"Panic!"* gave me the introduction to crash analysis I have always wanted.

I will admit I'd always wondered why people wanted core dumps so badly—what more can we really get than from a stack trace I would wonder. But from *"Panic!"* I was able to take my first real steps into practically doing crash analysis, and it took me on a journey that tried bleeding edge kernel debugging tools on FreeBSD. Let me show you what I learned. Don't worry, we won't have to wait around for lemon-soaked, paper napkins.

Getting a Kernel Dump

I'm going to be up front, I know you will not try to do kernel dump analysis until you really need to. When you have a hung machine. On that path lies a life of printf debugging.

Getting a core to play with isn't hard—your system needs to be set up to take crash dumps (see dumpon(8)) and then to enter the debugger. Normally, the system will help you by panicking, assisting you in your journey to reach the debugger. FreeBSD helpfully offers the ability to panic a kernel even when nothing has gone wrong yet. Setting debug.kdb.panic sysctl to 1 on a test system will drop you to a debug prompt:

```
# sysctl debug.kdb.panic=1
```

If you ran that on your desktop or a vital work machine in the cloud, you might be in a bit of trouble (and if it was your desktop, this article might have vanished). I would recommend learning about kernel debugging on a virtual machine, or at least something that won't cause too much trouble continually crashing.

Additionally, FreeBSD virtual machine images come configured by default to run **savecore** on boot and save out your crash dump file.

Once you set the debug.kdb.panic, you will be dropped to a ddb(4) prompt. ddb is a full fledged live system debugger—it can be a great analysis tool, but it isn't what we want today.

From ddb we can dump the running kernel with the dump command.

```
ddb> dump
Dumping 925 out of 16047 MB:..2%..11%..21%..32%..42%..51%..61%..71%..82%..92%
```

Panicking makes the system unusable, so you need to reboot to continue.

```
ddb> reboot
```

As your VM comes back up, there will be a message from **savecore** about extracting and saving your core file.

The core will be placed in **/var/crash** a long with some other files.

```
$ ls /var/crash
bounds          core.txt.0      info.0          info.last       minfree
vmcore.0        vmcore.last
```

The core file from our test is **vmcore.0**, and it comes with matching **info.0** and **core.txt.0**. The info file is a summary of the host and dump, and the **core.txt** is a summary of the dump file, any unread portions of the message buffer, and the panic string and stack trace if there is one.

```
Dump header from device: /dev/nvd0p3
  Architecture: amd64
  Architecture Version: 2
  Dump Length: 970199040
  Blocksize: 512
  Compression: none
  Dumptime: 2023-05-17 14:07:58 +0100
  Hostname: displacementactivity
  Magic: FreeBSD Kernel Dump
  Version String: FreeBSD 14.0-CURRENT #2 main-n261806-d3a49f62a284: Mon Mar 27 16:15:25
UTC 2023
    tj@displacementactivity:/usr/obj/usr/src/amd64.amd64/sys/GENERIC
  Panic String: Duplicate free of 0xfffff80339ef3000 from zone 0xfffffe001ec2ea00(mal-
loc-2048) slab 0xfffff80325789168(0)
  Dump Parity: 3958266970
  Bounds: 0
  Dump Status: good
```

The bounds file lets the dumper know the next coredump will be called **vmcore.1** and right now bounds on this machine:

```
# cat /var/crash/bounds
1
```

Finally, vmcore.last is a link to the most recent coredump file, in case you are having an interesting week and have lost track of the most recent crash.

Symbols

The second thing we need to along with the coredump are the kernel symbols. Kernel symbols for releases are available from the **`kernel-dbg`** package and are installed to `/usr/lib/debug/` or can be pulled out of your kernel build directory.

