



Blueprints
Installing and Using SystemTap





Blueprints

Installing and Using SystemTap

Note

Before using this information and the product it supports, read the information in “Notices” on page 31.

Second Edition (May 2009)

© Copyright International Business Machines Corporation 2008, 2009.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction	v	Chapter 5. Debuginfo-less probing for function boundary tracing	19
<hr/>		Chapter 6. Available command line options for SystemTap	21
Part 1. Installing and Using SystemTap	1	Chapter 7. Root versus non-root user probing	23
Chapter 1. SystemTap overview	3	<hr/>	
Chapter 2. Installing SystemTap	5	Part 2. Appendixes.	25
Installing SystemTap on Red Hat Enterprise Linux (RHEL) 5.2	5	Troubleshooting tips.	27
Installing SystemTap on SUSE Linux Enterprise Server (SLES) 10 SP2 and 11	6	Related information and downloads	29
Optional: Installing on a custom kernel	7	Notices	31
Chapter 3. SystemTap script examples explained.	11	Trademarks	32
syscalls_by_proc.stp example	11	Terms and conditions	32
fork.stp example.	13		
iotop.stp example	15		
Chapter 4. SystemTap safety features	17		

Introduction

SystemTap is a tool that allows you to run reusable scripts to examine Linux[®] kernel activities. By doing so, you can determine underlying causes of poor performance or functional problems. This Blueprint provides information about installing SystemTap on a supported system and detail about three SystemTap example scripts. By examining the information about how these scripts are written, you can learn to write your own SystemTap scripts.

Intended audience

This Blueprint is intended for Linux system administrators and programmers who want use SystemTap to learn more about system resource usage. You can also use SystemTap as a tracing tool to debug your code or to find information about the basic flow of kernel code.

Scope and purpose

SystemTap probes the Linux kernel using a dynamic instrumentation technique called *kprobes*. You can probe function calls or returns and the execution of kernel statements. This Blueprint helps you get started with SystemTap.

If you are interested in learning about the constructs of the SystemTap script language, you can read the SystemTap *Language Reference Guide*. This document is bundled with SystemTap packages in newer distributions (for example, in Red Hat Enterprise Linux 5.2, the guide is available in the `/usr/share/doc/systemtap-<version>/langref.pdf` file). You can also find the guide online at <http://sourceware.org/systemtap/langref/>.

Other considerations

The installation, execution, and screen logs in this document are explained regarding the following distributions:

Red Hat Enterprise Linux Server

(<http://www.redhat.com/rhel/server/>)

SUSE Linux Enterprise Server

(<http://www.novell.com/products/server/>)

The behavior of SystemTap is intended to be uniform across various Linux distributions, distribution releases, and kernel versions; however, some combinations might require specific modifications. You can ask general questions on the SystemTap mailing list (<http://sources.redhat.com/ml/systemtap/> and systemtap@sources.redhat.com). If you have questions about SystemTap packages provided by your Linux distribution, contact your distribution provider for assistance.

Software requirements

Find your distribution in the following chart and install the corresponding package groups. Use the package manager supplied with your distribution.

Red Hat Enterprise Linux (RHEL) 5.2

Development Tools, Development Libraries

SUSE Linux Enterprise Server (SLES) 10 SP2

C/C++ Compiler and Tools (also install the *libcap-devel* individual package)


Hardware requirements

SystemTap is available on x86_32, x86_64, PowerPC®, and s390 architectures and is tested on these platforms. You can use SystemTap on ia64 and ARM platforms, although this configuration is not fully supported.

IBM Services


Linux offers flexibility, options, and competitive total cost of ownership with a world class enterprise operating system. Community innovation integrates leading-edge technologies and best practices into Linux.

IBM® is a leader in the Linux community with over 600 developers in the IBM Linux Technology Center working on over 100 open source projects in the community. IBM supports Linux on all IBM servers, storage, and middleware, offering the broadest flexibility to match your business needs.

For more information about IBM and Linux, go to [ibm.com/linux](https://www.ibm.com/linux)  (<https://www.ibm.com/linux>).

IBM Support

Questions and comments regarding this documentation can be posted on the developerWorks SystemTap Blueprint Community Forum:

<http://www-128.ibm.com/developerworks/forums/forum.jspa?forumID=1323>  (<http://www-128.ibm.com/developerworks/forums/forum.jspa?forumID=1323>).

The IBM developerWorks® discussion forums let you ask questions, share knowledge, ideas, and opinions about technologies and programming techniques with other developerWorks users. Use the forum content at your own risk. While IBM will attempt to provide a timely response to all postings, the use of this developerWorks forum does not guarantee a response to every question that is posted, nor do we validate the answers or the code that are offered.

Typographic conventions

The following typographic conventions are used in this Blueprint:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
<i>Italics</i>	Identifies parameters whose actual names or values are to be supplied by the user.
Monospace	Identifies examples of specific data values, examples of text like what you might see displayed, examples of portions of program code like what you might write as a programmer, messages from the system, or information you should actually type.

Part 1. Installing and Using SystemTap

Chapter 1. SystemTap overview

SystemTap is a tool that allows developers and administrators to write and reuse simple scripts to finely examine the activities of a live Linux system. Data can be extracted, filtered, and summarized quickly and safely, to enable diagnoses of complex performance or functional problems.

