Systemtap

FrOSCon (25. August 2013)

Stefan Seyfried
Linux Consultant & Trainer
B1 Systems GmbH
seife@b1-systems.de

Systemtap

What is Systemtap?

- systemtap is a scriptable monitoring and analysis environment
- used for kernel
  - monitoring
  - profiling
  - tracing
- comparable to userspace tools like `top` and `strace`

How Does Systemtap Work?

- uses existing kernel facilities, which must be enabled:

  `CONFIG_RELAY` logging/transfer interface from kernel to userspace

  `CONFIG_KPROBES` enables setting of breakpoints at arbitrary places in the kernel and execute own code

- `Kprobes` provides different "probes":

  `Kprobes` can set breakpoints at arbitrary places in the kernel

  `Jprobes` can be placed in kernel function head (access to the argument list)

  `Return Probes` (aka. kretprobes) will be called at the end of functions and has access to the return values

How Does Systemtap Work?

- systemtap provides a simple scripting language
- systemtap's scripting language makes use of `kprobes` and other kernel facilities

```
probe kernel.function("sys_open") {
    printf ("%s(%d) called sys_open\n", execname(), pid())
}
```

Interaction of kprobes, relay and systemtap

1. a systemtap script is started by systemtap
2. systemtap parsers translate the script into C $\rightarrow$ autogenerated sourcecode of a kernel module
3. the module uses kprobes' functionality to set needed probes
4. uses `relay` functions to transfer output from kernel to the systemtap process
5. a C compiler compiles the kernel module sourcecode

Interaction of kprobes, relay and systemtap

1. a systemtap script is started by systemtap
2. systemtap parsers translate the script into C $\rightarrow$ autogenerated sourcecode of a kernel module
3. the module uses kprobes' functionality to set needed probes
4. uses `relay` functions to transfer output from kernel to the systemtap process
5. a C compiler compiles the kernel module sourcecode

Interaction of kprobes, relay and systemtap

1. a systemtap script is started by systemtap
2. systemtap parsers translate the script into C $\rightarrow$ autogenerated sourcecode of a kernel module
3. the module uses kprobes' functionality to set needed probes
4. uses `relay` functions to transfer output from kernel to the systemtap process
5. a C compiler compiles the kernel module sourcecode

Interaction of kprobes, relay and systemtap

1. a systemtap script is started by systemtap
2. systemtap parsers translate the script into C $\rightarrow$ autogenerated sourcecode of a kernel module
3. the module uses kprobes' functionality to set needed probes
4. uses `relay` functions to transfer output from kernel to the systemtap process
5. a C compiler compiles the kernel module sourcecode

Interaction of kprobes, relay and systemtap

1. a systemtap script is started by systemtap
2. systemtap parsers translate the script into C $\rightarrow$ autogenerated sourcecode of a kernel module
3. the module uses kprobes' functionality to set needed probes
4. uses `relay` functions to transfer output from kernel to the systemtap process
5. a C compiler compiles the kernel module sourcecode

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Interaction of kprobes, relay and systemtap

1. module is loaded by systemtap
2. `kprobes` calls register the probes at the selected functions
3. `kprobes` handlers call the compiled function stub
4. the function stub tranfers information to systemtap via the `relay` interface
5. systemtap receives the information via so called `relay channels`
6. systemtap processes the received information and prints them (depending on the script) to `stdio`
7. module is unloaded if systemtap is interrupted or the script/kernel module has reached a controlled end point

Influence on the System – Support

- by using self compiled kernel modules, problems with Enterprise support might occur
- value of `/proc/sys/kernel/tainted` is no longer "0"

Influence on the System – Performance

- only little overhead (microseconds) (depending on function's complexity)
- long time use is not problematic as memory is limited
- too "slow" probes which are called very often are skipped
- if a probe is skipped 100 times, the complete systemtap script is stopped and unloaded

Influence on the System – Stability

- possible system crashes when script is loaded
- test scripts thoroughly on dedicated test machines
- if a crash occurs, it mostly occurs instantly when loading the script
- upstream developers work on avoiding such crashes which strongly depend on the kernel version used

Installation

Installation

- systemtap is packaged for all distributions
- the kernel-development and debuginfo/debugsource packages are also needed
- if the `debuginfo`-package is missing, running a script which uses kernel functions leads to errors:

```
# stap open.stp
semantic error: no match while resolving probe point
 syscall.open
Pass 2: analysis failed.  Try again with another
 '--vp 01' option.
```

Compiling Your Own Kernel

- when using a self compiled kernel, certain options need to be set:

```
CONFIG_DEBUG_INFO=y
CONFIG_KPROBES=y
CONFIG_RELAY=y
CONFIG_DEBUG_FS=y
CONFIG_MODULES=y
CONFIG_MODULE_UNLOAD=y
```

