DTrace for Linux

Tomas Jedlicka <tomas.jedlicka@oracle.com>

2018-02-03 Sat

Introduction

Overview

DTrace has been released in 2005 for Sun's *Solaris* operating system.

Today it has become adopted by other operating systems as well

- Apple's *Mac Os X*
- Joyent's *Smart OS*
- *FreeBSD*
- Oracle's *Linux* . . . WIP

Why another Linux tracer?

- It is not a Linux only tracer
- Can trace kernel and user space
- Tracing happens directly in kernel
- Very simple and powerful syntax
- Used for over a decade on production machines (Solaris)
- Lot of books/examples about the tool
- No need for compilation/additonal kernel module loading

Architecture

High-level overview

Workflow:

1. User writes a tracing script in D language
2. The script is compiled into intermediate bytecode (DIF)
3. The bytecode is sent to the kernel and verified for safety
4. Kernel enables probes and starts action processing and recording to in-kernel buffers.
5. Userspace periodically extracts the buffers and processes records

Components

DTrace is built around following components:

- probe - event of interest. Identified by unique tuple of (provider, module, function, name)
- provider - responsible for creation/firing of probes.
- consumer - enables probe of interest and collects recorded data
- framework - core component that manages probes, providers, consumers

DTrace providers

provider	description
dtrace	Default framework probes
profile	Time based probing
sdt	Statically defined tracing
fbt	Function boundary tracing
fasttrap	Userspace tracing (USDT at this moment)
syscall	Syscall tracing

Subsystems

- kernel and modules framework/providers
- libdtrace core library
- libproc wrapper around procfs
- libdtrace-ctf CTF helper library
- dtrace enduser consumer application

Source code

For more details check the opensource project page:

`https://oss.oracle.com/projects/DTrace/`

How to use DTrace

D language syntax

- *awk* style syntax
- executed from top to bottom

```
provider:module:function:probe,
...
/ predicate /
{
    var = 5;
    ... actions ...
}
```

Actions and sub-routines

- A subroutine is a function returning value that can be called from predicate.
- An action is a statment than can be performed from probe action block.
- Action can execute in kernel or in userspace
- Action may be destructive.
    - requires explicit -w argument
    - may have destructive sideffect on traced system

DTrace provider

There are three probes always available in the DTrace:

- dtrace:::BEGIN - fires as the first event
- dtrace:::END - fires as the last event
- dtrace:::ERROR - fires when probe execution causes error

Example (Hello World)

```
BEGIN
{
    printf("Hello World!\n");
    exit(0);
}
```

List of available probes

Part of the debugging is to know what is available

Output

```
dtrace -l
   ID   PROVIDER            MODULE                          FUNCTION NAME
    1     dtrace                                                     BEGIN
    2     dtrace                                                     END
    3     dtrace                                                     ERROR
    4        fbt           vmlinux                  run_init_process entry
    5        fbt           vmlinux                  run_init_process return
    6        fbt           vmlinux           try_to_run_init_process entry
    7        fbt           vmlinux           try_to_run_init_process return
    8        fbt           vmlinux                     name_to_dev_t entry
    9        fbt           vmlinux                     name_to_dev_t return
   10        fbt           vmlinux                   calibrate_delay entry
   11        fbt           vmlinux                   calibrate_delay return
   12        fbt           vmlinux                  native_read_tscp entry
   13        fbt           vmlinux                  native_read_tscp return
   14        fbt           vmlinux   10                  native_read_cr4 entry
```

Listing probe details

Get more details about the probe itself (option -v)

Output

```
   ID   PROVIDER            MODULE                          FUNCTION NAME
72709   lockstat           vmlinux          mutex_lock adaptive-acquire

        Probe Description Attributes
                Identifier Names: Private
                Data Semantics:   Private
                Dependency Class: Unknown

        Argument Attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
                Dependency Class: ISA

        Argument Types
                args[0]: struct mutex *
```

Aggregations

Aggregation variable

- key is a tuple of values
- value is result of aggregating functions

DTrace aggregates directly in-kernel.

Example (Aggregations)

```
@variable[key1, key2, key3] = aggfunct(...args...)
```

Built-in variables

There are also built-in variables available for you.

