Vmcore Analysis Training Primer

Lesson 3

Written and developed by : **Stephen Johnston**
Principal Software Maintenance Engineer
Red Hat North America GSS-SEG
October 2014

WARNING

The information contained in this course may contain proprietary and confidential information from customers, vendors and Red Hat.

UNDER NO CIRCUMSTANCE IS ANY OF THIS INFORMATION TO BE SHARED WITH ANYONE ELSE INCLUDING CUSTOMERS, FRIENDS, OTHER EMPLOYEES. NO EXCEPTIONS.

There is no inferred permission nor authorization given for making any copies of this material. Do not not share it with anyone. If someone is interested in this material/course please have them contact Red Hat North America GSS-SEG management or the author.

Table of Contents

3.2.3 - stap

3.2.3.1 - What needs to be installed to support stap

First. Checking the correct packages are installed

You do need the following packages (kernel release isn't the issue here, it's that they need **kernel-debuginfo** [and **kernel-debuginfo-common** if RHEL6]):

```
kernel-2.6.32-279.14.1.el6.x86_64                              Sat 02 Feb 2013 12:44:36 AM EST
kernel-debuginfo-2.6.32-279.14.1.el6.x86_64                    Tue 04 Jun 2013 12:41:47 PM EDT
kernel-debuginfo-common-x86_64-2.6.32-279.14.1.el6.x86_64      Tue 04 Jun 2013 12:40:40 PM EDT
kernel-devel-2.6.32-279.14.1.el6.x86_64                        Tue 04 Jun 2013 12:42:43 PM EDT
```

You need **stap** of course:

```
systemtap-1.8-7.el6.x86_64                                     Mon 03 Jun 2013 06:35:11 PM EDT
systemtap-client-1.8-7.el6.x86_64                              Mon 03 Jun 2013 06:35:11 PM EDT
systemtap-devel-1.8-7.el6.x86_64                               Mon 03 Jun 2013 06:35:08 PM EDT
systemtap-runtime-1.8-7.el6.x86_64                             Sat 01 Jun 2013 10:24:36 PM EDT
```

You also need to have **gcc**

```
gcc-4.4.7-3.el6.x86_64                                         Mon 03 Jun 2013 06:35:07 PM EDT
libgcc-4.4.7-3.el6.i686                                        Sat 01 Jun 2013 10:23:32 PM EDT
libgcc-4.4.7-3.el6.x86_64                                      Sat 01 Jun 2013 10:21:52 PM EDT
```

If in doubt, perform a **yum** upgrade:

```
# yum install kernel-devel kernel-debuginfo kernel-debuginfo-common gcc systemtap
systemtap-runtime
```

yum will ignore packages already up-to-date.

Testing that stap packages are all installed correctly

To test if it's installed correctly, try this:

```
# stap -v -e 'probe vfs.read {printf("\nstap test: read performed\n\n"); exit()}'
```

It should report the following 5 phases. Obviously if it does not report Pass 5, somethings wrong

```
Pass 1: parsed user script and 68 library script(s) using 20400virt/12480res/2080shr kb, in
320usr/90sys/754real ms.
Pass 2: analyzed script: 1 probe(s), 11 function(s), 3 embed(s), 1 global(s) using
212600virt/133872res/77256shr kb, in 2360usr/640sys/3748real ms.
Pass 3: translated to C into "/tmp/stapfOwkvX/stap_c5d8d2140cb8593b00da32b0f460fdfe_5574.c"
using 212600virt/135136res/78520shr kb, in 1580usr/60sys/1863real ms.
Pass 4: compiled C into "stap_c5d8d2140cb8593b00da32b0f460fdfe_5574.ko" in
10630usr/3410sys/37735real ms.
Pass 5: starting run.

stap test: read performed

Pass 5: run completed in 20usr/100sys/321real ms.
```

Problem persists, what other information can you gather?

Try one or all of the following:

```
# stap -L 'kernel.trace("*")' >stap_trace.log
# stap -vvvv -L 'kernel.trace("*")' >stap_trace_vvvv.log
# stap-report
# stap -vvvv -g dropwatch_src.stp 2>&1
```

The first will list all kernel.trace packages

```
# stap -L 'kernel.trace("*")'
kernel.trace("__extent_writepage") $page:struct page* $inode:struct inode* $wbc:struct
writeback_control*
kernel.trace("block_bio_backmerge") $q:struct request_queue* $bio:struct bio*
kernel.trace("block_bio_bounce") $q:struct request_queue* $bio:struct bio*
kernel.trace("block_bio_complete") $q:struct request_queue* $bio:struct bio*
kernel.trace("block_bio_frontmerge") $q:struct request_queue* $bio:struct bio*
kernel.trace("block_bio_queue") $q:struct request_queue* $bio:struct bio*
kernel.trace("block_getrq") $q:struct request_queue* $bio:struct bio* $rw:int
kernel.trace("block_plug") $q:struct request_queue*
- - - - - - - - 8< - - - - - - - - -
```