Scripting

Hello World

- the complex part: scripting language
- allows complicated procedures
- simplest example:

```
# cat hw.stp
probe begin {
    print ("hello world\n")
    exit ()
}

# stap hw.stp
hello world
```

Variables

- only two basic types: `integer` and `string` (nothing else!)
- variables starting with $ depict the variables in the source code of the instrumented function
- "own" variables do **not** start with $
- default: local variables, global variables are declared with `global` *`var`*
- unused variables are optimized away by systemtap

Variables

```
global foo

probe begin {
        bar = 5
        foo = 42
        printf("hello: %d -> %d\n", foo, bar)
        foo = bar
        exit()
}

probe end {
        printf("byebye: %d\n", foo)
}
```

Conditionals (if)

- exactly like C:

```
if ( expression ) {
        foo
} else {
        bar
}
```

Loops (for, while, foreach)

- `foreach` used to iterate through arrays
- `while` only as top-controlled loop
- `for` like C with two semicolon-separated fields:

```
for (i=0; i < 42; i++) {
        printf("#%d Element: %d\n", i, array[i])
}

i=42
while (i > 0) {
        i = do_something_fancy(i);
}
```

Command Line Arguments

- like normal applications or scripts, arguments can be passed on the command line
- argument variables are: $$n$ (example: $1 $2 ...)
- to interpret a variable as string, use @ instead of $ (example: @2)
- **Note:** for every call with a different command line, the script needs to be recompiled as arguments will be hardcoded in the compiled code

Helper Functions

Systemtap provides a large set of helper functions:

`printf(`*`format`*`)` / `sprintf(`*`format`*`)` same as in C

`print(`*`string`*`)` output a string

`strlen(`*`string`*`)` determine the length of a string

`isinstr(strA, substrB)` check for a substring

`strtol(`*`string`*`, `*`base`* as in C, converts a string to an integer

`exit()` end the systemtap script

Context Helper Functions

Systemtap provides context specific helpers to relate a call to a userspace process:

pid()	Process ID of the userspace process which triggered the instrumented kernel function
execname()	Name of the userspace process
cpu()	Current CPU ID the userspace process is running on
uid()	Current user ID of the userspace process
probefunc()	Name of the instrumented kernel function, useful with generic probes

Tapsets

- tapsets call predefined probe functions
- Example:

```
probe vfs.read {}
```

- excerpt from the corresponding tapset:

```
probe vfs.read = kernel.function ("vfs_read")
```

- it is basically an *Alias*
- systemtap copies the function `vfs_read` and inserts the scripted block
- predefined tapsets in `/usr/share/systemtap/tapset/*`

Examples – Provided Scripts

- example scripts under `/usr/share/doc/packages/systemtap/examples/`
- an explanation for every example is located in `/usr/share/doc/packages/systemtap/examples/index.html`

Example – forktracker.stp

- example script: process/forktracker.stp
- traces the creation of new processes in the system
- consists basically of those two probes:

```
probe kprocess.create {
  printf("%-25s: %s (%d) created %d\n",
      ctime(gettimeofday_s()), execname(), pid(),
      new_pid)
}
probe kprocess.exec {
  printf("%-25s: %s (%d) is exec'ing %s\n",
      ctime(gettimeofday_s()), execname(), pid(),
      filename)
}
```

- used kernel functions:
  - copy_process
  - do_execve, compat_do_execve

Example Output – forktracker.stp

Example output of forktracker.stp

```
# stap process/forktracker.stp
Wed Feb  2 15:38:54 2011 : bash (3747) created 6879
Wed Feb  2 15:38:54 2011 : bash (6879) is exec'ing /bin/ls
Wed Feb  2 15:39:20 2011 : bash (3747) created 6880
Wed Feb  2 15:39:20 2011 : bash (6880) is exec'ing /usr/bin/touch
Wed Feb  2 15:39:26 2011 : bash (3747) created 6881
Wed Feb  2 15:39:26 2011 : bash (6881) is exec'ing /bin/rm
```

Example – `disktop.stp`

- example script: `io/disktop.stp`
- provides a "top ten" digest of disc access (reading/writing) in the system every five minutes
- monitoring "docks" onto kernel functions `vfs_read` and `vfs_write`, saves collected data in arrays

Example Output – disktop.stp

Example output – disktop.stp

```
# stap io/disktop.stp

Wed Feb  2 13:18:19 2011 , Average:   0Kb/sec, Read:       0Kb, Write:      0Kb

     UID      PID     PPID                    CMD   DEVICE    T        BYTES
       0     1833        1               syslog-ng     vda2    W          199

Wed Feb  2 13:18:49 2011 , Average:   0Kb/sec, Read:       0Kb, Write:      0Kb

     UID      PID     PPID                    CMD   DEVICE    T        BYTES
       0     3438        1                 master     vda2    W            1
      51     3468     3438                 pickup     vda2    R            1
```

Example – `iotime.stp`

- Example script: `io/iotime.stp`
- Syntax:

```
iotime.stp [-c <Programm>]
```

- Which read calls last how long?

