SystemTap: Full System Observability for Linux

Adrien Kunysz
adk@redhat.com

RMLL, Bordeaux, France
8 July 2010

Who is Adrien Kunysz?

- Senior Technical Support Engineer at Red Hat UK
  - when you call support, I pick up the phone
- co-founder of FSUGAr (Belgium)
- Krunch or adk on Freenode
- I like to look at core files and to read code
  - . . . but sometimes you need something more dynamic
- I am just a happy SystemTap user (not a developer)

What are we going to discuss?

- Explaining SystemTap
- Practical Examples
- Requirements and safety
- Comparison to Other Tools
- Conclusion
- More examples?

What is SystemTap?

According to http://sourceware.org/systemtap/

> *SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.*

- I like to think of it as a system-wide code injection framework
- with facilities for common tracing/debugging jobs
- makes it very easy to observe anything about a live system
  - . . . the problem is to figure out what you want to observe

How does it work?

1. write or choose a script describing what you want to observe
2. stap translates it into a kernel module
3. stap loads the module and communicates with it
4. just wait for your data

The five stap passes

```
# stap -v test.stp
Pass 1: parsed user script and 38 library script(s) in
    150usr/20sys/183real ms.
Pass 2: analyzed script: 1 probe(s), 5 function(s), 14
    embed(s), 0 global(s) in 110usr/110sys/242real ms.
Pass 3: translated to C into
    "/tmp/stapEjEd0T/stap_6455011c477a19ec8c7bbd5ac12a9cd0_13
     in 0usr/0sys/0real ms.
Pass 4: compiled C into
    "stap_6455011c477a19ec8c7bbd5ac12a9cd0_13608.ko" in
    1250usr/240sys/1685real ms.
Pass 5: starting run.
[...script output goes here...]
Pass 5: run completed in 20usr/30sys/4204real ms.
```

SystemTap probe points examples

SystemTap is all about executing certain actions when hitting certain probe points.

- `syscall.read`
  - when entering read() system call
- `syscall.close.return`
  - when returning from the close() system call
- `module("floppy").function("*")`
  - when entering any function from the "floppy" module
- `kernel.function("*@net/socket.c").return`
  - when returning from any function in file `net/socket.c`
- `kernel.statement("*@kernel/sched.c:2917")`
  - when hitting line 2917 of file `kernel/sched.c`

More probe points examples

- `timer.ms(200)`
  - every 200 milliseconds
- `process("/bin/ls").function("*")`
  - when entering any function in /bin/ls (not its libraries or syscalls)
- `process("/lib/libc.so.6").function("*malloc*")`
  - when entering any glibc function which has "malloc" in its name
- `kernel.function("*init*")`, `kernel.function("*exit*").return`
  - when entering any kernel function which has "init" in its name or returning from any kernel function which has "exit" in its name

RTFM for more (`man stapprobes`).

SystemTap programming language

- mostly C-style syntax with a feeling of awk
- builtin associative arrays
- builtin aggregates of statistical data
  - very easy to collect data and do statistics on it (average, min, max, count,...)
- many helper functions (builtin and in tapsets)

RTFM: *SystemTap Language Reference* shipped with SystemTap (langref.pdf)

Some helper functions you'll see a lot

`pid()` which process is this?

`uid()` which user is running this?

`execname()` what is the name of this process?

`tid()` which thread is this?

`gettimeofday_s()` epoch time in seconds

`probefunc()` what function are we in?

`print_backtrace()` figure out how we ended up here

There are many many more. RTFM (`man stapfuncs`) and explore `/usr/share/systemtap/tapset/`.

Some cool stap options

- `-x` trace only specified PID (only for userland probing)
- `-c` run given command and only trace it and its children (will still trace all threads for kernel probes)
- `-L` list probe points matching given pattern along with available variables
- `-d` load given module debuginfo to help with symbol resolution in backtraces
- `-g` embed C code in stap script
  - unsafe, dangerous and fun

Agenda

Example 1: trace processes execution

Listing 1: exec.stp

```
1 probe syscall.exec* {
2     printf("exec %s %s\n", execname(), argstr)
3 }
```

```
$ stap -L 'syscall.exec*'
syscall.execve name:string filename:string
   args:string argstr:string

# stap exec.stp
exec gnome-terminal /bin/bash
exec bash /usr/bin/id -gn
exec bash /usr/bin/id -un
exec bash /bin/uname -s
exec bash /bin/uname -r
```

Example 2: real support case

Customer	Hello, the saslauthd service mysteriously stops every now and then, can you help?
Support	Sure, what does strace say?
Customer	It gets a SIGKILL.
Support	OK, let's figure out who is sending the signal.

