Linux eBPF/bcc uprobes

08 Feb 2016

User-level dynamic tracing support was just added to bcc[1], a front-end to Linux eBPF[2]. As a spooky example, let's trace interactive commands entered on all running bash shells, system-wide:

```
# ./bashreadline
TIME      PID    COMMAND
05:28:25  21176  ls -l
05:28:28  21176  date
05:28:35  21176  echo hello world
05:28:43  21176  foo this command failed
05:28:45  21176  df -h
05:29:04  3059   echo another shell
05:29:13  21176  echo first shell again
```

This even sees commands that failed. bash doesn't need to be run in any special debug mode for this to work: all running bash shells are instrumented immediately, and new ones. You can walk up to a system that's never run eBPF before, and say: "so what's bash doing?" and then see. It's like a superpower.

Here's the entire bashreadline bcc/eBPF program:

```
#!/usr/bin/python
# [...]
from __future__ import print_function
from bcc import BPF
from time import strftime

# load BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
int printret(struct pt_regs *ctx) {
    if (!ctx->ax)
        return 0;

    char str[80] = {};
    bpf_probe_read(&str, sizeof(str), (void *)ctx->ax);
    bpf_trace_printk("%s\\n", &str);

    return 0;
};
"""
b = BPF(text=bpf_text)
b.attach_uretprobe(name="/bin/bash", sym="readline", fn_name="printr

# header
print("%-9s %-6s %s" % ("TIME", "PID", "COMMAND"))

# format output
```

```
while 1:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    except ValueError:
        continue
    print("%-9s %-6d %s" % (strftime("%H:%M:%S"), pid, msg))
```

bashreadline.py hosted with ❤ by GitHub view raw

This is tracing the return of the readline() function from /bin/bash, using a uretprobe (user-level return probe). The uretprobe runs a custom eBPF program, printret(), which prints the returned string. Because eBPF programs only operate on their own stack memory (improving safety), we need to use bpf_probe_read() to pull in the string for later operations (bpf_trace_printk()). This may go away: there's a lot of work in bcc to automatically do the bpf_probe_read()s for you, so you can write tools more easily.

This currently accesses the return value from the x86_64 `%ax` register[3], however, bcc should really provide an alias for this (eg, "rval"; it's bug #225[4]).

gethostlatency

As another example, gethostlatency traces name lookups (DNS) system-wide:

```
# ./gethostlatency
TIME       PID    COMM           LATms HOST
06:10:24   28011  wget           90.00 www.iovisor.org
06:10:28   28127  wget            0.00 www.iovisor.org
06:10:41   28404  wget            9.00 www.netflix.com
06:10:48   28544  curl           35.00 www.netflix.com.au
06:11:10   29054  curl           31.00 www.plumgrid.com
06:11:16   29195  curl            3.00 www.facebook.com
06:11:25   29404  curl           72.00 foo
06:11:28   29475  curl            1.00 foo
```

This time it's tracing three libc library functions, system wide: getaddrinfo(), gethostbyname(), and gethostbyname2(). The relevant code from the program is:

```
b.attach_uprobe(name="c", sym="getaddrinfo", fn_name="do_entry")
b.attach_uprobe(name="c", sym="gethostbyname", fn_name="do_entry")
b.attach_uprobe(name="c", sym="gethostbyname2", fn_name="do_entry")
b.attach_uretprobe(name="c", sym="getaddrinfo", fn_name="do_return")
b.attach_uretprobe(name="c", sym="gethostbyname", fn_name="do_return")
b.attach_uretprobe(name="c", sym="gethostbyname2", fn_name="do_return")
```

This attaches custom eBPF functions to the library function calls and function returns, via uprobes and uretprobes. The name="c" is referring to libc. Tracing both calls and returns was necessary to save and retrieve timestamps for calculating the latency. Check out the full program: gethostlatency.

Summary

User-level dynamic tracing is a new bcc/eBPF feature. I demonstrated it by tracing user-level functions and system library functions, and with two new tools: bashreadline and gethostlatency.

We have some work to improve uprobe usage, but that you can now use them at all is a big milestone.

References & Links

1. https://github.com/iovisor/bcc
2. http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html
3. https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI
4. https://github.com/iovisor/bcc/issues/225

Thanks Brenden Blanco (PLUMgrid) for uprobe support!

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