A visitation of sysdig

Project Report

Jan-Willem Selij, Eric van den Haak

June 1, 2014

UNIVERSITY OF AMSTERDAM

Abstract

This research focuses on using sysdig, a tool that caotures system calls, for a forensic purpose. First, implications regarding requiring a kernel module to be able to run sysdig have been researched. This was performed by trying to run sysdig on a common Linux distribution with a default and a self-compiled non-modular kernel. Complications arise when a non-modular kernel is used as sysdig can not run without its kernel module. Thereafter, completeness of the output has been researched. Source code shows that sysdig drops events when its buffer gets full. Finally, the research is focused upon the verifiability of sysdig. Multiple experiments have been set up in which sysdig has been actively running. Results illustrate that sysdig's output is very alike but never the same. The investigation of sysdig shows that sysdig can be useful to find out what is going on on a system as it logs as many events as possible and does not slow down the system. The ability of running sysdig without root privileges or on a non-modular kernel proves to be almost impossible.

Contents

1 Introduction

Sysdig [2] is a system-level exploration and troubleshooting tool for Linux. It captures system calls and other system level events using a Linux kernel facility called tracepoints. It requires a driver to be present in the kernel space because it has to register tracepoints for system calls. The information gets "packetized" which allows it to be saved to trace files, similar to the way tcpdump parses network packets. These captures can be filtered using extensions called chisels. This allows for more specific reports on the current systems state, for example on CPU, I/O and network usage of a certain process. While the main intended purpose is aimed at troubleshooting Linux-based systems, this research focuses on looking at the tool from a forensics angle. Because sysdig provides insight in system activities, it might be useful as a monitoring tool.

1.1 Related research

Sysdig was initially released on April 3, 2014. Because of its infancy, no research has been done yet.

1.2 Research questions

Our main purpose is to determine if sysdig is suitable to be used in forensic environments. Therefore it is necessary to research complications of getting the tool running on a Linux environment. For forensics, the tool has to give a complete output. We thus have to test if all, or to what extend, our system calls will be in sysdig's output. The tool also has to be verifiable. This might be a problem because the tool monitors runtime system calls and can therefore (almost) never give the same output on another time on the same system. We have to come up with a method to verify this ourselves. If it is possible to circumvent the tool, forensic usage of it might not be ideal. If time allows us, we can test whether we can circumvent the tool by for example a rootkit.

Is sysdig suitable for forensics?

- What are the implications of requiring a kernel driver?
- Is sysdig's output complete?
- Is sysdig's output verifiable?
  - Is it possible to circumvent sysdig?

2 Research

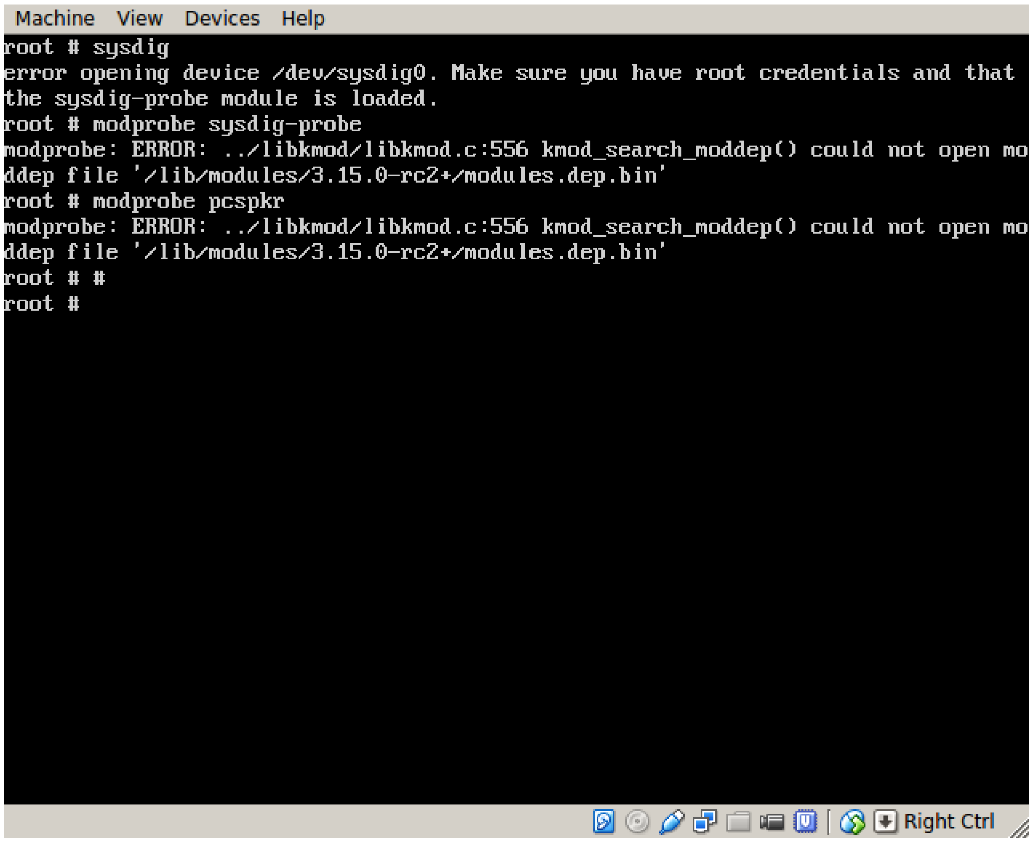
This section describes the research done.