The second will provide a very verbose debug of the listing

```
# stap -vvvv -L 'kernel.trace("*")'
Systemtap translator/driver (version 1.8/0.152 non-git sources)
Copyright (C) 2005-2012 Red Hat, Inc. and others
```

```
This is free software; see the source for copying conditions.
enabled features: AVAHI LIBRPM LIBSQLITE3 NSS TR1_UNORDERED_MAP NLS
Created temporary directory "/tmp/stap70XEIP"
Session arch: x86_64 release: 2.6.32-358.6.2.el6.x86_64
Parsed kernel "/lib/modules/2.6.32-358.6.2.el6.x86_64/build/.config", containing 3166
tuples
Parsed kernel /lib/modules/2.6.32-358.6.2.el6.x86_64/build/Module.symvers, which
contained 5541 vmlinux exports
Searched: " /usr/share/systemtap/tapset/x86_64/*.stp ", found: 4, processed: 4
Searched: " /usr/share/systemtap/tapset/*.stp ", found: 82, processed: 82
Pass 1: parsed user script and 86 library script(s) using
195356virt/24284res/3104shr/21288data kb, in 100usr/0sys/106real ms.
Attempting to extract kernel debuginfo build ID from /lib/modules/2.6.32-
358.6.2.el6.x86_64/build/vmlinux.id
- - - - - - - - 8< - - - - - - - - -
```

The third tells us about their stap version

```
# stap-report
== stap -V ==
Systemtap translator/driver (version 1.8/0.152 non-git sources)
Copyright (C) 2005-2012 Red Hat, Inc. and others
This is free software; see the source for copying conditions.
enabled features: AVAHI LIBRPM LIBSQLITE3 NSS TR1_UNORDERED_MAP NLS
== which stap ==
/usr/bin/stap
== locate --regex '/stap(run)?$' | xargs ls -ald ==
-rwxr-xr-x. 1 root root    2135296 Dec 13 09:10 /usr/bin/stap
---s--x---. 1 root stapusr  162584 Dec 13 09:10 /usr/bin/staprun
== printenv | egrep '^PATH=|^LD_LIBRARY_PATH=|^SYSTEMTAP_.*=|^XDG_DATA.*=' ==
PATH=/usr/lib64/qt-
3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
== stap -vv -p4 -e 'probe begin {exit()}' ==
- - - - - - - - 8< - - - - - - - - -
```

The fourth, executes the script (**-g** guru mode is required in this specific script specifically because it has embedded c code in it. It has nothing to do with the **-v** needed for verbose debugging) and provides a very verbose debug of each of the 5 phases (passes). Here's the script also used in this example:

```
# cat -v -t dropwatch_src.stp
#!/usr/bin/stap

# Array to hold the list of drop points we find
global locations

%{
#include <linux/tcp.h>
```

```
#include <linux/ip.h>
#include <linux/skbuff.h>
#include <net/sock.h>
%}

function srcport:long(skb:long)
%{
   struct tcphdr *th;
   th = tcp_hdr((struct sk_buff *)THIS->l_skb);
   THIS->__retvalue = (long) th->source;
%}

function srcaddr:long(skb:long)
%{
   struct iphdr *iph;
   iph = ip_hdr((struct sk_buff *)THIS->l_skb);
   THIS->__retvalue = (long) iph->saddr;
%}

function family:long(skb:long)
%{
   struct sock* sk;
   sk = ((struct sk_buff *) THIS->l_skb)->sk;
   if (sk)
      THIS->__retvalue = sk->__sk_common.skc_family;
   else
      THIS->__retvalue = 0;
%}

# Note when we turn the monitor on and off
probe begin { printf("Monitoring for dropped packets\n") }
probe end { printf("Stopping dropped packet monitor\n") }

probe kernel.trace("kfree_skb") {
   if (family($skb) == 2){
      port = srcport($skb)
      addr = srcaddr($skb)
      locations[addr,port,$location] <<<1
   }
}

# Every 5 seconds report our drop locations
probe timer.sec(5)
{
        printf("\n%s\n", ctime(gettimeofday_s()))
        foreach ([addr,port,l] in locations-) {
                printf("\t%d (%x, %x)\t%s (%p)\n", @count(locations[addr,port,l]),
addr, port, symname(l), l)
        }
        delete locations
```

```
}
```

```
# stap -vvvv -g dropwatch_src.stp
Systemtap translator/driver (version 1.8/0.152 non-git sources)
Copyright (C) 2005-2012 Red Hat, Inc. and others
This is free software; see the source for copying conditions.
enabled features: AVAHI LIBRPM LIBSQLITE3 NSS TR1_UNORDERED_MAP NLS
Created temporary directory "/tmp/stapiDtSvW"
Session arch: x86_64 release: 2.6.32-358.6.2.el6.x86_64
Parsed kernel "/lib/modules/2.6.32-358.6.2.el6.x86_64/build/.config", containing 3166
tuples
Parsed kernel /lib/modules/2.6.32-358.6.2.el6.x86_64/build/Module.symvers, which
contained 5541 vmlinux exports
Searched: " /usr/share/systemtap/tapset/x86_64/*.stp ", found: 4, processed: 4
Searched: " /usr/share/systemtap/tapset/*.stp ", found: 82, processed: 82
Pass 1: parsed user script and 86 library script(s) using
195692virt/24300res/3100shr/21624data kb, in 100usr/0sys/120real ms.
- - - - - - - - 8< - - - - - - - - -
```

What are those 5 phases (Pass 1 thru 5)?

