DTrace: New Additions to an Old Tracing System

BY DOMAGOJ STOLFA

DTrace is a software tracing framework built into FreeBSD that allows users to inspect and modify the currently running system in real time. It is highly extensible and was originally built for Solaris, but has since been ported many times to other environments such as FreeBSD, macOS, Windows and Linux. This article will focus on DTrace usage in FreeBSD with examples and give a summary of recent developments in the DTrace space on FreeBSD.

DTrace in Short

Operating systems are very complicated pieces of software which have many components. A single tracing system attempting to support tracing of nearly the entire OS can be overwhelming given their complexity. In order to simplify this as well as to account for future extensions, DTrace introduces the notion of a *provider*. Providers live in the kernel as kernel modules by default in FreeBSD and are responsible for implementing the necessary functionality to instrument a particular component of the OS. They expose DTrace *probes* which are names for locations in the OS code that can be dynamically instrumented with scriptable routines written in the D programming language. Some example providers shipped with FreeBSD include the function boundary tracing provider (`fbt.ko`) — responsible for instrumentation of kernel function entry and exit points, the profile provider (`profile.ko`) which provides probes associated with a fixed time-based interrupt specified by the script-writer, the PID provider (`fasttrap.ko`) which implements `fbt` but for user processes and the libraries they link to and various others. While deep knowledge of DTrace is not required in order to understand this article, those wishing to know more about DTrace may want to check out the user guide[1], specification[2], FreeBSD Handbook Page[3], whitepaper[4], book[5] and various FreeBSD wiki pages that can be found such as the list of one-liners[6]. Furthermore, a number of previous FreeBSD Journal editions featured articles on DTrace[7,8,9].

Simple Examples

Probes are specified via a `provider:module:function:name` 4-tuple. Each of the entries can be globbed or left blank to mean "everything". We use an example toy snooper script as an introduction to D. The script tells us which programs users are running. Note that we specify the `-x quiet` option to avoid additional information that DTrace would otherwise output.

```
# dtrace -x quiet -n 'proc:::exec { printf("user = %u, gid = %u: %s\n", uid, gid, 
stringof(args[0])); }'
user = 1001, gid = 1001: /usr/sbin/service
user = 1001, gid = 1001: /bin/kenv
user = 1001, gid = 1001: /sbin/sysctl
user = 1001, gid = 1001: /sbin/env
user = 1001, gid = 1001: /bin/env
user = 1001, gid = 1001: /usr/sbin/env
user = 1001, gid = 1001: /usr/bin/env
user = 1001, gid = 1001: /etc/rc.d/sendmail
user = 1001, gid = 1001: /bin/kenv
user = 1001, gid = 1001: /sbin/sysctl
user = 1001, gid = 1001: /bin/ls
```

As we can see, D is very similar to C in its syntax aside from a couple of special forms of syntax specific to it. Unlike C, it does not support loops so any form of looping must be done by manually unwinding the loop. In the above example we can access the user and group id through `uid` and `gid` built-in variables.

DTrace also supports aggregating the trace results together in various ways. For example, we can count up the system calls each program is doing:

```
# dtrace -n 'syscall:::entry { @syscall_agg[execname, pid] = count(); }'
dtrace: description 'syscall:::entry ' matched 1148 probes
 sh                                                         46569                 7
 sh                                                         46570                 7
 syslogd                                                      703                16
 sshd                                                         848                17
 devd                                                         501                20
 ntpd                                                         771                24
 sh                                                         46565                93
 dtrace                                                     46568               138
 ps                                                         46570               254
 sshd                                                       46564             27517
 ls                                                         46569             35755
```

Using `@` as a prefix to a variable makes it an aggregate variable. `@syscall_agg` is indexed by two keys, however one can keep adding keys. The aggregation output for `@syscall_agg` should be read as:

```
execname                                                     pid               count
```

Our final example will be one with stack traces. DTrace allows the user to gather stack traces both in the kernel and userspace using `stack()` and `ustack()` routines respectively. Furthermore, DTrace can be extended with language-specific stack unwinders. One such example is the `jstack()` action, which provides the user a legible backtrace from a Java program. In our example, we focus on `stack()`:

```
# dtrace -x quiet -n 'io:::start { @[stack()] = count(); }'
              zfs.ko`zio_vdev_io_start+0x2f5
              zfs.ko`zio_nowait+0x15f
              zfs.ko`vdev_mirror_io_start+0xfd
              zfs.ko`zio_vdev_io_start+0x1eb
              zfs.ko`zio_nowait+0x15f
              zfs.ko`arc_read+0x14aa
              zfs.ko`dbuf_read+0xc84
              zfs.ko`dmu_tx_check_ioerr+0x84
              zfs.ko`dmu_tx_count_write+0x191
              zfs.ko`dmu_tx_hold_write_by_dnode+0x64
              zfs.ko`zfs_write+0x500
              zfs.ko`zfs_freebsd_write+0x39
              kernel`VOP_WRITE_APV+0x194
              kernel`vn_write+0x2ce
              kernel`vn_io_fault_doio+0x43
              kernel`vn_io_fault1+0x163
              kernel`vn_io_fault+0x1cc
              kernel`dofilewrite+0x81
              kernel`sys_writev+0x6e
              kernel`amd64_syscall+0x12e
                1

              zfs.ko`zio_vdev_io_start+0x2f5
              zfs.ko`zio_nowait+0x15f
              zfs.ko`zil_lwb_write_done+0x360
              zfs.ko`zio_done+0x10d6
              zfs.ko`zio_execute+0xdf
              kernel`taskqueue_run_locked+0xaa
              kernel`taskqueue_thread_loop+0xc2
              kernel`fork_exit+0x80
              kernel`0xffffffff810a35ae
                1
```