Example Output – iotime.stp

Example output – iotime.stp

```
# stap io/iotime.stp -c 'hostname'
sles11a
WARNING: Number of errors: 0, skipped probes: 4
7445 5459 (hostname) access /etc/ld.so.cache read: 0 write: 0
7716 5459 (hostname) access /lib64/libc.so.6 read: 832 write: 0
7719 5459 (hostname) iotime /lib64/libc.so.6 time: 3
8951 5459 (hostname) access /usr/share/locale/locale.alias read: 8192 write: 0
8955 5459 (hostname) iotime /usr/share/locale/locale.alias time: 16
9031 5459 (hostname) access /usr/lib/locale/en_US.utf8/LC_CTYPE read: 0 write: 0
9106 5459 (hostname) access <unknown> read: 0 write: 0
```

Example – `nettop.stp`

- example script: `network/nettop.stp`
- summarizes network access of processes in 5 second intervals
- used kernel functions: `dev_queue_xmit` and `netif_receive_skb`

Example Output – nettop.stp

Example output – nettop.stp

```
# stap network/nettop.stp
  PID   UID DEV     XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND

  PID   UID DEV     XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
    0     0 eth0          0       1       0       0 swapper
 3720  1000 eth0          1       0       0       0 sshd

  PID   UID DEV     XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
    0     0 eth0          0       2       0       0 swapper
 3720  1000 eth0          1       0       0       0 sshd
```

Probes

Probes in General

- probes start with `probe` *`probename`*
- probe lists are possible: `probe syscall.open, syscall.close`
- overview of available probes: `man 3stap stapprobes`

Optional Probes

- optional probes allow to write generic systemtap scripts
- if the probe is not available for this kernel version, no error occurs
- optional probes are postfixed with a question mark "?"
- should a list of possible probes not be tried furthter after the first match, use an exclamation mark (!) instead of the question mark

```
probe foo ?, bar ? { ... }

probe green !, yellow !, red { ... }
```

Conditional Probes

- probe is only executed if condition is met
- `probe` *`name`* `if (`*`expression`*`) { ... }`
- useful to "filter" probes
- either source code variables, command line arguments or global variables are possible

Special Probes: begin, end, never

begin is executed once during start

end is executed once at the end (`exit()`, `Ctrl + c`)

never is never executed, but analysed by the compiler (useful for testing of optional probes)

Return Probes

- return probes instrument the end of a function
- `probe` *`name`*`.return`
- `$return`: return value of the function
- sometimes `$return` is not available if code is optimized (e.g. inlined functions). Better with GCC 4.5+

Kernel Functions

kernel function `kernel.function("do_fork") / kernel.function("do_fork").return`

wildcard `kernel.function("*copy*")`

wildcard in source file `kernel.function("*@kernel/fork.c")`

kernel module functions `module("name").function("*")`

Timer Probes

- `probe timer.s(`*N*`)`
- `probe timer.ms(`*N*`)`
- `probe timer.us(`*N*`)`
- `probe timer.ns(`*N*`)`
- `probe timer.hz(`*N*`)`
- `probe timer.ms(`*N*`).randomize(`*M*`)`
- `...`

Functions

Formatted Output

- `thread_indent(`*N*`)`
- relative indenting to display the calling order of functions

Example:

```
probe kernel.function("*@mm/*.c") {
  printf("%s > %s\n", thread_indent(2), probefunc())
}
probe kernel.function("*@mm/*.c").return {
  printf("%s < %s\n", thread_indent(-2), probefunc())
}
```

Formatted Output

Result:

```
     0 usb-storage(21043):   > blk_complete_request
     7 usb-storage(21043):     > __blk_complete_request
    14 usb-storage(21043):     < __blk_complete_request
    18 usb-storage(21043):   < blk_complete_request
```

Command Execution

- `system("`*`command`*`")`
- executes a command
- executing in the background to not block the probe
- useful for long running systemtap analysis to signal conditions

errno_str

- `errno_str:string (e:long)`
- returns the symbol for a given errno
- Example: 12, for ENOMEM. (`Cannot allocate memory`)
- useful to translate errnos into readable symbols

Thank you!

Stefan Seyfried
seife@b1-systems.de