Q. Can you step through a stap run and see the various passes/phases?

A. Yes. You can run a **stap** script with **-p <*n*>** providing a number from 1-5, e.g stap -p2 myscript.stp. This will stop after phase/pass 2. Stopping a script after a phase also provides for additional output to be displayed for that pass level ONLY. This information is not normally seen but is basic debug data (assumed you want it because you are stopping after that phase/pass level). See each pass below for clarification as to what they provide.

pass 1

The translator begins pass 1 by parsing the given input script, and all scripts (files named ***.stp**) found in a tapset directory. The directories listed with **-I** are processed in sequence, each processed in "guru mode". For each directory, a number of subdirectories are also searched. These subdirectories are derived from the selected kernel version (the **-R** option), in order to allow more kernel-version-specific scripts to override less specific ones. For example, for a kernel version 2.6.12-23.FC3 the following patterns would be searched, in sequence: **2.6.12-23.FC3/*.stp**, **2.6.12/*.stp**, **2.6/*.stp**, and finally ***.stp**

Stopping the translator after pass 1 causes it to print the parse trees.

pass 2

In pass 2, the translator analyzes the input script to resolve symbols and types. References to variables, functions, and probe aliases that are unresolved internally are satisfied by searching through the parsed tapset script files. If any tapset script file is selected because it defines an unresolved symbol, then the entirety of that file is added to the translator's resolution queue. This process iterates until all symbols are resolved and a subset of tapset script files is selected.

Next, all probe point descriptions are validated against the wide variety supported by the translator. Probe points that refer to code locations ("synchronous probe points") require the appropriate kernel debugging information to be installed. In the associated probe handlers, target-side variables (whose names begin with "$") are found and have their run-time locations decoded.

Next, all probes and functions are analyzed for optimization opportunities, in order to remove variables, expressions, and functions that have no useful value and no side-effect. Embedded-C functions are assumed to have side-effects unless they include the magic string /* pure */. Since this optimization can hide latent code errors such as type mismatches or invalid $target variables, it sometimes may be useful to disable the optimizations with the **-u** option.

Finally, all variable, function, parameter, array, and index types are inferred from context (literals and operators).

Stopping the translator after pass 2 causes it to list all the probes, functions, and variables, along with all inferred types. Any inconsistent or unresolved types cause an error.

pass 3

In pass 3, the translator writes C code that represents the actions of all selected script files, and creates a Makefile to build that into a kernel object. These files are placed into a temporary directory.

Stopping the translator at this point causes it to print the contents of the .C file.

pass 4

In pass 4, the translator invokes the Linux kernel build system to create the actual kernel object file. This involves running make in the temporary directory, and requires a kernel module build system (headers, config and Makefiles) to be installed in the usual spot /lib/modules/VERSION/build.

Stopping the translator after pass 4 is the last chance before running the kernel object. This may be useful if you want to archive the file.

pass 5

In pass 5, the translator invokes the systemtap auxiliary program staprun program for the given kernel object. This program arranges to load the module then communicates with it, copying trace data from the kernel into temporary files, until the user sends an interrupt signal. Any run-time error encountered by the probe handlers, such as running out of memory, division by zero, exceeding nesting or runtime limits, results in a soft error indication. Soft errors in excess of **MAXERRORS** block of all subsequent probes (except errorhandling probes), and terminate the session. Finally, staprun unloads the module, and cleans up.

3.2.3.2 - Some Example scripts:

Here is an example of a number of things.

- Embedded C code is surrounded by %{ and %}. In this example, we're merely using a include statement.
- An example of **using cast to assign a structure** and/or a struct element to a variable
- An example of how to simply **print a structure**
- **Access variables/elements** that exist in the probed function itself

Before we go further let me explain the cast. You should notice that this script is using an element called **%sd** (highlighted). What is that??? if you look in the source code of the probed function you will discover:

```
find_busiest_group(struct sched_domain *sd, int this_cpu,
                   unsigned long *imbalance, enum cpu_idle_type idle,
                   int *sd_idle, const struct cpumask *cpus, int *balance)
```

You'll notice I highlighted the **sched_domain** structure assigned a pointer ***sd**. Therefore as this element is accessible/visible to the function, you can use it directly in your stap script. Yes, that means anything locally visible to the source code is directly usable in your script too without redefinition etc.

What then is a cast? Cast allows you to assign other elements not directly used in a function but which do have an indirect link. That is, "**sched_group**" below is not in the probed function but "**sched_domain**" is and it has a "**sched_group**" structure pointer in its structure. Therefore, there's no direct variable reference in the probed function so you have to create a variable in your stap to assign it. That's what the cast does. As you can see, you can not only assign the struct, you can assign a variable in that substructure.

One other point. Highlighted in yellow is something else you can do. You can supply a structure pointer and print (display) it in your stap script also.

```
#!/usr/bin/env stap

%{
#include <linux/sched.h>
%}

probe kernel.function("find_busiest_queue")  {

        start = curr = $sd->groups;
        power = $sd->groups->cpu_power;

        printf("Initial Current group address = 0x%x. CPU Power is %d\n", start, power);

        curr = $sd->groups->next;
```

```
        while (curr != start) {
                power = @cast(curr, "sched_group")->cpu_power;
                printf("Current group address = 0x%x. CPU Power is %d\n", curr, power);

                curr = @cast(curr, "sched_group")->next;
                printf("Struct is %s\n", $sd$$);
        }
}
```

Here's another example that shows an example of **calling a kernel callable function**, in this case **panic()**

```
%{
#define LOAD_INT(x) ((x) >> FSHIFT)
%}

function panic(msg:string) %{
      panic("%s", STAP_ARG_msg);
%}

function panic_string:string(nr_active:long, threshold:long) {
      return sprintf("nr_active: %d: is >= %d\n",
            nr_active, threshold);
}

probe begin {
      printf("Starting tracing...\n");
}

function get_avenrun:long () %{
      STAP_RETVALUE = LOAD_INT(avenrun[0] + (FIXED_1/200));
%}

probe kernel.function("nr_active").return {
      if ($return >= $1 && get_avenrun() >= $1)
            panic(panic_string($return, $1))
}
```