Looking at a Core with gdb (first)

First lets look at a core very quickly with **`kgdb`** to give ourselves a point of comparison for how far along lldb crash dump debugging is.

```
$ kgdb kernel.debug vmcore.0

Unread portion of the kernel message buffer:
panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432
cpuid = 2
time = 1706644478
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe0047d2f480
vpanic() at vpanic+0x132/frame 0xfffffe0047d2f5b0
panic() at panic+0x43/frame 0xfffffe0047d2f610
tcp_discardcb() at tcp_discardcb+0x25b/frame 0xfffffe0047d2f660
tcp_usr_detach() at tcp_usr_detach+0x51/frame 0xfffffe0047d2f680
sorele_locked() at sorele_locked+0xf7/frame 0xfffffe0047d2f6b0
tcp_close() at tcp_close+0x155/frame 0xfffffe0047d2f6e0
rack_check_data_after_close() at rack_check_data_after_close+0x8a/frame 0xfffffe0047d2f720
rack_do_fin_wait_1() at rack_do_fin_wait_1+0x141/frame 0xfffffe0047d2f7a0
rack_do_segment_nounlock() at rack_do_segment_nounlock+0x243b/frame 0xfffffe0047d2f9a0
rack_do_segment() at rack_do_segment+0xda/frame 0xfffffe0047d2fa00
tcp_input_with_port() at tcp_input_with_port+0x1157/frame 0xfffffe0047d2fb50
tcp_input() at tcp_input+0xb/frame 0xfffffe0047d2fb60
ip_input() at ip_input+0x2ab/frame 0xfffffe0047d2fbc0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fc20
ether_demux() at ether_demux+0x17a/frame 0xfffffe0047d2fc50
ether_nh_input() at ether_nh_input+0x39f/frame 0xfffffe0047d2fca0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fd00
ether_input() at ether_input+0xd9/frame 0xfffffe0047d2fd60
vtnet_rxq_eof() at vtnet_rxq_eof+0x73e/frame 0xfffffe0047d2fe20
vtnet_rx_vq_process() at vtnet_rx_vq_process+0x9c/frame 0xfffffe0047d2fe60
ithread_loop() at ithread_loop+0x266/frame 0xfffffe0047d2fef0
fork_exit() at fork_exit+0x82/frame 0xfffffe0047d2ff30
fork_trampoline() at fork_trampoline+0xe/frame 0xfffffe0047d2ff30
--- trap 0, rip = 0, rsp = 0, rbp = 0 ---
KDB: enter: panic

Reading symbols from /boot/kernel/zfs.ko...
Reading symbols from /usr/lib/debug//boot/kernel/zfs.ko.debug...
Reading symbols from /boot/kernel/tcp_rack.ko...
Reading symbols from /usr/lib/debug//boot/kernel/tcp_rack.ko.debug...
Reading symbols from /boot/kernel/tcphpts.ko...
Reading symbols from /usr/lib/debug//boot/kernel/tcphpts.ko.debug...
__curthread () at /usr/src/sys/amd64/include/pcpu_aux.h:57
57              __asm("movq %%gs:%P1,%0" : "=r" (td) : "n" (offsetof(struct pcpu,
(kgdb)
```

kgdb starts up by showing its license (removed), and then printing out the final parts of the message buffer which is a handy addition from **kgdb**. The final part of the message buffer tells us the panic message, info and a stack trace.

