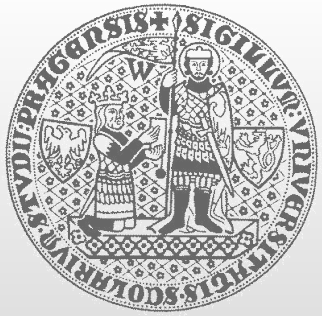
mdb

Crash Dump Analysis 2015/2016



CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Department of Distributed and Dependable Systems

ORACLE®

SUSE®

redhat®

mdb – modular debugger

- **Low-level tool**
  - Assembly-level debugging, printing C structures
  - Support for both user space and kernel debugging
  - Support for live/post-mortem debugging
  - Allows writing custom modules with new debugger commands
  - Bundled in all Solaris distributions

kmdb – kernel mdb

- **Allows debugging of live kernel**
  - Shares most of mdb features
    - Both mdb and kmdb built from same sources
  - Full control over kernel
    - Debugger stops kernel
    - Requires console access
    - Allows single stepping, breakpoints and watchpoints inside kernel

References

- **Oracle Solaris 11.2 Modular Debugger Guide**
  - http://docs.oracle.com/cd/E36784_01/html/E36864/toc.html
- **Man pages**
  - mdb(1), kmdb(1)

mdb evolution

Debugger	Year	OS	Note
adb(1)	1979	7th Edition UNIX	assembly-level debugger
crash(1M)	1984	System V R3	examine control structures of kernel
kadb(1M)	1986	SunOS 3.5	kernel adb
mdb(1)	1999	Solaris 8	technology preview
mdb(1)	2001	Solaris 9	CTF – debugging information in kernel
mdb(1)	2003	Solaris 10	user space CTF support
kmdb(1)	2004	Solaris 10	in situ kernel debugger

Starting mdb

- **User space binary**
  - Syntax: `mdb` *`program`*
  - User space core dump
    - Syntax: `mdb` `[`*`program`*`]` *`core`*
- **Kernel crash dump**
  - Crash dump is pair of files `unix.X` and `vmcore.X`
    - Syntax: `mdb` *`X`* or `mdb unix.`*`X`* `vmcore.`*`X`*
  - Live kernel
    - Syntax: `mdb -k` or `mdb /dev/ksyms /dev/kmem`

Basic commands

- **Command syntax: [value [,count]] command**
  - General commands

Command	Function
::help	Get help
::quit	Quit debugger (Ctrl-D works in same way)
<command> ! <shell>	Pipes command output to shell
::log	Log session to file

  - dcmd
    - Commands are often referred as dcmds (debugger commands)

Expressions

- **Constants**
  - By default all numbers are hexadecimal!
  - Use prefixes to change radix (zero and letter)

prefix	radix
0t	decimal
0x	hexadecimal (default)
0o	octal
0i	binary

- **Symbols**
  - Function names, variables are virtual addresses
    - Symbolic information is based on ELF symbols and CTF

Expressions (2)

- **Operators**
  - Common operators are +, -, *
  - Division is %
- **Variables**
  - Value of variable can be used in expression using '<' in form '<variablename'
- **Example expression**
  - main+4f – address of “instruction” in function main at offset 0x4f

Displaying data

- **Read data from memory**
  - address [, count] dcmd format [format ...]
  - Dcmd is either '/', '\', '?'

dcmd	function
/	read data from virtual address
\	read data from physical address
?	read data from primary target
=	display (convert)

- **Print value in given format**
  - value = format

Displaying data (2)

- **Formats**
  - Format is single character (::formats)
  - Format groups: read, write, search, special
- **Read formats**
  - Hexadecimal byte (B), Hexadecimal int (X)
  - Decimal int (D), Hexadecimal long long (J)
  - Instruction (I), Symbol (p)
  - C string (s)

Displaying data (3)

- **Write formats**
  - address [, count] dcmd format value
  - 1 byte (v), 4 bytes (W)
- **Special formats**
  - Newline (n), address (a)
- **Examples**

```
> 1234=D
                4660

> ip_debug/X
ip_debug:
ip_debug:       0

> ip_debug/W 5
ip_debug:       0x0             =       0x5
```