This following example highlights some more usefuly stap features:

- How to pass **parameters into your stap script**
- Using **begin and end** to cleanly wrap around your stap script
- How to **group call and returns** into probes
- It shows **the timer function** which can terminate the stap script for you after a select period of time.

```
#!/usr/bin/env stap
#
# Written by Steve Johnston
#            Principal Software Maintenance Engineer, RedHat North America GSS-SEG
#
# Purpose of the script is to time/test the sys_nanosleep() call sequence to see how
#         long we are deep in the kernel and also where we are stalling.
#         As there are 3000+ sleep processes in play, let's start by timing and
#         monitoring a specific single case (use 'pid nnnnn xxxx')
#
#                                    @1    @2       $3
# run:        stap -g nanosleep.stp pid  <nr>   <secs timer>     (timer = 1 to 86400)
#        or   stap -g nanosleep.stp name <proc> <secs timer>     (timer = 1 to 86400)

global target_pid = 999999
global target_name = ""

probe begin {
      printf("nanosleep stap script starting monitoring %s = %s for %d secs\nPress
Ctrl/C to terminate\n\n", @1, @2, $3)
      %( $# == 3 %?
         if(@1 == "pid")
         target_pid = strtol(@2, 10)
         if(@1 == "name")
         target_name = @2
      %)
}

probe end {
       printf("\n\nDisabling nanosleep Probes.\n\n");
}


# trace all functions called by sys_nanosleep()

probe kernel.function("sys_nanosleep").call,
      kernel.function("hrtimer_wakeup").call,
      kernel.function("hrtimer_nanosleep").call,
      kernel.function("do_nanosleep").call,
      kernel.function("hrtimer_try_to_cancel").call,
      kernel.function("hrtimer_start").call,
```

```
      kernel.function("enqueue_hrtimer").call {
      if ( pid() == target_pid || execname() == target_name ) {
         message = sprintf("%s : %d | %s -> %-30s", ctime(gettimeofday_s()),
             gettimeofday_ns(), thread_indent(0), probefunc());
       printf("%s\n", message);
      }
}

probe kernel.function("sys_nanosleep").return,
      kernel.function("hrtimer_wakeup").return,
      kernel.function("hrtimer_nanosleep").return,
      kernel.function("do_nanosleep").return,
      kernel.function("hrtimer_try_to_cancel").return,
      kernel.function("hrtimer_start").return,
      kernel.function("enqueue_hrtimer").return {
      if ( pid() == target_pid || execname() == target_name ) {
         message = sprintf("%s : %d | %s <- %-30s", ctime(gettimeofday_s()),
             gettimeofday_ns(), thread_indent(0), probefunc());
       printf("%s\n", message);
      }
}

probe timer.sec($3) {
      printf ("\nTermination timer fired after %d secs\n", $3)
      exit()
}
```

This following example has multiple aspects to it:

- **Passing arguments on the command line**
- **Creating dynamic arrays** to hold information and then deleting them
- Using a **C-like "IF ? TRUE : FALSE"** statement
- **Timing how long it takes a TID to process in a piece of code** by timing when it enters and exits.

I want to point out a simple technique this script shows. **pfaults.stp** is probing a kernel call and kernel return for a function. We want to determine how long we are IN that function. So a timestamp is acquired on entry and one on exit. However, how do you know it's for the same process? You don't. Some page faults may take only nanoseconds, others might take microseconds and other milliseconds. With a large Multi CPU environment, you don't know how long each will take or how many are actively and concurrently handing page faults. So this stap saves the information in an array element based on the TID. Why not the PID? Because multiple threads can run under the same PID, we need the time window for a TID to be sure of uniqueness. The script therefore stores various pieces of collected data in a number of arrays.

```
# cat pfaults.stp
#! /usr/bin/env stap

#      Steve Johnston
#      Principal Software Maintenance Engineer
#      Red Hat North America GSS-SEG

# You can run this script in one of 3 ways
#
#      stap -g pfaults.stp              Report on all page faults
#      stap -g pfaults.stp tid <number>       Report on page faults for tid = number
#      stap -g pfaults.stp name <execname>    Report on page faults for all running
<execname> processes

global fault_entry_time, fault_address, fault_access, execnames
global target_tid = 0, target_name = "", target = 0

probe begin {

%( $# == 1 %?
  log("Wrong number of arguments, use none, or 'tid nr', or 'name proc'")
  exit()
%)

%( $# == 2 %?
  if(@1 == "tid") {
     target_tid = strtol(@2, 10)
        target = 1
  }
  if(@1 == "name") {
```

```
    target_name = @2
      target = 1
  }
  if(target == 0) {
    log("Invalid arguments, use none, or 'tid nr', or 'name proc'")
      exit()
  }
%)

  printf("         nsec clock      |    TID | Process          |    Fault Address |RW
|Maj/Min| Time delta\n")
  printf("-------------------------+--------+------------------+------------------
+---+-------+-----------\n")
}

probe vm.pagefault {
  if (tid() == target_tid || execname() == target_name || target == 0) {
    t = gettimeofday_ns()
    fault_entry_time[tid()] = t
    fault_address[tid()] = address
    fault_access[tid()] = write_access ? "w" : "r"
    execnames[tid()] = execname()
  }
}

probe vm.pagefault.return {
  if (tid() == target_tid || execname() == target_name || target == 0) {
    t = gettimeofday_ns()
    if (!(tid() in fault_entry_time)) next
    e = t - fault_entry_time[tid()]
    if (vm_fault_contains(fault_type,VM_FAULT_MINOR)) {
      ftype="minor"
    } else if (vm_fault_contains(fault_type,VM_FAULT_MAJOR)) {
      ftype="major"
    } else {
      next              # only want to deal with minor and major page faults
    }

    printf("%24d : %6d : %-16s : %16x : %s : %s : %d\n",
       t, tid(), execnames[tid()], fault_address[tid()], fault_access[tid()], ftype,
e)
    # free up memory
    delete fault_entry_time[tid()]
    delete fault_address[tid()]
    delete fault_access[tid()]
    delete execnames[tid()]
  }
}
```