With the **kgdb bt** (backtrace) command, we can get a stack track, and with the frames command, we can move around the stack to see what was happening at the time of the panic.

```
(kgdb) bt
...
#10 0xffffffff80b51233 in vpanic (fmt=0xffffffff811f87ca "Assertion %s failed at %s:%d",
ap=ap@entry=0xfffffe0047d2f5f0) at /usr/src/sys/kern/kern_shutdown.c:953
#11 0xffffffff80b51013 in panic (fmt=0xffffffff81980420 <cnputs_mtx> "\371\023\025\201\377\
377\377\377") at /usr/src/sys/kern/kern_shutdown.c:889
#12 0xffffffff80d5483b in tcp_discardcb (tp=tp@entry=0xfffff80008584a80) at /usr/src/sys/
netinet/tcp_subr.c:2432
#13 0xffffffff80d60f71 in tcp_usr_detach (so=0xfffff800100b6b40) at /usr/src/sys/netinet/
tcp_usrreq.c:215
#14 0xffffffff80c01357 in sofree (so=0xfffff800100b6b40) at /usr/src/sys/kern/uipc_sock-
et.c:1209
#15 sorele_locked (so=so@entry=0xfffff800100b6b40) at /usr/src/sys/kern/uipc_socket.c:1236
#16 0xffffffff80d545b5 in tcp_close (tp=<optimized out>) at /usr/src/sys/netinet/tcp_
subr.c:2539
#17 0xffffffff82e37e0a in tcp_tv_to_usectick (sv=0xfffffe0047d2f698) at /usr/src/sys/neti-
net/tcp_hpts.h:177
#18 tcp_get_usecs (tv=0xfffffe0047d2f698) at /usr/src/sys/netinet/tcp_hpts.h:232
...
(kgdb) frame 12
#12 0xffffffff80d5483b in tcp_discardcb (tp=tp@entry=0xfffff80008584a80) at /usr/src/sys/
netinet/tcp_subr.c:2432
warning: Source file is more recent than executable.
2432
(kgdb) list
2427    #endif
2428
2429            CC_ALGO(tp) = NULL;
2430            if ((m = STAILQ_FIRST(&tp->t_inqueue)) != NULL) {
2431                    struct mbuf *prev;
2432
```

```
                STAILQ_INIT(&tp->t_inqueue);
                STAILQ_FOREACH_FROM_SAFE(m, &tp->t_inqueue, m_stailqpkt, prev)
                        m_freem(m);
        }
```

To review, I have listed the backtrace which led up to the panic, identified the call to panic around frame number **#11**, and asked **`kgdb`** to move to frame **#12** (the code which lead to the panic itself), then listed the code there. Further investigation from here would help us determine whatever led to the panic in this crash dump I had lying around.

These are the basic steps in kernel debugging, looking at what was going on and interrogating the crash dump to find out what values variables held. lldb needs to be able to do these tasks, for it to be useful in a kernel context.

lldb

FreeBSD has been moving to the more freeely licensed llvm/clang toolchain for the last decade. One missing piece for a while has been debugging, but in 2024 there are enough pieces in place that FreeBSD kernel debugging is possible with lldb.

In 2024 there are enough pieces in place that FreeBSD kernel debugging is possible with lldb.

lldb is able to import FreeBSD kernel dumps as core files and can move through stack frames.

lldb was developed by Apple. I remember when they changed the default debugger in gdb to lldb and I suffered severe culture shock. All of the debugging commands I had won from the cryptic documentation-less gnu world were gone, replaced with other weird commands.

lldb isn't really meant to be used as a command line interface, rather it is meant to be driven by software via an API. This shows in the verbosity of many commands. Thankfully, lldb has grown support for more gdb-like commands in its command line, meaning that more of the command interfaces match. Base commands such as printing now have compatible syntax, but many other options are different, and either better or much worse.

Poking Around with lldb

lldb doesn't need special configuration to analyze a kernel dump. Loading a crash dump in lldb is the same as kgdb just with the arguments swapped around a little:

```
$ lldb --core <corefile> path/to/kernel/symbols
```

For the examples that works out as:

```
$ lldb --core ../gdb/coredump/vmcore.0 ../gdb/coredump/kernel-debug/kernel.debug
(lldb) target create "../gdb/coredump/kernel-debug/kernel.debug" --core "../gdb/coredump/
vmcore.0"
Core file '/home/tj/code/scripts/gdb/coredump/vmcore.0' (x86_64) was loaded.
(lldb)
```

That is much quieter than the start up **`kgdb`**, which is nice, but it is also missing out on some important context from our crash dump. What exactly led to this being dumped?

kgdb isn't able to perform any magic (if so it would have a 'fix' command to match the 'break' command). All it is doing is looking for well-known symbols in the crash dump and printing them for us on start up.

