# DTrace for Developers

George V. Neville-Neil

August 9, 2018

# A Look Inside FreeBSD with DTrace

## What is DTrace?

George V. Neville-Neil  Robert N. M. Watson

August 9, 2018

# What is DTrace?

- A dynamic tracing framework for software
- Low impact on overall system performance
- Does not incur costs when not in use

# What can DTrace show me?

- When a function is being called
- A function's arguments
- The frequency of function calls
- A whole lot more...

# A Simple Example

```
1   dtrace -n syscall:::
2   dtrace: description 'syscall:::' matched 2148 probes
3   CPU     ID                    FUNCTION:NAME
4     1  51079                     ioctl:return
5     1  51078                      ioctl:entry
6     1  51079                     ioctl:return
7     1  51078                      ioctl:entry
8     1  51079                     ioctl:return
9     1  51632                sigprocmask:entry
10    1  51633               sigprocmask:return
11    1  51784                  sigaction:entry
```

- Look at all system calls

# How does DTrace Work?

- Various probes are added to the system
- The probes are activated using the dtrace program
- A small number of assembly instructions are modified at run-time to get the system to run in the probe

# A more complex example

```
1   dtrace -n 'syscall::write:entry /arg2 != 0/ { printf("write size %d\n", arg2); }'
2   dtrace: description 'syscall::write:entry ' matched 2 probes
3   CPU     ID                    FUNCTION:NAME
4   0   50978                     write:entry write size 1
5   0   50978                     write:entry write size 55
6   0   50978                     write:entry write size 2
```

# DTrace Glossary

| | |
|---|---|
| **Probe** | A way of specifying what to trace |
| **Provider** | A DTrace defined module that provides information about something in the system |
| **Module** | A software module, such as `kernel` |
| **Function** | A function in a module, such as `ether_input` |
| **Predicate** | A way of filtering DTrace probes |
| **Action** | A set of D language statements carried out when a probe is matched |

# Providers

| | |
|---|---|
| **fbt** | Function Boundary Tracing (50413) |
| **syscall** | System Calls (2148) |
| **profile** | Timing source |
| **proc** | Process Operations |
| **sched** | Scheduler |
| **io** | I/O calls |
| **ip** | Internet Protocol |
| **udp** | UDP |
| **tcp** | TCP |
| **vfs** | Filesystem Routines |

# Dissecting a Probe

- `syscall::write:entry`

  **Provider** syscall
  **Module** None
  **Function** write
  **Name** entry

- `fbt:kernel:ether_input:entry`

  **Provider** fbt
  **Module** kernel
  **Function** `ether_input`
  **Name** entry

# DTrace Requirements

- A kernel with DTrace support built in
  - Default on FreeBSD 10 or later
- The ability to sudo or be root
- The complete command syntax is covered in the dtrace manual page

# Finding Probes

- Listing all the probes gets you 50000 to choose from
- Judicious use of providers, modules and grep
- e.g. `dtrace -l -P syscall`

# Probe Arguments

- Use verbose (-v) mode to find probe arguments
- `sudo dtrace -lv -f syscall:freebsd:read`

```
 ID   PROVIDER            MODULE
57177    syscall             freebsd

Argument Types
args[0]: int
args[1]: void *
args[2]: size_t
```

# The D Language

- A powerful subset of C
- Includes features specific to DTrace, such as aggregations
- Anything beyond some simple debugging usually required a D script

# DTrace One-Liners

- A set of useful single line scripts

```
1  # Trace file opens with process and filename:
2  dtrace -n 'syscall::open*:entry { printf("%s %s", execname, copyinstr(arg0)); }'
3
4  # Count system calls by program name:
5  dtrace -n 'syscall:::entry { @[execname] = count(); }'
6
7  # Count system calls by syscall:
8  dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
```

# Count System Calls

```
1   dtrace -n 'syscall:::entry { @[probefunc] = count(); }'
2   dtrace: description 'syscall:::entry ' matched 1072 probes
3   ^C
4     fstat                                                         1
5     setitimer                                                     1
6     getpid                                                        2
7     read                                                          2
8     sigreturn                                                     2
9     write                                                         3
10    getsockopt                                                    4
11    select                                                        6
12    sigaction                                                     6
13    _umtx_op                                                      7
14    __sysctl                                                      8
15    munmap                                                       18
16    mmap                                                         19
17    sigprocmask                                                  23
18    clock_gettime                                                42
19    ioctl                                                        45
```

