Problem Solving with SystemTap

Eugene Teo <eteo@redhat.com>

Wednesday, May 9th 2007

What is SystemTap

- A troubleshooting and analysis tool
- A scripting language translator
- An instrumentation framework
- A free software (GPL)

Have you ever wondered...

- Who killed my process?
- Why is there so much I/O going on?
- Is this program an exploit or backdoor?
- Why did OOM killer start killing process?
- What performance statistics can i collect?
- Why does my battery drain so quickly?
- And the list goes on...

Quick demonstration

- Example #1: Trace when process executes
- Example #2: Trace SIGKILL signals
- Example #3: Trace top I/O “offenders”

What SystemTap is like

- Traditional tools
  - `strace`, `ltrace` – tracing system and library calls
  - `gdb`, `printk()` - tracing kernel functions and variable values
  - `sar`, `vmstat`, `iostat` – measuring systems performance
  - `Oprofile` – profiling running kernel code
  - `crash`, `gdb` – accessing kernel data structures
- SystemTap is capable to do all of the above, and more

What SystemTap offers

- System-wide and process-centric views
- Flexible extendable framework with tapsets
- Event/action, procedural scripting language
- Protected, and simple interface to kprobes

What isn’t SystemTap

- SystemTap isn’t sentient; requires user thinking process
- SystemTap isn’t a replacement for any existing tools

Who is SystemTap for

- System Administrators
- Software Developers
- Kernel Hackers
- Researchers
- End users