2.1 Implications

First, research has been done upon the implications to get sysdig running. It is given that in current state, it is necessary to insert a kernel module into the running kernel in order to be able to capture system calls. On normal Linux distributions this is possible when having root access. While focusing on live forensics, the current kernel of a system has to be used so that current system behavior can be monitored.

Non-modular kernel

In some cases, non-modular kernels are used to improve security. Since sysdig requires compiling a kernel module and loading it into the kernel, this should not work on a non-modular kernel. To verify this, a non-modular kernel was compiled and the sysdig module separately. Sysdig was ran which resulted in the system failing to open the *sysdig0* device. Manually trying to insert the module or any other failed as expected.



```
root # sysdig
error opening device /dev/sysdig0. Make sure you have root credentials and that
the sysdig-probe module is loaded.
root # modprobe sysdig-probe
modprobe: ERROR: ../libkmod/libkmod.c:556 kmod_search_moddep() could not open mo
ddep file '/lib/modules/3.15.0-rc2+/modules.dep.bin'
root # modprobe pcspkr
modprobe: ERROR: ../libkmod/libkmod.c:556 kmod_search_moddep() could not open mo
ddep file '/lib/modules/3.15.0-rc2+/modules.dep.bin'
root # #
root #
```

Figure 1: Trying to run sysdig on non-modular kernel.

Kernel memory

However, this might still be possible through an alternative way. The *inskmem* utility [4] allows one to load kernel modules through the use of */dev/kmem*. This device exposes the kernel memory where *inskmem*, with the use of the mapped memory, uses *kmalloc* in combination with some other system calls to load the module. This method has not been tested and comes with a few things to think about. Most distributions have already disabled access to the kernel memory through either *grsecurity* or compiling the kernel with the `CONFIG_DEVKMEM` set to false or `CONFIG_STRICT_DEVMEM`, as this device was also used by rootkits. The utility is fairly old and not tested on 3.x kernels, as well on 64-bit.

2.2 Completeness

Secondly, research has been done upon completeness of sysdig. Authors and sourcecode of sysdig shows that when the event buffer fills up, sysdig will drop events[3]. While this may result in an unwilling loss of events, sysdig won't slow down the system itself and thus won't interfere with current system's behavior as dtrace and strace might do.

Sysdig will show the total amount of events and the amount of dropped ones if ran with the verbose flag (*-v*), as indicated by the source code:

```
if(verbose)
{
  fprintf(stderr, "Driver Events:%" PRIu64 "\nDriver Drops:%" PRIu64 "\n"
      ,
  cstats.n_evts,
  cstats.n_drops);
```

Listing 1: /userspace/sysdig/sysdig.cpp

It also logs are more exact result on why this is to *dmesg*:

```
pr_info("closing ring %d, evt:%llu, dr_buf:%llu, dr_pf:%llu, pr:%llu, cs
    :%llu\n",
           ring_no,
           ring->info->n_evts,
           ring->info->n_drops_buffer,
           ring->info->n_drops_pf,
           ring->info->n_preemptions,
           ring->info->n_context_switches);
```

Listing 2: driver/main.c

Where `dr_buff` stands for Drops/Buffer and `dr_pf` for Drops/Page fault.

2.3 Verifiability

It is important for forensic utilities that they yield the same output results if ran multiple times with the same source. Sysdig should output the same results if ran multiple times on a very same scenario. While this sounds near impossible because of the fact that system calls behavior differ upon each system run, a few tests were conducted.