Simple conversion from hexadecimal to decimal number

Reading and writing from/to a variable "ip_debug"

Displaying data (4)

- **Macros**
  - Obsolete, prior mdb format characters were used to dump complex C structures, even containers
  - C structures were converted to set of macro(s) containing primitive dcmds

```
.>x
<x+0=""
+/"code"16t"result"16t"flags"nXXX
+/"counter"16t"cb_arg"16tnXX
+/"mutex"
.$<<mutex

...
```

Displaying data (5)

- **Printing C structures**
  - Syntax: “addr::print [-ta] [type] [member] ...”
  - Requires CTF debugging symbols

```
> 0000030002d924a0::print -ta proc_t
{
    30002d924a0 struct vnode *p_exec = 0x30008028800
    30002d924a8 struct as *p_as = 0x300015eca88
    30002d924b0 struct plock *p_lockp = 0x300011f8940
    30002d924b8 kmutex_t p_crlock = {
        30002d924b8 void *[1] _opaque = [ 0 ]
    }
    30002d924c0 struct cred *p_cred = 0x300003b2a58

...

> 0000030002d924a0::print -ta proc_t p_user.u_psargs
30002d92a41 char [80] p_user.u_psargs = [ "/usr/lib/inet/inetd start" ]
```

Displaying instructions

- **Disassembler**
  - Dcmd address::dis
  - Disassemble single instruction (/i)

```
> main+0x20/i
main+0x20:      call    -0x579   <PLT:textdomain>

> main+40::dis -n 3
main+0x34:                      call      -0x9c924      <clock_tick_init_pre>
main+0x38:                      nop
main+0x3c:                      call      -0x6b0ac      <clock_init>
main+0x40:                      nop
main+0x44:                      call      -0x10f5d8     <lgrp_plat_probe>
main+0x48:                      sethi     %hi(0x1815000), %i5
main+0x4c:                      ldx       [%i5 + 0xb8], %l7
```

Registers

- **Registers are internal variables in mdb**
  - Using register value
    - <rip::dis – disassemble instructions around program counter
  - ::regs - displaying registers
  - Trap frames (trap-frame-addr::print struct regs)
  - Alternatively use heuristic to find register values on stack

Program stack

- **Stack backtrace ($C)**

```
> $C
08046698 PLT:printf(8069c18, 8067490, 80466e8, 8052dc8)
080466e8 pem+0xdb(8069174, 80691a0, 0, 0, 0, 8069170)
080469a8 main+0x915(1, 80469d8, 80469e0, 80469cc)
080469cc _start+0x7d(1, 8046b38, 8067490, 8046b3b, 8046b5d, 8046b6d)
```

- **Stack backtrace of particular kernel thread (::findstack)**

```
> 300087bb100::findstack
stack pointer for thread 300087bb100: 2a101122db1
[ 000002a101122db1 cv_wait_sig+0x114() ]
  000002a101122e61 str_cv_wait+0x28()
  000002a101122f21 strwaitq+0x238()
  000002a101122fe1 strread+0x19c()
  000002a1011230b1 fop_read+0x20()
  000002a101123161 read+0x274()
  000002a1011232e1 syscall_trap+0xac()
```

Program stack (2)

- **Raw stack (addr,count/nap)**

```
> <esp,10/nap
0x804562c:
0x804562c:      pentry+0x600
0x8045630:      0x8056068
0x8045634:      0x8069c18
0x8045638:      0x8067488
0x804563c:      pentry+0x11
0x8045640:      0
0x8045644:      0
0x8045648:      0
0x804564c:      0
0x8045650:      0x2d90002
0x8045654:      0x509ee
0x8045658:      0
0x804565c:      0x816d
0x8045660:      0
0x8045664:      0x1008c
0x8045668:      0x4979996c
```

- Prints one line for one value on stack
  - The debugger provides hint (p) as it translates the value on stack to symbolic name (+ offset)

Running program

- **Run**
  - ::run arglist
- **Continue execution**
  - :c
- **Single stepping**
  - ::step
  - ::step over (skips function calls)
  - ::step out (returns from current function)

