

Introduction to SystemTap

A practical approach

Breno Leitão
breno.leitao@gmail.com
Linux Technology Center

June 19, 2009



Agenda

Introduction

How it works

Installation

Core

TapSets

Advanced Topics

What is SystemTap

SystemTap is a tool for the Linux Operating System that allows developers and system administrators to deeply investigate the behavior of the kernel and even user space applications in order to discover error conditions, performance issues, or just to understand how the system works, similarly to DTrace.

Tracing

What is tracing:

- ▶ Understand application's behavior
- ▶ Check the variables and memory position
- ▶ Check which function is being called, and how often
- ▶ Check the call site, and function parameters
- ▶ Used when it's not possible to use a interactive debugger.

More about SystemTap

- ▶ It was created by Red Hat, IBM, Intel, Hitachi, Oracle.
- ▶ Runs on x86, x86_64, Power PC® , and System 390.
- ▶ It is licenced under GPL.

Benefits

- ▶ Easy to summarize and visualize information
- ▶ Without requiring you to do a tedious kernel backup, modification, compile, install and reboot
- ▶ Script based
- ▶ Fast

Script

SystemTap run a script which define events and handlers.

- ▶ Similar to AWK and C
- ▶ Easy and simple
- ▶ It is not designed to be an extensive, general purpose programming language.
- ▶ Divided in blocks

<http://sourceware.org/systemtap/langref/>

How to use

```
[root@sanx1003 systemtap]# stap foo.stp
Hello World.
[root@sanx1003 systemtap]# cat foo.stp
probe begin{
    print("Hello World.\n")
    exit()
}
```

Dissection of a script

```
function is_open_creating:long (flag:long){
    CREAT_FLAG = 4 // 0x4 = 00000100b
    if (flag & CREAT_FLAG)
        return 1
    return 0
}

probe kernel.function("sys_open") {
    creating = is_open_creating($mode)
    if (creating)
        printf("Creating file %s\n", user_string($filename))
    else
        printf("Opening file %s\n",
            user_string($filename))
}
```

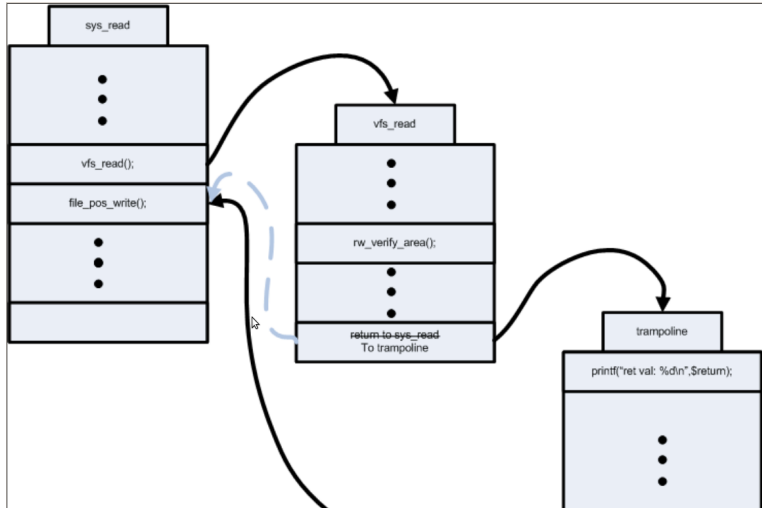
Steps

1. Translates the .stp script into the C source code. This implements the instrumentation using kprobe.
2. Compiles the C file to create the module (.ko file).
3. Loads the module into the kernel.
4. The module runs, reporting events of interest as specified by the script. Stap collects this information from the kernel and reports it to stdout.
5. The session ends and the module is unloaded when you send stap a CTRL-C, or the module decides it is done.

