IBM Research

Debugging the Linux kernel with the Crash Utility

Lucas Villa Real
IBM Almaden Research Center
Oct 1st, 2009

About Crash

- **The Crash Utility is a tool which integrates with GDB to inspect live Linux kernels and kernel core dump files (kdump)**
- **Provides a couple of useful features**
  - Access to the kernel log buffer
  - Display kernel stack traces of all processes
  - Source code disassembly
  - Formatted kernel data structures and variable displays
  - Virtual memory data
  - Dump of linked lists
  - Translation of addresses to symbols and vice-versa

When to use it

- When the kernel oopses, but stays running
  - Live inspection
- When curious about what the state of some component is
  - Live inspection
- When the machine hangs
  - NMI button → kdump → post mortem analysis
- When someone emails you telling that they've got a kdump for you!
  - Post mortem analysis

Invoking Crash

- On a live system
  - crash /path/to/vmlinux
- On a kdump file
  - crash /path/to/vmlinux /path/to/vmlinux.debug vmcore_file

Using Crash

```
      KERNEL: /boot/vmlinux-2.6.16.46-360d-smp
   DEBUGINFO: /usr/lib/debug/boot/vmlinux-2.6.16.46-360d-smp.debug
    DUMPFILE: vmcore
        CPUS: 4
        DATE: Sat Jul 18 13:22:15 2009
      UPTIME: 00:17:27
LOAD AVERAGE: 2.96, 3.08, 1.99
       TASKS: 454
    NODENAME: ianode1
     RELEASE: 2.6.16.46-360d-smp
     VERSION: #1 SMP Thu Jul 16 21:19:51 UTC 2009
     MACHINE: x86_64  (2992 Mhz)
      MEMORY: 7.9 GB
       PANIC: "Oops: 0000 [1] SMP " (check log for details)
         PID: 9545
     COMMAND: "mmnfsnodeback"
        TASK: ffff81017ea5e840  [THREAD_INFO: ffff810185c88000]
         CPU: 1
       STATE: TASK_RUNNING (PANIC)

crash>
```

Using Crash: *backtrace*

```
crash> bt

PID: 9545   TASK: ffff81017ea5e840  CPU: 1   COMMAND: "mmnfsnodeback"
 #0 [ffff810185c89b20] crash_kexec at ffffffff801516c3
 #1 [ffff810185c89ba8] flush_cpu_workqueue at ffffffff801422a3
 #2 [ffff810185c89be0] __die at ffffffff802f5854
 #3 [ffff810185c89c20] do_page_fault at ffffffff802f70fa
 #4 [ffff810185c89c50] printk at ffffffff80132cd3
 #5 [ffff810185c89d10] error_exit at ffffffff8010bc15
    [exception RIP: flush_cpu_workqueue+13]
    RIP: ffffffff801422a3  RSP: ffff810185c89dc8  RFLAGS: 00010282
    RAX: ffff810235430c00  RBX: 0000000000000000  RCX: 0000000000000005
    RDX: 0000000000000000  RSI: 00000000ffffffff  RDI: 0000000000000000
    RBP: 00000000ffffffff   R8: ffffffff80465260   R9: 0000000000000020
    R10: 0000000000000000  R11: ffff810185c89f50  R12: 0000000000000020
    R13: ffff810185c89f50  R14: 0000000000000000  R15: 00002af96d3a2aa0
    ORIG_RAX: ffffffffffffffff  CS: 0010  SS: 0018
...
```