This D script counts up all of the kernel stack traces that lead to I/O on a block device. We omit the aggregation name as we only have one aggregation in this script and our key is `stack()` — a built-in DTrace action returning an array of program counters which are later resolved to symbols when printing results. DTrace can also gather stacks using the `profile` provider in order to gather on-CPU stack traces, making it possible to generate Flame Graphs[10].

New Developments

dwatch

A new tool called `dwatch` was developed by Devin Teske (dteske@freebsd.org) and upstreamed to FreeBSD 11.2. `dwatch` makes DTrace much easier to use for common use-cases than the **dtrace** command line tool. Going back to our toy snooping example, one can simply run:

```
# dwatch execve
```

to get nicely filtered output with more information than our simple snooper shown above.

```
# dwatch execve
INFO Watching 'syscall:freebsd:execve:entry' ...
2022 Nov 24 18:46:53 1001.1001 sh[46565]: sudo ps auxw
2022 Nov 24 18:46:53 0.0 sudo[46920]: ps auxw
2022 Nov 24 18:46:55 1001.1001 sh[46565]: ls
2022 Nov 24 18:47:01 1001.1001 sh[46565]: ls -lapbtr
2022 Nov 24 18:47:09 1001.1001 sh[46924]: kenv -q rc.debug
2022 Nov 24 18:47:09 1001.1001 sh[46924]: /sbin/sysctl -n -q kern.boottrace.enabled
2022 Nov 24 18:47:09 1001.1001 sh[46565]: env -i -L -/daemon HOME=/ PATH=/sbin:/
bin:/usr/sbin:/usr/bin /etc/rc.d/sendmail onestop
2022 Nov 24 18:47:09 1001.1001 env[46565]: /bin/sh /etc/rc.d/sendmail onestop
2022 Nov 24 18:47:09 1001.1001 sh[46924]: kenv -q rc.debug
2022 Nov 24 18:47:09 1001.1001 sh[46924]: /sbin/sysctl -n -q kern.boottrace.enabled
```

Furthermore, **dwatch** supports filtering based on jails, groups, processes and many other features that make it worthwhile to learn for even the most seasoned DTrace users. All along the dwatch tower[11] is an excellent talk that introduces **dwatch** and goes over its features in detail. Similarly, the **dwatch(1)** man page in FreeBSD has a lot of good examples for those interested to try out.

CTFv3

Compact C Type Format (CTF) is a format used to encode C type information in FreeBSD ELF binaries. It allows DTrace to know C type layouts for target binaries (processes, the kernel) so that scripts written by users can refer to those types and explore them. In the past DTrace only supported a total of **2^15** C types in a single binary encoded as CTF due to the way that CTFv2 was implemented. This limitation was a source of many bug reports in FreeBSD relating to DTrace. In March of this year, Mark Johnston (markj@freebsd.org) committed changes which switches DTrace to use CTFv3 instead which raises not only the number of C types that can be manipulated by DTrace, but also various other limits in CTF.

dwatch supports filtering based on jails, groups, processes and many other features that make it worthwhile to learn.

kinst – A New DTrace Provider for Instruction-level Tracing

A 2022 Google Summer of Code project successfully completed by Christos Margiolis (christos@freebsd.org) and mentored by Mark Johnston (markj@freebsd.org) implemented and upstreamed instruction-level tracing to FreeBSD. The provider that implements this functionality is called **kinst**. It reuses parts of the **fbt** mechanism and extends it to instrument arbitrary points of a kernel function, rather than just the entry and exit points.