Kprobe, jprobe and kretprobes - the magicians

- ▶ Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively.
- ▶ You can trap at almost any kernel code address, specifying a handler routine to be invoked when the breakpoint is hit
- ▶ kprobe - any instruction in the kernel.
- ▶ jprobe - entry to a kernel function (function parameters)
- ▶ kretprobe - fires when a specified function returns

Kretprobe example



What is required to run SystemTap

- ▶ The SystemTap binaries
- ▶ A kernel with debuginfo section

Installation on distro

Ubuntu

- ▶ `sudo apt-get install systemtap`
- ▶ `sudo apt-get install linux-image-debug-generic`
- ▶ `sudo ln -s /boot/vmlinuz-debug-$(uname -r)
/lib/modules/$(uname -r)/vmlinuz`

SystemTap devel version

- ▶ `git clone git://sources.redhat.com/git/systemtap.git`
- ▶ `git clone git://git.fedorahosted.org/git/elfutils.git`
- ▶ `cd systemtap ; ./configure --with-elfutils=../elfutils`

Building a custom Kernel

Enable the following options on your `.config`:

- ▶ `CONFIG_DEBUG_FS`
- ▶ `CONFIG_DEBUG_KERNEL`
- ▶ `CONFIG_DEBUG_INFO`
- ▶ `CONFIG_KPROBES`

Event and handlers

SystemTap monitors the operating system for the specified events and executes the handlers as they occur.

Events are:

- ▶ Any kernel function¹
- ▶ Any line in any source code
- ▶ In any time
- ▶ In any event defined by a tapset
 - ▶ `tcp.receive`
 - ▶ `syscall.close`
 - ▶ `nfs.fop.llseek`
 - ▶ `vm.oom_kill`

¹Some exceptions as inlines

Sketch

Probing a core kernel function

```
probe kernel.function("XXX"){  
  // do something  
}
```

or/and

Probing a module function

```
probe module("module_name").function("XXX"){  
  // do something  
}
```

Special probe points

Begin and End special probe points

```
probe begin{  
  \\ Executed before anything else  
}  
  
....  
probe end{  
  \\ Executed when the script ends  
}
```

When the script ends?

Condition probing

```
probe kernel.function("__kmalloc").return {  
    if (i < 10){  
        printf("Hooked\n")  
    } else {  
        exit()  
    }  
}
```

First example

Example

```
probe module("ext3").function("*").call {  
    printf("\%s -> \%s\n", thread_indent(1), probefunc())  
}  
probe module("ext3").function("*").return {  
    printf("\%s <- \%s\n", thread_indent(-1), probefunc())  
}
```

Output of first example

```
0 ls(6218): -> ext3_permission
3 ls(6218): <- ext3_permission
0 ls(6218): -> ext3_dirty_inode
3 ls(6218): -> ext3_journal_start_sb
8 ls(6218): <- ext3_journal_start_sb
12 ls(6218): -> ext3_mark_inode_dirty
16 ls(6218): -> ext3_reserve_inode_write
20 ls(6218): -> ext3_get_inode_loc
23 ls(6218): -> __ext3_get_inode_loc
28 ls(6218): <- __ext3_get_inode_loc
31 ls(6218): <- ext3_get_inode_loc
35 ls(6218): <- ext3_reserve_inode_write
39 ls(6218): -> ext3_mark_iloc_dirty
43 ls(6218): <- ext3_mark_iloc_dirty
```

Second example

Example

```
probe syscall.mkdir{
    printf("Creating directory %s using mode %d...",
           user_string($pathname), $mode)
}
probe syscall.mkdir.return {
    if (!$return)
        printf("created\n")
    else
        printf("failed\n")
}
```

Output of second example

```
Creating file foo using mode 511...failed  
Creating file /tmp/foo using mode 511...created  
Creating file /tmp/stapgIfyVs using mode 448...created
```

Third example

Example

```
probe kernel.function("path_get").return {
    if($return < 0) {
        printf("In process [%s]\n", execname())
        print_regs()
        print_backtrace()
        print("-----\n")
    }
}
```