There are some really good stap examples at: **https://sourceware.org/systemtap/examples/** However, be advised some may have been written for older kernel releases and may not work on current releases. None the less, there are many nice simple scripts out there that will also help to give you encouragement to cut/paste and develop your own as needed.

Some stap one liners courtesy of Brendan Gregg. You'll note some of these are more stap environment than actual probing. None the less, these are excellent examples that show the wide power of stap in providing valuable data with minimal “programming”

```
#
# SystemTap Lightweight One-Liners
#
# Drop the "-v" to elide the debug information about phase execution.
#

# List non-debuginfo syscall probes:
stap -L 'nd_syscall.*'

# List kernel tracepoint-based probes:
stap -L 'kernel.trace("*")'

# Trace new processes (via exec() only), showing arguments:
stap -ve 'probe nd_syscall.execve.return { println(cmdline_str(); }'

# Trace new processes (via exec() only), showing time and arguments:
stap -ve 'probe nd_syscall.execve.return { printf("%s %s\n", ctime(gettimeofday_s()),
cmdline_str()); }'

# Trace file open()s with process name and filename (if faulted):
stap -ve 'probe nd_syscall.open { printf("%s %s\n", execname(), filename); }'

# Count syscalls by process name:
stap -ve 'global a; probe nd_syscall.* { a[execname()] <<< 1; }'

# Count syscalls by pid and process name:
stap -ve 'global a; probe nd_syscall.* { a[pid(), execname()] <<< 1; }'

# Count syscalls by syscall name:
stap -ve 'global a; probe nd_syscall.* { a[name] <<< 1; }'

# Count syscalls by syscall name, output formatted:
stap -ve 'global a; probe nd_syscall.* { a[name] <<< 1; } probe end { foreach (s in
a+) { printf("%-16s %8d\n", s, @count(a[s])); } }'

# Histogram of syscall reads by requested size:
stap -ve 'global a; probe nd_syscall.read { a <<< int_arg(3); } probe end
{ print(@hist_log(a)); }'
```

```
# Histogram of syscall writes by requested size:
stap -ve 'global a; probe nd_syscall.write { a <<< int_arg(3); } probe end
{ print(@hist_log(a)); }'

# Histogram of syscall reads by return value (bytes are > 0, errors are < 0):
stap -ve 'global a; probe nd_syscall.read.return { a <<< returnval(); } probe end
{ print(@hist_log(a)); }'

# Histogram of syscall read returns by size (bytes):
stap -ve 'global a; probe nd_syscall.read.return { rval = returnval(); if (rval > 0)
{ a <<< rval; } } probe end { print(@hist_log(a)); }'

# Histogram of read() syscall time, in nanoseconds:
stap -ve 'global a; probe nd_syscall.read.return { a <<< gettimeofday_ns() -
@entry(gettimeofday_ns()); } probe end { print(@hist_log(a)); }'

# Histogram of write() syscall time, in nanoseconds:
stap -ve 'global a; probe nd_syscall.write.return { a <<< gettimeofday_ns() -
@entry(gettimeofday_ns()); } probe end { print(@hist_log(a)); }'

# Trace storage I/O requests by process and size (bytes):
stap -ve 'probe ioblock_trace.request { printf("%d %s %d\n", pid(), execname(),
size); }'

# Histogram of storage I/O request size (bytes):
stap -ve 'global a; probe ioblock_trace.request { a <<< size; } probe end
{ print(@hist_log(a)); }'

# Histogram of storage I/O request size, using kernel tracepoint (bytes):
stap -ve 'global a; probe kernel.trace("block_rq_insert") { a <<< $rq->__data_len; }
probe end { print(@hist_log(a)); }'

# Histogram of storage I/O kernel queueing time (nanoseconds), using built-in
timestamps:
stap -ve 'global a; probe kernel.trace("block_rq_complete") { a <<< $rq-
>io_start_time_ns - $rq->start_time_ns; } probe end { print(@hist_log(a)); }'

# Count major faults by process name:
stap -ve 'global a; probe perf.sw.page_faults_maj { a[execname()] <<< 1; }'

# Count IRQs by device name and IRQ:
stap -ve 'global a; probe irq_handler.entry { a[kernel_string(dev_name), irq] <<<
1; }'

# Count kernel function calls beginning with "bio", using dynamic tracing:
stap -ve 'global a; probe kprobe.function("bio*") { a[ppfunc()] <<< 1; }'

# Histogram of kernel vfs_read() calls by requested size:
stap -ve 'global a; probe kprobe.function("vfs_read") { a <<< int_arg(3); } probe end
{ print(@hist_log(a)); }'
```

```
# Count block I/O requests, using the block_rq_insert tracepoint, by process name:
stap -ve 'global a; probe kernel.trace("block_rq_insert") { a[execname()] <<< 1; }'
```