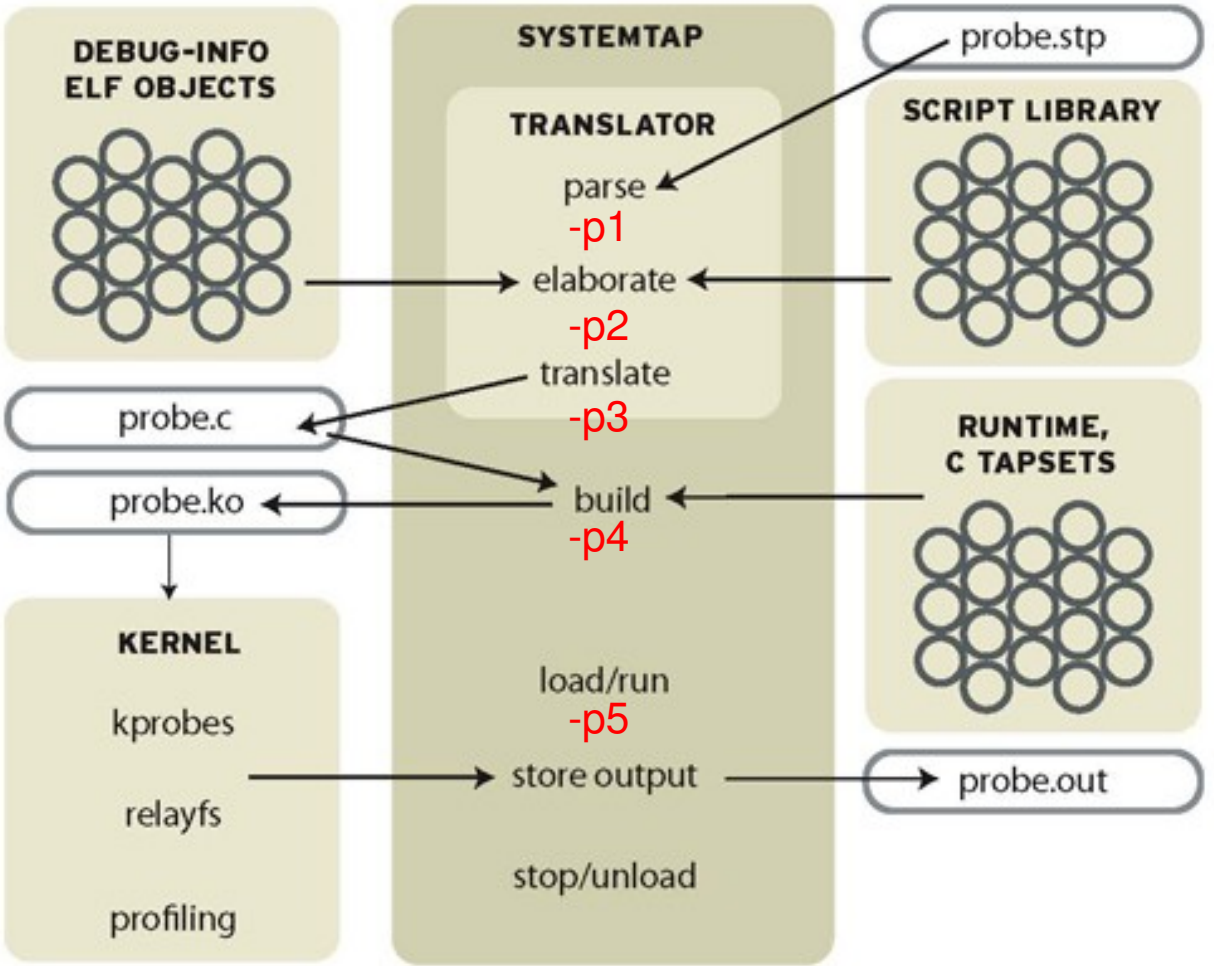
How system administrators use SystemTap

- Install SystemTap, and its prerequisites
- Run pre-packaged instrumentation scripts
- Pre-compile a script, and deploy it on production machines, if necessary
- Modify and share; blog about it
- Tell us your problem
- In exchange, we try to help you write a script to solve them

How developers use SystemTap

- Likewise, install SystemTap, and its prerequisites
- Identify problem areas (functions, directory/file:line), events
- Search tapset for available probe definitions
- Identify additional variables in probe context
- Write SystemTap script
- Distill area expertise into “tapset” scripts
- Share and blog about it

How SystemTap works



- `stap -p[1-5] <script>`
- If `-p[1-5]` is not specified, then `stap` will run all 5 passes

How to use SystemTap

- Writing one-liner or SystemTap scripts
  - See http://sources.redhat.com/systemtap/wiki/WarStories
- Within shell scripts
  - Helps to know shell scripting, and procedural programming languages
  - Possibilities are endless
- Run SystemTap Toolkit scripts (new project)
  - See http://sources.redhat.com/systemtap/wiki/ScriptsTools
- Use SystemTapGUI (IDE for SystemTap)
  - See http://stapgui.sourceforge.net

Components of SystemTap script

- Main outermost constructs: probes and functions
  - Probe definition:
    ```
    probe PROBEPOINT [, PROBEPOINT] { [stmt ...] }
    ```
  - Probe aliases:
    ```
    probe <alias> = <probepoint> { <prologue_stmts> }
    probe <alias> += <probepoint> { <epilogue_stmts> }
    ```
  - Global variables:
    ```
    global var1[=<value>], var2[=<value>]
    ```
  - Auxiliary function:
    ```
    function <name>[:<type>] (<arg1>[:<type>], ...)
            { <stmts> }
    ```
  - Embedded C function:
    ```
    function <name>[:<type>] (<arg1>[:<type>], ...)
            %{ <C_stmts> %}
    ```

Components of SystemTap script (cont'd)

- Within these outer constructs, are statements and expressions in C-like language
  - Comments
    `# ..., // ..., /* ... */`
  - do ... while loop
    `do STMT while (EXP)`
  - for loop
    `for (EXP1; EXP2; EXP3) STMT`
  - foreach loop
    `foreach (VAR in ARRAY) STMT`
  - `break`, and `continue`
  - Statistics (Aggregates)
    `<<<, @count, @avg, @hist_log, @hist_linear`
  - more...

Example #1: Trace when process executes

```
$ cat execve.stp
#!/usr/bin/env stap

probe syscall.exec* {
        printf("exec %s %s\n", execname(), argstr)
}
```

Example #2: Trace SIGKILL signals

```
$ cat sigkill.stp
#!/usr/bin/env stap

probe signal.send {
        if (sig_name == "SIGKILL")
                printf("%s was sent to %s (pid:%d) by uid:%d\n",
                        sig_name, pid_name, sig_pid, uid());
}
```

Example #3: Trace top I/O “offenders”

```
$ cat traceio.stp
#!/usr/bin/env stap

global reads, writes, total_io

probe kernel.function("vfs_read").return {
        reads[execname()] = total_io[execname()] += $return
}

probe kernel.function("vfs_write").return {
        writes[execname()] = total_io[execname()] += $return
}

probe timer.s(1) {
        foreach(p in total_io- limit 10)
                printf("%15s r: %8d KiB w: %8d KiB\n",
                        p, reads[p]/1024,
                        writes[p]/1024)
        printf("\n")
}
```

Example #4: Trace new processes created

- Have you ever wonder what happen when you run `man`?
- Also, an example of a one-liner SystemTap script

```
$ stap -e 'probe process.create { printf("%s (pid:%d)
created\n", task_execname(task), new_pid)}' \
        -c 'man stap 2>&1>/dev/null'
```

Example #5: Aggregate system calls count

```
$ cat syscall_graphs.stp
#!/usr/bin/env stap

global probes

probe syscall.* {
        if (execname() == "ls")
                probes[probefunc()] <<< 1
}

probe end {
        foreach (p in probes) {
                printf("%-30s %4d\n", p, @count(probes[p]))
                print(@hist_linear(probes[p], 1, 2, 1))
        }
}
```