To limit the influence from outside matters a dedicated hardware setup was used. This contained just the basics needed for running a computer; motherboard, CPU, memory and a hard disk. A Linux distributed was installed with the default configuration (in our case Arch Linux) and sysdig was installed as advised on the website. A script was ran directly after booting the machine which runs sysdig for 10,000 lines and stores the output. This was repeated five times. This test was also performed in single user mode, because there was a lot of system call noise by booting the regular way.

The same test was also performed by using a virtual machine with the help of Vagrant[1]. A standard Vagrant box was downloaded and modified to always start in single user mode followed by starting sysdig. The box was then exported as a new box. Vagrant then takes care of starting the box from the same exported point to minimize any influences caused by a regular reboot.

Overview of performed tests

Scenario	Description
Virtual	Boot a virtual machine in single user mode
Hardware	Boot a hardware machine in multi user mode
Hardware	Boot a hardware machine run single script

Each scenario ran 5 times and recorded 10000 system calls.

2.3.1 Findings

There are a few expected findings in the output. The system clock changed so the time in the log differs and the process id listed in the log differs.

Comparing the five logs from either test shows that the system calls are nearly the same. It happens often that one test shows a few more system calls which causes one to “drift”. This means the sequence of the system calls are the same, only with an offset.

It is noticeable that sysdig drops some system calls by looking at the event id. Dropping the logging of system calls only happens when the host system is too busy. To not slow the system down by letting the log catch up a number are dropped. The reason for this behavior is unknown because the system should have plenty of cycles left because it is in single user mode. This is also reflected later in the log because the only system calls that are made are context switches between running sysdig and other system processes.

2.3.2 Differences

In order to understand the differences between the captures they are laid out below each other. Lines are truncated as this improves clarity and does not hinder the comparison. The differences are indicated by a diff-like syntax where `>` highlights certain lines, `+` indicates an addition, `-` indicates deletion and `@@@` indicates a range between a the first number above and below it. Regular diffs would not clearly show any shift in system calls in addition that the event numbers change a lot more together with the time so no line would actually match.

```
30 13:47:06.437306006 0 rsyslogd (722) > gettimeofday
31 13:47:06.437310196 0 rsyslogd (722) < gettimeofday
> 32 13:47:06.437312990 0 rsyslogd (722) > gettimeofday
> 33 13:47:06.437313269 1 <NA> (0) > switch next=711(rs:main)
34 13:47:06.437316622 0 rsyslogd (722) < gettimeofday
```

Listing 3: Shifted sequence. Test 1

```
30 13:56:51.456925558 0 rsyslogd (690) > gettimeofday
31 13:56:51.456929190 0 rsyslogd (690) < gettimeofday
> 32 13:56:51.456931425 1 <NA> (0) > switch next=688(rs:main)
> 33 13:56:51.456932263 0 rsyslogd (690) > gettimeofday
34 13:56:51.456935894 0 rsyslogd (690) < gettimeofday
```

Listing 4: Shifted sequence. Test 2

The system calls exactly match until event number 32. The scheduler switch for test 2 is a little bit earlier. Note that no system call is actually omitted, the event enter of `gettimeofday` (indicated by the greater-than mark, `>`) isn't show in its usual pair on second test, but directly after the switch.

The following comparison shows that an extra event appears after the `gettimeofday` combo, called `switch`.

```
54 13:47:06.437381434 1 rs:main (711) > gettimeofday
55 13:47:06.437384787 1 rs:main (711) < gettimeofday
56 13:47:06.437388698 1 rs:main (711) > write fd=4(<f>/var/lo
57 13:47:06.437391771 1 rs:main (711) < write res=125 data=Ma
58 13:47:06.437394285 1 rs:main (711) > gettimeofday
```

Listing 5: Scheduler switch. Test 1

```
54 13:56:51.457027247 1 rs:main (688) > gettimeofday
55 13:56:51.457031717 1 rs:main (688) < gettimeofday
+ 56 13:56:51.457034231 0 <NA> (3) > switch next=0
57 13:56:51.457034790 1 rs:main (688) > write fd=1(<f>/var/lo
58 13:56:51.457038142 1 rs:main (688) < write res=125 data=Ma
```

Listing 6: Scheduler switch. Test 2

To make system calls and arguments readable, sysdig has a mapping which converts this internally.

event_table.c states the following about `switch`:

```
/* PPME_SCHEDSWITCH_E */{"switch", EC_SCHEDULER, EF_NONE, 1, {{"next",
   PT_PID, PF_DEC} } },
```

This means it's an extra scheduler switch to PID 0, not showing up the first test.