Using Crash: *backtrace*

```
...
 #6 [ffff810185c89dd0] try_to_del_timer_sync at ffffffff8013af61
 #7 [ffff810185c89df0] flush_workqueue at ffffffff801423b1
 #8 [ffff810185c89e10] cancel_rearming_delayed_workqueue at ffffffff80142470
 #9 [ffff810185c89e30] nfs4_state_shutdown at ffffffff8852a197
#10 [ffff810185c89e60] nfsd_svc at ffffffff88512472
#11 [ffff810185c89e80] write_threads at ffffffff88512fba
#12 [ffff810185c89eb0] get_zeroed_page at ffffffff80163a19
#13 [ffff810185c89ec0] simple_transaction_get at ffffffff801a1ef3
#14 [ffff810185c89ef0] nfsctl_transaction_write at ffffffff88512d08
#15 [ffff810185c89f10] vfs_write at ffffffff80182943
#16 [ffff810185c89f40] sys_write at ffffffff80182f0a
#17 [ffff810185c89f80] tracesys at ffffffff8010aeb1
    RIP: 00002af96d223400  RSP: 00007fff3e245828  RFLAGS: 00000246
    RAX: ffffffffffffffda  RBX: ffffffff8010aeb1  RCX: ffffffffffffffff
    RDX: 0000000000000003  RSI: 000000000058bc20  RDI: 0000000000000001
    RBP: 00000000ffffffff   R8: 000000000058eec1   R9: 000000000058eee0
    R10: 00002af96cb2b2ef  R11: 0000000000000246  R12: 0000000000000000
    R13: 00002af96d02ca00  R14: 00002af96d02cb80  R15: 000000006d3a2aa0
    ORIG_RAX: 0000000000000001  CS: 0033  SS: 002b

crash>
```

Using Crash: *dmesg*

```
crash> dmesg
...
nfsd: last server has exited
RPC: failed to contact portmap (errno -5).
RPC: failed to contact portmap (errno -5).
RPC: failed to contact portmap (errno -5).
Unable to handle kernel NULL pointer dereference at 0000000000000060 RIP:
<ffffffff801422a3>{flush_cpu_workqueue+13}
PGD 1915b9067 PUD 1947f5067 PMD 0
Oops: 0000 [1] SMP
last sysfs file: /devices/pci0000:00/0000:00:02.0/0000:1a:00.0/0000:1b:00.0/0000:1c:00.0/host3/rport-
3:0-1/target3:0:0/3:0:0:2/rev
CPU 1
Modules linked in: af_packet nfsd exportfs lockd nfs_acl sunrpc xt_tcpudp iptable_filter ip_tables
x_tables mmfs mmfslinux tracedev ipv6 bonding fuse_2_7_3 loop dm_r
dac dm_round_robin dm_multipath dm_mod qla2xxx hw_random firmware_class shpchp pci_hotplug
scsi_transport_fc i2c_i801 e1000 bnx2 i2c_core piix ata_piix libata ehci_h
cd sg uhci_hcd usbhid usbcore aacraid mptsas scsi_transport_sas mptspi mptscsih mptbase
scsi_transport_spi sr_mod cdrom sd_mod scsi_mod ide_disk ide_core
...
```

