Dynamic Instrumentation

Golang Meetup Bangalore XXIV
15 July 2017

@JeffryMolanus @openebs

Tracing

- To record information about a programs execution
  - Useful for understanding code, in particular a very large code base
  - Used during debugging, statistics, so on and so forth
- Dynamic tracing is the ability to ad-hoc add or remove certain instrumentation without making changes to the code that is subject to tracing or restarting the program or system
- In general, tracing should not effect the stability of the program that is being traced in production, during development its less of importance
- When no tracing is enabled there should be no overhead; when enabled the overhead depends on what is traced and how
- User land tracing requires abilities in kernel (which is the focus of this talk)
  - user space tracing has a little more overhead due to the induced context switch

Tracers on other platforms

- Illumos/Solaris and FreeBSD
  - Dtrace, very powerful and production safe used for many years
  - Compressed Type Format (CTF) data is available in binaries and libraries, no need for debug symbols to work with the types
  - Solaris uses the same CTF data for type information for debugging
- Event Tracing for Windows (EWT)
- Linux
  - Requires debug symbols to be downloaded depending on what you trace and how specific you want to trace
  - With DWARF data more can be done then with plain CTF however

Basic architecture of tracing

- There are generally, two parts of tracing in Linux
- Frontend tools to work/consume with/the in kernel tracing facilities
  - We will look briefly in ftrace, systemtap and BCC
- Backend subsystems
  - Kernel code that executes what ever code you want to be executed on entering the probes function or address
  - kprobes, probes, tracepoints, sysdig

ftrace

- **Tracepoints**; static probes defined in the kernel that can be enabled at run time
  - ABI is kept stable by kernel
  - static implies you have to know what you want to trace while developing the code
- Makes use of sysfs interface to interact with it
- Several wrappers exist to make things a little easier
  - tracecmd and kernelshark (UI)
  - Also check the excellent stuff from Brendan Gregg

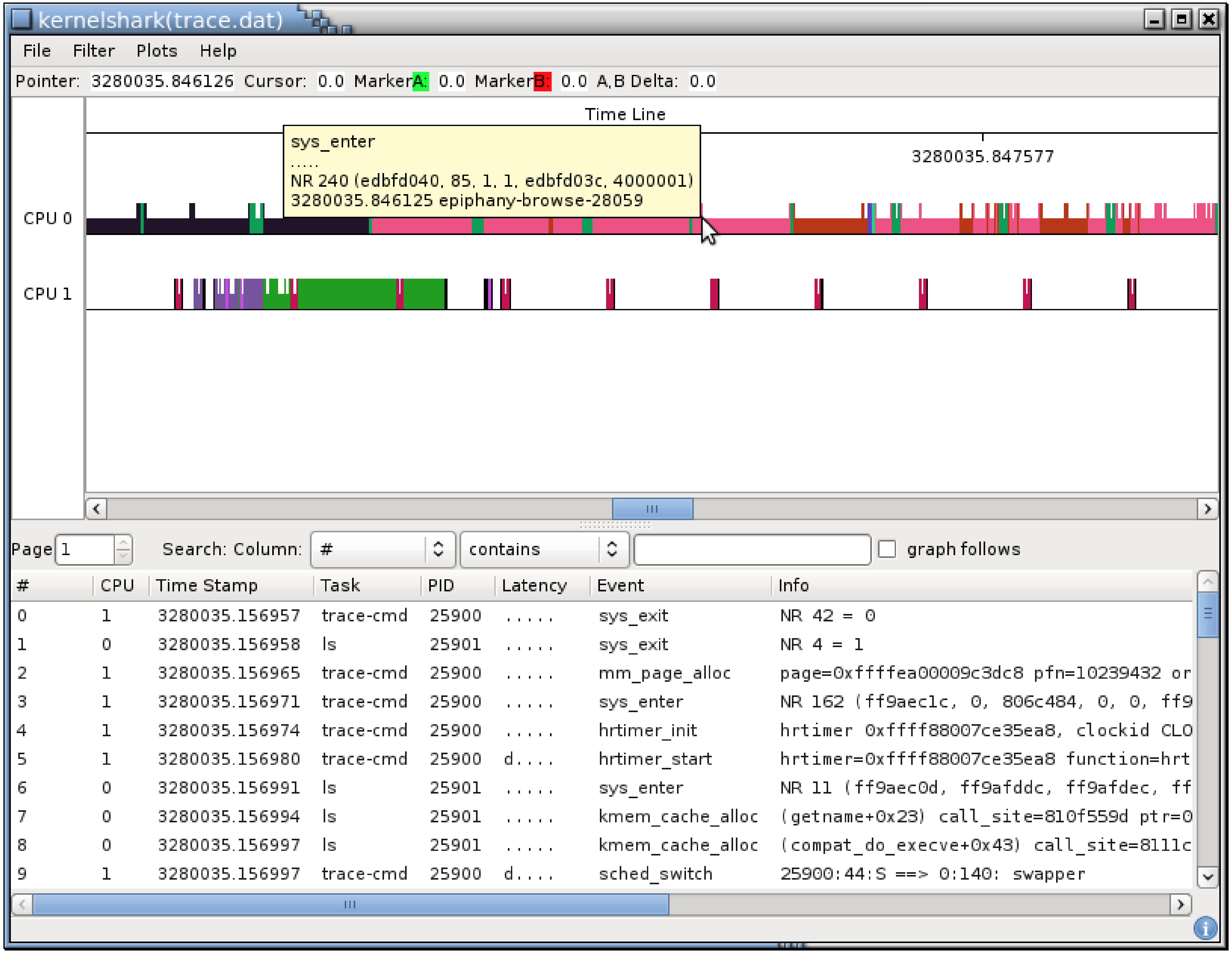
Adding a tracepoint

```
TRACE_EVENT(ext4_request_inode,
    TP_PROTO(struct inode *dir, int mode),

    TP_ARGS(dir, mode),

    TP_STRUCT__entry(
            __field(        dev_t,  dev                     )
            __field(        ino_t,  dir                     )
            __field(        __u16, mode                     )
    ),

    TP_fast_assign(
            __entry->dev    = dir->i_sb->s_dev;
            __entry->dir    = dir->i_ino;
            __entry->mode   = mode;
    ),

    TP_printk("dev %d,%d dir %lu mode 0%o",
              MAJOR(__entry->dev), MINOR(__entry->dev),
              (unsigned long) __entry->dir, __entry->mode)
);
```

