Hacking Linux USDT with Ftrace

03 Jul 2015

I previously thought it was impossible to use user-level statically defined tracing (USDT) probes on Linux, without adding SystemTap or LTTng. Linux's built-in tracers, ftrace and perf_events, don't support USDT.

But I've found a way to hack them in. **This is not recommended. Do not try this at home!**

```
# ./usdt -l /opt/node/node
PROBE
node:gc__start
node:gc__done
node:http__client__request
node:http__client__response
[...]
# ./usdt node:gc__start
Tracing uprobe node_gc__start (p:node_gc__start /opt/node/node:0x7f44b4). Ctrl-C to end.
             node-23467 [001] d... 19993502.867548: node_gc__start: (0xbf44b4)
             node-23484 [003] d... 19993502.983719: node_gc__start: (0xbf44b4)
             node-23484 [003] d... 19993503.354810: node_gc__start: (0xbf44b4)
             node-23484 [000] d... 19993503.433517: node_gc__start: (0xbf44b4)
[...]
```

My usdt tool (in this example, running on Node.js) is a proof of concept. What's important is not this tool, but that this is even possible on an unmodified Linux kernel. In this post I'll show how it's done, although I don't encourage anyone to try this until we have a better front-end.

(Update 2017: since this post, we now do have better front-ends: perf now supports USDT in recent Linux versions, and now so does bcc/eBPF USDT.)

User-Level Dynamic Tracing

USDT should not be confused with user-level *dynamic* tracing, where any user-level code can be instrumented. For example, tracing MySQL server queries by instrumenting the server's `dispatch_command()` function:

```
# ./uprobe 'p:cmd /opt/bin/mysqld:_Z16dispatch_command19enum_server_commandP3THDPcj +0(%dx):strin
Tracing uprobe cmd (p:cmd /opt/bin/mysqld:0x2dbd40 +0(%dx):string). Ctrl-C to end.
  mysqld-2855  [001] d... 19957757.590926: cmd: (0x6dbd40) arg1="show tables"
  mysqld-2855  [001] d... 19957759.703497: cmd: (0x6dbd40) arg1="SELECT * FROM numbers"
[...]
```

I'm using my uprobe tool, which in turn uses built-in Linux capabilities: ftrace (tracer) and uprobes (user-level dynamic tracing, which you'll want a recent Linux for, eg, 4.0-ish). Other tracers, including perf_events and SystemTap, can do this as well.

Many other MySQL functions can also be traced for further insight. Listing and counting them:

```
# ./uprobe -l /opt/bin/mysqld | more
account_hash_get_key
add_collation
add_compiled_collation
add_plugin_noargs
adjust_time_range
[...]
# ./uprobe -l /opt/bin/mysqld | wc -l
21809
```

That's 21 thousand functions. We can also trace library functions, and even individual instruction offsets.

User-level dynamic tracing is awesome, and can solve countless problems. But it can also be difficult to work with: figuring out which code to trace, processing function arguments, and dealing with code changes.

User-level Statically Defined Tracing

A USDT probe (or user-level "marker") is where the developer has added tracing macros to their code at interesting locations, with a stable and documented API. It makes tracing easier. (If you are a developer, see Adding User Space Probing for an example of how to add these.)

With USDT, instead of tracing `_Z16dispatch_command19enum_server_commandP3THDPcj`, the C++ symbol for `dispatch_command()`, I can simply trace a probe called `mysql:query__start`.

I can still trace `dispatch_command()` if I want to, and the 21 thousand other mysqld functions, but only when needed: drilling down when the USDT probes don't solve an issue.

USDT in Linux

Static tracepoints, in one form or another, have existed for decades. It was recently made popular by Sun's DTrace utility, leading to them being placed in many common applications, including MySQL, PostgreSQL, Node.js, and Java. SystemTap developed a way to consume these DTrace probes.

You may be running a Linux application that already contains USDT probes, or, it may require a recompilation (usually --enable-dtrace). Check using `readelf`. Eg, for Node.js:

```
# readelf -n node
[...]
Notes at offset 0x00c43058 with length 0x00000494:
  Owner                 Data size   Description
  stapsdt               0x0000003c   NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: gc__start
    Location: 0x0000000000bf44b4, Base: 0x0000000000f22464, Semaphore: 0x0000000001243028
    Arguments: 4@%esi 4@%edx 8@%rdi
[...]
  stapsdt               0x00000082       NT_STAPSDT (SystemTap probe descriptors)
    Provider: node
    Name: http__client__request
    Location: 0x0000000000bf48ff, Base: 0x0000000000f22464, Semaphore: 0x0000000001243024
    Arguments: 8@%rax 8@%rdx 8@-136(%rbp) -4@-140(%rbp) 8@-72(%rbp) 8@-80(%rbp) -4@-144(%rbp)
[...]
```

This is node recompiled with `--enable-dtrace`, and with the systemtap-sdt-dev package, which provided a "dtrace" utility to build USDT support. Two probes are shown here: `node:gc__start` (garbage collection start), and `node:http__client__request`.