Output of second example

```
In process [ls]
NIP: C00000000012FA84 XER: 20000000 LR: C000000001340B4 CTR: 000000000000FED
REGS: c0000001f23976f0 TRAP: 0700
MSR: 8000000000029032 CR: 22000248
DAR: 0000000000000000 DSISR: 00000000ffdd058
CPU: 0
GPR00: C0000001F2D0E80 C0000001F2397970 C0000000007D0670 C0000001F23979E0
GPR04: C0000001F2397B50 0000000000000001 C0000001F2397B50 0000000000000000
...
GPR16: 0000000000000000 0000000000000000 000000000FFF0008 0000000000000000
GPR20: 000000000FFDD058 000000000FFF0944 0000000000000000 0000000000000000
GPR24: 00000000FFFFFFF C0000001DD5F0980 C0000001F23979E0 C0000001DD5F0980
GPR28: C0000001F2397B50 C0000001F21D2B44 C00000000077E7E0 C0000001F2397970
NIP [c00000000012fa84] LR [c000000001340b4]
0xc00000000012fa84 : .path_get+0x0/0x90 [kernel]
[0xc0000001f23976f8] [0xc0000001f23979e0] 0xc0000001f23979e0 :
klist_next+0x1f1bcf388/0x0 [kernel] (unreliable)
[0xc0000001f2397970] [0xc0000001f2397a10] 0xc0000001f2397a10 :
klist_next+0x1f1bcf3b8/0x0 [kernel]
...
[0xc0000001f2397da0] [0xc0000000001a7a8] 0xc0000000001a7a8 :
.compat_sys_access+0x38/0x58 [kernel]
[0xc0000001f2397e30] [0xc00000000008534] 0xc00000000008534 :
.start_here_common+0x198/0x354 [kernel]
--- Exception: c00 at 0x00000000ffdd88c4
LR =0x00000000ffc41ec
-----
```

Third example

task_curr() at kernel/sched.c

```
inline int task_curr(const struct task_struct *p)
{
    return cpu_curr(task_cpu(p)) == p;
}
```

Example

```
probe kernel.function("task_curr"){
    printf("Current pid %d in ", $p->pid)
    if (!$p->state)
        printf("runnable state\n")
    else if ($p->state > 0)
        printf("stopped state\n")
    else // -1
        printf("unrunnable state\n")
}
```

Output of third example

```
Current pid 5463 in stopped state  
Current pid 15487 in stopped state  
Current pid 15487 in stopped state  
Current pid 15487 in stopped state  
Current pid 5353 in stopped state
```

Timer example

Example

```
probe timer.ms(500){
    printf("MS: %s\n", ctime(gettimeofday_s()))
}
probe timer.s(1){
    printf("S: %s\n", ctime(gettimeofday_s()))
}
```

Output of Timer example

```
MS: Tue Nov 11 16:20:47 2008  
MS: Tue Nov 11 16:20:47 2008  
S: Tue Nov 11 16:20:47 2008  
MS: Tue Nov 11 16:20:48 2008  
MS: Tue Nov 11 16:20:48 2008  
S: Tue Nov 11 16:20:48 2008  
MS: Tue Nov 11 16:20:49 2008  
MS: Tue Nov 11 16:20:49 2008  
S: Tue Nov 11 16:20:49 2008  
MS: Tue Nov 11 16:20:50 2008  
MS: Tue Nov 11 16:20:50 2008  
S: Tue Nov 11 16:20:50 2008  
MS: Tue Nov 11 16:20:51 2008
```

Profile example

Example

```
probe timer.profile {  
    printf("cpu: %d - process: %s\n",cpu(),execname())  
}
```

Output of profile example

```
cpu: 0 - process: sshd  
cpu: 1 - process: find  
cpu: 1 - process: find  
cpu: 0 - process: sshd  
cpu: 0 - process: swapper  
cpu: 1 - process: find  
cpu: 0 - process: sshd  
cpu: 1 - process: find  
cpu: 0 - process: sshd  
cpu: 1 - process: find  
cpu: 1 - process: find  
cpu: 0 - process: sshd
```

Timing syscall example

Example

```
global time
probe syscall.open {
    time[tid()] = gettimeofday_ns()
}
probe kernel.function("sys_open").return {
    if(time[tid()]) {
        printf("sys_open took %d ns to execute\n", g
time[tid()])
        delete time[tid()]
    }
}
```

