Linux bcc/BPF Node.js USDT Tracing

12 Oct 2016

You may know that Node.js has built-in USDT (user statically-defined tracing) probes for performance analysis and debugging, but did you know that Linux now supports using them? And now that V8 has tracing, is this too late to matter? In this post I'll explain things a little with a basic USDT example.

The Linux 4.x series has been adding enhancements to BPF (Berkeley Packet Filter) originally for software defined networks, but now can be used for programmatic tracing. Aka BPF superpowers. These are built into Linux, so sooner or later, this is coming to everyone who runs Linux.

I wrote an example of instrumenting the node http-server-request USDT probe with BPF:

```
# ./nodejs_http_server.py 24728
TIME(s)            COMM             PID    ARGS
24653324.561322998 node             24728  path:/index.html
24653335.343401998 node             24728  path:/images/welcome.png
24653340.510164998 node             24728  path:/images/favicon.png
```

The source is in bcc under examples/tracing/nodejs_http_server.py:

```
#!/usr/bin/python
#
# nodejs_http_server    Basic example of node.js USDT tracing.
#                       For Linux, uses BCC, BPF. Embedded C.
#
# USAGE: nodejs_http_server PID
#
# Copyright 2016 Netflix, Inc.
# Licensed under the Apache License, Version 2.0 (the "License")

from __future__ import print_function
from bcc import BPF, USDT
import sys

if len(sys.argv) < 2:
    print("USAGE: nodejs_http_server PID")
    exit()
pid = sys.argv[1]
debug = 0

# load BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
int do_trace(struct pt_regs *ctx) {
    uint64_t addr;
    char path[128];
```

```
    bpf_usdt_readarg(6, ctx, &addr);
    bpf_probe_read(&path, sizeof(path), (void *)addr);
    bpf_trace_printk("path:%s\\n", path);
    return 0;
};
"""

# enable USDT probe from given PID
u = USDT(pid=int(pid))
u.enable_probe(probe="http__server__request", fn_name="do_trace")
if debug:
    print(u.get_text())
    print(bpf_text)

# initialize BPF
b = BPF(text=bpf_text, usdt=u)

# header
print("%-18s %-16s %-6s %s" % ("TIME(s)", "COMM", "PID", "ARGS"))

# format output
while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
        print("value error")
        continue
    print("%-18.9f %-16s %-6d %s" % (ts, task, pid, msg))
```


bcc uses C for the kernel instrumentation (which it compiles into BPF bytecode) and Python (or lua) for user-level reporting. It gets the job done, but is verbose. This example should be even more verbose: I used a debug shortcut, bpf_trace_printk(), but if this were a tool intended for concurrent use it needs to use BPF_PERF_OUTPUT() instead (I explained how in the [bcc Python Tutorial]), which will inflate the code further.

This code ultimately runs the do_trace() function when the http__server__request probe is hit, which reads the 6th argument, the URL. You can see some argument definitions in src/node_provider.d, eg:

```
probe http__server__request(node_dtrace_http_server_request_t *h,
    node_dtrace_connection_t *c, const char *a, int p, const char *m,
    const char *u, int fd) : (node_http_request_t *h, node_connection_t *c,
```

Let's check those other strings (char *'s). The bcc trace program can print them out, which allows some powerful ad hoc one-liners to be developed:

```
# trace -p `pgrep -n node` 'u:node:http__server__request "%s %s %s", arg3, arg5, arg6'
TIME     PID    COMM         FUNC             -
21:28:14 827    node         http__server__request 127.0.0.1 GET /
21:28:15 827    node         http__server__request 127.0.0.1 GET /
21:28:16 827    node         http__server__request 127.0.0.1 GET /
^C
```

You can also use bcc's tplist to list probes, eg, on a file:

```
# tplist -l /mnt/src/node-v6.7.0/node
/mnt/src/node-v6.7.0/node node:gc__start
/mnt/src/node-v6.7.0/node node:gc__done
/mnt/src/node-v6.7.0/node node:http__server__response
/mnt/src/node-v6.7.0/node node:net__server__connection
/mnt/src/node-v6.7.0/node node:net__stream__end
/mnt/src/node-v6.7.0/node node:http__client__response
/mnt/src/node-v6.7.0/node node:http__client__request
/mnt/src/node-v6.7.0/node node:http__server__request
```