SystemTap scripts are driven by events during execution. There are several kinds of events: entering or exiting a function, a timer expiring, or the entire SystemTap session beginning or ending through the use of a *kprobe*. A *kprobe* is a Linux kernel tool that you can use to interrupt the kernel execution flow even when the system is running. When a probe is reached during execution, a predefined function called a *handler* can be executed as if it were a quick subroutine then resumes execution to collect debugging information. A *probe handler* is a series of script language statements that specify the work to be done whenever the event occurs. This work typically includes extracting data from the event context, storing them into internal variables, or printing well-formatted results. A handler can be made to run before each of these events.

SystemTap works by translating its script to a C-language file that is compiled into a Linux kernel module. A Linux kernel module is a binary file containing kernel code that is not loaded into the system during boot but can be dynamically loaded or unloaded from the system. In this case, the Linux kernel module is basically a *kprobes* module.

When the module is loaded, all probes that are defined in the script are activated and handlers are executed when the instrumented events occur. When the session stops, the probes or hooks are disconnected and the module is unloaded. This behavior ensures that there is no functional or performance impact when SystemTap is not running.

SystemTap has an advantage over other debugging techniques due to its powerful scripting language and built-in safety features. *Tapsets* are libraries of reusable scripts installed with SystemTap that provide helper functions and predefined variables. You can use the scripting language and tapsets to create instrumentation code that otherwise requires several thousands of lines of *kprobes* module code when written using C.

SystemTap can use Debugging With Attributed Record Formats (DWARF) debugging information that accompanies kernel images and modules, such as how the GDB debugger uses them, to provide fine granular control over debugging. DWARF is a standard for generating information by compilers, assemblers, and linkers that can be used for debugging purposes. Availability of debugging information conforming to the DWARF standard provides visibility into symbols and source code statements.

Enterprise Linux distributions provide the corresponding package containing the kernel debugging information (*kernel-debuginfo*) that SystemTap uses. This package is typically included in the supplementary installation medium. If you update the kernel, you must also update to the matching *kernel-debuginfo* package. You might have to download this package separately.

Chapter 2. Installing SystemTap

SystemTap is packaged as a system and debugging tool with enterprise Linux distributions and must be installed with the *kernel-debuginfo* packages that match the installed kernel version.

Install SystemTap on a supported software and hardware configuration. See “Software requirements” on page v and “Hardware requirements” on page vi for more information.

If you are running the enterprise distribution kernel, and if circumstances allow, update your machine to the latest kernel through the distribution update mechanism. If you are running a custom kernel, see Installing on a custom kernel for more information.

Installing SystemTap on Red Hat Enterprise Linux (RHEL) 5.2

SystemTap functionality is dependent upon the installation of several packages: *systemtap* and *systemtap-runtime* RPMs, *kernel-debuginfo* and *kernel-debuginfo-common* RPMs, and the *kernel-devel* RPM. The *kernel-debuginfo*, *kernel-debuginfo-common*, and *kernel-devel* packages must match the kernel version on your machine.

Installing the *systemtap* and *systemtap-runtime* packages

If SystemTap is not already installed, run the following command to install SystemTap on a system configured with YUM repositories for Red Hat.

```
$ sudo yum install systemtap
```

This command installs the *systemtap* and *systemtap-runtime* packages.

Installing the *kernel-debuginfo* and *kernel-debuginfo-common* packages

The kernel image, though compiled with debugging information, is typically stripped to reduce package size. Therefore, install the *kernel-debuginfo* package separately to use the Debugging With Attributed Record Formats (DWARF) information.

Determine if the *kernel-debuginfo* RPM from the Red Hat *rhel-debuginfo* repository matches your running kernel by typing the following command:

```
$ sudo yum --enablerepo=rhel-debuginfo list kernel-debuginfo
```

Compare the output of the previous command to the output of the following command:

```
$ uname -r
```

If these commands return the same kernel version, then the two kernel versions match and you can install the matching *kernel-debuginfo* RPMs. Type the following command:

```
$ sudo yum --enablerepo=rhel-debuginfo install kernel-debuginfo
```

Because the *kernel-debuginfo* package is dependent upon the *kernel-debuginfo-common* package, the *kernel-debuginfo-common* package is also installed when running the previous command.

If your system is running an older kernel (for example, the version that was packaged during release) and you have updated the kernel, you have two choices for continuing with the installation. The first option is to update your kernel to the latest version and use the previous instructions to install the *kernel-debuginfo* packages.

The second option is to browse through the Red Hat FTP site and manually download the kernel-debuginfo package and the dependent kernel-debuginfo-common package that corresponds to the kernel version running on your system. Follow these steps to manually download the matching kernel-debuginfo and kernel-debuginfo-common RPMs:

1. Go to the Red Hat FTP Web site at `ftp://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/os`.
2. Depending upon your platform, choose the appropriate architecture name. You can find your architecture name using the `uname -i` command. For ppc64 machines, the architecture name used on this page is ppc rather than ppc64.
3. Choose *Debuginfo* as the repository that you want to view. A page containing versions of all packages compiled with debugging information is displayed in format `<package>-<version>-debuginfo` RPMs.
4. Find the kernel-debuginfo and kernel-debuginfo-common packages that match your kernel version. For example:
 - For an i386 machine running the 2.6.18-92.el5PAE kernel, download the following RPMs:
 - a. kernel-PAE-debuginfo-2.6.18-92.el5.i686.rpm
 - b. kernel-debuginfo-common-2.6.18-92.el5.i686.rpm
 - For a System p[®] machine running the 2.6.18-92.el5 kernel, download the following RPMs:
 - a. kernel-debuginfo-2.6.18-92.el5.ppc64.rpm
 - b. kernel-debuginfo-common-2.6.18-92.el5.ppc64.rpm

Note: Do not confuse the kernel-debuginfo packages with kernel-debug-debuginfo packages.

