Trap tracing

Crash Dump Analysis 2014/2015

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Department of Distributed and Dependable Systems

When DTrace is not enough

- Mostly assembly language code without epilogues and prologues
- When there are no sdt probes
- When the context is very restricted, such as callback_handler or trap code
- If there is no DTrace, i. e. Solaris 9 and older

When kmdb is not enough

- Set a breakpoint, but the debugger fails when the breakpoint is hit
- Set a breakpoint, the kernel crashes without hitting the breakpoint
- Set a breakpoint, the breakpoint is hit, but the kernel crashes upon continuing

TRAPTRACE

- **Low-level tracing of essential system events**
  - Available in Solaris
    - IA-32/AMD64 and SPARC V9 (**sun4u**/sun4v)
      - Slightly different implementations
  - May be the only analysis aid left when ...
    - ... everything else fails
    - ... other techniques are not applicable

TRAPTRACE (2)

- **Compile-time choice**
  - Cannot be turned on if not present
  - Cannot be turned off if present
  - Enabled in debug kernels
  - When TRAPTRACE macro defined
- **MDB support**
  - Present the trace data from a crash dump
    - Requires post-mortem interpretation by a human

Implementation – sun4u

- Trace data stored in a per-CPU kernel circular buffer of records of the `struct trap_trace_record` type

```
struct trap_trace_record {
    uint16_t        tt_tl;
    uint16_t        tt_tt;
    uintptr_t       tt_tpc;
    uint64_t        tt_tstate;
    uint64_t        tt_tick;
    uintptr_t       tt_sp;
    uintptr_t       tt_tr;
    uintptr_t       tt_f1;
    uintptr_t       tt_f2;
    uintptr_t       tt_f3;
    uintptr_t       tt_f4;
};
```

Implementation – sun4u (2)

- **tt_tl**
  - corresponds to the TL register as it existed in the moment of the event
  - trap level
    - (0) – no trap in progress
    - (1) – a trap in progress
    - (>1) – nested trap in progress
      - depth of nesting

Implementation – sun4u (3)

- tt_tt
  - trap type
    - 0x0 – 0x1ff
      - identifies the type of the trap
        - page fault vs. interrupts vs. window trap etc.
    - >= 0x200
      - for non-trap events
        - such as TSB-miss / hit
        - passing a trace-point in the code

Implementation – sun4u (4)

- **tt_tpc**
  - corresponds to the TPC register as it existed in the moment of the trap
  - trap PC
    - records the address in code where the event occurred
- **tt_tstate**
  - snapshot of the TSTATE register as it existed in the moment of the trap
    - information about processor state

Implementation – sun4u (5)

- **tt_tick**
  - corresponds to the STICK register as it existed in the moment of the event
  - event timestamp
- **tt_sp**
  - snapshot of the SP register as it existed in the moment of the event

Implementation – sun4u (6)

- tt_tr
- tt_f1 - tt_f4
  - auxilliary fileds used by non-trap records
    - e. g. details about MMU faults, register windows configuration registers

Instrumentation – sun4u

- Spot the difference

```
> trap_table0+98*20,20/ai
0x1001300:
0x1001300: stx %l0, [%sp + 0x7ff]
...
0x100131c: stx %l7, [%sp + 0x837]
0x1001320: stx %i0, [%sp + 0x83f]
...
0x1001338: stx %fp, [%sp + 0x86f]
0x100133c: stx %i7, [%sp + 0x877]
0x1001340: ba +0x60dc
<0x100741c>
0x1001344: rd %pc, %l4
0x1001348: clr %l4
0x100134c: saved
0x1001350: retry
```

```
> trap_table0+98*20,20/ai
0x1001300:
0x1001300: stx %l0, [%sp + 0x7ff]
...
0x100131c: stx %l7, [%sp + 0x837]
0x1001320: stx %i0, [%sp + 0x83f]
...
0x1001338: stx %fp, [%sp + 0x86f]
0x100133c: stx %i7, [%sp + 0x877]
0x1001340: saved
0x1001344: retry
0x1001348: illtrap   0x0
0x100134c: illtrap   0x0
0x1001350: illtrap   0x0
```

Instrumentation – sun4u

- **Spot the difference**

```
> trap_table0+98*20,20/ai
0x1001300:
0x1001300: stx %l0, [%sp + 0x7ff]
...
0x100131c: stx %l7, [%sp + 0x837]
0x1001320: stx %i0, [%sp + 0x83f]
...
0x1001338: stx %fp, [%sp + 0x86f]
0x100133c: stx %i7, [%sp + 0x877]
0x1001340: ba +0x60dc
<0x100741c>
0x1001344: rd %pc, %l4
0x1001348: clr %l4
0x100134c: saved
0x1001350: retry
```

```
> trap_table0+98*20,20/ai
0x1001300:
0x1001300: stx %l0, [%sp + 0x7ff]
...
0x100131c: stx %l7, [%sp + 0x837]
0x1001320: stx %i0, [%sp + 0x83f]
...
0x1001338: stx %fp, [%sp + 0x86f]
0x100133c: stx %i7, [%sp + 0x877]
0x1001340: saved
0x1001344: retry
0x1001348: illtrap   0x0
0x100134c: illtrap   0x0
0x1001350: illtrap   0x0
```