# Aggregations

- `syscall::entry { @[probefunc] = count(); }`
- The `@[probefunc]` syntax
- Aggregates data during a run for later output
- Extremely powerful feature of D language

# Quantization

```
# Summarize requested write() sizes by program name, as power-of-2 distributions (bytes):
dtrace -n 'syscall::write:entry { @[execname] = quantize(arg2); }'
dtrace: description 'syscall::write:entry ' matched 2 probes
^C
  find
           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |                                         1
               4 |                                         17
               8 |@@                                       841
              16 |@@@@@@@@@@@@@@@@                         6940
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 13666
              64 |                                         59
             128 |                                         0
```

# Probing the stack

- Find out how we got where we are
- The `stack()` routine

# Who called malloc()?

```
1    1   29371                          malloc:entry
2                    kernel`cloneuio+0x2c
3                    kernel`vn_io_fault1+0x3b
4                    kernel`vn_io_fault+0x18b
5                    kernel`dofileread+0x95
6                    kernel`kern_readv+0x68
7                    kernel`sys_read+0x63
8                    kernel`amd64_syscall+0x351
9                    kernel`0xffffffff80d0aa6b
```

- Read upwards from the bottom

# DTrace Toolkit

- An open source set of tools written to use D scripts
- Originally specific to Solaris
- Exists as a FreeBSD port and package
- Currently being updated with new scripts

# An example script: hotkernel

```
1   ./hotkernel
2   Sampling... Hit Ctrl-C to end.
3   ^C
4   FUNCTION                                  COUNT   PCNT
5   kernel`lookup                                 1   0.1%
6   kernel`unlock_mtx                             1   0.1%
7   kernel`_vm_page_deactivate                    1   0.1%
8   ...
9   kernel`amd64_syscall                          9   0.5%
10  kernel`pmap_remove_pages                      9   0.5%
11  kernel`hpet_get_timecount                    13   0.7%
12  kernel`pagezero                              15   0.8%
13  kernel`0xffffffff80                          34   1.9%
14  kernel`spinlock_exit                        486  27.0%
15  kernel`acpi_cpu_c1                          965  53.6%
```

# Predicates

- Filtering probes based on relevant data
- Useful for excluding common conditions
- `/arg0 != 0/` Ignore a normal return value

# Tracking a Specific Process

- `pid` is used to track a Process ID
- Used in predicates
- `/pid == 1234/`

# Running a Program Under DTrace

- DTrace is most often used on running systems
- DTrace can be attached at runtime to a program
  - `dtrace -p pid ...`
- Run a program completely under the control of DTrace
  - `dtrace -c cmd ...`

# Going too far

- Overly broad probes slow down the system
  - Watching everything in the kernel
  - Registering a probe on a module

# The Probe Effect

- Each probe point has a cost
- Every action has a reaction
- Any action code requires time to run
- Impacts system performance

# DTrace Lab Exercises

- Bring up OSCourse Virtual Machine
- Find the current list of providers
- Count the probes available
- Trace all the system calls used by sshd
- Summarize requested write() sizes by program name
- Summarize return values from write() by program name
- Find and modify three (3) of the DTrace one-liners

# DTrace for Developers

## Tracepoints Outside the Kernel

George V. Neville-Neil

August 9, 2018

# Tracing Programs

- Much of DTrace has been about the kernel
- There is a lot more code in user space
- We can trace that code too

# The pid provider

- Let's you look into a process
- The `pid` and `target` variables
- Functions work like `fbt` the kernel

# Finding probes in user space

- Run the program with -c
- Attach to a running daemon with -p

# What probes exist in ls?

# Whoops, what happened there?

- DTrace continues to play it safe
- Protects the system against memory exhaustion
- Cut the probes in half by using `:entry`

# Dissecting a PID probe

**Provider** The PID

**Module** Program or Library

**Function** Function

**Name** entry, return or a hex offset

```
pid3722:ls:usage:entry
pid3722:libc.so.7:__malloc:entry
```

# Tracing an Instruction

- We can create a probe point on an instruction
- Listing a function without a `entry` or `return`
- Instructions are hexadecimal offsets

# The instructions in malloc

```
1   dtrace -ln 'pid$target::__malloc:' -c ls
```

# Getting the return value

- User space follows the same convention as kernel `fbt`s
  - Because they all follow the same ABI
- `arg0` Return address
- `arg1` Return value

# Tracing malloc()

```
1   dtrace -qn 'pid$target::malloc:return \
2       { printf ("allocated 0x%x returned from offset 0x%x", arg1, arg0); }' -p 600
3   allocated 0x8040431c0 returned from offset 0x8e
```

# Does malloc ever complete?

# Predicates and PIDs

```
1   dtrace -qn 'pid$target::malloc:return /arg0 != 0x8e/
2       { printf ("allocated 0x%x returned from offset 0x%x\n", arg1, arg0); }' -c ls
```

# PID Provider Lab

- How many probe points exist in `ping` ?
- What libraries are used by `vi`?
- What does the `usage()` function return?
- Which function in `ls` calls `malloc` most frequently?
- Write a D script to track `malloc()` pointers.