Kernel developers reading this might already see the potential of **kinst** when it comes to analyzing call stacks from certain branches in a function. As a result finding bugs and performance issues in FreeBSD could be made easier and faster. For a demonstration, we consider scenarios resembling the following C-style pseudo-code:

```
if (__predict_false(rarely_true)) {
     return (slow_operation());
} else {
     return (get_from_cache());
}
```

In this example, we focus on a particular function in the FreeBSD kernel that has behavior similar to this. The simplified and stripped down version of it is:

```
void
_thread_lock(struct thread *td)
{
        ...
        if (__predict_false(LOCKSTAT_PROFILE_ENABLED(spin__acquire)))
                goto slowpath_noirq;
        spinlock_enter();
        ...
        if (__predict_false(m == &blocked_lock))
                goto slowpath_unlocked;
        if (__predict_false(!_mtx_obtain_lock(m, tid)))
                goto slowpath_unlocked;
        ...
        _mtx_release_lock_quick(m);
slowpath_unlocked:
        spinlock_exit();
slowpath_noirq:
        thread_lock_flags_(td, 0, 0, 0);
}
```

It's immediately noticeable that there are two slow paths: **slowpath_unlocked** and **slowpath_noirq**. In the two slow paths, either **spinlock_exit()** or **thread_lock_flags_()** is called, whereas **_mtx_release_lock_quick()** is just an atomic compare-and-swap instruction on amd64. In order to use **kinst** to identify the call stacks which end up in the slow paths, we first need to disassemble the function in some way. One possible way of doing so is using **kgdb** in FreeBSD (**pkg install gdb**):

```
# kgdb
(kgdb) disas /r _thread_lock
Dump of assembler code for function _thread_lock:
```

```
...
0xffffffff80bc7dcc <+124>:   5d       pop    %rbp
  0xffffffff80bc7dcd <+125>:   e9 4e 72 09 00   jmp    0xffffffff80c5f020
<witness_lock>
  0xffffffff80bc7dd2 <+130>:   48 c7 43 18 00 00 00 00 movq   $0x0,0x18(%rbx)
  0xffffffff80bc7dda <+138>:   e8 e1 43 4e 00   call   0xffffffff810ac1c0
<spinlock_exit>
  0xffffffff80bc7ddf <+143>:   8b 75 d4         mov    -0x2c(%rbp),%esi
...
  0xffffffff80bc7df2 <+162>:   41 5d    pop    %r13
  0xffffffff80bc7df4 <+164>:   41 5e    pop    %r14
  0xffffffff80bc7df6 <+166>:   41 5f    pop    %r15
  0xffffffff80bc7df8 <+168>:   5d       pop    %rbp
  0xffffffff80bc7df9 <+169>:   e9 82 00 00 00   jmp    0xffffffff80bc7e80
<thread_lock_flags_>
```

In this case, we can take the instructions at offset +138 and +169, which are the function calls to `spinlock_exit()` and `thread_lock_flags_()`. Using those offsets, we can now write our DTrace script:

```
# dtrace -n 'kinst::_thread_lock:138,kinst::_thread_lock:169 { @[stack(),
probename] = count(); }'
...
              0xcf566bb0
              kernel`ipi_bitmap_handler+0x87
              kernel`0xffffffff810a48b3
              kernel`vm_fault_trap+0x71
              kernel`trap_pfault+0x22d
              kernel`trap+0x48c
              kernel`0xffffffff810a2548
 138                                                                      8
```

Those familiar with DTrace might notice that this could have easily been implemented using speculative tracing instead of needing to use **kinst**. However, one can easily imagine scenarios where the "slow path" or its equivalent is not a simple function call or where the same function call might be present in all of the branches.

kinst also has other implications on the DTrace ecosystem on FreeBSD. Historically, there has been a problem with instrumentation of inlined functions in the kernel using **fbt**. The mechanisms used to implement **kinst** could help extend **fbt** in order to support reliable tracing of inlined functions.

Ongoing work

DTrace and eBPF – a Comparison

Mateusz Piotrowski (0mp@FreeBSD.org) has been working on the performance analysis of DTrace on FreeBSD and how it compares to eBPF on Linux. Some of the results were presented[12] this year at EuroBSDcon 2022. This work could lead to interesting results which could serve as a basis for further optimization of DTrace. This would make enabling instrumentation on performance-critical systems less disruptive.