stap is insanely powerful

stap is an extremely powerful tool. It can also suffer from that power.... That is to say, you are probing kernel functions. If those functions are called frequently in the kernel, you could easily overflow output buffers, you could also run out of time to handle the volume of probing being requested. So there are a number of runtime options that can be used to "solve" various conditions. We discuss these in more detail shortly. **-s** is a megabyte in-memory buffer size

```
# To build the .ko:
# stap -g -p 4 -r 2.6.32-279.22.1.el6.x86_64 mmap2-ENOMEM.stp -s2560 \
#      -DSTP_NO_OVERLOAD -DTRYLOCKDELAY=2000 -DMAXTRYLOCK=2000 -m mmap2stapv2
#
#      NOTE. -s1024 works fine if you set the probe timer to 15 secs
#            set to -s2048 if you set the probe timer to 60 secs
#            Other options set to overcome skipped probes errors
#
# To run:
# staprun mmap2stapv2.ko > mmap2stap.log
```

The comments above came from an stap script that I wrote and required numerous options to solve various timing issues that were occurring. It is also worth emphasizing that installing an stap script CAN induce an overhead in the kernel and cause performance issues. It certainly can impact the timing of the code to a point that some problems which are caused by timing windows, may change so much as to not occur with stap running. So be careful, be cautious and be open minded about what your script is trying to achieve.

What can you do with stap? Honestly it's limited only by your imagination (OK it does have limits but few will ever reach them...). You can code up some very exotic functions.

One final example. What if you want to utilize a kernel callable function for example, **test_bit()**. Sounds like it should be easy? Well it is but like everything in life... it's easy when someone shows you. When you don't have that luxury, believe me this can turn into a nightmare of a “problem” to solve.

I've also included an example of using the **jiffies** timer to determine “Time since boot”. You might think this is easy... Yep it is.... if you know... You'll see I take the jiffies and do some maths on it.

```
t = ((jiffies() + 300000) – 0x100000000);
```

I'm going to leave you to ponder on this mathematical conversion. I'll explain why during the class.

```
#!/usr/bin/env stap

%{ #include <asm/bitops.h> %}

# stap lock_page_or_retry3.stp name mmaptest > /tmp/stap-trace-$(date +"%m%d%y-%H%M%S").log
# -or-
# stap lock_page_or_retry3.stp pid 12345 > /tmp/stap-trace-$(date +"%m%d%y-%H%M%S").log

# If you have stap problems with overflow etc... add "-s2560 -DSTP_NO_OVERLOAD"

global target_tid = 0;
global target_name = " ";
global pagelocked = 0;
global pglocked = " ";
global flags = 0;

%{
#define PAGE_LOCKED       (0)
%}

probe begin {

        %( $# < 2 %?
                log("Wrong number of arguments, use none, 'tid nr' or 'name proc'")
                exit()
        %)

        %( $# == 2 %?
                if(@1 == "tid")
                        target_tid = strtol(@2, 10)
                if(@1 == "name")
                        target_name = @2
        %)

  printf("Beginning probes. Ctrl/c to terminate\n\n")
  printf("   Jiffies    | CPU |   Args\n")
  printf("-------------+-----+-----------------------------------------------\n")
}
```

```
function _tester:string(pflags:long)
          %{
          unsigned long pf = (unsigned long)STAP_ARG_pflags;
          if (test_bit(PAGE_LOCKED, &pf))
             strlcat(STAP_RETVALUE, "LOCKED", MAXSTRINGLEN);
          else
             strlcat(STAP_RETVALUE, "UNLOCKED", MAXSTRINGLEN);
          %}

probe kernel.function("__lock_page_or_retry") {
   if (tid() == target_tid || execname() == target_name) {
         t = ((jiffies() + 300000) - 0x100000000);
         printf("%12d : %3d : __lock_page_or_retry\(\) page->flags = 0x%lx\n",
                t, cpu(), $page->flags);
   }
}

probe kernel.function("vm_normal_page").return {
   if (pagelocked == 0) {
      if (tid() == target_tid || execname() == target_name) {
         t = ((jiffies() + 300000) - 0x100000000);
         if ($return != 0) {
            flags = @cast($return, "page")->flags;
            pglocked = _tester(@cast($return, "page")->flags);
            if (pglocked == "LOCKED") {
               pagelocked = $return;
            }
         } else {
         flags = 0;
         }
         printf("%12d : %3d : vm_normal_page\(\)      page = 0x%lx, page->flags = 0x%lx -
%s\n",
                t, cpu(), $return, flags, pglocked);
      }
   }
}


probe kernel.function("unlock_page").return {
   if (pagelocked == $page) {
         t = ((jiffies() + 300000) - 0x100000000);
         flags = @cast($page, "page")->flags;
         pglocked = _tester(@cast($page, "page")->flags);
         printf("%12d : %3d : unlock_page\(\)         page = 0x%lx, page->flags = 0x
%lx - %s\n",
                t, cpu(), $page, flags, pglocked);
   }
}
```

```
probe kernel.function("do_wp_page") {
   if (tid() == target_tid || execname() == target_name) {
         t = ((jiffies() + 300000) - 4294967296);
         printf("%12d : %3d : do_wp_page\(\)          address = 0x%lx\n",
                t, cpu(), $address);
   }
}
```

No one expects you to understand all these example scripts. They are here for “reference”. Attached to this lesson you will find an stap Tutorial written by Frank Eigler (Red Hat).