The final comparison shows that, in the first test quite a few events don't show up (which are dropped). They are, however, visible on the second test.

```
200 13:47:06.437846298 1 rs:main (711) > futex addr=20C3194 o
201 13:47:06.437861383 1 rs:main (711) > switch next=0
202 13:47:06.445224609 0 <NA> (0) > switch next=3
203 13:47:06.445265117 0 <NA> (3) > switch next=0
204 13:47:06.457241500 0 <NA> (0) > switch next=3
205 13:47:06.457266084 0 <NA> (3) > switch next=0
206 13:47:06.461236701 0 <NA> (0) > switch next=3
207 13:47:06.461262961 0 <NA> (3) > switch next=0
208 13:47:06.461277209 0 <NA> (0) > switch next=3
209 13:47:06.461289221 0 <NA> (3) > switch next=0
210 13:47:06.467217349 0 <NA> (0) > switch next=785(sysdig)
- 211
@@@
- 217
- 9xx
978 13:47:06.516773361 0 <NA> (0) > switch next=785(sysdig)
986 13:47:06.547071368 0 <NA> (0) > switch next=785(sysdig)
994 13:47:06.577511798 0 <NA> (0) > switch next=785(sysdig)
1002 13:47:06.607772090 0 <NA> (0) > switch next=785(sysdig)
1010 13:47:06.638048000 0 <NA> (0) > switch next=785(sysdig)
1018 13:47:06.668338993 0 <NA> (0) > switch next=785(sysdig)
1026 13:47:06.698588387 0 <NA> (0) > switch next=785(sysdig)
1034 13:47:06.728837449 0 <NA> (0) > switch next=785(sysdig)
1042 13:47:06.759141930 0 <NA> (0) > switch next=785(sysdig)
1050 13:47:06.789469668 0 <NA> (0) > switch next=785(sysdig)
1054 13:47:06.795728300 1 <NA> (0) > switch next=563(dhclient
1055 13:47:06.795741150 1 dhclient3 (563) < select res=0
1056 13:47:06.795744503 1 dhclient3 (563) > gettimeofday
1057 13:47:06.795748414 1 dhclient3 (563) < gettimeofday
1058 13:47:06.795760147 1 dhclient3 (563) > gettimeofday
1059 13:47:06.795763500 1 dhclient3 (563) < gettimeofday
1060 13:47:06.795766014 1 dhclient3 (563) > select
```

Listing 7: Dropped events. Test 1

```
200 13:56:51.457465571 1 rs:main (688) < gettimeofday
201 13:56:51.457468364 1 rs:main (688) > futex addr=213F194 o
202 13:56:51.457483171 1 rs:main (688) > switch next=0
203 13:56:51.468859478 0 <NA> (0) > switch next=3
204 13:56:51.468922894 0 <NA> (3) > switch next=0
205 13:56:51.472832609 0 <NA> (0) > switch next=3
206 13:56:51.472844063 0 <NA> (3) > switch next=0
207 13:56:51.472870882 0 <NA> (0) > switch next=3
208 13:56:51.472881218 0 <NA> (3) > switch next=0
209 13:56:51.472894907 0 <NA> (0) > switch next=3
210 13:56:51.472901891 0 <NA> (3) > switch next=0
+ 211 13:56:51.472915301 0 <NA> (0) > switch next=3
+ 212 13:56:51.472922285 0 <NA> (3) > switch next=0
+ 213 13:56:51.472935694 0 <NA> (0) > switch next=3
+ 214 13:56:51.472942678 0 <NA> (3) > switch next=0
+ 215 13:56:51.477600254 0 <NA> (0) > switch next=3
+ 216 13:56:51.477610311 0 <NA> (3) > switch next=0
+ 217 13:56:51.487764115 0 <NA> (0) > switch next=764(sysdig)
989 13:56:51.520816637 0 <NA> (0) > switch next=764(sysdig)
997 13:56:51.557827768 0 <NA> (0) > switch next=764(sysdig)
1005 13:56:51.588148658 0 <NA> (0) > switch next=764(sysdig)
1013 13:56:51.618454469 0 <NA> (0) > switch next=764(sysdig)
1021 13:56:51.648792680 0 <NA> (0) > switch next=764(sysdig)
1029 13:56:51.679088713 0 <NA> (0) > switch next=764(sysdig)
1037 13:56:51.709355120 0 <NA> (0) > switch next=764(sysdig)
1045 13:56:51.739652284 0 <NA> (0) > switch next=764(sysdig)
1053 13:56:51.769935147 0 <NA> (0) > switch next=764(sysdig)
1061 13:56:51.778850268 1 <NA> (0) > switch next=548(dhclient
```