... or on a running process:

```
# tplist -p `pgrep node`
/mnt/src/node-v6.7.0/out/Release/node node:gc__start
/mnt/src/node-v6.7.0/out/Release/node node:gc__done
/mnt/src/node-v6.7.0/out/Release/node node:http__server__response
/mnt/src/node-v6.7.0/out/Release/node node:net__server__connection
/mnt/src/node-v6.7.0/out/Release/node node:net__stream__end
/mnt/src/node-v6.7.0/out/Release/node node:http__client__response
/mnt/src/node-v6.7.0/out/Release/node node:http__client__request
/mnt/src/node-v6.7.0/out/Release/node node:http__server__request
/lib/x86_64-linux-gnu/libc-2.23.so libc:setjmp
/lib/x86_64-linux-gnu/libc-2.23.so libc:longjmp
/lib/x86_64-linux-gnu/libc-2.23.so libc:longjmp_target
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_heap_new
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_sbrk_less
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_arena_reuse_free_list
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_arena_reuse_wait
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_arena_reuse
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_arena_new
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_arena_retry
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_heap_free
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_heap_less
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_heap_more
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_sbrk_more
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_malloc_retry
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_free_dyn_thresholds
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_realloc_retry
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_memalign_retry
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_calloc_retry
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_mxfast
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_arena_max
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_arena_test
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_mmap_max
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_mmap_threshold
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_top_pad
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_trim_threshold
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_perturb
/lib/x86_64-linux-gnu/libc-2.23.so libc:memory_mallopt_check_action
/lib/x86_64-linux-gnu/libc-2.23.so libc:lll_lock_wait_private
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:pthread_start
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:pthread_create
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:pthread_join
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:pthread_join_ret
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_init
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_destroy
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_acquired
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_entry
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_timedlock_entry
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_timedlock_acquired
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:mutex_release
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:rwlock_destroy
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:rdlock_acquire_read
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:rdlock_entry
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:wrlock_acquire_write
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:wrlock_entry
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:rwlock_unlock
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:cond_init
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:cond_destroy
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:cond_wait
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:cond_timedwait
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:cond_signal
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:cond_broadcast
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:lll_lock_wait_private
/lib/x86_64-linux-gnu/libpthread-2.23.so libpthread:lll_lock_wait
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowatan2_inexact
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowatan2
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowlog_inexact
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowlog
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowatan_inexact
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowatan
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowtan
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowasin
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowacos
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowsin
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowcos
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowexp_p6
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowexp_p32
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowpow_p10
```

```
/lib/x86_64-linux-gnu/libm-2.23.so libm:slowpow_p32
/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21 libstdcxx:catch
/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21 libstdcxx:throw
/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21 libstdcxx:rethrow
/lib/x86_64-linux-gnu/ld-2.23.so rtld:init_start
/lib/x86_64-linux-gnu/ld-2.23.so rtld:init_complete
/lib/x86_64-linux-gnu/ld-2.23.so rtld:map_failed
/lib/x86_64-linux-gnu/ld-2.23.so rtld:map_start
/lib/x86_64-linux-gnu/ld-2.23.so rtld:map_complete
/lib/x86_64-linux-gnu/ld-2.23.so rtld:reloc_start
/lib/x86_64-linux-gnu/ld-2.23.so rtld:reloc_complete
/lib/x86_64-linux-gnu/ld-2.23.so rtld:unmap_start
/lib/x86_64-linux-gnu/ld-2.23.so rtld:unmap_complete
/lib/x86_64-linux-gnu/ld-2.23.so rtld:setjmp
/lib/x86_64-linux-gnu/ld-2.23.so rtld:longjmp
/lib/x86_64-linux-gnu/ld-2.23.so rtld:longjmp_target
```

This has picked up many other USDT probes (wow), from libc, libpthread, libm, libstdcxx, and rtld. Nice!

Node.js and USDT

Last time I built Node.js with these USDT probes I used these steps:

```
$ sudo apt-get install systemtap-sdt-dev   # adds "dtrace", used by node build
$ wget https://nodejs.org/dist/v6.7.0/node-v6.7.0.tar.gz
$ tar xvf node-v6.7.0.tar.gz
$ cd node-v6.7.0
$ ./configure --with-dtrace
$ make -j8
```

If you don't have bcc setup, you can use readelf to check that the USDT probes are built into the binary, which show up as "SystemTap probe descriptors":

```
# readelf -n /mnt/src/node-v6.7.0/node
[...]
Displaying notes found at file offset 0x01814014 with length 0x000003c4:
  Owner                 Data size   Description
  stapsdt              0x0000003c   NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: gc__start
    Location: 0x00000000011552f4, Base: 0x0000000001a444e4, Semaphore: 0x0000000001e13fdc
    Arguments: 4@%esi 4@%edx 8@%rdi
[...]
```

bcc supports both USDT probes and IS-ENABLED USDT probes.

There is another way to create USDT probes: the dtrace-provider library, which allows your Node.js code to dynamically declare new USDT probes. Last I checked, that library did not compile on Linux, however, with the new bcc/BPF support it should be fixable.

In order to use USDT probes with bcc, you'll need a Linux kernel that's new and shiny. By Linux 4.4 (which is used by Ubuntu 16.04 LTS), there's enough BPF to do USDT event tracing, latency measurements, and histograms. Linux 4.6 adds stack trace support.

V8 Tracing & Future Use

V8 has recently added an --enable-tracing option that generates a v8_trace.json file for loading in Google Chrome's trace viewer. Once this is widely available in node, many common tracing needs may be solved.

The long term value of USDT support with bcc may be the instrumentation of other subsystems: node internals, system libraries, and the kernel, and exposing these with Node.js context fetched from the USDT probes. BPF/bcc can instrument kernel functions, kernel tracepoints, and user-level functions as well. These:

```
# objdump -j .text -tT node | head

node:     file format elf64-x86-64

SYMBOL TABLE:
000000000079bd00 l    d  .text  0000000000000000              .text
000000000079e070 l     F .text  0000000000000000              deregister_tm_clones
000000000079e0b0 l     F .text  0000000000000000              register_tm_clones
000000000079e0f0 l     F .text  0000000000000000              __do_global_dtors_aux
000000000079e110 l     F .text  0000000000000000              frame_dummy
000000000079e140 l     F .text  000000000000002b              ssl_callback_ctrl
# objdump -j .text -tT node | wc
  76170  492908 9373788
# wc /proc/kallsyms
108919 339047 4659123 /proc/kallsyms
```

That's over 75 thousand user-level probe points, plus over 108 thousand kernel probe points, plus those extra USDT probes I listed previously.

If you can solve your performance issues using v8/node-specific capabilities like V8 tracing, then great! But for times when you really need to dig into the depths of your application and its OS interactions: bcc/BPF can do it, and you can make custom tools to automate it.

Adding USDT support to bcc is just the beginning, and I've shared some details on how it works in this post. Next is to build useful tools and GUIs that make use of it.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).

About this blog