At this point you can use SystemTap or LTTng to trace these. The built-in Linux tracers, ftrace and perf_events, cannot (although support for perf_events is in development). However...

Hacking with Ftrace: Tracing Addresses

From the `readelf` output above, `node:gc__start` is at 0xbf44b4. What is this address?

```
$ gdb node
(gdb) print/a 0xbf44b4
$1 = 0xbf44b4 <_ZN4node15dtrace_gc_startEPN2v87IsolateENS0_6GCTypeENS0_15GCCallbackFlagsE+4>
(gdb) disas _ZN4node15dtrace_gc_startEPN2v87IsolateENS0_6GCTypeENS0_15GCCallbackFlagsE
Dump of assembler code for function _ZN4node15dtrace_gc_startEPN2v87IsolateENS0_6GCTypeENS0_15GCC
   0x0000000000bf44b0 <+0>:     push   %rbp
   0x0000000000bf44b1 <+1>:     mov    %rsp,%rbp
   0x0000000000bf44b4 <+4>:     nop
   0x0000000000bf44b5 <+5>:     pop    %rbp
   0x0000000000bf44b6 <+6>:     retq
End of assembler dump.
```

It's a nop (no-operation). Ftrace and uprobes can instrument instructions, so we can instrument this address.

Since this is not a shared library, we need to know and subtract the base load address:

```
# objdump -x /opt/node/node | more
/opt/node/node:     file format elf64-x86-64
/opt/node/node
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000617770

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
         filesz 0x00000000000001f8 memsz 0x00000000000001f8 flags r-x
  INTERP off    0x0000000000000238 vaddr 0x0000000000400238 paddr 0x0000000000400238 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
         filesz 0x0000000000c2ada4 memsz 0x0000000000c2ada4 flags r-x
    LOAD off    0x0000000000c2bca8 vaddr 0x000000000122bca8 paddr 0x000000000122bca8 align 2**21
         filesz 0x0000000000017384 memsz 0x00000000000020cb0 flags rw-
[...]
```

It is 0x400000 (typical for x86_64). The uprobes documentation, uprobetracer.txt, gets this from /proc/PID/maps, however, that technique requires a running process.

Subtracting 0x400000 from 0xbf44b4 = 0x7f44b4, which is the address to use with uprobes (because it is not a shared library). Tracing it using my uprobe tool (mentioned earlier):

WARNING: the address must be correct and instruction-aligned

```
# ./uprobe 'p:gc_start node:0x7f44b4'
Tracing uprobe gc_start (p:gc_start /opt/node/node:0x7f44b4). Ctrl-C to end.
             node-23484 [002] d... 19993896.988891: gc_start: (0xbf44b4)
             node-23467 [003] d... 19993897.079658: gc_start: (0xbf44b4)
             node-23484 [003] d... 19993897.326284: gc_start: (0xbf44b4)
             node-23484 [003] d... 19993897.402107: gc_start: (0xbf44b4)
             node-23467 [001] d... 19993897.605167: gc_start: (0xbf44b4)
[...]
```

Neat. I gave it the alias "gc_start". We can see GC from two node processes occurring.

This is something we'd prefer a better front-end for, one that has more safety checks (eg, perf_events). Ftrace assumes you know what you're doing.

```
perf_events> look
YOU SEE A CUP OF POISON.

perf_events> drink cup
I'M AFRAID I WON'T DO THAT. BECAUSE IT IS POISON.

ftrace> drink cup
GLUG GLUG GLUG!
```

If you get the address wrong, and you're lucky, the target process will immediately crash ("illegal instruction"). If you're unlucky, it will be in an unknown corrupted state.

Hacking with Ftrace: Is-Enabled

Let's use the previous technique with the second probe I highlighted: `node:http__client__request`:

```
# ./uprobe p:node:0x7f48ff
Tracing uprobe node_0x7f48ff (p:node_0x7f48ff /opt/node/node:0x7f48ff). Ctrl-C to end.
^C
Ending tracing...
```

Nothing. This probe should be firing (there is load), but isn't. The reason was in the `readelf` output:

```
    Location: 0x0000000000bf48ff, Base: 0x0000000000f22464, Semaphore: 0x0000000001243024
```

This probe uses a semaphore for the implementation of is-enabled: a feature from DTrace where the tracer can inform the target process that a particular event is being traced. The target process can then choose to do some more expensive processing, usually fetching and formatting arguments for a USDT probe.