Check the doc for more details

variable	purpose
curthread	Pointer to current thread/task
curpsinfo	Pointer to psinfo_t for current process
timestamp	High precision counter
walltimestamp	Wall time clock
arg0 .. arg9	Probe arguments

CPU profiling

Show processes that consume most CPU cycles

```
profile-997hz /arg1/ { @[pid, execname] = count(); }
```

Result

```
dtrace: description 'profile-997hz ' matched 1 probe
^C

     3358  find                                                        1
     3073  bash                                                        2
     3357  ls                                                          2
     3072  sshd                                                        3
     3359  top                                                        51
```

CPU profiling

Show kernel top functions consuming CPU

```
profile-997hz /arg0/ { @[func(arg0)] = count(); }
```

Result

```
dtrace: description 'profile-997hz ' matched 1 probe
^C

  vmlinux`xen_hypercall_event_channel_op                              1
...
  vmlinux`__call_rcu                                                  3
  vmlinux`smp_call_function_single                                    6
  vmlinux`_raw_spin_unlock_irqrestore                                 9
  vmlinux`native_safe_halt                                       111648
```

Variables

Scope of variables

- this per probe
- self per kernel thread
- global scope

Variable can be dynamically allocated:

1. allocated at first store
2. freed when 0 is stored in it

Arrays are always associative

Compact Type Format

D is following C type notation.

CTF stores information about C object types.

Why?

- DTrace depends on type info to provide most of its features
- CTFs are smaller than DWARF (kernel build time deduplication)
- Can be stored directly in a kernel module or in an external archive
- Production environment may not have debuginfo with DWARF available

CPU profiling

Threads on a CPU based on their name

```
profile-997hz { @[stringof(curthread->comm)] = count()
```

Result

```
dtrace: description 'profile-997 ' matched 1 probe
^C

  NetworkManager                                                  1
  sshd                                                            1
  irqbalance                                                      2
  rcuos/0                                                         2
  dtrace                                                          4
  kworker/0:0                                                     4
  top                                                            18
  swapper/10                                                   6962
  swapper/0                                                    6977
```

Options

Many options directly from command line. Or -x for additional options.

Buffering policy

- switch policy
- ring policy
- fill policy

Lazy attach -Z

Allow destructive actions -w

Speculative tracing

Allows tracing of data but decide later to keep it or discard it.

1. Allocate alternate buffers with speculation subroutine
2. Switch output to alternate set of buffers by speculate
3. Later decide to keep data or not (commit or discard)

Locks observations (SDT)

```
lockstat:vmlinux::adaptive-acquire
{ @[stack()] = count(); }
```

Result

```
vmlinux‘do_error_trap+0x67
vmlinux‘do_invalid_op+0x20
vmlinux‘invalid_op+0x1e
vmlinux‘kernfs_iop_permission+0x36 <== kernfs_mutex
vmlinux‘__inode_permission+0x72
vmlinux‘inode_permission+0x18
vmlinux‘link_path_walk+0x22e
vmlinux‘path_init+0xb9
vmlinux‘path_openat+0x72
vmlinux‘do_filp_open+0x49
vmlinux‘do_sys_open+0x137
vmlinux‘SyS_open+0x1e
vmlinux‘system_call_fastpath+0x12
528
```

Network (SDT)

```
tcp:::receive { @[args[2]->ip_saddr] =
quantize(args[2]->ip_plength); }
```

Result

```
dtrace: description 'tcp:::receive ' matched 2 probes
^C


  10.163.45.28
   value  ------------- Distribution ------------- count
      16 |                                         0
      32 |@@@@@@@@@@@@@@@@@@@@@                    15
      64 |@@@@@@@@@@@@@                            9
     128 |@                                        1
     256 |@@@                                      2
     512 |                                         0
    1024 |@                                        1
    2048 |                                         0
```

Drilling down into a system

Which module is most active in the system?

```
fbt:::entry { @[probemod] = count(); }
```

Result

```
dtrace: description 'fbt:::entry ' matched 35452 probes
^C

  sunrpc                                                         1
  iptable_filter                                                 3
  iptable_security                                               3
...
  nf_conntrack                                                2244
  vmlinux                                                   700408
```

Drilling down into a system

OK it is *vmlinux*. What is it doing?

```
fbt:vmlinux::entry { @k[probefunc] = count(); }
```