Example 2 continued: sigkill.stp

Listing 2: examples/process/sigkill.stp

```
1 # Copyright (C) 2007 Red Hat, Inc., Eugene Teo
2 [...GPL blah...]
3 probe signal.send {
4   if (sig_name == "SIGKILL")
5     printf("%s was sent to %s (pid:%d) by %s uid:%d\n",
6            sig_name, pid_name, sig_pid, execname(), uid())
7 }
```

```
$ stap -L signal.send
signal.send name:string sig:long task:long
   sinfo:long shared:long send2queue:long
   sig_name:string sig_pid:long pid_name:string
    si_code:string $sig:int

# stap /usr/share/systemtap/tapset/signal.stp
SIGKILL was sent to saslauthd (pid:6202) by
   AntiCloseWait.s uid:0
```

Example 2 continued: fixing

```
$ find sosreport/ -name AntiCloseWait.s*
sosreport/etc/cron.hourly/AntiCloseWait.sh
```

Support Fix your cronjob.

Customer Thanks.

Example 3: monitoring file read/write

Listing 3: filewatch.stp

```
1  probe kernel.function("vfs_write"),
       kernel.function("vfs_read")
2  {
3      dev_nr   = $file->f_path->dentry->d_inode->i_sb->s_dev
4      inode_nr = $file->f_path->dentry->d_inode->i_ino
5
6      if (dev_nr == stat2dev($1) && inode_nr == $2)
7          printf("%s(%d) %s 0x%x/%u\n",
8                 execname(), pid(), probefunc(),
9                 dev_nr, inode_nr)
10 }
11
12 # convert "stat -c %d" output to a proper device number
13 function stat2dev(s)
14 {
15     return ((s & 0xff00) << 12) | (s & 0xff)
16 }
```

Example 3 continued: using filewatch.stp

```
# stat -c 'device: %d, inode: %i' /etc/passwd
device: 64768, inode: 1805253
# stap filewatch.stp 64768 1805253
bash(28549) vfs_read 0xfd00000/1805253
id(28553) vfs_read 0xfd00000/1805253
crontab(28579) vfs_read 0xfd00000/1805253
id(28585) vfs_read 0xfd00000/1805253
vim(28620) vfs_read 0xfd00000/1805253
```

Agenda

Requirements

- you need kprobes (`CONFIG_KPROBES=y`)
- for source-level tracing, you need debug symbols of the code you want to trace
  - `package-debuginfo` on RPM distros
  - `package-dbg` on .deb distros
  - build your application with `gcc -g`
  - for kernel it's `CONFIG_DEBUG_INFO=y`
- for userland tracing you need the utrace kernel patch
  - not in mainline (yet?)
  - in Red Hat Enterprise Linux 5+, Fedora,...

Performances and safety

- language-level safety features
  - no pointers
  - no unbounded loops
  - type inference
- runtime safety features
  - stap enforces maximum run time for each probe handler
  - various concurrency constraints are enforced
  - overload processing (don't allow stap to take up all the CPU time)
  - many things can be overriden manually if you really want
  - see `SAFETY AND SECURITY` section of `stap(1)`

Agenda

SystemTap vs DTrace

SystemTap is often described as "DTrace for Linux". I never used DTrace but from what I read...

- DTrace is specific to Solaris, FreeBSD, NetBSD and OS X (for now; SystemTap is just for Linux)
- DTrace won't allow you to inject arbitrary C code (no -g)
- DTrace won't allow you to probe arbitrary symbolic statements? (limited to functions boundaries and explicit markers?)
- DTrace scripts are interpreted by a virtual machine in the kernel (no loading of binary module)

SystemTap vs auditd(8)

- auditd can only trace system calls
- auditd cannot really do any proper decoding/filtering of the syscall arguments
- auditd might be easier to set up on your distro

The sigkill.stp example with auditd:

```
# auditctl -a entry,always -S kill -F a1=9
```