We can do that ourselves.

First, the panic message in the kernel is stored in the string panicstr and is set by **vpanic** (in **kern/kern_shutdown.c**). We can easily extract this from the dump from lldb:

```
(lldb) p panicstr
(const char *) 0xffffffff819c1a00 "Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/neti-
net/tcp_subr.c:2432"
```

This might be enough for someone to start debugging. I also like stack traces which we can get with bt in lldb:

```
(lldb) bt
* thread #1, name = '(pid 1025) tcplog_dumper'
  * frame #0: 0xffffffff80b83d2a kernel.debug`sched_switch(td=0xfffff800174be740,
flags=259) at sched_ule.c:2297:26
    frame #1: 0xffffffff80b5e9e3 kernel.debug`mi_switch(flags=259) at kern_synch.c:546:2
    frame #2: 0xffffffff80bb0dc4 kernel.debug`sleepq_switch(wchan=0xffffffff817e1448,
pri=0) at subr_sleepqueue.c:607:2
    frame #3: 0xffffffff80bb11a6 kernel.debug`sleepq_catch_signals(wchan=0xffffffff817e1448,
pri=0) at subr_sleepqueue.c:523:3
    frame #4: 0xffffffff80bb0ef9 kernel.debug`sleepq_wait_sig(wchan=<unavailable>,
pri=<unavailable>) at subr_sleepqueue.c:670:11
    frame #5: 0xffffffff80b5df3c kernel.debug`_sleep(ident=0xffffffff817e1448,
lock=0xffffffff817e1428, priority=256, wmesg="tcplogdev", sbt=0, pr=0, flags=256) at kern_
synch.c:219:10
    frame #6: 0xffffffff8091190e kernel.debug`tcp_log_dev_read(dev=<unavailable>,
uio=0xfffffe0079b4ada0, flags=0) at tcp_log_dev.c:303:9
    frame #7: 0xffffffff809d99ce kernel.debug`devfs_read_f(fp=0xfffff80012857870,
uio=0xfffffe0079b4ada0, cred=<unavailable>, flags=0, td=0xfffff800174be740) at devfs_vn-
ops.c:1413:10
    frame #8: 0xffffffff80bc9bc6 kernel.debug`dofileread [inlined] fo_read(f-
p=0xfffff80012857870, uio=0xfffffe0079b4ada0, active_cred=<unavailable>, flags=<unavail-
able>, td=0xfffff800174be740) at file.h:340:10
    frame #9: 0xffffffff80bc9bb4 kernel.debug`dofileread(td=0xfffff800174be740, fd=3,
fp=0xfffff80012857870, auio=0xfffffe0079b4ada0, offset=-1, flags=0) at sys_generic.c:365:15
    frame #10: 0xffffffff80bc9712 kernel.debug`sys_read [inlined] kern_readv(td=0xfffff-
800174be740, fd=3, auio=0xfffffe0079b4ada0) at sys_generic.c:286:10
    frame #11: 0xffffffff80bc96dc kernel.debug`sys_read(td=0xfffff800174be740, ua-
p=<unavailable>) at sys_generic.c:202:10
    frame #12: 0xffffffff810556a3 kernel.debug`amd64_syscall [inlined] syscallenter(t-
d=0xfffff800174be740) at subr_syscall.c:186:11
    frame #13: 0xffffffff81055581 kernel.debug`amd64_syscall(td=0xfffff800174be740,
traced=0) at trap.c:1192:2
    frame #14: 0xffffffff8102781b kernel.debug`fast_syscall_common at exception.S:578
```