5. Install the kernel-debuginfo and kernel-debuginfo-common RPMs by running the following command:
`$ sudo rpm -ivh kernel-debuginfo-*.rpm`

Note: You are prompted to download and install kernel-devel and kernel-headers packages if you have not already done so. If you selected **Development Tools** and **Development Libraries** package groups during installation, the necessary packages are already installed.

6. Check that the installed kernel-debuginfo, kernel-debuginfo-common, and kernel-devel packages match your kernel version. For example, for a System p machine running the 2.6.18-92.el5 kernel, note the output generated by the following command:

```
# rpm -qa|grep kernel
kernel-2.6.18-92.el5
kernel-debuginfo-2.6.18-92.el5
kernel-debuginfo-common-2.6.18-92.el5
kernel-devel-2.6.18-92.el5
```

Run this command on your machine and check that all required packages are listed and that they match your installed kernel version. If the kernel-devel package is not installed, install the software requirements listed in “Software requirements” on page v.

The installation is completed.

Installing SystemTap on SUSE Linux Enterprise Server (SLES) 10 SP2 and 11

SystemTap functionality depends upon the installation of the *systemtap* and *kernel-debuginfo* packages. The kernel-debuginfo package must match the running kernel version on your machine.

The following instructions correspond to SLES 10 SP2 and were tested on both SLES 10 SP2 and SLES 11. Use the instructions as a guide to install SystemTap on SLES 11.

Installing the systemtap package

Install SystemTap on a SUSE Linux Enterprise Server machine by typing the following command:

```
$ sudo yast -i systemtap
```

Installing the kernel-debuginfo package

The kernel image, though compiled with debugging information, is stripped to reduce package size. Therefore, you must install the kernel-debuginfo package separately to use Debugging With Attributed Record Formats (DWARF) data.

You have two options for installing the kernel-debuginfo packages. The first option is to configure debuginfo channels for the package manager installed on your system. See the instructions at [How to add the channel for debuginfo packages \(http://www.novell.com/support/documentLink.do?externalID=3074997\)](http://www.novell.com/support/documentLink.do?externalID=3074997).

The second option is to manually download the packages from the SUSE patch support database. Follow these steps to manually download the matching kernel-debuginfo RPM:

1. Go to SUSE Linux Enterprise patch support database site at <http://support.novell.com/linux/psdb/byproduct.html#sledebug>. You might have to page down or click a product link to find the SUSE Linux Enterprise DEBUGINFO section.
2. Select the DEBUGINFO link that corresponds to your operating system and architecture. For example, if your operating system is SLES 10 SP2 on the i386 architecture, select *SUSE Linux Enterprise DEBUGINFO 10 Service Pack 2 for x86 (i386)*.
3. On the SuSE Linux Enterprise Server Patch Support Database (PSDB) page, find the patch group called **Security update for Linux kernel** and click that link. There might be more than one patch group called **Security update for Linux kernel**. Determine which one matches your running kernel version. You can repeat these steps if the patch group that you initially chose does not match your kernel. Check for older patches by clicking **archive**.
4. On the **Security update for Linux kernel** page, select *SUSE Linux Enterprise 10 SP2 DEBUGINFO for <architecture>*. For example, if your machine is an i386, the link would be *SUSE Linux Enterprise 10 SP2 DEBUGINFO for x86 (i386)*.
5. On the **Linux kernel patch** page, there is a list of available kernel-debuginfo packages with the same version number. Check the version number to see if it matches your kernel version. If you cannot find your version, click **Back** in your browser and repeat step 3 to find another **Security update for Linux Kernel** option to match your kernel patch date.
6. If the correct kernel-debuginfo RPM version is available, click **proceed to download**.
7. Log in.
8. Select the matching debuginfo kernel package to download. For example, if you are running an SMP kernel of version 2.16.60-0.27 on an i586 machine, select the RPM named `kernel-smp-debuginfo-2.16.60-0.27.i586.rpm`. Click **download**.
9. Install the package by typing the following command:

```
$ sudo yast -i <downloaded_path>/kernel-<default/smp/ppc64>-debuginfo.<version_number>.<arch>.rpm
```

The installation is complete.

Optional: Installing on a custom kernel

If you are running a kernel not provided by your distribution, you must compile your kernel with debug information and install *systemtap* and *elfutils* from source packages.

Note: Most Linux distributions do not support products when run on custom kernels.

The following steps describe how to activate kernel options required for SystemTap, how to build a kernel, and how to install SystemTap from source:

1. Compile your kernel with the following options chosen during make menuconfig:

```
General setup --->
[*] Kprobes
[*] Kernel->user space relay support (formerly relayfs)
Kernel hacking --->
[*] Debug Filesystem
[*] Kernel debugging
[*] Compile the kernel with debug info
Security Options --->
[*] Default Linux Capabilities
```

Note: The CONFIG_RELAY kernel option, which corresponds to user space relay support, is added after kernel 2.6.17.