Trace points in sysfs

```
➜  ext4_request_inode pwd
/sys/kernel/debug/tracing/events/ext4/ext4_request_inode
➜  ext4_request_inode cat format
name: ext4_request_inode
ID: 920
format:
	field:unsigned short common_type;	offset:0;	size:2;	signed:0;
	field:unsigned char common_flags;	offset:2;	size:1;	signed:0;
	field:unsigned char common_preempt_count;	offset:3;	size:1;	signed:0;
	field:int common_pid;	offset:4;	size:4;	signed:1;

	field:dev_t dev;	offset:8;	size:4;	signed:0;
	field:ino_t dir;	offset:16;	size:8;	signed:0;
	field:__u16 mode;	offset:24;	size:2;	signed:0;
```

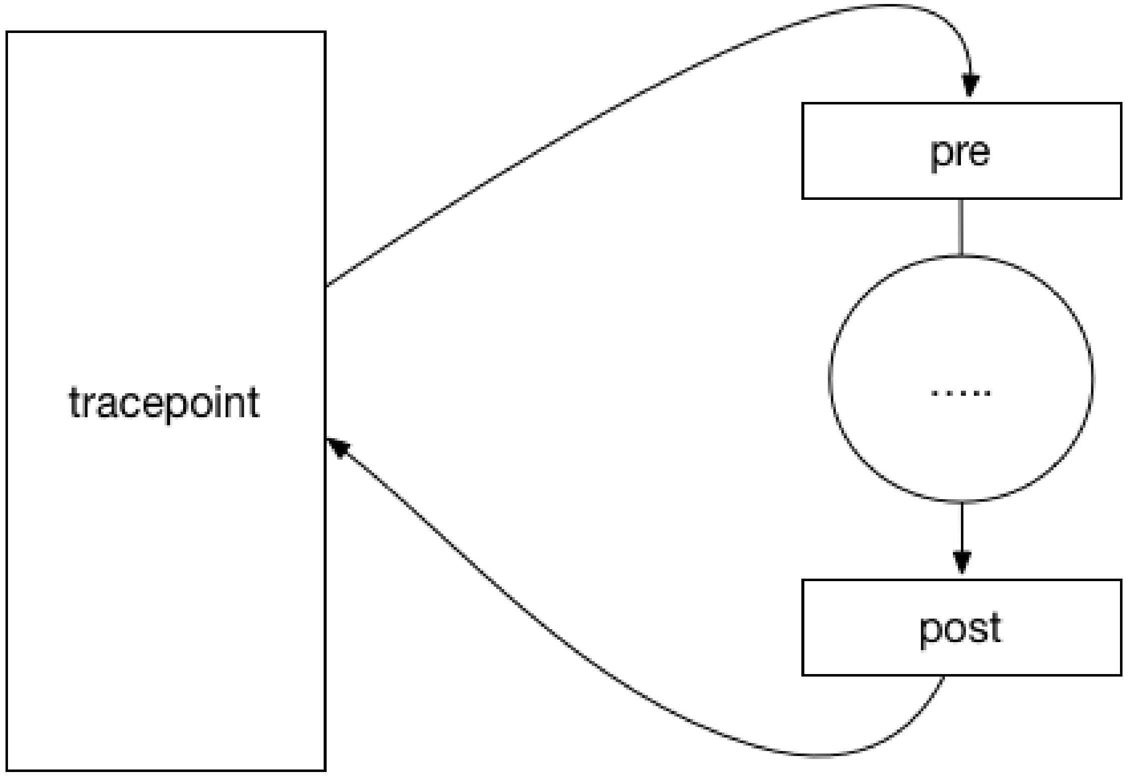
kernelshark



kprobes

- kprobes is defined in multiple sub categories
  - **jprobes**: trace function entry (optimised for function entry, copy stack)
  - **kretprobes**: trace function return
  - **kprobes**: trace at any arbitrary instruction in the kernel
- To use it one has to write a kernel module which needs to be loaded at run time
  - this is not guaranteed to be safe
- A kprobe replaces the traced instruction with a break point instruction
  - On entry, the pre_handler is called after instrumenting, the post handler

kprobes



Kprobe example

```c

/*For each probe you need to allocate a kprobe structure*/
static struct kprobe kp;

/*kprobe pre_handler: called just before the probed instruction is executed*/
int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    printk("pre_handler: p->addr=0x%p, eip=%lx, eflags=0x%lx\n",
        p->addr, regs->eip, regs->eflags);
    dump_stack();
    return 0;
}

/*kprobe post_handler: called after the probed instruction is executed*/
void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned long flags)
{
    printk("post_handler: p->addr=0x%p, eflags=0x%lx\n",
        p->addr, regs->eflags);
}

/* fault_handler: this is called if an exception is generated for any
 * instruction within the pre- or post-handler, or when Kprobes
 * single-steps the probed instruction.
 */
int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapnr)
{
    printk("fault_handler: p->addr=0x%p, trap #%dn",
        p->addr, trapnr);
    /* Return 0 because we don't handle the fault. */
    return 0;
}
```

Kprobe example

```
int init_module(void)
{
    int ret;
    kp.pre_handler = handler_pre;
    kp.post_handler = handler_post;
    kp.fault_handler = handler_fault;
    kp.addr = (kprobe_opcode_t*) kallsyms_lookup_name("do_fork");
    /* register the kprobe now */
    if (!kp.addr) {
        printk("Couldn't find %s to plant kprobe\n", "do_fork");
        return -1;
    }
    if ((ret = register_kprobe(&kp) < 0)) {
        printk("register_kprobe failed, returned %d\n", ret);
        return -1;
    }
    printk("kprobe registered\n");
    return 0;
}

void cleanup_module(void)
{
    unregister_kprobe(&kp);
    printk("kprobe unregistered\n");
}
```