We can pick an interesting frame to look at from lldb too:

```
(lldb) frame select 12
frame #12: 0xffffffff810556a3 kernel.debug`amd64_syscall [inlined] syscallenter(td=0xfffff-
800174be740) at subr_syscall.c:186:11
   183                     if (!sy_thr_static)
   184                             syscall_thread_exit(td, se);
   185             } else {
-> 186                     error = (se->sy_call)(td, sa->args);
   187                     /* Save the latest error return value. */
   188                     if (__predict_false((td->td_pflags & TDP_NERRNO) != 0))
   189                             td->td_pflags &= ~TDP_NERRNO;
```

Getting the Kernel Buffer

Printing stuff and moving around the stack is most of what we need for kernel crash dump debugging. The start-up message from gdb is quite nice though, showing us the last part of the kernel message buffer as if it had come directly off the local console.

lldb doesn't yet offer a nice start-up command like that. Thankfully, "Panic!" gives us some hints as to how we might pull out this information ourselves. "Panic!" uses a macro called "**`msgbuf`**" to print the kernel message buffer from a **`struct msgbuf`**.

Some poking in the FreeBSD source, and we have something similar available:

```
(lldb) p *msgbufp
(msgbuf) {
  msg_ptr = 0xfffff8001ffe8000 "---<<BOOT>>---\nCopyright (c) 1992-2023 The FreeBSD
Project.\nCopyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994\n\
tThe Regents of the University of California. All rights reserved.\nFreeBSD is a reg-
istered trademark of The FreeBSD Foundation.\nFreeBSD 15.0-CURRENT #0 main-272a40604:
Wed Nov 29 13:42:38 UTC 2023\n    tj@vpp:/usr/obj/usr/src/amd64.amd64/sys/GENERIC amd64\
nFreeBSD clang version 16.0.6 (https://github.com/llvm/llvm-project.git llvmorg-16.0.6-
0-g7cbf1a259152)\nWARNING: WITNESS option enabled, expect reduced performance.\nVT:
init without driver.\nCPU: 12th Gen Intel(R) Core(TM) i7-1260P (2500.00-MHz K8-class
CPU)\n  Origin=\"GenuineIntel\"  Id=0x906a3  Family=0x6  Model=0x9a  Stepping=3\n  Fea-
tures=0x9f83fbff<FPU,VME,DE,PSE,TSC,MSR,PAE,MCE,CX8,APIC,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE36,M-
MX,FXSR,SSE,SSE2,SS,HTT,PBE>\n  Features2=0xfeda7a17<SSE3,PCLMULQDQ,DTES64,DS_CPL,SSSE3,SD-
BG,FMA,CX16,xTPR,PCID,SSE4.1,SSE4.2,MOVBE,POPCNT,AESNI,XSAVE,OSXSAVE,AVX,F16C,RDRAND,HV>\n
AMD Features=0x2c100800<SYSCALL,"...
  msg_magic = 405602
  msg_size = 98232
  msg_wseq = 16777
  msg_rseq = 15001
  msg_cksum = 1421737
  msg_seqmod = 1571712
  msg_lastpri = -1
  msg_flags = 0
  msg_lock = {
    lock_object = {
      lo_name = 0xffffffff81230bcc "msgbuf"
      lo_flags = 196608
      lo_data = 0
      lo_witness = NULL
    }
    mtx_lock = 0
  }
}
```

We have a struct **msgbuf** globally visible in the kernel implementing the kernel's message **buffer. lldb** shows us the start of the buffer. The fields **msg_wseq** and **msg_rseq** tell us where we have written to and where we have read from.