You must choose the CONFIG_DEBUG_INFO=y option before building the image. The debuginfo image is vmlinux. SystemTap assumes that the image with debugging information is available at any of the following paths:

```
/boot/vmlinux-`uname -r`
/usr/lib/debug/lib/modules/`uname -r`/vmlinux
/lib/modules/`uname -r`/vmlinux
```

You can override this search path by setting the SYSTEMTAP_DEBUGINFO_PATH environment variable:

Using an invalid path can cause a failure such as:

```
semantic error: libdwfl failure [...]
```

2. Install the **elfutils** snapshot if it is not already installed:
 - a. Download the latest **elfutils** snapshot from <https://fedorahosted.org/releases/e/1/elfutils/>, as follows:

```
$ wget https://fedorahosted.org/releases/e/1/elfutils/elfutils-<version number>.tar.gz
```

For example:

```
$ wget https://fedorahosted.org/releases/e/1/elfutils/elfutils-0.134.tar.gz
```

- b. Untar the package in a directory *ELFUTILS-SOURCE-DIR*:

```
$ tar -zxvf elfutils-<version_number>.tar.gz
```

For example:

```
$ tar -zxvf elfutils-0.134.tar.gz
```

3. Install SystemTap:

- a. Download the latest SystemTap sources snapshot from <ftp://sources.redhat.com/pub/systemtap/snapshots/>, as follows:

```
$ wget ftp://sources.redhat.com/pub/systemtap/snapshots/systemtap-<version>.tar.bz2
```

For example:

```
$ wget ftp://sources.redhat.com/pub/systemtap/snapshots/systemtap-20080823.tar.bz2
```

- b. Untar the package:

```
$ tar -jxvf systemtap-<version>20080823.tar.bz2
```

Alternatively, you can get the source by using the **git** command:

```
$ git clone git://sources.redhat.com/git/systemtap.git
```

or

```
$ git clone http://sources.redhat.com/git/systemtap.git
```

- c. Build SystemTap from source:

```
cd into the systemtap source directory
```

```
$ ./configure --with-elfutils=<ELFUTILS-SOURCE-DIR>[other autoconf options]
```

or

```
$ ./configure (if elfutils package was already available)
$ make
$ sudo make install
```

The installation is complete.

Chapter 3. SystemTap script examples explained

You can learn the language constructs and understand the working of a SystemTap script by examining sample scripts.

If you are interested in learning about the components of the SystemTap script language, you can read the SystemTap Language Reference guide. This guide is bundled with SystemTap packages in newer distributions (for example, in Red Hat Enterprise Linux 5.2, the guide is available in the `/usr/share/doc/systemtap-<version>/langref.pdf` file). You can also find the guide online at <http://sourceware.org/systemtap/langref/>

The scripts discussed in this section are also available online. You can download the scripts directly from the link provided with each script to try on your machine.

`syscalls_by_proc.stp` example

This script is a small script that measures the number of system calls made by each process and lists a user-defined number of them in descending order.

```
1. #
2. # Print the system call count by process name in descending order.
3. #
4.
5. global syscalls
6.
7. probe begin {
8.     print ("Collecting data... Type Ctrl-C to exit and display results\n")
9. }
10.
11. probe syscall.* {
12.     syscalls[execname()]++
13. }
14.
15. probe end {
16.     printf ("%10s %-s\n", "#SysCalls", "Process Name")
17.     foreach (proc in syscalls-)
18.         printf ("%10d %-s\n", syscalls[proc], proc)
19. }
```

This script is available at http://sourceware.org/systemtap/examples/process/syscalls_by_proc.stp.

The SystemTap scripting language mostly mimics awk scripting language in its syntax. It lacks data-types, declarations (other than global variables), and most indirection, but contains arrays and simplified form of string processing.

Comments in SystemTap script are denoted by using the number sign (#) at the beginning of a line. You can also use other forms such as two forward slashes (//) or a C-style forward slash and asterisk combination (/*). In this example, lines 1 - 3 are commented out and are not executed when the script is run.

Line 5 declares a global variable by the name `syscalls`. The datatype of `syscalls` is not indicated during declaration but is dynamically inferred from its usage in the script. In this script, `syscalls` is being used as an array.

SystemTap scripts are event-driven. The script is run when an occurrence of interest triggers a series of actions defined by the user. The events include invocation of a function, return from a function, or executing a kernel statement. When such events occur, SystemTap takes control and runs the handler

script. The events of interest are defined in terms of *probe* points. For example, if you are interested in creating a probe for the function `sys_fork()` in the kernel (a system call used to create a child process), define a probe as:

```
probe kernel.function("do_fork")
```

Examine lines 7 - 9 to find the probe handler that prints out a statement. Unlike the `sys_fork()` probe discussed above, the probes that are defined in lines 7 - 9 and line 15 - 19 are special probe points in the sense that they are not driven by any kernel action. Instead, as their name suggests, these probes are run when SystemTap is started for begin and end. This behavior also implies that the probes are run once during the lifetime of the script. The begin probes can be used to indicate that SystemTap has begun execution or to set up any global variables that would be eventually used. The end probes can be used to print out the data collected using the probes as in this case. This procedure is like the begin and end constructs in the awk language.

Line 8 uses a print statement to output data to the standard console. A semicolon (;) at the end of the statement can be present and it is up to you to place them.

This script is intended to count the number of system calls made by each process in the system. Therefore, create a probe that invokes the handler every time a system call is made. In Line 11, a counter that is stored in `syscall[execname()]` is incremented every time the probe is hit.