Output of timing syscall example

```
sys_open took 6002 ns to execute  
sys_open took 5686 ns to execute  
sys_open took 5628 ns to execute  
sys_open took 5707 ns to execute  
sys_open took 5362 ns to execute  
sys_open took 6234 ns to execute  
sys_open took 5777 ns to execute  
sys_open took 11314 ns to execute  
sys_open took 1091570 ns to execute
```

Output of timing syscall example

Tapsets

Tapsets in SystemTap are libraries of useful functions, and predefined probes that may be useful in writing scripts. They can serve to simplify scripts when the implementation details of gathering useful data from a particular kernel function are complicated.

Example:

- ▶ TCP/IP probes
- ▶ Userspace addresses
- ▶ `retstr = returnstr()`

How to use Tapsets

Tapsets do not have to be explicitly included. The tapsets installed on a system can be found in the `/usr/share/systemtap/tapset` directory.

Just use it

Example

A Tapset for usb_submit_urb^a

^aIssue an asynchronous transfer request for an endpoint

```
probe usb.submit_urb = kernel.function("usb_submit_urb")
{
    urb = $urb
    dev = $urb->dev
    flags = $mem_flags
}
```

Example

```
probe usb.submit_urb
{
    printf("usb_submit_urb called on device at %x\n", dev)
}
```

A lot of Tapsets available

There are a lot of tapset already implemented

- ▶ System Calls
- ▶ Signals
- ▶ Networking
- ▶ Virtual memory
- ▶ NFS

Find more on </usr/share/systemtap/tapset>

Guru mode

SystemTap supports some advanced features that are enabled in the guru mode. In this mode, which is enabled when you run SystemTap using the `-g` argument, you do not have the memory and data protection normally provided by the SystemTap access restrictions.

Running on guru mode

```
# stap -g myultrascript.stp
```

SystemTaping for fun and profit

In guru mode you can change the behavior of the kernel, altering how it executes a function. The example above demonstrates how powerful SystemTap is. In this example, the write syscall is probed and the output is changed, so that the first six characters are not displayed. To do so, the buf pointer is shifted 6 characters.

write syscall

write syscall

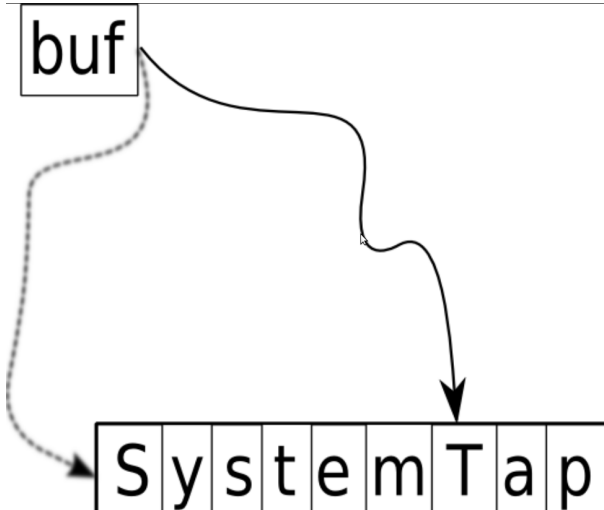
```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,  
                size_t, count) {  
    struct file *file;  
    ssize_t ret = -EBADF;  
    int fput_needed;  
  
    file = fget_light(fd, &fput_needed);  
    if (file) {  
        loff_t pos = file_pos_read(file);  
        ret = vfs_write(file, buf, count, &pos);  
        file_pos_write(file, pos);  
        fput_light(file, fput_needed);  
    }  
}  
  
return ret;
```

Guru mode

Example

```
probe syscall.write {
    if (isinstr(user_string($buf), "SystemTap")){
        $buf = $buf + 0x06 // Shift 6 characters
    }
}
```

Kretprobe example



Guru mode

Running on guru mode

```
# echo SystemTap  
Tap  
#
```

Thank you!
Doubts!?