Result

```
dtrace: description 'fbt:vmlinux::entry ' matched 28438 probes
^C

  SyS_nanosleep                                                     1
  __alloc_skb                                                       1
  __bitmap_and                                                      1
...
  strcmp                                                        12826
  _find_next_bit.part.0                                         31833
```

Drilling down into a system

So most of the time a *vmlinux* calls *strcmp* function. Why?

```
fbt:vmlinux:strcmp:entry { @[stack()] = count(); }
```

Result

```

  vmlinux‘security_context_to_sid+0x16
  vmlinux‘selinux_inode_setsecurity+0x72
  vmlinux‘selinux_inode_notifysecctx+0x1d
  vmlinux‘security_inode_notifysecctx+0x16
  vmlinux‘kernfs_refresh_inode+0x70
  vmlinux‘kernfs_iop_permission+0x41
  vmlinux‘__inode_permission+0x72
  vmlinux‘inode_permission+0x18
  vmlinux‘link_path_walk+0x22e
  vmlinux‘path_init+0xb9
  vmlinux‘path_openat+0x72
  vmlinux‘do_filp_open+0x49
  vmlinux‘do_sys_open+0x137
  vmlinux‘SyS_open+0x1e
  vmlinux‘system_call_fastpath+0x12
10368
```

Drilling down into a system

It is called from syscall open. Who is doing that?

```
syscall::open:entry { @[pid,execname] = count(); }
```

Result

```
dtrace: description 'syscall::open:entry ' matched 1 probe
^C

      850  irqbalance                                                   10
     1702  dtrace                                                      128
```

Drilling down into a system

So it is *dtrace*. Why is dtrace issuing open syscalls?

```
syscall::open:entry /execname == "dtrace"/
{ @[copyinstr(arg0)] = count(); }
```

Or strace

```
strace -b open dtrace -n 'syscall::open:entry { @[pid,execname] = count(); }'
```

In the end DTrace is trying to open various:
/sys/devices/system/cpu/cpu78/online

Flow tracing (script)

What happens during syscall open in a kernel? Let's use -F option to coalesce functions during tracing.

Example

```
syscall::open:entry
{
    self->t = 1;
}

syscall::open:return
/ self->t /
{
    self->t = 0;
}

fbt::: / self->t / {}
```

Flow tracing (result)

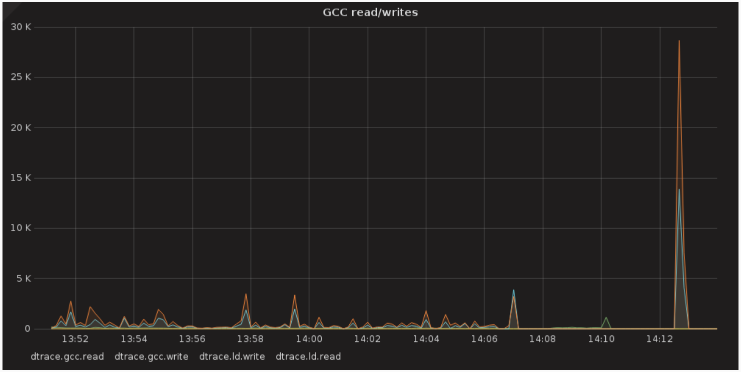
```
CPU FUNCTION
  3  -> SyS_open
  3    -> getname
  3      -> getname_flags
  3        -> kmem_cache_alloc
  3        <- kmem_cache_alloc
  3      <- getname_flags
  3    <- getname
...
  3        <- _raw_spin_lock
  3        -> _raw_spin_unlock
  3        <- _raw_spin_unlock
  3      <- __fd_install
  3    <- fd_install
  3    -> putname
  3      -> kmem_cache_free
  3      <- kmem_cache_free
  3    <- putname
  3  <- SyS_open
```

Telemetrics (script)

```
syscall::write:entry
/ execname == "gcc" /
{
    @writes[execname] = count();
}
...
tick-1s
{
    printa("dtrace.%s.write %@u", @writes);
    printf(" %lli\n", walltimestamp / 1000000000);
    clear(@writes);
}
```

Telemetrics (result)

Use: dtrace -s metrics.d | nc carbonserver <port>



Thank you!