Reading out the unread portion of the message buffer is easy:

```
(lldb) p msgbufp->msg_ptr+msgbufp->msg_rseq
(char *) 0xfffff8001ffeba99 "panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/ne-
tinet/tcp_subr.c:2432\ncpuid = 2\ntime = 1706644478\nKDB: stack backtrace:\ndb_trace_self_
wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe0047d2f480\nvpanic() at vpanic+0x132/
frame 0xfffffe0047d2f5b0\npanic() at panic+0x43/frame 0xfffffe0047d2f610\ntcp_discardcb()
at tcp_discardcb+0x25b/frame 0xfffffe0047d2f660\ntcp_usr_detach() at tcp_usr_detach+0x51/
frame 0xfffffe0047d2f680\nsorele_locked() at sorele_locked+0xf7/frame 0xfffffe0047d2f6b0\
ntcp_close() at tcp_close+0x155/frame 0xfffffe0047d2f6e0\nrack_check_data_after_close() at
rack_check_data_after_close+0x8a/frame 0xfffffe0047d2f720\nrack_do_fin_wait_1() at rack_
do_fin_wait_1+0x141/frame 0xfffffe0047d2f7a0\nrack_do_segment_nounlock() at rack_do_seg-
ment_nounlock+0x243b/frame 0xfffffe0047d2f9a0\nrack_do_segment() at rack_do_segment+0x-
da/frame 0xfffffe0047d2fa00\ntcp_input_with_port() at tcp_input_with_port+0x1157/frame
0xfffffe0047d2fb50\ntcp_input() at tcp_input+0xb/frame 0xfffffe0047d2fb60\nip_input() at
ip_in"...
```

The output isn't formatted in a very friendly way, control characters are just printed out, but we can read out the kernel message buffer. The output is truncated before the full backtrace is available. Let's try some other commands:

```
(lldb) x/b msgbufp->msg_ptr+msgbufp->msg_rseq
0xfffff8001ffeba99: "panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/
tcp_subr.c:2432\ncpuid = 2\ntime = 1706644478\nKDB: stack backtrace:\ndb_trace_self_wrap-
per() at db_trace_self_wrapper+0x2b/frame 0xfffffe0047d2f480\nvpanic() at vpanic+0x132/
frame 0xfffffe0047d2f5b0\npanic() at panic+0x43/frame 0xfffffe0047d2f610\ntcp_discardcb()
at tcp_discardcb+0x25b/frame 0xfffffe0047d2f660\ntcp_usr_detach() at tcp_usr_detach+0x51/
frame 0xfffffe0047d2f680\nsorele_locked() at sorele_locked+0xf7/frame 0xfffffe0047d2f6b0\
ntcp_close() at tcp_close+0x155/frame 0xfffffe0047d2f6e0\nrack_check_data_after_close() at
rack_check_data_after_close+0x8a/frame 0xfffffe0047d2f720\nrack_do_fin_wait_1() at rack_
do_fin_wait_1+0x141/frame 0xfffffe0047d2f7a0\nrack_do_segment_nounlock() at rack_do_seg-
ment_nounlock+0x243b/frame 0xfffffe0047d2f9a0\nrack_do_segment() at rack_do_segment+0x-
da/frame 0xfffffe0047d2fa00\ntcp_input_with_port() at tcp_input_with_port+0x1157/frame
0xfffffe0047d2fb50\ntcp_input() at tcp_input+0xb/frame 0xfffffe0047d2fb60\nip_input() at
ip_i"
warning: unable to find a NULL terminated string at 0xfffff8001ffeba99. Consider increasing
```

```
the maximum read length.
(lldb) x/2048b msgbufp->msg_ptr+msgbufp->msg_rseq
error: Normally, 'memory read' will not read over 1024 bytes of data.
error: Please use --force to override this restriction just once.
error: or set target.max-memory-read-size if you will often need a larger limit.
```

We hit a printing limit, and try as I might, I can't convince lldb to go further. Time for a higher tool.

Some Help From the Moon

lldb also offers a scripting interface for control, which explains why many of the commands are incredibly verbose to type out. Currently, lldb supports scripting with C++, Python and has experimental support for Lua. FreeBSD ships Lua in base, and the FreeBSD builds of lldb in 2024 include Lua support by default.