The `syscall[execname()]` expression requires some explanation. The *syscall* global variable is used as an array of counters that are indexed based on the name of the process. Unlike programming languages such as C or Java, where array indexes have to be purely integers, SystemTap can use arrays that are uniquely indexed using any of the allowed data types (data types supported by SystemTap language are described in detail in the Language Reference Guide). An alternative way to understand is to think of `syscall[]` as "bins" with each "bin" being named using the process name that is obtained through the built-in `tapset execname()`. Whenever a *syscall* is made, the counter in the "bin" corresponding to that system call is incremented by one. You can find other such built-in functions by using the `man stapfuncs` command.

The `syscalls_by_proc.stp` script continues to collect data throughout its execution period. The script is terminated using SIGKILL or **Ctrl C**. The end probe runs before the script is terminated and the collected information from `syscalls[]` is outputted to your console.

In line 16 in the end probe, a string containing the headers of the various fields of collected data are outputted. In line 17, there is a looping construct called `foreach`.

```
17.  foreach (proc in syscalls-)
```

The variable *proc* is an index variable that iterates over the range of values possible for the array index of *syscalls*. Also note the en dash (-) after *syscalls* that denotes that the iteration runs in reverse order. This character ensures that the number of system calls made print in descending order. To print in ascending order, change the script to `syscalls+`.

Note: Do not confuse this *proc* variable with the `/proc` interface in Linux. The variable used in this script could be any unused variable name

Line 18 ensures that the counter containing the number of invocations of system call is printed along with the process name.

```
18.  printf("%-10d %-s\n", syscalls[proc], proc)
```

The *proc* variable intelligently replaces the array index when used as `syscalls[proc]` and the `execname()` string during the second usage in the previous line.

With this understanding, run the `syscalls_by_proc.stp` script. For illustration purposes, the screen logs of execution obtained from a `x86_64` machine running Red Hat Enterprise Linux 5.2 is shown in the following output.

```
#> stap -v syscalls_by_proc.stp
Pass 1: parsed user script and 38 library script(s) in 110usr/10sys/129real ms.
Pass 2: analyzed script: 383 probe(s), 9 function(s), 14 embed(s), 1 global(s) in 11360usr/120sys/11484real ms.
Pass 3: using cached /root/.systemtap/cache/1d/stap_1dcde028e9869c50e497e23e130e427c_95302.c
Pass 4: using cached /root/.systemtap/cache/1d/stap_1dcde028e9869c50e497e23e130e427c_95302.ko
Pass 5: starting run.
Collecting data... Type Ctrl-C to exit and display results
#SysCalls Process Name
273      pcscd
40       stapio
26       hald-addon-stor
17       sshd
10       rpc.idmapd
9        automount
6        syslogd
5        gam_server
Pass 5: run completed in 0usr/20sys/4372real ms.
```

The `-v` causes more verbose output. You can increase the verbosity level by adding more `vs` to the option, for example, `-vvv`. The `-v` option has no bearing on the amount of output from your script, but only from the SystemTap runtime. The SystemTap run time runs the script five times.

Pass 1: Parse the script

Pass 2: Elaborate – the various “tapsets” and pre-built functions and link them

Pass 3: Translate – the SystemTap script into C

Pass 4: Compile the kernel module in C into object files

Pass 5: Run - Insert the kernel modules and run them

In Pass 3, a `.c` C file is created under `/tmp` directory and a `.ko` module in Pass 4, while in Pass 5, the actual execution of the script happens starting with the probe begin and terminating after executing probe end.

You can find more information about using SystemTap command options in the section Chapter 6, “Available command line options for SystemTap,” on page 21.

fork.stp example

In this example, the output of each of these SystemTap passes is more verbose. This script (using the “`-vvv`” option) detects the new processes created by placing a probe on the `do_fork()` - worker routine of the `fork()` system call.

```
1. #!/usr/bin/stap
2.
3. global proc_counter
4.
5. probe begin {
6.   print ("Started monitoring creation of new processes....Press ^C to
   terminate\n")
7.   printf ("%25s %-10s %-s\n", "Process Name", "#Process ID", "#Clone
   Flags")
8. }
9.
10. probe kernel.function("do_fork") {
11.   proc_counter++
12.   printf ("%25s %-10d 0x%-x\n", execname(), pid(), $clone_flags)
13. }
14.
```

```

15. probe end {
16.     printf ("\n%d processes forked during the observed period\n",
            proc_counter)
17.}

```

This script is available at <http://sourceware.org/systemtap/wiki/systemtapstarters>.

Like the script in the section “syscalls_by_proc.stp example” on page 11, lines 1-8 and lines 15-17 contain a global variable declaration and probes for begin and end. The script is structured slightly different in this example. The output is logged on to the screen from within the handler of the probe, displaying real-time, continuous data as opposed to logged data as in previous script.

Also note that the probe definition in Line 10 uses the **probe kernel.function()** to place the probe specifically on the function invocation. One of the parameters of the **do_fork()** kernel function is accessed in line 12 using *\$arg_name* (in this case, *\$clone_flags*).