# The USDT Provider

- User Space Dynamic Tracing
- Different from the `pid` provider
- Add probes to your own programs
- Dynamic logging!

# Adding Tracepoints to User Programs

- Like `SDT` but different.
- More powerful in some ways
- Less powerful in others

# Hello World, again

```
 1  #include <stdio.h>
 2  #include <unistd.h>
 3
 4  int main (int argc, char **argv) {
 5          int i;
 6          for (i = 0; i < 5; i++) {
 7                  printf("Hello world\n");
 8                  sleep(1);
 9          }
10  }
```

- Thanks to Alan Hargreaves who wrote this for the DTrace book

# Adding a single probe

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <sys/sdt.h> /* <- new header file */
4
5   int main (int argc, char **argv) {
6           int i;
7           for (i = 0; i < 5; i++) {
8                   DTRACE_PROBE1(world, loop, i); /* <- probe point */
9                   printf("Hello world\n");
10                  sleep(1);
11          }
12  }
```

# Probe Description File

```
1  provider world {
2          probe loop(int);
3  };
4
5  #pragma D attributes Evolving/Evolving/Common provider world provider
6  #pragma D attributes Private/Private/Common provider world module
7  #pragma D attributes Private/Private/Common provider world function
8  #pragma D attributes Evolving/Evolving/Common provider world name
9  #pragma D attributes Evolving/Evolving/Common provider world args
```

# Updated Build Process

```
1  cc -c hello.c
2  dtrace -G -s probes.d hello.o
3  cc -o hello -ldtrace probes.o hello.o
```

# Dissecting a USDT Probe

# Stability

**Name** Do we think the name will change?

**Data** Are we committed to the data format?

**Dependency** Is this probe OS or hardware dependent?

# Stability

| | |
|---|---|
| **Internal** | Part of DTrace |
| **Private** | Vestige of Sun Microsystems |
| **Obsolete** | Will be removed in upcoming release, do not use. |
| **External** | Vestige of Sun Microsystems |
| **Unstable** | Can change at any time. |
| **Evolving** | Could change but becoming more stable. |
| **Stable** | Will not change within a major revision. |
| **Standard** | Defined by a standards body (POSIX, IETF etc.) |

# Dependency Classes

**Unknown**

| | |
|---|---|
| **CPU** | SPARC, Intel |
| **Platform** | FreeBSD, Illumos, MacOS |
| **Group** | Similar to ISA on FreeBSD |
| **ISA** | Instruction set architecture (amd64, arm32) |
| **Common** | Nearly all user space probes should use this. |

# Reviewing our probe file

```
1   provider world {
2           probe loop(int);
3   };
4
5   #pragma D attributes Evolving/Evolving/Common provider world provider
6   #pragma D attributes Private/Private/Common provider world module
7   #pragma D attributes Private/Private/Common provider world function
8   #pragma D attributes Evolving/Evolving/Common provider world name
9   #pragma D attributes Evolving/Evolving/Common provider world args
```

# USDT Translator Lab

- Install /usr/src onto your VM
- Add the following probes to `/usr/src/bin/echo/echo.c`
  - Probe the value of `nflag`.
  - Probe the value of `len` in the argument loop.
  - Probe the value of `veclen` in the final loop.

# A Look Inside FreeBSD with DTrace

## Kernel SDTs

George V. Neville-Neil  Robert N. M. Watson

August 9, 2018

# Converting Logging Code

- Most code littered with `printf`
- Many different `DEBUG` options
- Most can be converted

# TCPDEBUG Case Study

- TCBDEBUG added in the original BSD releases
- Rarely enabled kernel option that shows:
  - direction
  - state
  - sequence space
  - `rcv_nxt`, `rcv_wnd`, `rcv_up`
  - `snd_una`, `snd_nxt`, `snx_max`
  - `snd_wl1`, `snd_wl2`, `snd_wnd`

# TCPDEBUG Before

- 127 lines of code
- 14 calls to printf
- Statically defined ring buffer of 100 entries
- Static log format

# TCPDEBUG After

- Four (4) new tracepoints
  - debug-input
  - debug-output
  - debug-user
  - debug-drop
- Access to TCP and socket structures
- Flexible log format

# Convenient Macros

- `SDT_PROVIDER_DECLARE` Declare a provider in an include file
- `SDT_PROVIDER_DEFINE` Instantiate a provider in C code
- `SDT_PROBE_DECLARE` Declare a probe in a n include file
- `SDT_PROBE_DEFINE` Define a probe of X arguments (0-6)
- `SDT_PROBE_DEFINE_XLATE` Define a probe of N arguments with translation
- Only available for kernel code

# TCP Debug Desclarations