Using Crash: *dmesg*

```
...
Pid: 9545, comm: mmnfsnodeback Tainted: GF    U 2.6.16.46-360d-smp #1
RIP: 0010:[<ffffffff801422a3>] <ffffffff801422a3>{flush_cpu_workqueue+13}
RSP: 0018:ffff810185c89dc8  EFLAGS: 00010282
RAX: ffff810235430c00 RBX: 0000000000000000 RCX: 0000000000000005
RDX: 0000000000000000 RSI: 00000000ffffffff RDI: 0000000000000000
RBP: 00000000ffffffff R08: ffffffff80465260 R09: 0000000000000020
R10: 0000000000000000 R11: ffff810185c89f50 R12: 0000000000000020
R13: ffff810185c89f50 R14: 0000000000000000 R15: 00002af96d3a2aa0
FS:  00002af96d3a2b00(0000) GS:ffff81023803a6c0(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: 0000000000000060 CR3: 00000001b6212000 CR4: 00000000000006e0
Process mmnfsnodeback (pid: 9545, threadinfo ffff810185c88000, task ffff81017ea5e840)
Stack: 0000000000000282 ffffffff8013af61 ffffffff885444d0 0000000000000002
       ffff810234e942a0 ffffffff801423b1 ffffffff8853ff80 ffff810234e942a0
       0000000000000020 ffffffff80142470
Call Trace: <ffffffff8013af61>{try_to_del_timer_sync+81}
       <ffffffff801423b1>{flush_workqueue+14} <ffffffff80142470>{cancel_rearming_delayed_workqueue+42}
       <ffffffff8852a197>{:nfsd:nfs4_state_shutdown+27} <ffffffff88512472>{:nfsd:nfsd_svc+712}
       <ffffffff88512f4b>{:nfsd:write_threads+0} <ffffffff88512fba>{:nfsd:write_threads+111}
       <ffffffff80163a19>{get_zeroed_page+48} <ffffffff801a1ef3>{simple_transaction_get+139}
       <ffffffff88512f4b>{:nfsd:write_threads+0} <ffffffff88512d08>{:nfsd:nfsctl_transaction_write+66}
       <ffffffff80182943>{vfs_write+215} <ffffffff80182f0a>{sys_write+69}
       <ffffffff8010aeb1>{tracesys+209}

Code: 48 8b 57 60 65 48 8b 04 25 00 00 00 00 48 39 c2 75 33 ff c5
RIP <ffffffff801422a3>{flush_cpu_workqueue+13} RSP <ffff810185c89dc8>

crash>
```

Using Crash: *gdb <command>*

```
crash> gdb list *flush_cpu_workqueue+13
or
crash> gdb list *0xffffffff801422a3

0xffffffff801422a3 is in flush_cpu_workqueue (kernel/workqueue.c:234).
229      * If cpu == -1 it's a single-threaded workqueue and the caller does not hold
230      * cpu_hotplug lock
231      */
232     static void flush_cpu_workqueue(struct cpu_workqueue_struct *cwq, int cpu)
233     {
234             if (cwq->thread == current) {
235                     /*
236                      * Probably keventd trying to flush its own queue. So simply
run
237                      * it by hand rather than deadlocking.
238                      */
```

Using Crash: *gdb <command>*

```
crash> gdb list *flush_cpu_workqueue+13

0xffffffff801422a3 is in flush_cpu_workqueue (kernel/workqueue.c:234).
229      * If cpu == -1 it's a single-threaded workqueue and the caller does not hold
230      * cpu_hotplug lock
231      */
232     static void flush_cpu_workqueue(struct cpu_workqueue_struct *cwq, int cpu)
233     {
234             if (cwq->thread == current) {
235                     /*
236                      * Probably keventd trying to flush its own queue. So simply
run
237                      * it by hand rather than deadlocking.
238                      */
```

NULL pointer dereference must come from cwq == NULL.

Next step is to verify who calls flush_cpu_workqueue() and why cwq is NULL.