Listing 8: Dropped events. Test 2

offset of one extra system call not corrected in the comparison

There are a couple more scheduler switches between PID 0 and PID 3 (*ksoftirqd/0*).

Also note the difference in event numbers. The first test reports far less system calls in the 200-range. The system calls on the on the second test might actually have happened, but are dropped. If the scheduler switches are disregarded for a moment it is apparent that both systems continue their path by calling *dhclient3* instead of deviating from it.

Aside from some extra reported scheduler switches, both follow the same path.

2.3.3 Graphical view

In order to make sysdig's output easier to compare, the output has been put into graphs. On each graph the process along with an event is shown as is the number of system calls belonging to this combination.

Virtual machine

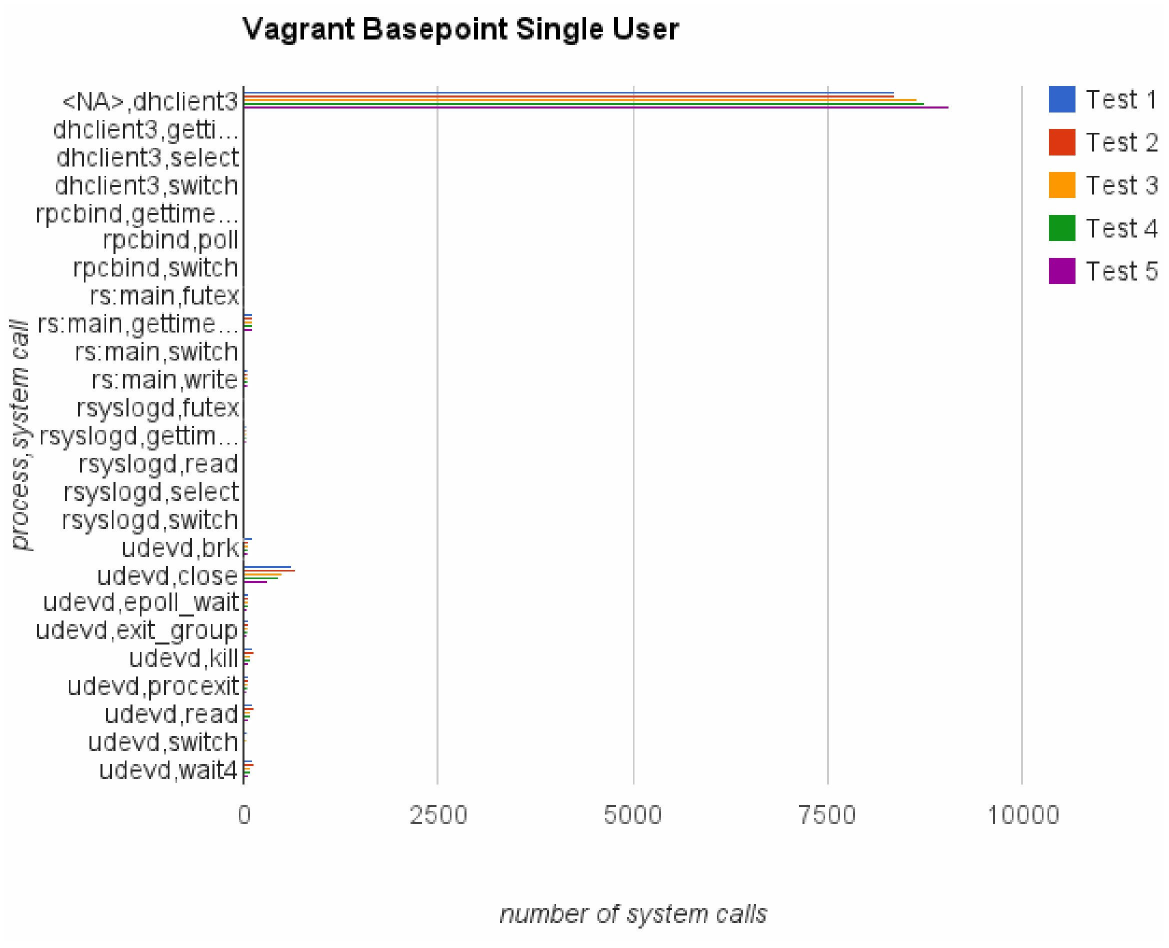


Figure 2: Virtual, all system calls

It is clear that two processes claim most system calls. To give a better view, these are filtered in the next image.