Components of Tapset script

- Probes and functions that can be reused by other scripts
- Tapset to encapsulate knowledge about a kernel subsystem
- Also ends with .stp (do not be confused)
- Think of this as a library or a class file that other scripts can create or instantiate objects without knowing how it works, as long as it works
- Located at `/usr/share/systemtap/tapset` by default
- Additional tapsets?

```
$ stap
[...]
   -I DIR      look in DIR for additional .stp script files, in
               addition to
               /usr/local/share/systemtap/tapset
               /usr/local/share/systemtap/tapset/LKET
```

Quick example of Tapset script

```
$ cat /usr/share/systemtap/tapset/syscalls2.stp
[...]
# long sys_open(const char __user * filename,
                int flags, int mode)
probe syscall.open = kernel.function("sys_open") ?,
                     kernel.function("compat_sys_open") ?,
                     kernel.function("sys32_open") ?
{
        name = "open"
        filename = user_string($filename)
        flags = $flags
        mode = $mode
[...]
                argstr = sprintf("%s, %s",
                         user_string_quoted($filename),
                         _sys_open_flag_str(flags))
}
```

What's coming next!

- Dynamic tracing of user-space applications
- Static probing markers (as opposed to dynamic instrumentation)
- Cross-compilation of SystemTap scripts
- Stay tuned to SystemTap mailing list

Dynamic tracing of user-space applications

- Observability across entire software/kernel stack
- Examine software execution from user to kernel-space
- Uses utrace (already in Fedora kernel, upstreaming it via -mm kernel)
- Uses uprobes (an utrace module/client)
- Uprobes exploits, and extends utrace to provide kprobe-like, breakpoint-based probing to user-space applications
- Need to extend SystemTap translator to interface with uprobes
- Work-in-progress

Static probing markers

- Designate points in functions as being candidates for SystemTap-style probing
- Using direct instructions rather than int3 breakpoints
- Faster and more precise than kprobes
- Based on Linux Kernel Markers

http://marc.info/?l=linux-kernel&m=117797543716260&w=2
List: linux-kernel
Subject: 2.6.22 -mm merge plans
From: Andrew Morton <akpm () linux-foundation ! org>
Date: 2007-04-30 23:20:07
[...]
Static markers. Will merge.

Cross-compilation of SystemTap scripts

- Possible to cross-compile for machines with different sub-architectures
- The stock SystemTap assumes that the host and target systems are the same sub-architecture
- If the host SystemTap is a i686 machine and the target is a i586 (OLPC laptop), the generated kernel module for the target machine will fail to load because of a runtime architecture test, i.e. i686 != i586

```
$ stap -e 'probe syscall.open {}' -p3
[...]
    if (strcmp (machine, "i686")) {
      _stp_error ("module machine mismatch (%s vs %s)",
          machine, "i686");
      rc = -EINVAL;
    }
```

- But there is a quick hack to bypass that check. Proper implementation is in the TODO list

Interesting work being done on OLPC

- One Laptop per Child (OLPC) project
- pstimeouts – how often processes wake up due to timeouts
- idle1.stp – how much time is spent in power saving halted mode
- boottap – see what machine is doing when it boots up
- Red Hat Summit talk: Thursday 15:00 One Laptop Per Child (Chris Blizzard)

How often processes wake up due to timeouts

- Monitor timers that may cause a process to wake up
- Probe points: poll, select, epoll, “real time” interval timer, schedule_timeout

```
[eteo@kerndev ~]$ ./pstimeouts -u
  uid     pid |    poll   select    epoll   itimer    other| process
  500    3463 |     116        0        0        0        0| gnome...
  500    3467 |       0        0        0        0        0| scim...
  500    3482 |     220        0        0        2        0| metacity
[...]
[eteo@kerndev ~]$ ./pstimeouts -ua
  uid     pid |    poll   select    epoll   itimer    other| process
    0    2311 |       0        6        0        0        0| pcscd
    0    2342 |       0        0        0        0        2| automount
   38    2404 |       0        0        0        0        0| ntpd
[...]
```