Using Crash: *back to dmesg...*

```
Pid: 9545, comm: mmnfsnodeback Tainted: GF    U 2.6.16.46-360d-smp #1
RIP: 0010:[<ffffffff801422a3>] <ffffffff801422a3>{flush_cpu_workqueue+13}
RSP: 0018:ffff810185c89dc8  EFLAGS: 00010282
RAX: ffff810235430c00 RBX: 0000000000000000 RCX: 0000000000000005
RDX: 0000000000000000 RSI: 00000000ffffffff RDI: 0000000000000000
RBP: 00000000ffffffff R08: ffffffff80465260 R09: 0000000000000020
R10: 0000000000000000 R11: ffff810185c89f50 R12: 0000000000000020
R13: ffff810185c89f50 R14: 0000000000000000 R15: 00002af96d3a2aa0
FS:  00002af96d3a2b00(0000) GS:ffff81023803a6c0(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 000000008005003b
CR2: 0000000000000060 CR3: 00000001b6212000 CR4: 00000000000006e0
Process mmnfsnodeback (pid: 9545, threadinfo ffff810185c88000, task ffff81017ea5e840)
Stack: 0000000000000282 ffffffff8013af61 ffffffff885444d0 0000000000000002
       ffff810234e942a0 ffffffff801423b1 ffffffff8853ff80 ffff810234e942a0
       0000000000000020 ffffffff80142470
Call Trace: <ffffffff8013af61>{try_to_del_timer_sync+81}
       <ffffffff801423b1>{flush_workqueue+14}
<ffffffff80142470>{cancel_rearming_delayed_workqueue+42}
       <ffffffff8852a197>{:nfsd:nfs4_state_shutdown+27} <ffffffff88512472>{:nfsd:nfsd_svc+712}
       <ffffffff88512f4b>{:nfsd:write_threads+0} <ffffffff88512fba>{:nfsd:write_threads+111}
       <ffffffff80163a19>{get_zeroed_page+48} <ffffffff801a1ef3>{simple_transaction_get+139}
       <ffffffff88512f4b>{:nfsd:write_threads+0}
<ffffffff88512d08>{:nfsd:nfsctl_transaction_write+66}
       <ffffffff80182943>{vfs_write+215} <ffffffff80182f0a>{sys_write+69}
       <ffffffff8010aeb1>{tracesys+209}

Code: 48 8b 57 60 65 48 8b 04 25 00 00 00 00 48 39 c2 75 33 ff c5
RIP <ffffffff801422a3>{flush_cpu_workqueue+13} RSP <ffff810185c89dc8>
```

Using Crash: *gdb <command>*

```
crash> gdb list *cancel_rearming_delayed_workqueue+42

0xffffffff80142470 is in cancel_rearming_delayed_workqueue (kernel/workqueue.c:479).
474      */
475     void cancel_rearming_delayed_workqueue(struct workqueue_struct *wq,
476                                            struct work_struct *work)
477     {
478             while (!cancel_delayed_work(work))
479                     flush_workqueue(wq);
480     }
481     EXPORT_SYMBOL(cancel_rearming_delayed_workqueue);
482
483     /**
```

Problem: NFS functions weren't loaded by kdump, as they are provided by a kernel module, nfsd.ko

```
crash> gdb list *nfs4_state_shutdown+27
No symbol "nfs4_state_shutdown" in current context.
```

Using Crash: *loading external modules*

```
crash> mod -s nfsd /lib/modules/2.6.16.46-360d-smp/kernel/fs/nfsd/nfsd.ko
     MODULE       NAME    SIZE OBJECT FILE
ffffffff88540280  nfsd  269704 /lib/modules/2.6.16.46-360d-smp/kernel/fs/nfsd/nfsd.ko
```

Once the module is loaded we can proceed debugging that function:

```
crash> gdb list *nfs4_state_shutdown+27
0xffffffff8852a197 is in nfs4_state_shutdown (fs/nfsd/nfs4state.c:3261).
3256
3257    void
3258    nfs4_state_shutdown(void)
3259    {
3260            cancel_rearming_delayed_workqueue(laundry_wq, &laundromat_work);
3261            destroy_workqueue(laundry_wq);
3262            nfs4_lock_state();
3263            nfs4_release_reclaim();
3264            __nfs4_state_shutdown();
3265            nfsd4_free_slabs();
```

Using Crash: *disassembling code to find where a variable lives*

```
crash> print laundry_wq
Cannot access memory at address 0x8840
```