Breakpoints & watchpoints

- **Breakpoints**
  - ::bp
- **Watchpoints**
  - ::wp [-rwx] [-L size]
- **Display breakpoints and watchpoints**
  - ::events
- **Remove breakpoints and watchpoints**
  - ::delete [n]

Walkers

- **Walker**
  - Command which iterates over data structures
  - The output of walker is list of addresses
  - Usage: “::walk walkername”
  - Common walkers are “::walk proc”, “::walk thread”
  - Full list of walkers “::walkers”

```
> ::walk thread
ffffffffbc2c6e0
ffffff0003c05c60
ffffff0003c0bc60
...
```

Pipelines

- '|'
  - Interconnect output and input of commands
  - Typical usage with walkers

```
> 0t3694::pid2proc | ::walk thread | ::findstack
stack pointer for thread ffffff0164c0f900: ffffff00049b8a40
[ ffffff00049b8a40 _resume_from_idle+0xf1() ]
  ffffff00049b8a70 swtch+0x160()
  ffffff00049b8ad0 cv_wait_sig+0x14b()
  ffffff00049b8b30 str_cv_wait+0xbc()
  ffffff00049b8be0 strwaitq+0x1fe()
  ffffff00049b8c60 strread+0x159()
  ffffff00049b8ce0 spec_read+0x85()
  ffffff00049b8d50 fop_read+0x6b()
  ffffff00049b8e90 read+0x2b8()
  ffffff00049b8ec0 read32+0x22()
  ffffff00049b8f10 _sys_sysenter_post_swapgs+0x14b()
```

1. get pointer to proc_t structure of process 3694
2. walk threads of that process
3. prints stack backtrace

Longer version of 0t3694::pid2proc | ::walk thread

```
> 0t3694::pid2proc | ::print proc_t p_tlist | ::list kthread_t t_forw
ffffff0164c0f900
```

CTF

- **Compact debugging information**
  - Designed for production systems
  - Describes C structures, types, functions, arguments
  - No line information
  - Present in kernel/libraries

CTF (2)

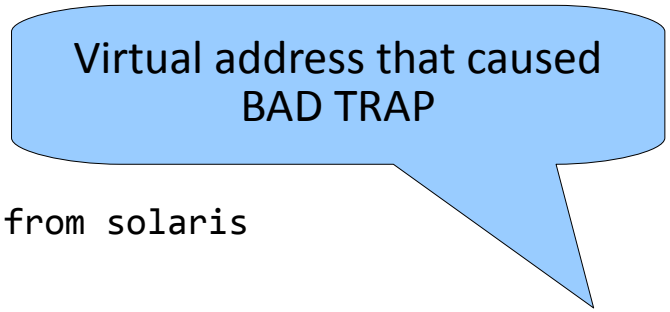
- **CTF tools**
  - Part of ON sources – ctfdump, ctfmerge, ctfconvert
- **Dcmds using CTF**
  - Show type definition - ::print [-ta] <type>
  - Build internal type graph - ::typegraph
  - Print size of a type ::sizeof <type>
  - Attempt to determine what type corresponds to an address <addr>::whattype
    - Requires built type graph

Example crash dump

- **Example crash dump analysis**
  - Run mdb

```
/var/crash/solaris$ mdb 2
Loading modules: [ unix genunix specfs dtrace cpu.generic uppc pcplusmp
scsi_vhci zfs ip hook neti sctp arp usba fctl md lofs fcip fcp cpc random
crypto logindmux ptm ufs sppp nsmb sd ipc ]
>
```

  - Run ::status




```
> ::status
debugging crash dump vmcore.2 (64-bit) from solaris
operating system: 5.11 snv_101b (i86pc)
panic message:
BAD TRAP: type=e (#pf Page fault) rp=ffffff0001306ac0 addr=ffffff0012341254
dump content: kernel pages only
```

Example crash dump (2)