We can simply try this out with the following:

```
(lldb) script
>>> print("hello esteemed FreeBSD Journal readers!")
hello esteemed FreeBSD Journal readers!
>>> quit
```

With the >>> prompt indicating that we have moved into the Lua interpreter.

From "Panic!" we learn that adb the SunOS/Solaris debugger had a handy and easy-to-understand macro for finding and printing the message buffer:

```
msgbuf/"magic"16t"size"16t"bufx"16t"bufr"n4X
+,(*msgbuf+0t8)-*(msgbuf+0t12)))&80000000$<msgbuf.wrap
.+*(msgbuf+0t12),(*(msgbuf+0t8)-*(msfbuf+0t12))/c
```

Implementing a similar mechanism with Lua should be no problem at all with that as an example.

The lldb lua interface is generated from swig bindings, this is a C++ format for describing interfaces between libraries. The Python and Lua bindings are generated the same way. For any questions you have about the API or how to use it, you can figure it out from working from the Python API documentation which is available from the lldb project. This is a very clunky way to do things, but it is possible.

I quickly get sick of running commands in the interpreter and, considering the length of some of them, they can be annoying to try. lldb can load your Lua script from a file once the interpreter has been run once. From a fresh session:

```
$ lldb --core coredump/vmcore.1 coredump/kernel-debug/kernel.debug
(lldb) target create "coredump/kernel-debug/kernel.debug" --core "coredump/vmcore.1"
Core file '/home/tj/code/scripts/gdb/coredump/vmcore.1' (x86_64) was loaded.
(lldb) script
>>> print("hello")
hello
>>> quit
(lldb) command script import ./hello.lua
hello from the script hello.lua
```

Assuming the file **`hello.lua`** contains:

```
print("hello from the script hello.lua")
```

The lldb Lua environment provides a **`lldb`** variable with members enabling access to the target, debugger, frame, process, and thread. These objects map to ones described in the Python API.

I'm not really a fan of the lldb api, it can be quite clunky to write and difficult to understand if you are having a problem with your choice of function or how variables are laid out in memory.

Once you have some experience, it gets easier to understand what it wants from you.

Let me illustrate how to use the lldb Lua bindings with an example of printing out the message buffer from a crash dump.

From the lldb Lua variable we can access files in the dump image. A big block for me when I first started doing core dump analysis was understanding how to locate things in memory. There are various kernel global variables that you can access as starting points, and most subsystems have something you can build from.

As we saw before, **`msgbufp`** is a global instance of the kernels message buffer. From lldb Lua we can access this with:

```
msgbuf = lldb.target:FindFirstGlobalVariable("msgbufp")
```

This gives us an instance of a SBValue representing this instance of the struct in the memory from the core dump. We can access the child members of the struct with the **`GetChildMemberWithName`** method and a name such as **`msg_rseq`**.

The **`lldb.process`** object gives us the ability to read out memory from our kernel dump. Sometimes it can take a bit of juggling to get together the correct references, addresses and values to perform the operations you want.

With these methods, we can assemble a point to the start of the message buffer, read it out of the core dump, and print it using Lua. I've put all of this into a script called **`msgbuf.lua`**:

```
msgbuf = lldb.target:FindFirstGlobalVariable("msgbufp")

msgbuf_start = msgbuf:GetChildMemberWithName("msg_rseq"):GetValue()
msgbuf_end = msgbuf:GetChildMemberWithName("msg_wseq"):GetValue()
unread_len = msgbuf_end - msgbuf_start

msgbuf_addr = msgbuf:GetChildMemberWithName("msg_ptr")
                   :Dereference()
                   :GetLoadAddress() + msgbuf_start
msgbuf_ptr = lldb.process:ReadMemory(msgbuf_addr, unread_len, lldb.SBError())

print("Unread portion of the kernel message buffer:")
print(msgbuf_ptr)
```