We need to disassemble nfs4_state_shutdown() to find the address where laundry_wq lives.

```
crash> dis -r nfs4_state_shutdown+27
0xffffffff8852a17c <nfs4_state_shutdown>:       push   %rbp
0xffffffff8852a17d <nfs4_state_shutdown+1>:     mov    $0xffffffff8853ff80,%rsi
0xffffffff8852a184 <nfs4_state_shutdown+8>:     push   %rbx
0xffffffff8852a185 <nfs4_state_shutdown+9>:     xor    %ebx,%ebx
0xffffffff8852a187 <nfs4_state_shutdown+11>:    sub    $0x18,%rsp
0xffffffff8852a18b <nfs4_state_shutdown+15>:    mov    142126(%rip),%rdi        #
0xffffffff8854ccc0
0xffffffff8852a192 <nfs4_state_shutdown+22>:    callq  0xffffffff80142446
<cancel_rearming_delayed_workqueue>
0xffffffff8852a197 <nfs4_state_shutdown+27>:    mov    142114(%rip),%rdi        #
0xffffffff8854ccc0

crash> sym 0xffffffff8854ccc0
ffffffff8854ccc0 (b) laundry_wq
```

Using Crash: *inspecting data structures*

```
crash> whatis laundry_wq
struct workqueue_struct *laundry_wq;

crash> struct workqueue_struct
struct workqueue_struct {
    struct cpu_workqueue_struct *cpu_wq;
    const char *name;
    struct list_head list;
}
SIZE: 32

crash> struct workqueue_struct 0xffffffff8854ccc0
struct workqueue_struct {
  cpu_wq = 0xffff810234e942a0,
  name = 0x0,                          <----- oops!
  list = {
    next = 0x0,                        <----- this looks...
    prev = 0x0                         <----- ...very wrong!
  }
}

crash> struct cpu_workqueue_struct 0xffff810234e942a0
struct cpu_workqueue_struct {
  lock = {
    raw_lock = {
      slock = 3401380863               <----- ewwww!
    }
  },
  remove_sequence = -2007820453,       <----- definitely corrupted!
  ...
```

Using Crash: *inspecting data structures*

- So far, we found that:
  - Crash gives us access to the kernel log / stack trace
  - We can inspect specific functions in the context of the task that crashed
  - It's possible to get the contents of a specific variable, although it's not always as easy as one would expect it to be
- Now what?
  - We have a data structure corrupted and we have some hints about where to start looking in the code
  - Crash doesn't do magic – we would need to dig the source for more answers now :-)

Using Crash: *other useful features*

- The Crash Utility offers some other very handful features:
  - Inspecting any other tasks' state
    - List of open files
    - Reading non-structured data from memory
  - Getting kernel memory information (SLABs)

Using Crash: *inspecting other tasks' state*

```
crash> ps | grep auditd
   4110      1   1  ffff810235954040  IN   0.0   14468   1376  auditd
   4138      1   1  ffff810232729780  IN   0.0   14468   1376  auditd
   4140     15   0  ffff810237842780  IN   0.0       0      0  [kauditd]

crash> set 4110
    PID: 4110
COMMAND: "auditd"
   TASK: ffff810235954040  [THREAD_INFO: ffff810231dc0000]
    CPU: 1
  STATE: TASK_INTERRUPTIBLE

crash> files
PID: 4110   TASK: ffff810235954040  CPU: 1   COMMAND: "auditd"
ROOT: /    CWD: /
 FD       FILE            DENTRY           INODE       TYPE PATH
  0 ffff810233b16b80 ffff810237628e50 ffff8102377d5730 CHR  /dev/null
  1 ffff810233b16b80 ffff810237628e50 ffff8102377d5730 CHR  /dev/null
  2 ffff810233b16b80 ffff810237628e50 ffff8102377d5730 CHR  /dev/null
  3 ffff810232a631c0 ffff810234805220 ffff81023436ea10 SOCK socket:/[12736]
  4 ffff8102336718c0 ffff810234896e50 ffff810235d13758 REG  /var/log/audit/audit.log
  5 ffff81023428fec0 ffff8102339e5a40 ffff81023436e1d0 SOCK socket:/[12740]
  6 ffff8102349e7cc0 ffff8101b9eeebe0 ffff8101b8c5a040 REG
/tiam/col1/utility/raw_logs/172.31.1.1.rawlog
```