- **Check kernel message buffer**
  - ::msgbuf

```
panic[cpu0]/thread=ffffff008d30f3a0:
BAD TRAP: type=e (#pf Page fault) rp=ffffff0001306ac0 addr=ffffff0012341254


hald:
#pf Page fault
Bad kernel fault at addr=0xffffff0012341254
pid=312, pc=0xffffffffff79a1314, sp=0xffffff0001306bb0, eflags=0x10246
cr0: 8005003b<pg,wp,ne,et,ts,mp,pe> cr4: 6b8<xmme,fxsr,pge,pae,pse,de>
cr2: ffffff0012341254
cr3: 35e8000
cr8: c

        rdi:                0 rsi:                0 rdx:                b
        rcx:                0  r8:               31  r9:               11
        rax:               31 rbx:                d rbp: ffffff0001306bf0
        r10:                a r11: ffffff00013069b0 r12:              27d
        r13: ffffff008d7a9f50 r14:                0 r15: ffffff0012341234
        fsb:                0 gsb: fffffffffbc2bc70  ds:                4b
         es:               4b  fs:                0  gs:               1c3
        trp:                e err:                0 rip: ffffffffff79a1314
         cs:               30 rfl:            10246 rsp: ffffff0001306bb0
         ss:               38

ffffff00013069a0 unix:die+10f ()
ffffff0001306ab0 unix:trap+1752 ()
ffffff0001306ac0 unix:_cmntrap+e9 ()
ffffff0001306bf0 mntfs:mntfs_global_len+34 ()
ffffff0001306c70 mntfs:mntfs_snapshot+102 ()
ffffff0001306ce0 mntfs:mntread+66 ()
ffffff0001306d50 genunix:fop_read+6b ()
ffffff0001306e90 genunix:read+2b8 ()
ffffff0001306ec0 genunix:read32+22 ()
ffffff0001306f10 unix:brand_sys_sysenter+1e6 ()

syncing file systems...
 done
dumping to /dev/zvol/dsk/rpool/dump, offset 65536, content: kernel
```

Example crash dump (3)

- **Find immediate root cause**
  - In case of BAD TRAP, find faulty instruction
  - Display instruction on instruction pointer

```
> <rip/i
mntfs_global_len+0x34:          testl  $0x2,0x20(%r15)
```

  - Verify instruction operands

```
> <r15=K
                fffffff0012341234
> <r15+20/X
mdb: failed to read data from target: no mapping for address
0xfffffff0012341254:
```

  - Register r15 + 0x20 points to unmapped memory

Example crash dump (4)

- **Continue to find the root cause**
  - First need to back trace where the value came from
  - Disassemble and analyze last function, which executed before BAD TRAP
  - Primarily check assignments to r15

```
> mntfs_global_len::dis ! grep r15
mntfs_global_len+0x17:          pushq  %r15
mntfs_global_len+0x20:          movq   +0x42e5811(%rip),%r15    <rootvfs>
mntfs_global_len+0x34:          testl  $0x2,0x20(%r15)
mntfs_global_len+0x3e:          movq   (%r15),%r15
mntfs_global_len+0x4c:          movq   %r15,%rdi
mntfs_global_len+0x57:          movq   (%r15),%r15
mntfs_global_len+0x5a:          cmpq   +0x42e57d7(%rip),%r15    <rootvfs>
mntfs_global_len+0x6e:          popq   %r15
>
```

Example crash dump (5)

- **Register value life-cycle**
  - Finding where register value came from may be difficult, especially if it's passed through several function calls
  - In this simple case, there are only 3 assignments

```
mntfs_global_len+0x20:          movq    +0x42e5811(%rip),%r15    <rootvfs>
mntfs_global_len+0x3e:          movq    (%r15),%r15
mntfs_global_len+0x57:          movq    (%r15),%r15
```

  - The function contains loop. The initial value of r15 is value of rootvfs variable, then it loads pointer from some “structure” – apparently sort of a linked list.

Example crash dump (6)

- **Walk the linked list by hand**

```
> rootvfs/K
rootvfs:
rootvfs:        ffffffffbcad650
> ffffffffbcad650/K
root:
root:           ffffffffbcad4b0
> ffffffffbcad4b0/K
devices:
devices:        ffffffffbcad580
> ffffffffbcad580/K
dev:
dev:            ffffff0085727ea8
> ffffff0085727ea8/K
0xffffff0085727ea8:             ffffff0085727dd8
...
> ffffff0085727278/K
0xffffff0085727278:             ffffff0012341234
> ffffff0012341234/K
mdb: failed to read data from target: no mapping for address
0xffffff0012341234:
>
```