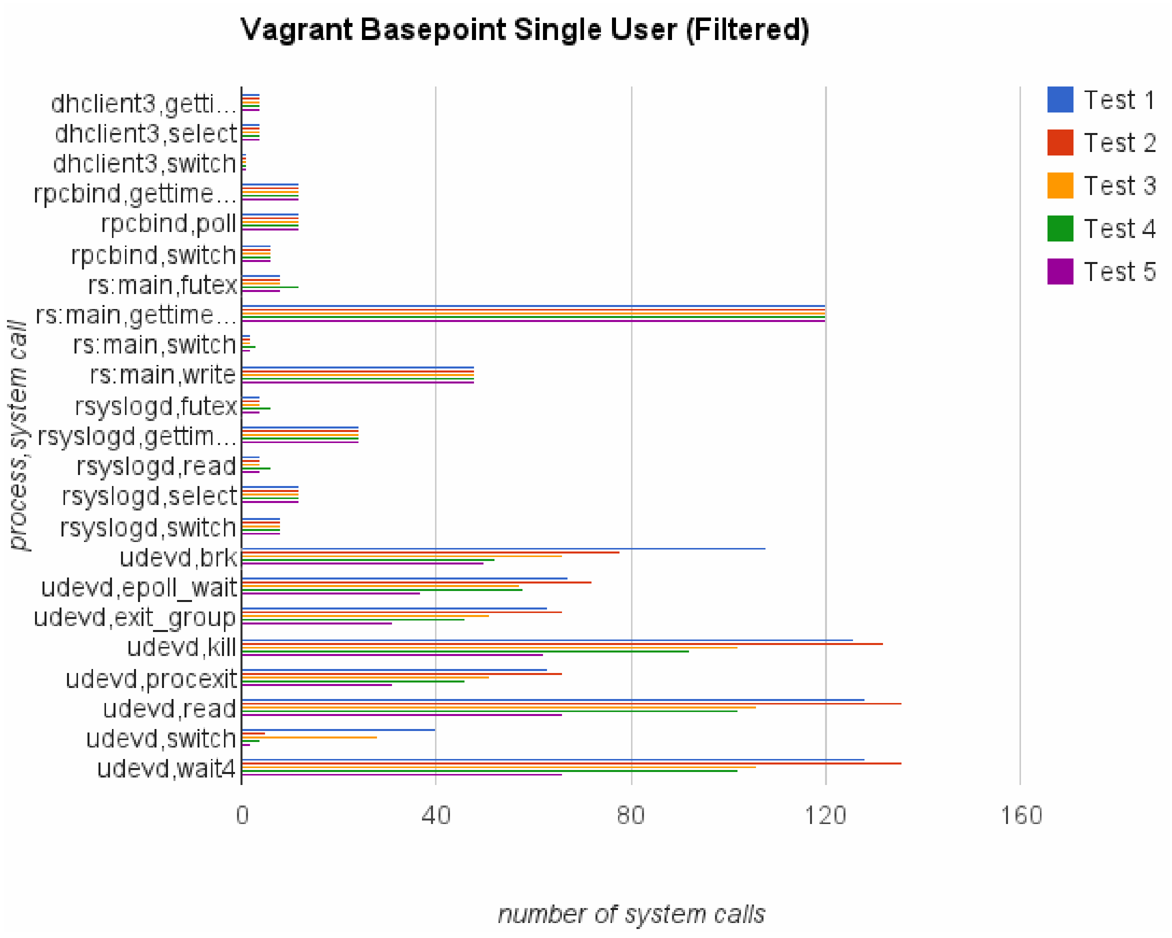


Figure 3: Virtual, filtered.

This indicates that, except for the *udevd* process (hardware related), system call behavior is very alike.

Hardware machine, multi user boot

On the multi user boot, the amount of system calls that occured did not allow for a clear graph. Therefore these graphs show the number of total system calls per process.