Look at an example output of running the SystemTap **-vvv fork.stp**. Some of the logs have been shortened for brevity:

```

#> stap -vvv fork.stp
SystemTap translator/driver (version 0.6.2/0.131 built 2008-03-12)
Copyright (C) 2005-2008 Red Hat, Inc. and others
This is free software; see the source for copying conditions.
Created temporary directory "/tmp/stapVF2Dvm"
Searched '/usr/share/systemtap/tapset/x86_64/*.stp', found 1
Searched '/usr/share/systemtap/tapset/*.stp', found 37
Pass 1: parsed user script and 38 library script(s) in 120usr/0sys/130real ms.
control symbols: kts: 0xffffffff80064c68 kte: 0xffffffff80067956 stext: 0xffffffff80001000
parsed 'do_fork' -> func 'do_fork'
blacklist regexps:
blfn: ^(atomic_notifier_call_chain|default_do_nmi|__die|die_nmi|do_debug|do_general_protection|
do_int3|do_IRQ|do_page_fault|do_sparc64_fault|do_trap|dummy_nmi_callback|flush_icache_range|
ia64_bad_break|ia64_do_page_fault|ia64_fault|io_check_error|mem_parity_error|nmi_watchdog_tick|
notifier_call_chain|oops_begin|oops_end|program_check_exception|single_step_exception|sync_regs|unhandled_fault|
unknown_nmi_error|.*raw.*lock.*|.*read.*lock.*|.*write.*lock.*|.*spin.*lock.*|.*rwlock.*lock.*|
.*rwsem.*lock.*|.*mutex.*lock.*|raw.*|.*seq.*lock.*|.*apic.*|.*APIC.*|.*softirq.*|.*IRQ.*|.*intr.*|
__delay|.*kernel_text.*|get_current|
current.*|.*exception_tables.*|.*setup_rt_frame.*|.*preempt_count.*|preempt_schedule|__switch_to)$
blfn_ret: ^(do_exit|sys_exit|sys_exit_group)$
blfile: ^(kernel/kprobes.c|arch/.*kernel/kprobes.c)$
focused on module 'kernel' = [0xffffffff80000000-0xffffffff804d6b64, bias 0x0] file
/usr/lib/debug/lib/modules/2.6.18-92.el5/vmlinux ELF machine x86_64 (code 62)
selected function do_fork
probe do_fork@kernel7/fork.c:1443 kernel section=.text pc=0xffffffff80030e65
finding location for local 'clone_flags' near address ffffffff80030e65, module bias 0
Eliding unused function print_regs
Eliding unused function print_backtrace
...
.....
.....
Eliding unused function caller_addr
Eliding unused function caller
Pass 2: analyzed script: 3 probe(s), 3 function(s), 0 embed(s), 1 global(s) in 140usr/40sys/189real ms.
Pass 3: using cached /root/.systemtap/cache/8b/stap_8b2d8ab41b8269222f01c53167f646ea_998.c
Pass 4: using cached /root/.systemtap/cache/8b/stap_8b2d8ab41b8269222f01c53167f646ea_998.ko
Pass 5: starting run.
Running /usr/bin/staprun -v -v -d 23468 /tmp/stapVF2Dvm/stap_8b2d8ab41b8269222f01c53167f646ea_998.ko
staprun:main:236
....
....
staprun:exit_cleanup:203 something exited...
Pass 5: run completed in 10usr/10sys/43941real ms.
Running rm -rf /tmp/stapVF2Dvm
]

```

From this output, notice the lists of blacklisted functions and files, as shown in the following output, which is pruned to avoid duplication.

```
blfn: ^(atomic_notifier_call_chain|default_do_nmi|__die|die_nmi|do_debug|.....|unhandled_fault|
unknown_nmi_error|.*raw_.*lock.*|.read_.*lock.*|.write_.*lock.*|.spin_.*lock.*|
.....|.preempt_count.*|preempt_schedule)$
blfn_ret: ^(do_exit|sys_exit|sys_exit_group|__switch_to)$
blfile: ^(kernel/kprobes.c|arch./kernel/kprobes.c)$
```

The line beginning with **blfn** shows the list of functions over which probes cannot be placed. The line beginning with **blfn_ret** indicates the same for function return probes (using *probe kernel.function(PATTERN).return*). The line beginning with **blfile** contains list of blacklisted files. Such blacklisted functions and files are identified to prevent the system from entering unsafe states during execution. This feature is one of the SystemTap safety features to ensure that the probed system does not become unstable as a result of executing the script. Be aware of the list of blacklisted functions to avoid creating probes for those functions. You can easily identify these functions by following the steps in FAQ 18 at the following Web site: SystemTap FAQ.

iotop.stp example

In this script, examine the I/O operations performed on the system. Such statistics are useful in understanding performance-related issues.

```
1. #!/usr/bin/stap
2.
3. global reads, writes, total_io
4.
5. probe kernel.function("vfs_read") {
6.     reads[execname()] += $count
7. }
8.
9. probe kernel.function("vfs_write") {
10.    writes[execname()] += $count
11. }
12.
13. # print top 10 IO processes every 5 seconds
14. probe timer.s(5) {
15.     foreach (name in writes)
16.         total_io[name] += writes[name]
17.     foreach (name in reads)
18.         total_io[name] += reads[name]
19.     printf ("%16s\t%10s\t%10s\n", "Process", "KB Read", "KB Written")
20.     foreach (name in total_io- limit 10)
21.         printf ("%16s\t%10d\t%10d\n", name,
22.             reads[name]/1024, writes[name]/1024)
23.     delete reads
24.     delete writes
25.     delete total_io
26.     print("\n")
27. }
```

This script is available at <http://sourceware.org/systemtap/examples/io/iotop.stp>.