If we run this from our lldb session we get the following output:

```
(lldb) command script import ./msgbuf.lua
Unread portion of the kernel message buffer:
panic: Assertion !tcp_in_hpts(tp) failed at /usr/src/sys/netinet/tcp_subr.c:2432
cpuid = 2
time = 1706644478
KDB: stack backtrace:
db_trace_self_wrapper() at db_trace_self_wrapper+0x2b/frame 0xfffffe0047d2f480
vpanic() at vpanic+0x132/frame 0xfffffe0047d2f5b0
panic() at panic+0x43/frame 0xfffffe0047d2f610
tcp_discardcb() at tcp_discardcb+0x25b/frame 0xfffffe0047d2f660
tcp_usr_detach() at tcp_usr_detach+0x51/frame 0xfffffe0047d2f680
sorele_locked() at sorele_locked+0xf7/frame 0xfffffe0047d2f6b0
tcp_close() at tcp_close+0x155/frame 0xfffffe0047d2f6e0
rack_check_data_after_close() at rack_check_data_after_close+0x8a/frame 0xfffffe0047d2f720
rack_do_fin_wait_1() at rack_do_fin_wait_1+0x141/frame 0xfffffe0047d2f7a0
rack_do_segment_nounlock() at rack_do_segment_nounlock+0x243b/frame 0xfffffe0047d2f9a0
rack_do_segment() at rack_do_segment+0xda/frame 0xfffffe0047d2fa00
tcp_input_with_port() at tcp_input_with_port+0x1157/frame 0xfffffe0047d2fb50
tcp_input() at tcp_input+0xb/frame 0xfffffe0047d2fb60
ip_input() at ip_input+0x2ab/frame 0xfffffe0047d2fbc0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fc20
ether_demux() at ether_demux+0x17a/frame 0xfffffe0047d2fc50
ether_nh_input() at ether_nh_input+0x39f/frame 0xfffffe0047d2fca0
netisr_dispatch_src() at netisr_dispatch_src+0xad/frame 0xfffffe0047d2fd00
ether_input() at ether_input+0xd9/frame 0xfffffe0047d2fd60
vtnet_rxq_eof() at vtnet_rxq_eof+0x73e/frame 0xfffffe0047d2fe20
vtnet_rx_vq_process() at vtnet_rx_vq_process+0x9c/frame 0xfffffe0047d2fe60
ithread_loop() at ithread_loop+0x266/frame 0xfffffe0047d2fef0
fork_exit() at fork_exit+0x82/frame 0xfffffe0047d2ff30
fork_trampoline() at fork_trampoline+0xe/frame 0xfffffe0047d2ff30
--- trap 0, rip = 0, rsp = 0, rbp = 0 ---
KDB: enter: panic
```

Lua has kindly expanded the control characters in the buffer for us giving us nice formatted output from the message buffer.

Better Debugging Possibilities

lldb is new on the block when it comes to kernel debugging and there are still many features not available in the Lua environment, but it has enough functionality to be a useful tool. Old timer gdb users might be struggling to see the value of these examples, after all lldb adds a much more complicated syntax and it might seem like change for changes sake.

A big value that lldb and its built-in lua brings is shipping in the release FreeBSD images. lldb Lua is freely licensed and is compatible with FreeBSD, and from the start of 2024, it was enabled by default in CURRENT builds. This allows kernel developers and trouble shooters to write scripts in lldb Lua and provide them to users for analysis.

kgdb has had support for gdb script for a long time, but it isn't the most pleasant scripting language to program. Lua, on the other hand, while a little weird, is commonly used in many environments and is part of the FreeBSD boot loader. I have written a tool to extract TCP log files from crashed kernel images--the major headaches were figuring out how to get the memory. Once I had the data, creating and writing these to files was an easy job.

Kernel dumps have everything in them and can contain sensitive information. A reasonable scripting language makes it possible for developers to provide scripts to extract further debugging information from a kernel image without the need to move around large core dumps, and without needing to handle the worries of trusting a stranger with possibly sensitive information.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.