3.2.3.3 - Run time parameters to solve stap warnings/errors

Exhausted resources

Systemtap attempts to assure that probe handlers run within strict time/space limits. A typical script producing a modest amount of trace data will not encounter these limits, but maybe your script does. What can you do?

It depends on what resource ran out.

There is a simple reason that you should try to determine the root cause of your warning/error messages. It can easily undermine the accuracy of your stap script. That probe that is skipped may have been the one that you were testing for.

Error Message	Exhausted Resource	Possible Remedy
`MAXACTION exceeded`	time -- number of statements executed by probe handler	Recompile with larger **`-DMAXACTION=NN`** or **`-DMAXACTION_INTERRUPTIBLE=NN`**
		Impose limits on iteration - **`foreach (... limit NN)`** or **`break`**)
`MAXNESTING exceeded`	time/space -- function recursion too deep	Reduce recursion
		Recompile with larger **`-DMAXNESTING=NN`**
`division by zero`	time/space -- universe is too small to represent exact value	Try dividing by one instead
`aggregation overflow`	space -- number of distinct index tuples in aggregation array	Recompile with larger **`-DMAXMAPENTRIES=NN`**
		Declare that array only with larger size -- **`global my_array[NN]`**
		Delete unneeded elements, or use **`% array global`** declaration suffix
`Array overflow`	space -- number of distinct index tuples in ordinary array	As above
`string truncation`	space -- strings are silently limited to a maximum length	Recompile with larger **`-DMAXSTRINGLEN=NN`**

`WARNING: ... skipped probes: NN`	time -- cross-processor contention over global script variables. **NOTE. Further expanded upon after this table**	Reduce number of globals
		Maybe recompile with larger **`-DMAXTRYLOCK=NN`**
		Use aggregates and **`@count()`** instead of integer counters
	space -- too many return probes may be pending	Add larger **`.maxaction(NN)`** to **`.return`** probe point
	space -- too little kernel stack available for probe handler run	Move probe points higher in the call stack
		risky, but maybe recompile with smaller **`-DMINSTACKSPACE=NN`**
`probe overhead exceeded threshold`	time -- probe handlers are taking too high fraction of total real time	Shrink probe handler code
		Recompile with larger **`-DSTP_OVERLOAD_THRESHOLD`**
		Disable this heuristic with **`-DSTP_NO_OVERLOAD`**
`There were NN transport failures`	space -- amount of data printed by script exceeded staprun's capability to copy it to the output file	Print less stuff
		Run with larger in-memory trace buffers: **`stap -s NN`** (This size is a per processor MB size)

Understanding skipped probes

Systemtap probe handlers may be skipped sometimes. When enough of them do so, a script exits.

```
WARNING: Number of errors: 0, skipped probes: 100
```

There are many different reasons. As a first step for debugging, rerun your script with the **`-t`** (timing) flag. Listed below are the common possibilities.

The number of skipped probes that trigger an overall script error exit is governed by the **`-DMAXSKIPPED=nnn`** parameter. You may increase that number dramatically and try running the script again.

Skipped due to global 'VAR' lock timeout

The problem is excessive contention for a script-level "global" variable: where too many concurrently running probe handlers are trying to modify the same global(s). Each new probe handler waits up to a limited amount of time for the locks to be released; otherwise the probe is skipped. This can sometimes be worked around by optimizing script code, or by enlarging the **-DTRYLOCKDELAY=mmm** and **-DMAXTRYLOCK=nnn** parameters.

Skipped due to low stack

Some probes were triggered in a context where too little kernel stack was available to safely attempt execution of the handlers. For example, on architectures that don't allocate separate exception stacks, kprobes-based probes may be placed in deeply nested kernel contexts. The amount minimum free kernel stack space at probe entry is about **-DMINSTACKSPACE=nnn** bytes. You could try to reduce this amount below its default (slightly and carefully), or place your probes higher up in the call stack.

Skipped due to reentrancy

Most probe handlers run in **-DINTERRUPTIBLE=1** mode by default. This means that hardware interrupts may occur while a probe handler is run. If those interrupt handlers are in turn instrumented somehow, then the second systemtap probe could try to be invoked while the first one is still active. This sort of reentrancy is detected and prevented by skipping the new reentrant probe. You can reduce this phenomenon by placing probes out of interrupt handler paths, or by running with **-DINTERRUPTIBLE=0** (but NMI can still interrupt things). You may request a more detailed trace record about each reentrancy event by specifying **-DDEBUG_REENTRANCY=1**.

Skipped due to uprobe register failure

A user-space probe registration (activation) attempt has failed. This could be because of having too many concurrent user-space probes, so that a fixed-sized table was exhausted (**-DMAXUPROBES=mmm**), or because the addresses were somehow invalid. Running with **-DDEBUG_UPROBES=1** should generate some extra tracing into the kernel printk logs (check the ring buffer/dmesg log file).

Skipped due to uprobe unregister failure

A user-space probe unregistration (deactivation) attempt has failed. This suggests some kind of internal error. **-DDEBUG_UPROBES=1** may give some clues.

Skipped due to missed kretprobe/1 on 'FOO'

A **kernel/module.function().return** probe was requested, but the number of preallocated pending-kretprobe table slots was exhausted, because too many other instances of the given function have started but not yet returned. Add a **.maxactive(nnnn)** at the end of the **kernel/module.function().return** probe point specification to reserve more slots. Or if

you're using a tapset alias or wildcard, you could increase the systemwide default with the **-DKRETACTIVE=mmm** parameter. You could set it to hundreds or thousands if you suspect that kernel threads can block in or under that function for a long time.