Instrumentation – sun4u (2)

- **TT_TRACE(label) macro**
  - trace_gen
  - trace_win
  - trace_tsbmiss
  - trace_tsbhit

Instrumentation – sun4u (3)

- **SYSTRAP_TRACE**
  - tracing the sys_trap() trace-point
- **Directly embedded**
  - pil_interrupt()

MDB Support

- MDB can present the TRAPTRACE data collected before crash
- The data can be used to reconstruct events which lead to a crash
- Syntax
  - `[cpuid]::ttrace [-x]`

MDB Support (2)

```
> ::ttrace
CPU %tick            %tt               %tl  %tpc
  0 00000000c40ced44 0024 cleanwin     0001 000000000108e4c0 vsnprintf
  0 00000000c40ced1f 0268 ?            0001 0000000001087704 panicsys+0x120
  0 00000000c40cecfb 0098 spill-6-norm 0001 00000000010086e4 flush_windows+4
  0 00000000c40cecf5 0098 spill-6-norm 0001 00000000010086e4 flush_windows+4
```

MDB Support (3)

```
> 0::ttrace -x
%tick            %tstate          %tt  %tl  %tpc             %sp
TR               F1-4
00000000c40ced44 0000000000001606 0024 0001 000000000108e4c0 0000000000000000
0000000000009999 [15,7030003,3000e,0]
00000000c40ced1f 00000000000001c0 0268 0001 0000000001087704 0000000070002000
000000003f575c00 [ffffffffffffffff,1087708,3f680010,0]
00000000c40cecfb 0000009900001603 0098 0001 00000000010086e4 000000000180d5d1
0000000000009999 [15,2050001,3000e,1087be4]
00000000c40cecf5 0000009900001603 0098 0001 00000000010086e4 000000000180d681
0000000000009999 [15,1040002,3000e,102ae9c]
```

Real life example – Scenario

- **Demonstration of memory debugging techniques**
  - Scenario
    - We used *netcat* as our target
    - We used *libwatchmalloc* for memory debugging
  - Found buffer overrun in *netcat*, an application that had been running fine for quite a few years
  - Overrun not found in old *netcat* version

Real life example – netcat

- **Attempt #1 - Something changed in netcat**
  - Source code history does not show any changes
  - Maybe change in some dynamically linked libraries
  - Tried running with *libumem* ... no memory issue discovered
  - Does not seem to be caused by *netcat* itself

Real life example – watchmalloc

- **Attempt #2 - Problem in *watchmalloc* library**
  - The algorithm as follows:
    - Preload own *alloc/free* functions that get used instead the system ones
    - Append header and/or footer to each block
    - Protect header/footer using *watchpoints*
  - Simple code, no obvious bug
  - And yet allegedly the header/footer was touched

Real life example – back in netcat

- **Attempt #3 - back in *netcat***
  - The function where we stopped makes a copy of an 48byte wide buffer
  - It is implemented as 3x 16byte moves via MMX/SSE instructions
  - Older version of uses 6x 8byte moves via GPRs
  - Finally! Something has changed - the compiler
  - Unfortunately neither assembly versions should trigger the watchpoint and stop

redhat SUSE ORACLE Department of Distributed and Dependable Systems D3S

Real life example – watchpoints

- **Attempt #4 - Problem with watchpoints**
  - A watchpoint is interval *(start, start+len)* of memory that we care about
  - The MMU works with 4k pages. Once you enable watchpoint, kernel has to protect whole page to catch any access
  - During every protection fault it needs to determine interval of the fault and compare it to list of watchpoint intervals and either continue or take action
  - The range of the fault is calculated in a following way:
    - The start address is provided by the MMU hardware
    - The length is depending on the instruction that was running
  - Unfortunately the in-kernel disassembler understands SSE/MMX and knows that
  - the access is 16byte wide. There is no way to hit the watchpoint.

Real life example – which tool

- **Attempt #5 - take a look at protection fault**
  - This is not task for DTrace
    - predicates too complicated to catch so precise fault
    - Otherwise it produces lot of data
  - *kmdb* is helpful to debug trap code, but we would like to check the HW fault and this is place where kmdb could have troubles

Real life example – which tool

- **Attempt #5 - take a look at protection fault**
  - TrapTrace
    - As it is user-space fault, the registers contain user-space values. Including *%rip*
    - We know the instruction range and its place in memory
    - Thus we can find corresponding trap trace data for that particular %rip

Real life example – what we found

- **What we found**
  - The register *%cr2* does not contain expected value. The value in it is +8 bytes
  - Based on this every interval kernel needs to check is shifted 8 bytes to the right
  - Last 16-byte move instruction will cross the watchpoint boundary

Real life example – what we found (2)

- **How can this happen?**
  - The CPU reports correct *%cr2* for 8byte memory access, but not for 16 byte memory access
  - Except latest Xeons, no cpu guarantees atomic 128-bit memory stores. Our CPU performs 2x 8-byte stores internally and reports address in the middle in its *%cr2*
  - You can find this in AMD's cpus errata. There is no fix/workaround available as it does not break paging in operating systems, just causes troubles with debuggers.