When this script is run, the output provides the names of the top 10 applications that invoked top Input and Output routines until the script is terminated by pressing **Ctrl C**. A sample output is shown here for reference:

```
> stap -v iotop.stp
Pass 1: parsed user script and 43 library script(s) in 280usr/10sys/468real ms.
Pass 2: analyzed script: 3 probe(s), 2 function(s), 0 embed(s), 3 global(s) in
400usr/610sys/18447real ms.
Pass 3: using cached
/root/.systemtap/cache/b8/stap_b89ee0f2da107f773a50e3ea08367da5_1245.c
Pass 4: using cached
/root/.systemtap/cache/b8/stap_b89ee0f2da107f773a50e3ea08367da5_1245.ko
```

Pass 5: starting run.

Process	KB Read	KB Written
Xorg	26422	0
firefox	14000	0
stapio	3216	0
gnome-screensav	2860	0
wnck-applet	312	0
metacity	288	0
swriter.bin	200	0
notification-da	184	0
gnome-settings-	112	0
dbus-daemon	104	0

Chapter 4. SystemTap safety features

Apart from the identification of potentially unsafe functions for probing and blacklisting them, SystemTap contains several other safety features to ensure the stability of the system.

These features include defining a maximum value for stacks, string sizes, and timeouts while waiting for locks, permitted number of nested calls, and so on. Many programs tend to become unstable or exhibit undefined behavior primarily due to these factors. SystemTap provides these values and enforces them for you. When required, these limits can be overridden by using the `-D MACRO=VALUE` command-line option.

For a complete discussion of SystemTap language constructs, see the SystemTap Language Reference that is bundled along with SystemTap as `langref.pdf` or else view a copy online at <http://sourceware.org/systemtap/langref/>.

SystemTap allows pure C language code to be embedded within SystemTap functions, using the constructs ampersand left bracket (`%{`) and ampersand right bracket (`%}`). This feature is especially useful when you want deep levels of pointer indirection – a feature typically used by SystemTap tapsets to create powerful tapset functions. However, these SystemTap safety features do not work on the embedded code. Therefore, embedded code is susceptible to the faults that are prevented in SystemTap scripts, like infinite loops.

Chapter 5. Debuginfo-less probing for function boundary tracing

SystemTap allows you to insert probes on specific kernel statements. You can do this if an image containing all debugging information is available at a location specified by the `SYSTEMTAP_DEBUGINFO_PATH` environment variable.

This feature is available only in the more recent versions of SystemTap such as SystemTap version 0.7 or newer.

If your machine is running the SystemTap package that comes with Red Hat Enterprise Linux 5.2 (version 0.6.2) or SUSE Linux Enterprise Server 10 SP2 (version 0.5.8), download and install the matching kernel-debuginfo RPMs. See “Installing SystemTap on Red Hat Enterprise Linux (RHEL) 5.2” on page 5 or “Installing SystemTap on SUSE Linux Enterprise Server (SLES) 10 SP2 and 11” on page 6 for instructions. Next, download and install a recent version of the `elfutils` package and SystemTap as described in step 2 and 3 in “Optional: Installing on a custom kernel” on page 7.

For most enterprise distributions, there is a separate package containing the kernel image built with debugging information. However, this image might not be available for your system. SystemTap is versatile enough to operate without this information by using function boundary tracing. In this case, probes triggered during function entry and exit can be placed using:

```
probe kernel.function(<function_name>)
```

The function parameters can be accessed by the probe handler using `$arg<n>` where ‘n’ is the argument number. For example, use `$arg1`, `$arg2`, and so on.

The script described in “fork.stp example” on page 13 to trace the new processes can be rewritten to work in a debuginfo-less argument, as follows:

```
#!/usr/bin/stap
global proc_counter
probe begin {
    print ("Started monitoring creation of new processes....Press ^C to terminate\n")
    printf ("%25s %10s %s\n", "Process Name", "#Process ID", "#Clone Flags")
}

probe kernel.function("do_fork") {
    proc_counter++
    printf ("%25s %10d 0x%x\n", execname(), pid(), ulong_arg(1))
}

probe end {
    printf ("\n%d processes forked during the observed period\n", proc_counter)
}
```

You can download this script at <http://sourceware.org/systemtap/wiki/systemtapstarters>.

For this script, pass in the symbol table when running SystemTap, as follows:

```
# /usr/bin/stap -v --kmap=/proc/kallsyms fork.stp
```

Chapter 6. Available command line options for SystemTap

Previous examples demonstrated how the SystemTap command can be used. Included here is a list of the various options.

```
-- end of translator options, script options follow
-v increase verbosity [0]
-h show help
-V show version
-k keep temporary directory
-u unoptimized translation
-w suppress warnings
-g guru mode
-P prologue-searching for function probes
-b bulk (percpu file) mode
-s NUM buffer size in megabytes, instead of 0
-p NUM stop after pass NUM 1-5, instead of 5
(parse, elaborate, translate, compile, run)
-I DIR look in DIR for additional .stp script files, in addition to
/usr/local/share/systemtap/tapset
-D NM=VAL emit macro definition into generated C code
-R DIR look in DIR for runtime, instead of
/usr/local/share/systemtap/runtime
-r RELEASE cross-compile to kernel RELEASE, instead of 2.6.25.6-55.fc9.i686
-m MODULE set probe module name, instead of stap_15237
-o FILE send output to file, instead of stdout
-c CMD start the probes, run CMD, and exit when it finishes
-x PID sets target() to PID
-t collect probe timing information
-q generate information on tapset coverage
--kelf make do with symbol table from vmlinux
--kmap[=FILE]
make do with symbol table from nm listing
--ignore-vmlinux
for testing, pretend vmlinux can't be found
--ignore-dwarf
for testing, pretend vmlinux and modules lack debug info
```

Chapter 7. Root versus non-root user probing

SystemTap is a translator tool that parses, elaborates (like linking in compiler parlance), and translates the script to a 'C' language kernel module, then compiles and runs the module.