Using Crash: *reading structured and non-structured data*

```
crash> struct dentry.d_inode,d_name ffff8101b9eeebe0
  d_inode = 0xffff8101b8c5a040,
  d_name = {
    hash = 2631539605,
    len = 17,
    name = 0xffff8101b9eeec8c "172.31.1.1.rawlog"
  },

crash> struct dentry.d_name ffff8101b9eeebe0
  d_name = {
    hash = 2631539605,
    len = 17,
    name = 0xffff8101b9eeec8c "172.31.1.1.rawlog"
  },

crash> rd -8 0xffff8101b9eeec8c 20
ffff8101b9eeec8c:  31 37 32 2e 33 31 2e 31 2e 31 2e 72 61 77 6c 6f   172.31.1.1.rawlo
ffff8101b9eeec9c:  67 00 00 00                                       g...

crash> rd -64 0xffff8101b9eeec8c 4
ffff8101b9eeec8c:  312e31332e323731 6f6c7761722e312e   172.31.1.1.rawlo
ffff8101b9eeec9c:  0000000000000067 0000000000000000   g...............
```

Using Crash: *getting memory information*

```
crash> kmem -s
CACHE            NAME                 OBJSIZE  ALLOCATED     TOTAL  SLABS  SSIZE
ffff8101ca51b1c0 rpc_buffers             2048          8         8      4     4k
ffff8101ca51b840 rpc_tasks                384          8        20      2     4k
ffff8102297fe180 rpc_inode_cache          832          0         0      0     4k
ffff8102297fe800 gpfsInodeCache           832      26008     26020   6505     4k
ffff8102297ff140 gpfsBufChunk             528          0        70     10     4k
ffff8102297ff7c0 gpfsShMemDesc             40        156       184      2     4k
ffff810232c86100 fib6_nodes                64         17        59      1     4k
...

crash> kmem -S gpfsInodeCache
CACHE            NAME                 OBJSIZE  ALLOCATED     TOTAL  SLABS  SSIZE
ffff8102297fe800 gpfsInodeCache           832      26008     26020   6505     4k

SLAB              MEMORY            TOTAL  ALLOCATED  FREE
ffff810194a530c0  ffff810194a53100      4          1     3
FREE / [ALLOCATED]
   ffff810194a53100  (cpu 0 cache)
   ffff810194a53440  (cpu 0 cache)
   ffff810194a53780  (cpu 0 cache)
  [ffff810194a53ac0]
SLAB              MEMORY            TOTAL  ALLOCATED  FREE
ffff8101aa6d9080  ffff8101aa6d90c0      4          4     0
FREE / [ALLOCATED]
  [ffff8101aa6d90c0]
  ...
```

Using Crash: *dumping contents from SLAB memory*

- Initial address: ffff8101aa6d90c0
- Word size: 64 bits
- Object size: 832 bytes → 832/64=104

```
crash> rd -64 ffff8101aa6d90c0 104
ffff8101aa6d90c0:  0000000000000000 ffff810005bbc0a0   ................
ffff8101aa6d90d0:  ffff8101aa6d9410 ffff810194a53ad0   ..m.....:......
...
ffff8101aa6d93d0:  ffff8101aa6d93d0 ffff8101aa6d93d0   ..m.......m.....
ffff8101aa6d93e0:  0000000000000001 0000000000000000   ................
ffff8101aa6d93f0:  0000000000000000 0000000000000000   ................
```

- When CONFIG_DEBUG_SLAB=y, the last word contains the address of the function which allocated that object.
- We can also verify poisoned memory
  - 0xa5 == uninitialized memory
  - 0xa6 == memory already made free

Using Crash

- That's pretty much it!
- In short:
  - Crash is a great tool – use it!
  - Crash helps to find problems and may require integration with shell scripts for some tasks, such as extracting memory owner from all objects of a given SLAB
  - It doesn't do everything – reading the code is still essential to find where the actual culprit hides

Resources

- http://people.redhat.com/anderson/crash_whitepaper
- Haren :)

Thanks!