Skipped due to missed kretprobe/2 on 'FOO'

Similarly, a kernel .return probe was requested, but something went wrong with the corresponding function-entry kprobe. This should not happen, except perhaps in extreme low-kernel-memory conditions.

Skipped due to missed kprobe on 'FOO'

Similarly, a kernel function entry probe was requested, but something went wrong. This should not happen, except perhaps in extreme low-kernel-memory conditions.

-D options

MAXNESTING The maximum number of recursive function call levels. The default is 10.

MAXSTRINGLEN The maximum length of strings. The default is 128.

MAXTRYLOCK The maximum number of iterations to wait for locks on global variables before declaring possible deadlock and skipping the probe. The default is 1000.

MAXACTION The maximum number of statements to execute during any single probe hit. The default is 1000. Note that for straight-through probe handlers lacking loops or recursion, due to optimization, this parameter may be interpreted too conservatively.

MAXACTION_INTERRUPTIBLE Maximum number of statements to execute during any single probe hit which is executed with interrupts enabled (such as begin/end probes), default (**MAXACTION** * 10).

MAXBACKTRACE Maximum number of stack frames that will be be processed by the stap runtime unwinder as produced by the backtrace functions in the [u]context-unwind.stp tapsets, default 20.

MAXMAPENTRIES Default maximum number of rows in any single global array, default 2048. Individual arrays may be declared with a larger or smaller limit instead: global big[10000],little[5]. From testing I believe this is a hardcoded max of 65K (65,536) rows/entries in an array. Changing **MAXMAPENTRIES** to larger values like 100,000 had no affect. Additionally creating arrays with array[100000] did not work either.
You can denote the array with % to make them wrap-around automatically as follows: myarray%[65536]

MAXERRORS The maximum number of soft errors before an exit is triggered. The default is 0 which means that the first error will exit the script. Note that with the **--suppress-handler-errors** option, this limit is not enforced.

MAXSKIPPED The maximum number of skipped reentrant probes before an exit is triggered. The default is 100. Running systemtap with **-t** (timing) mode gives more details about skipped probes. With the default **-DINTERRUPTIBLE=1** setting, probes skipped due to reentrancy are not accumulated against this limit. Note that with the **--suppress-handler-errors** option, this limit is not enforced.

MINSTACKSPACE The minimum number of free kernel stack bytes required in order to run a probe handler. This number should be large enough for the probe handler's own needs, plus a safety margin. The default is 1024.

MAXUPROBES Maximum number of concurrently armed user-space probes (uprobes), default somewhat larger than the number of user-space probe points named in the script. This pool needs to be potentialy large because individual uprobe objects (about 64 bytes each) are allocated for each process for each matching script-level probe.

STP_MAXMEMORY Maximum amount of memory (in kilobytes) that the systemtap module should use, default unlimited. The memory size includes the size of the module itself, plus any additional allocations. This only tracks direct allocations by the systemtap runtime. This does not track indirect allocations (as done by kprobes/uprobes/etc. internals).

STP_PROCFS_BUFSIZE Size of procfs probe read buffers (in bytes). Defaults to **MAXSTRINGLEN**. This value can be overridden on a per-procfs file basis using the procfs read probe .maxsize(MAXSIZE) parameter.

STP_NO_OVERLOAD In addition to the methods outlined above, the generated kernel module also uses overload processing to make sure that probes can't run for too long. If more than **STP_OVERLOAD_THRESHOLD** cycles (default 500,000,000) have been spent in all the probes on a single cpu during the last **STP_OVERLOAD_INTERVAL** cycles (default 1,000,000,000), the probes have overloaded the system and an exit is triggered. By default, overload processing is turned on for all modules. If you would like to disable overload processing, define **STP_NO_OVERLOAD** (or its alias **STAP_NO_OVERLOAD**).

STP_OVEROAD_THRESHOLD
STP_OVERLOAD_INTERVAL Maximum number of machine cycles spent in probes on any cpu per given interval, before an overload condition is declared and the script shut down. The defaults are 500,000,000 and 1,000,000,000 billion, so as to limit stap script cpu consumption at around 50%.

INTERRUPTIBLE With scripts that contain probes on any interrupt path, it is possible that those interrupts may occur in the middle of another probe handler. The probe in the interrupt handler would be skipped in this case to avoid reentrance. To work around this issue, execute stap with the option **-DINTERRUPTIBLE=0** to mask interrupts throughout the probe handler. This does add some extra overhead to the probes, but it may prevent reentrance for common problem cases. However, probes in NMI handlers and in the callpath of the stap runtime may still be skipped due to reentrance.

RELAYHOST Multiple scripts can write data into a relay buffer concurrently. A host script

RELAYGUEST provides an interface for accessing its relay buffer to guest scripts. Then, the output of the guests are merged into the output of the host. To run a script as a host, execute stap with **-DRELAYHOST[=name]** option. The name identifies your host script among several hosts. While running the host, execute stap with **-DRELAYGUEST[=name]** to add a guest script to the host. Note that you must unload guests before unloading a host. If there are some guests connected to the host, unloading the host will be failed.

NOTE. In case something goes wrong with stap or staprun after a probe has already started running, one may safely kill both user processes, and remove the active probe kernel module with **`rmmod`**. Any pending trace messages may be lost.