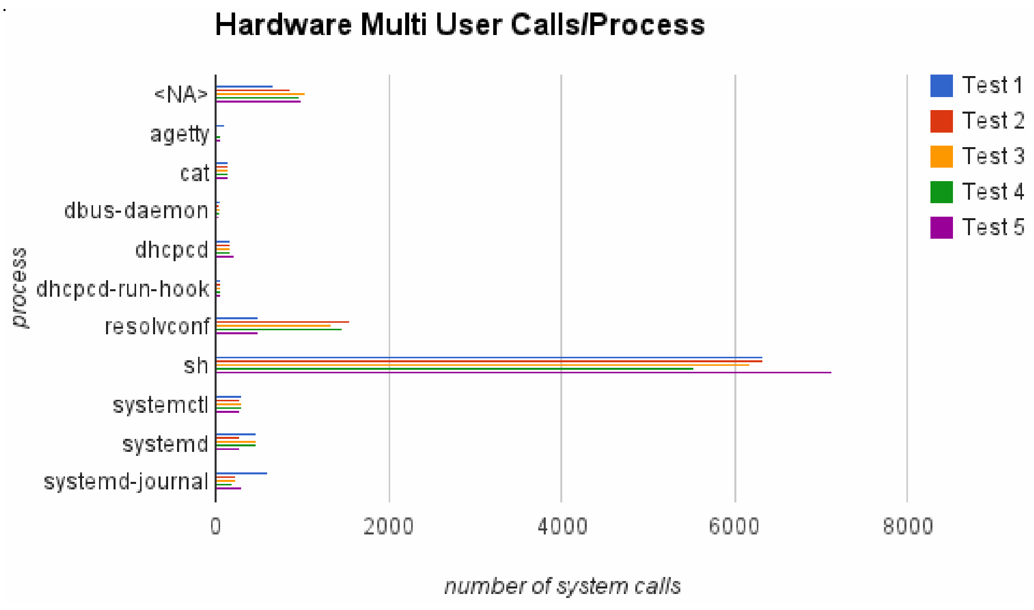


Figure 4: Hardware, multi user boot, system calls per process

The *sh* (shell) process is filtered out in the next image.

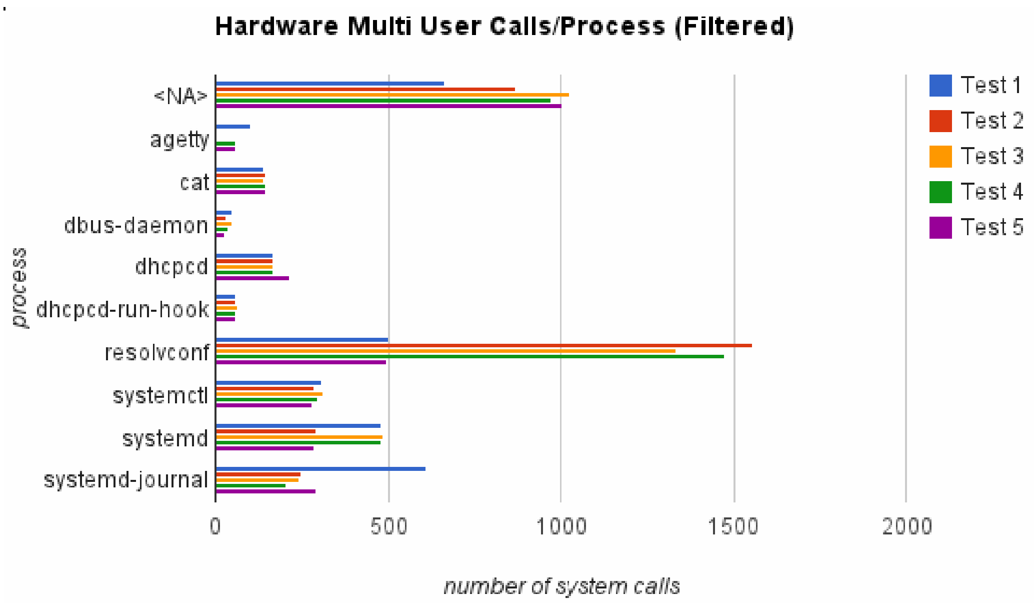


Figure 5: Hardware, multi user boot, filtered

The output is quite alike. The one that has a totally different behavior is *resolvconf*. This is expected because this process requires networking and that causes unpredictable delays.