We need to set the semaphore. The current value should be zero, and is:

```
# dd if=/proc/12647/mem bs=1 count=1 skip=$(( 0x1243026 )) 2>/dev/null | xxd
0000000: 00                                        .
```

Let's set this to 1 (given it was zero to start with):

WARNING: This is a proof of concept, and not intended for real use

```
# printf "\x1" | dd of=/proc/12647/mem bs=1 count=1 seek=$(( 0x1243024 ))
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.5286e-05 s, 28.3 kB/s
# dd if=/proc/12647/mem bs=1 count=1 skip=$(( 0x1243024 )) 2>/dev/null | xxd
0000000: 01                                        .
```

Yes, I'm piping the output of bash directly over target memory. You should probably never, ever, do this!

It did work, and I can now trace the probe:

```
# ./uprobe 'p:client_request /opt/node/node:0x7f48ff' | head -5
Tracing uprobe client_request (p:client_request /opt/node/node:0x7f48ff). Ctrl-C to end.
            node-12647 [001] d... 20025472.106062: client_request: (0xbf48ff)
            node-12647 [001] d... 20025472.106601: client_request: (0xbf48ff)
            node-12647 [001] d... 20025472.107240: client_request: (0xbf48ff)
            node-12647 [001] d... 20025472.107813: client_request: (0xbf48ff)
```

Good. I should decrement the semaphore when done.

This needs a better front-end to do semaphores properly. My bash one-liner has no error checking.

```
bash|dd|/dev/mem> look
YOU ARE STANDING IN A FOREST. YOU SEE THE TARGET PROCESS TO THE NORTH.

bash|dd|/dev/mem> northh
I DON'T UNDERSTAND "NORTHH". YOU DIE.
```

If you get the address wrong, you will corrupt memory.

Hacking with Ftrace: Arguments

Now that we can trace USDT probes, how about their arguments? They were shown in `readelf` as well, eg:

```
    Name: gc__start
    Arguments: 4@%esi 4@%edx 8@%rdi

    Name: http__client__request
    Arguments: 8@%rax 8@%rdx 8@-136(%rbp) -4@-140(%rbp) 8@-72(%rbp) 8@-80(%rbp) -4@-144(%rbp)
```

This shows their register and stack locations.

The arguments to `gc__start` can be determined from the node source that builds the USDT probe. Eg:

```
src/node_provider.d:
 54 provider node {
[...]
 78         probe gc__start(int t, int f, void *isolate);

src/node_dtrace.cc:
293 void dtrace_gc_start(Isolate* isolate, GCType type, GCCallbackFlags flags) {
294   // Previous versions of this probe point only logged type and flags.
295   // That's why for reasons of backwards compatibility the isolate goes last.
296   NODE_GC_START(type, flags, isolate);
297 }
```

So the first argument is "type", and is stored in `4@%esi`: size 4, and register %esi. (Here's the reference for the syntax, which is based on assembly.) Tracing it, and naming it "gctype":

```
# ./uprobe 'p:client_request /opt/node/node:0x7f44b4 gctype=%si:u32'
Tracing uprobe client_request (p:client_request /opt/node/node:0x7f44b4 gctype=%si:u32). Ctrl-C t
            node-12617 [000] d... 20031814.978011: client_request: (0xbf44b4) gctype=0x1
            node-12647 [002] d... 20031814.985857: client_request: (0xbf44b4) gctype=0x1
            node-12647 [001] d... 20031815.323407: client_request: (0xbf44b4) gctype=0x1
            node-12647 [001] d... 20031815.412093: client_request: (0xbf44b4) gctype=0x2
            node-12617 [000] d... 20031815.535844: client_request: (0xbf44b4) gctype=0x1
[...]
```

Great! From deps/v8/include/v8.h: type 1 is scavange, and type 2 is mark-sweep-compact.