To actually run the kernel objects it builds, your user ID must be one of the following:

- the root user
- a member of the `stapdev` group
- a member of the `stapusr` group

Members of the `stapusr` group can only use modules located in the `/lib/modules/VERSION/systemtap` directory. This directory must be owned by root and not be world-writable.

The kernel modules generated by SystemTap program are run by the `staprun` program. This program is a part of SystemTap that is dedicated to module loading and unloading (but only in the white zone) and kernel-to-user data transfer. Because `staprun` does not perform any additional security checks on the kernel objects it is given, do not add untrusted users to the `stapdev` or `stapusr` groups.

Part 2. Appendixes

Troubleshooting tips

This topic discusses troubleshooting tips and caveats.

Probing blacklisted functions

If you want to probe a blacklisted function, you can probe it by placing “kernel markers” in the code. Next, create a SystemTap marker handler using the *probe kernel.mark* function. Kernel markers cannot be dynamically inserted into a running system. Therefore, recompile the kernel image and reboot the system into the newly built kernel. You can find detailed instructions at <http://sourceware.org/systemtap/wiki/UsingMarkers>.

Kernel tracing

One of the primary uses of SystemTap is kernel tracing such as collecting logs over a time to better understand the kernel activity. Collecting logs typically produces large amounts of data. Data collection is accomplished in the respective probe handlers. Store the data in global variables and output the data in the **end** probe. Consider also the use of *aggregates*.

For a complete discussion of SystemTap language constructs, see the SystemTap Language Reference provided with SystemTap (as *langref.pdf*) or see <http://sourceware.org/systemtap/langref/>.

Symbol name resolution

SystemTap is highly dependent upon symbol name resolution from the debuginfo image and from the kernel symbol table. If SystemTap scripts are run on a system that does not have the kernel-debuginfo package installed, it might create errors like:

```
# stap -v syscall.stp
```

```
Pass 1: parsed user script and 38 library script(s) in 120usr/10sys/212real ms.
semantic error: libdwfl failure (missing kernel 2.6.18-92.el5 x86_64 debuginfo): No such file or
directory while resolving probe point kernel.function("sys_accept")?
semantic error: no match while resolving probe point kernel.function("sys_access")
```

This error is caused by installing the wrong version of kernel-debuginfo package.

Mismatches in kernel versions

Mismatches in kernel versions might lead to incorrect inferences by the SystemTap runtime. For example, the SystemTap runtime might run a SystemTap script with an older version of kernel-debuginfo image.

Another common oversight is running a SystemTap script without installing the kernel-debuginfo image or not having a vmlinux image built with CONFIG_DEBUG_INFO in any one of these paths. This oversight is most common for custom kernels, for example:

```
/boot/vmlinux-`uname -r`
/usr/lib/debug/lib/modules/`uname -r`/vmlinux
/lib/modules/`uname -r`/vmlinux
```

Non-inlined functions

It is possible that, as a result of compiler optimization, a function that was previously non-inlined is turned into an inline function. For example, compiler optimizations can cause a function to be inlined at a few call-sites, but non-inlined in others. SystemTap cannot use its typical function probing techniques due to the absence of a function call-stack and function prologue in case of non-inlined functions.

Therefore it cannot be probed using the `probe kernel.function()` syntax and can result in various types of error messages, depending upon the script usage. Consider verifying if the probed function is non-inlined, even if the original function is not explicitly declared inline. The dwarves library provides this feature through a command named `pfunct -cc_inlined <object_file>` that lists such functions in the object file.






SystemTap development community

The SystemTap development community maintains a list of bugs that were reported by users or discovered during testing. You can search through this list to match and identify any issue seen with any listed bugs.

Related information and downloads

You can find additional information about the processes and tools described in these procedures.

Related information

- SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems 
<http://www.redbooks.ibm.com/redpapers/pdfs/redp4469.pdf>
- Kernel Marker 
<http://sourceware.org/systemtap/wiki/UsingMarkers>
- Language Reference Guide 
<http://sourceware.org/systemtap/langref/>
- SystemTap FAQ 
<http://sourceware.org/systemtap/wiki/SystemTapFAQ>
- Red Hat FTP
<ftp://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/os>
- How to add the channel for debuginfo packages
<http://www.novell.com/support/documentLink.do?externalID=3074997>
- SUSE Linux Enterprise patch support database
<http://support.novell.com/linux/psdb/byproduct.html#sledebug>
- DeveloperWorks SystemTap Blueprint Community Forum 
<http://www-128.ibm.com/developerworks/forums/forum.jspa?forumID=1323>

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 903
11501 Burnet Road
Austin, TX 78758-3400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com)[®] are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol ([®] and [™]), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml

Adobe[®], the Adobe logo, PostScript[®], and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java[™] and all Java-based trademarks and logos are registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX[®] is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of the manufacturer.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of the manufacturer.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any data, software or other intellectual property contained therein.

The manufacturer reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by the manufacturer, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

THE MANUFACTURER MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THESE PUBLICATIONS ARE PROVIDED $\hat{\Delta}$ AS-IS $\hat{\Delta}$ AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.



Printed in USA