And what it looks like in the logs (killing `sleep(1)`):

```
type=SYSCALL msg=audit(1275595476.234:430): arch=40000003
    syscall=37 success=yes exit=0 a0=5b25 a1=9 a2=5b25
    a3=5b25 items=0 ppid=23188 pid=23189 auid=500 uid=500
    gid=500 euid=500 suid=500 fsuid=500 egid=500 sgid=500
    fsgid=500 tty=pts2 ses=35 comm="bash" exe="/bin/bash"
    subj=user_u:system_r:unconfined_t:s0 key=(null)
type=OBJ_PID msg=audit(1275595476.234:430): opid=23333
    oauid=500 ouid=500 oses=35
    obj=user_u:system_r:unconfined_t:s0 ocomm="sleep"
```

SystemTap vs userland tools

- strace can only handle system calls
- ltrace can only handle userland functions
- {s,l}trace can only monitor specific processes
- {s,l}trace cannot process traces on the fly (statistics, advanced filtering,...)
- gdb is more aimed at interactive debugging

References and questions

- SystemTap wiki: http://sourceware.org/systemtap/wiki
- lot of excellent documentation included:
  - `man {stap,stapprobes,stapfuncs,stapvars,...}`
  - file:///usr/share/doc/systemtap*
- there is probably already a script to do what you want: http://sourceware.org/systemtap/examples/
- systemtap@sources.redhat.com
- irc://chat.freenode.net/#systemtap

Agenda

- Explaining SystemTap
- Practical Examples
- Requirements and safety
- Comparison to Other Tools
- Conclusion
- More examples?

Example 4: callgraph for anything

Listing 4: examples/general/para-callgraph.stp (simplified a bit)

```
1  function trace(entry_p, extra) {
2          printf("%s%s%s %s\n",
3                  thread_indent (entry_p),
4                  (entry_p>0?"->":"<-"),
5                  probefunc (),
6                  extra)
7  }
8
9  probe $1.call   { trace(1, $$parms) }
10 probe $1.return { trace(-1, $$return) }
```

Example 4: using para-callgraph.stp

```
# stap examples/general/para-callgraph.stp
    'process("/usr/sbin/sendmail").function("*")'
     0 sendmail(4523):->doqueuerun
  1736 sendmail(4523):<-doqueuerun return=0x0
     0 sendmail(4523):->sm_blocksignal sig=0xe
    56 sendmail(4523):<-sm_blocksignal return=0x0
     0 sendmail(4523):->curtime
    22 sendmail(4523):<-curtime return=0x4c06fb34
     0 sendmail(4523):->refuseconnections name=0x93ad8b0
         e=0x343a80 d=0x0 active=0x0
    59 sendmail(4523): ->sm_getla
   109 sendmail(4523):  ->getla
   930 sendmail(4523):   ->sm_io_open type=0x3432c0
       timeout=0xfffffffffffffffe info=0x3231cd flags=0x2
       rpool=0x0
  1733 sendmail(4523):    ->sm_flags flags=0x2
  1771 sendmail(4523):    <-sm_flags return=0x10
  1876 sendmail(4523):    ->sm_fp t=0x3432c0 flags=0x10
      oldfp=0x0
 12409 sendmail(4523):    <-sm_fp return=0x372d7c
```

Example 5: block I/O requests monitoring

Listing 5: examples/io/ioblktime.stp (part 1 of 2)

```
1  global req_time, etimes
2
3  probe ioblock.request {
4    req_time[$bio] = gettimeofday_us()
5  }
6
7  probe ioblock.end {
8    t = gettimeofday_us()
9    s =  req_time[$bio]
10   delete req_time[$bio]
11   if (s) {
12     etimes[devname, bio_rw_str(rw)] <<< t - s
13   }
14 }
```

This is just to collect the data (no printing).

Example 5 continued: printing the collected data

Listing 6: continuation of examples/io/ioblktime.stap (part 2 of 2)

```
15 probe timer.s(10), end {
16   ansi_clear_screen()
17   printf("%10s %3s %10s %10s %10s\n",
18          "device", "rw", "total (us)", "count", "avg (us)")
19   foreach ([dev,rw] in etimes- limit 20) {
20     printf("%10s %3s %10d %10d %10d\n", dev, rw,
21            @sum(etimes[dev,rw]), @count(etimes[dev,rw]),
22            @avg(etimes[dev,rw]))
23   }
24   delete etimes
25 }
```

Example 5 continued: what it looks like

```
# stap examples/io/ioblktime.stp
 device   rw total (us)        count    avg (us)
    sda    W  270301266         2160      125139
   dm-0    W  270344450         2160      125159
   dm-0    R      30010            4        7502
    sda    R      28615            4        7153
```