- See http://dev.laptop.org/ticket/110

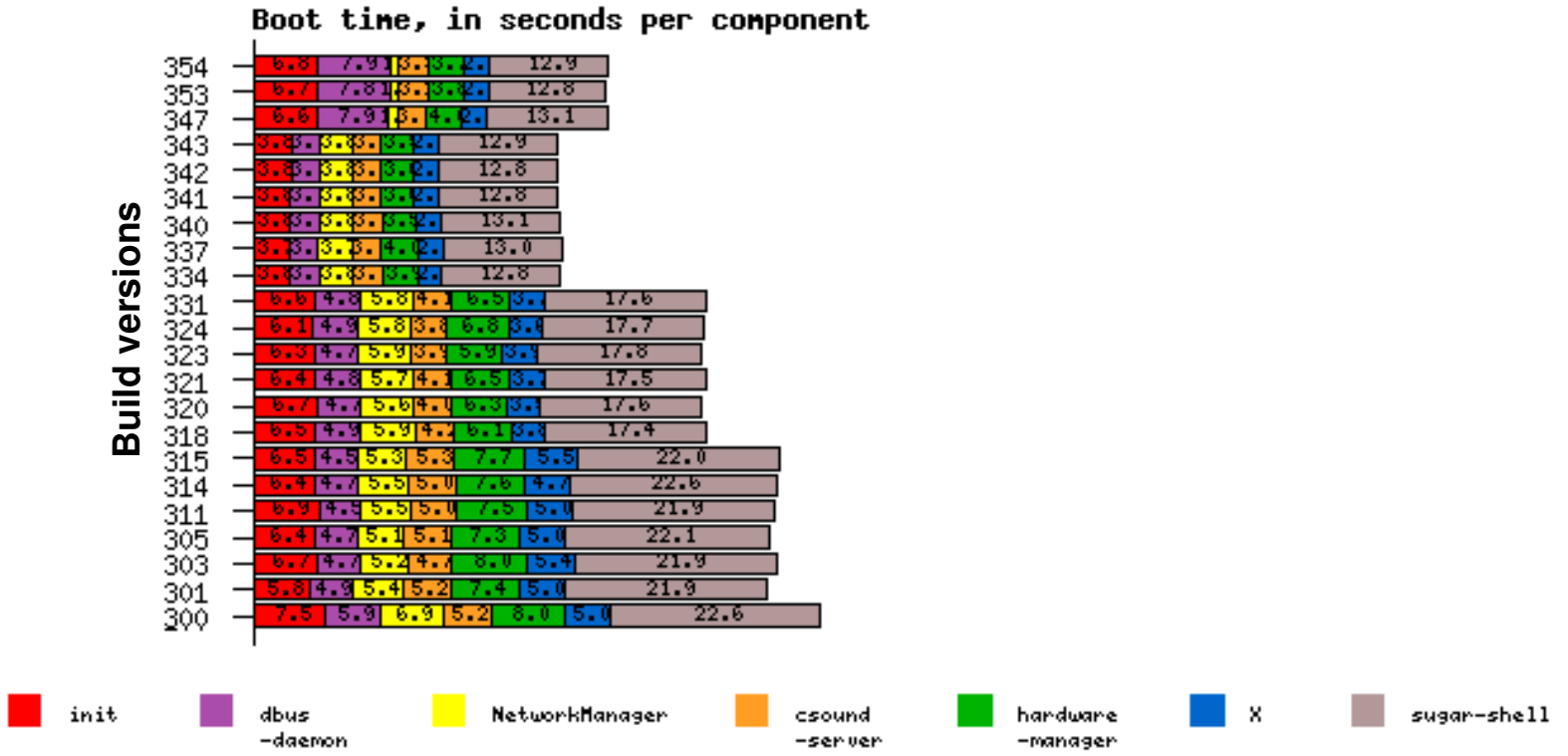
How much time spent in halt mode

- ACPI C1 state. “suspend on hlt”, causes CPU core to suspend on every hlt
- Probe points: safe_halt, irq_enter

```
Ran for 60544934 us <-- about 1 minute
entered default_halt 9609
exited default_halt 9609
percent halted: 17 <-- 17% for unloaded olpc
count: 9609
avg_time: 1081 <-- slightly over 1 ms
usec                distribution                                    count
    2 |                                                             18
    4 |@                                                           219
    8 |@                                                           295
   16 |                                                             56
   32 |@                                                           279
   64 |                                                            149
  128 |                                                            147
  256 |@                                                           275
  512 |@@@                                                         542
 1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  7605
[...]
```

See what machine is doing when it boots up

- Measuring boot time, in seconds per component
- Components: init, dbus-daemon, NetworkManager, csound-server, hardware-manager, X, and sugar-shell
- See OLPC Tinkerbox http://learn.laptop.org/tinderbox/



Exploring SystemTap!

- This has been just an introduction to SystemTap
- There's much, much more...
- Learn from `man stap`, `stapprobes`, `stapfuncs`, `stapex`
- Learn from existing tapsets `/usr/share/systemtap/tapset`
- Website: http://sources.redhat.com/systemtap/
- Wiki: http://sources.redhat.com/systemtap/wiki/
- War Stories: http://sources.redhat.com/systemtap/wiki/WarStories
- Tutorial (needs updating): http://sourceware.org/systemtap/tutorial.pdf
- Utrace: http://people.redhat.com/roland/utrace/
- Join our mailing list: systemtap@sources.redhat.com
- And hang out at #systemtap on irc.freenode.net

Problem Solving with SystemTap

Eugene Teo <eteo@redhat.com>

Wednesday, May 9th 2007