```
1   SDT_PROBE_DECLARE(tcp, , , debug__input);
2   SDT_PROBE_DECLARE(tcp, , , debug__output);
3   SDT_PROBE_DECLARE(tcp, , , debug__user);
4   SDT_PROBE_DECLARE(tcp, , , debug__drop);
```

# TCP Debug Call Sites

```
1   #ifdef TCPDEBUG
2           if (tp == NULL || (tp->t_inpcb->inp_socket->so_options & SO_DEBUG))
3                   tcp_trace(TA_DROP, ostate, tp, (void *)tcp_saveipgen,
4                           &tcp_savetcp, 0);
5   #endif
6           TCP_PROBE3(debug__input, tp, th, mtod(m, const char *));
```

# TCP Debug Translators

```
1  SDT_PROBE_DEFINE3_XLATE(tcp, , , debug__input,
2      "struct tcpcb *", "tcpsinfo_t *" ,
3      "struct tcphdr *", "tcpinfo_t *",
4      "uint8_t *", "ipinfo_t *");
5
6  SDT_PROBE_DEFINE3_XLATE(tcp, , , debug__output,
7      "struct tcpcb *", "tcpsinfo_t *" ,
8      "struct tcphdr *", "tcpinfo_t *",
9      "uint8_t *", "ipinfo_t *");
10
11 SDT_PROBE_DEFINE2_XLATE(tcp, , , debug__user,
12     "struct tcpcb *", "tcpsinfo_t *" ,
13     "int", "int");
14
15 SDT_PROBE_DEFINE3_XLATE(tcp, , , debug__drop,
16     "struct tcpcb *", "tcpsinfo_t *" ,
17     "struct tcphdr *", "tcpinfo_t *",
18     "uint8_t *", "ipinfo_t *");
```

# TCP Debug Example Script

```
1   tcp:kernel::debug-input
2   /args[0]->tcps_debug/
3   {
4           seq = args[1]->tcp_seq;
5           ack = args[1]->tcp_ack;
6           len = args[2]->ip_plength - sizeof(struct tcphdr);
7           flags = args[1]->tcp_flags;
8
9           printf("%p %s: input [%xu..%xu]", arg0,
10                  tcp_state_string[args[0]->tcps_state], seq, seq + len);
11
12          printf("@%x, urp=%x", ack, args[1]->tcp_urgent);
```

# TCP Debug Example Script Part 2

```
 1            printf("%s", flags != 0 ? "<" : "");
 2            printf("%s", flags & TH_SYN ? "SYN," :"");
 3            printf("%s", flags & TH_ACK ? "ACK," :"");
 4            printf("%s", flags & TH_FIN ? "FIN," :"");
 5            printf("%s", flags & TH_RST ? "RST," :"");
 6            printf("%s", flags & TH_PUSH ? "PUSH," :"");
 7            printf("%s", flags & TH_URG ? "URG," :"");
 8            printf("%s", flags & TH_ECE ? "ECE," :"");
 9            printf("%s", flags & TH_CWR ? "CWR" :"");
10            printf("%s", flags != 0 ? ">" : "");
11
12            printf("\n");
13            printf("\trcv_(nxt,wnd,up) (%x,%x,%x) snd_(una,nxt,max) (%x,%x,%x)\n",
14                   args[0]->tcps_rnxt, args[0]->tcps_rwnd, args[0]->tcps_rup,
15                   args[0]->tcps_suna, args[0]->tcps_snxt, args[0]->tcps_smax);
16            printf("\tsnd_(wl1,wl2,wnd) (%x,%x,%x)\n",
17                   args[0]->tcps_swl1, args[0]->tcps_swl2, args[0]->tcps_swnd);
```

# How Much Work is That?

- 200 line code change
- 167 lines of example code
- A few hours to code
- A day or two to test
- Now we have always on TCP debugging

# Lab Exercise: Adding Kernel Tracepoints