Hardware machine, single script

This test had a single script (called sysdigscript) running which performed a few thousand disk writes and a few thousand calls to `stdout` using *echo*. This made this test the most predictable because the behavior was known, besides executing sysdig at the same time.

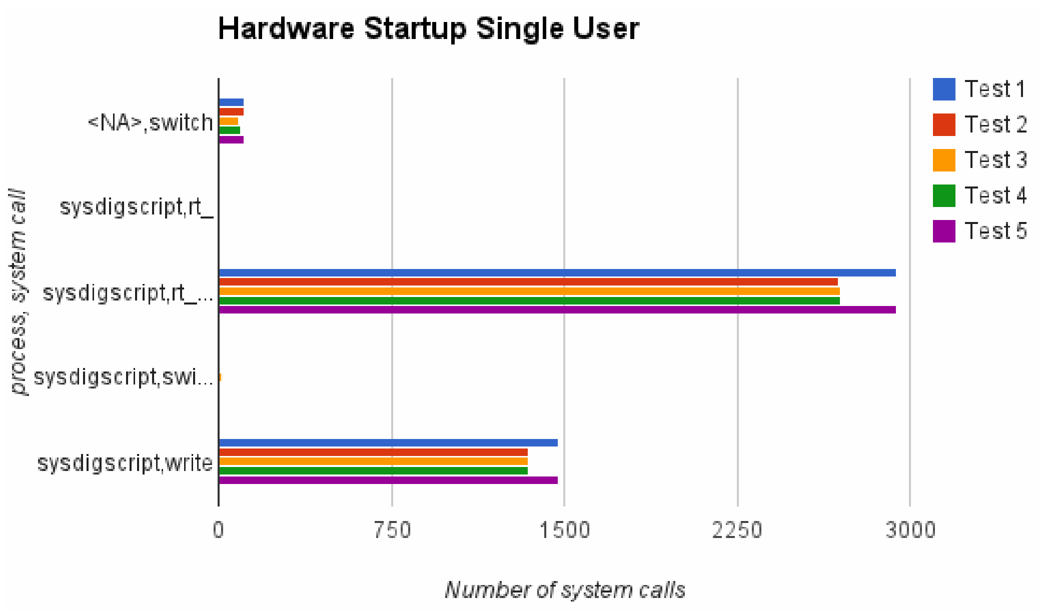


Figure 6: Hardware, single user bootup script

Output is very alike, but still differs among tests

3 Conclusion

This section concludes our findings.

What are the implications of requiring a kernel driver?

Having root access to a system which has a modular Linux kernel running results in no implications at all. However, when there is no root access or the kernel does not allow loading of modules, it is not possible to run sysdig[1].

Is sysdig's output complete?

Sysdig's output is as complete as it can be, meaning that it only drops events when necessary to not slow down the system. Therefore it can not be concluded that the output is complete, as this is not always the case. Source code however does show that given output is the raw output received via the kernel hook.

Is sysdig's output verifiable?

After running multiple tests in the same scenarios, findings suggest that sysdig's output is never the same but is very alike. As shown in figure 3, most differences stem from hardware system calls. As shown in figure 6, it can be stated that despite the fact the number of system calls differ upon each run, the behavior is the same.

Is sysdig suitable for forensics?

Sysdig can be very helpful for forensics as it can run on the system without slowing it down and provides a good insight in what is going on on the system. As output is "packetized", results can be hashed afterwards and stored in a sound way. However, it is not recommended to fully rely on sysdig as it can omit events when the system is under high load resulting in too many system calls.

[1]There is a possible */dev/kmem* solution which is listed as further research.

4 Future work

In order to gain completeness with sysdig, the source code might be altered to optionally disable event dropping. This can result in a complete view of the system but might slow it down. As research time was limited this can be a topic for future research.

Another aspect what can be researched is trying to get sysdig running on a non-modular kernel. As this research shows, there is a theoretical possibility with *inskmem*.

Since sysdig relies on system calls, it can be interesting to look at the effects of having a root kit active. If it is possible to use such a kit to circumvent sysdig, it might not be providing a good view of current system behavior.

References

[1] HashiCorp. Vagrant, 2013. http://www.vagrantup.com/.

[2] DRAIOS INC. sysdig, 2014. http://www.sysdig.org/.

[3] DRAIOS INC. sysdig, 2014. http://draios.com/sysdig-vs-dtrace-vs-strace-a-technical-discussion/.

[4] Thomas Wana. inskmem, 2004. https://www.wana.at/inskmem/.