- Looks like corruption in linked list

Example crash dump (7)

- **Walk the linked list automatically using ::map**

```
> rootvfs>x
> ,30::map "*(<x+0x0)>x"
ffffffffbcad650
ffffffffbcad4b0
ffffffffbcad580
fffffff0085727ea8
fffffff0085727dd8
fffffff0085727d08
fffffff0085727c38
fffffff0085727b68
fffffff0085727a98
fffffff00857279c8
fffffff00857278f8
fffffff0085727828
fffffff0085727758
fffffff0085727688
fffffff00857275b8
fffffff00857274e8
fffffff0085727348
fffffff0085727278
fffffff0012341234
mdb: failed to read from address fffffff0012341234: no mapping for address
>
```

Example crash dump (8)

- **What is rootvfs?**
  - Use ::typegraph and ::whattype to find the type

```
> ::typegraph
typegraph:                              pass => initial
typegraph:                     maximum nodes => 550694
typegraph:                      actual nodes => 488307
typegraph:                    anchored nodes => 5622
typegraph:          time elapsed, this pass => 7 seconds
typegraph:              time elapsed, total => 7 seconds
typegraph:
...

> rootvfs::whattype
ffffffffbc86b18 is ffffffffbc86b18+0, struct vfs *
```

Example crash dump (9)

- **Display how struct vfs looks like**
  - Use ::typegraph and ::whattype to find the type

```
> ::print -ta struct vfs
{
    0 struct vfs *vfs_next
    8 struct vfs *vfs_prev
    10 vfsops_t *vfs_op
    18 struct vnode *vfs_vnodecovered
    20 uint_t vfs_flag
    24 uint_t vfs_bsize
...
```

  - At offset 0x0 of structure, there is pointer to next list element vfs_next

Example crash dump (10)

- **Walk linked list using type information**
  - Use [addr]::list type member [variable]

```
> rootvfs::list struct vfs vfs_next
ffffffffbc86b18
ffffffffbcad650
ffffffffbcad4b0
ffffffffbcad580
ffffff0085727ea8
...
ffffff00857275b8
ffffff00857274e8
ffffff0085727348
ffffff0085727278
ffffff0012341234
mdb: failed to read next pointer from object ffffff0012341234: no mapping for address
```

  - Same result again, the last good pointer was at 0xffffff0085727278

Example crash dump (11)

- **Display last good vfs structure**

```
> ffffff0085727278::print -ta struct vfs
{
    ffffff0085727278 struct vfs *vfs_next = 0xffffff0012341234
    ffffff0085727280 struct vfs *vfs_prev = 0xffffff0085727348
    ffffff0085727288 vfsops_t *vfs_op = vfssw+0x5b8
    ffffff0085727290 struct vnode *vfs_vnodecovered = 0xffffff0086973500
    ffffff0085727298 uint_t vfs_flag = 0x2420
    ffffff008572729c uint_t vfs_bsize = 0x1000
    ffffff00857272a0 int vfs_fstype = 0xb
    ffffff00857272a4 fsid_t vfs_fsid = {
...
    ffffff0085727300 refstr_t *vfs_resource = 0xffffff00877f24e0
    ffffff0085727308 refstr_t *vfs_mntpt = 0xffffff0084dcf1f0
...
```

- **It's not a software bug, this structure was corrupted by administrator on purpose**

Example crash dump (12)

- **Display last good vfs_resource and vfs_mntpt**

```
> 0xffffff00877f24e0::print -ta refstr_t
{
    ffffff00877f24e0 uint32_t rs_size = 0xd
    ffffff00877f24e4 uint32_t rs_refcnt = 0x1
    ffffff00877f24e8 char [1] rs_string = [ "s" ]
}
> ffffff00877f24e8/s
0xffffff00877f24e8:             swap
> 0xffffff0084dcf1f0::print -ta refstr_t
{
    ffffff0084dcf1f0 uint32_t rs_size = 0x11
    ffffff0084dcf1f4 uint32_t rs_refcnt = 0x1
    ffffff0084dcf1f8 char [1] rs_string = [ "/" ]
}
> ffffff0084dcf1f8/s
0xffffff0084dcf1f8:             /var/run
```