Other arguments can get a lot tricker, but are still possible for the really determined. For example, fetching the url and method for `node:http__client__request`:

```
# ./uprobe 'p:client_request /opt/node/node:0x7f48ff +0(+0(%ax)):string' | head -5
Tracing uprobe client_request (p:client_request /opt/node/node:0x7f48ff +0(+0(%ax)):string). Ctrl
            node-12647 [003] d... 20031007.438200: client_request: (0xbf48ff) arg1="/"
            node-12647 [003] d... 20031007.438776: client_request: (0xbf48ff) arg1="/"
            node-12647 [003] d... 20031007.439322: client_request: (0xbf48ff) arg1="/"
            node-12647 [003] d... 20031007.439921: client_request: (0xbf48ff) arg1="/"

# ./uprobe 'p:client_request /opt/node/node:0x7f48ff +0(+8(%ax)):string' | head -5
Tracing uprobe client_request (p:client_request /opt/node/node:0x7f48ff +0(+8(%ax)):string). Ctrl
            node-12647 [000] d... 20032205.934697: client_request: (0xbf48ff) arg1="GET"
            node-12647 [000] d... 20032205.935255: client_request: (0xbf48ff) arg1="GET"
            node-12647 [000] d... 20032205.935832: client_request: (0xbf48ff) arg1="GET"
            node-12647 [000] d... 20032205.936367: client_request: (0xbf48ff) arg1="GET"
```

Thy argument `+0(+8(%ax)):string` begins with a register (%ax), then dereferences offsets (+8(), then +0), then casts it (:string). This syntax is similar to the earlier argument notation, but is still basically brainf*ck. It is documented in uprobetracer.txt and kprobetrace.txt (see FETCHARGS), but I don't know if it has a real name.

But the real point, as with the other hacking, is that I am getting this to work without modifying the kernel. What we want is a better front end.

Safer Hacking

Now that I've shown that this possible, I'll summarize a few different ways to make this safer.

- **Rewrite the above in C**. Include error and safety checks.
- **Tracing of the parent function**. The earlier `node::gc__start` was called directly by `dtrace_gc_start()`, which we could just trace using user-level dynamic tracing, as supported by ftrace, perf_events, and other tracers. Check the code: this works when there are no branches between the probe macro and function entry, so that tracing them is essentially the same.
- **Tracing of another canary function**. If the parent function isn't suitable, there may be a prior unique function call in the code that is, and can be traced using user-level dynamic tracing.
- **Tracing via the function offset**. A USDT probe location can be traced using the offset from the parent function. Eg, with perf_events, using `perf probe -x binary 'func+offset'`. This may be safer as it can involve more error checking (depending on the tracer), including ensuring the offset is in the bounds of the function, and that it is instruction-aligned.
- **Tracing USDT using another tracer**. SystemTap or LTTng support USDT.
- **Wait for perf_events or ftrace support**. At least two people have worked on proper perf_events USDT support so far. Can't wait!

As for my prototype usdt tool:

```
#./usdt -ip 12647 'node:http__client__request +0(+0(%ax)):string'
Tracing uprobe node_http__client__request (p:node_http__client__request /opt/node/node:0x7f48ff +
    node-12647 [001] d... 20047728.328532: node_http__client__request: (0xbf48ff) arg1="/"
    node-12647 [001] d... 20047728.329050: node_http__client__request: (0xbf48ff) arg1="/"
    node-12647 [001] d... 20047728.329577: node_http__client__request: (0xbf48ff) arg1="/"
    node-12647 [001] d... 20047728.330153: node_http__client__request: (0xbf48ff) arg1="/"
    node-12647 [001] d... 20047728.330689: node_http__client__request: (0xbf48ff) arg1="/"
[...]
# ./usdt -h
USAGE: usdt [-FhHisv] [-d secs] [-p PID] {[-lL] target |
            usdt_probe [filter]}
                -F              # force. trace despite warnings.
                -d seconds      # trace duration, and use buffers
                -i              # enable isenabled probes. Needs -p.
                                # WARNING: writes to target memory.
                -l target       # list usdt probes from this executable
                -L target       # list usdt probes and arguments
                -p PID          # PID to match
                -v              # view format file (don't trace)
                -H              # include column headers
                -s              # show user stack traces
                -h              # this usage message
[...]
```

For the warnings described earlier, I figured it would be reckless to share this in its current form. I'll consider improving its safety once my company is on a newer kernel, and we're more likely to use it (it uses uprobes, which aren't stable on the kernels we are running: a lot of 3.13).

perf_events USDT support might arrive later this year, and I can rewrite uprobe to use it. That will be nice, but in the meantime there are other options, including SystemTap for USDT, and both ftrace and perf_events for user-level dynamic tracing. It's not like we're missing out.

Conclusion

There is a way to access USDT probes using built-in Linux kernel functionality: ftrace and uprobes. Right now it's a proof of concept, and we lack a decent front-end tool with error and safety checking. (But you may have a desperate need, and this is useful to know anyway; although, if so, do check out SystemTap and LTTng.) In the future, built-in USDT support can only get better from here!

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).