HyperTrace

HyperTrace is a framework built on top of DTrace which allows the user to apply DTrace-like tracing techniques using the D programming language to tracing virtual machines. It grew out of the CADETS project at the University of Cambridge in the UK. As a simple example, we look at our original snooper script and extend it to use HyperTrace:

```
# dtrace -x quiet -En 'FreeBSD-14*:proc:::exec { printf("%s: user = %u, gid = %u:
%s\n", vmname, uid, gid, stringof(args[0])); }'
scylla1-webserver-0: user = 0, gid = 0: /usr/sbin/dtrace
scylla1-webserver-0: user = 0, gid = 0: /sbin/ls
scylla1-webserver-0: user = 0, gid = 0: /bin/ls
scylla1-client-0: user = 0, gid = 0: /usr/sbin/sshd
scylla1-client-0: user = 0, gid = 0: /bin/csh
scylla1-client-0: user = 0, gid = 0: /usr/bin/resizewin
scylla1-client-0: user = 0, gid = 0: /usr/sbin/iperf
scylla1-client-0: user = 0, gid = 0: /usr/bin/iperf
scylla1-client-0: user = 0, gid = 0: /usr/local/bin/iperf
host: user = 0, gid = 0: /bin/sh
host: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-0: user = 0, gid = 0: /bin/sh
scylla1-client-0: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-1: user = 0, gid = 0: /bin/sh
scylla1-client-1: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-2: user = 0, gid = 0: /bin/sh
scylla1-client-2: user = 0, gid = 0: /usr/libexec/atrun
scylla1-client-3: user = 0, gid = 0: /bin/sh
scylla1-client-3: user = 0, gid = 0: /usr/libexec/atrun
scylla1-webserver-0: user = 0, gid = 0: /bin/sh
scylla1-webserver-0: user = 0, gid = 0: /usr/libexec/atrun
```

We modified the script in order two new things: the prefix of where each of the processes was executed using the built-in variable **`vmname`** and a 5th tuple entry in the probe specification: **`FreeBSD-14*`**. This allows the user to specify which target machines (VMs) to instrument and can be controlled through command-line flags to support name resolution via things like the OS version or the machine's hostname.

Similar changes can be made to our block I/O example:

```
# dtrace -x quiet -En 'scylla1-*:io:::start { @[vmname, immstack()] = count(); }'
...
 scylla1-webserver-0
             devstat_start_transacti+0x90
             g_disk_start+0x316
             g_io_request+0x2d7
             g_part_start+0x289
```

```
              g_io_request+0x2d7
              g_io_request+0x2d7
              ufs_strategy+0x83
              VOP_STRATEGY_APV+0xd2
              bufstrategy+0x3e
              bufwriteL+0x80
              vfs_bio_awrite+0x24f
              flushbufqueues+0x52a
              buf_daemon+0x1f1
              fork_exit+0x80
              fork_trampoline+0xe
              aio_process_rw+0x10c
              aio_daemon+0x285
              fork_exit+0x80
              fork_trampoline+0xe
              fork_trampoline+0xe
            10598
scylla1-client-0
              g_disk_start+0x316
              g_io_request+0x2d7
              g_part_start+0x289
              g_io_request+0x2d7
              g_io_request+0x2d7
              ufs_strategy+0x83
              VOP_STRATEGY_APV+0x9e
              bufstrategy+0x3e
              bufwriteL+0x3e
              vfs_bio_awrite+0x24f
              flushbufqueues+0x52a
              buf_daemon+0x1f1
              fork_exit+0x80
              fork_trampoline+0xe
              fork_trampoline+0xe
              aio_process_rw+0x10c
              aio_daemon+0x285
              fork_exit+0x80
              fork_trampoline+0xe
              fork_trampoline+0xe
            10605
```

Here a new DTrace action **immstack()** is used which works similar to **stack()** but symbol resolution happens in the kernel rather than during time of printing output.

HyperTrace works by aiming to execute the entire D script on the host kernel rather than running DTrace inside the guest, while each of the guests is responsible for instrumenting itself and issuing a synchronous hypercall (akin to a system call in an OS) to the host when the probe is executed on the guest. This kind of design enables keeping global state across all of the guests and host in one place — increasing the overall expressiveness of D when it comes to tracing VMs. The work is still in progress and can be viewed on GitHub[13].

Further Reading

1. https://illumos.org/books/dtrace/preface.html#preface
2. https://github.com/opendtrace
3. https://docs.freebsd.org/en/books/handbook/dtrace/
4. https://www.cs.princeton.edu/courses/archive/fall05/cos518/papers/dtrace.pdf
5. https://www.brendangregg.com/dtracebook/
6. https://wiki.freebsd.org/DTrace/One-Liners
7. https://freebsdfoundation.org/wp-content/uploads/2014/05/DTrace.pdf
8. https://issue.freebsdfoundation.org/publication/?m=29305&i=417423&p=14&ver=html5
9. http://www.onlinedigeditions.com/publication/?m=29305&i=536657&p=4&ver=html5
10. https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
11. https://papers.freebsd.org/2018/bsdcan/teske-all_along_the_dwatch_tower/
12. https://github.com/freebsd/freebsd-papers/pull/112
13. https://github.com/cadets/freebsd

DOMAGOJ STOLFA is a Research Assistant at the University of Cambridge working on dynamic tracing of virtualized systems. He has been working with bhyve and DTrace on FreeBSD and contributing patches since 2016. Domagoj is also a teaching assistant on the Advanced Operating Systems courses at the University of Cambridge, teaching operating systems concepts with FreeBSD using DTrace and PMCs.