jprobes

- Note: function prototype needs t match the actual syscall

```c
/*
 * Jumper probe for do_fork.
 * Mirror principle enables access to arguments of the probed routine
 * from the probe handler.
 */

/* Proxy routine having the same arguments as actual do_fork() routine *
long jdo_fork(unsigned long clone_flags, unsigned long stack_start,
              struct pt_regs *regs, unsigned long stack_size,
              int __user * parent_tidptr, int __user * child_tidptr)
{
    printk("jprobe: clone_flags=0x%lx, stack_size=0x%lx, regs=0x%p\n",
              clone_flags, stack_size, regs);
    /* Always end with a call to jprobe_return(). */
    jprobe_return();
    /*NOTREACHED*/
    return 0;
}

static struct jprobe my_jprobe = {
    .entry = (kprobe_opcode_t *) jdo_fork
};
```

utrace/uprobes

- Roughly the the same as the kprobe facility in the kernel but focused on user land tracing
- current ptrace() in linux is implemented using the utrace frame work
  - tools like strace and GDB use ptrace()
  - Allows for more sophisticated tooling, one of which is uprobes
- Trace points are placed on the an inode:offset tuple
  - All binaries that map that address will have a SW breakpoint injected at that address

ftrace & user space

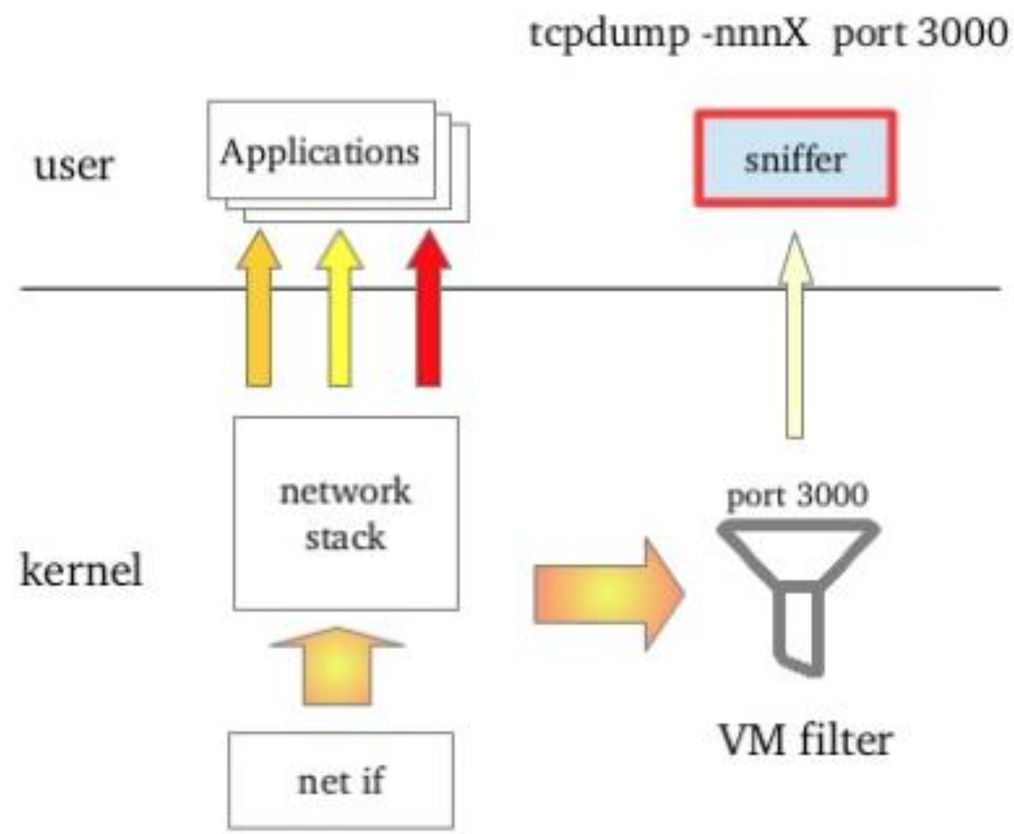
- The same ftrace interface is available for working with uprobes
- Behind the scene the kernel does the right thing (e.g use kprobe, tracepoints, or uprobes)
- The same sysfs interface is used, general work flow:
  - Find address to place the probe on
  - Enable probing
  - Disable probing
  - View results (flight recorder)

```
➜  tracing objdump -T /bin/zsh | grep -w zfree
000000000005de10 g    DF .text  0000000000000012  Base        zfree
➜  tracing echo 'p:zfree_entry /bin/zsh:0x5de10 %ip %ax' > uprobe_events

➜  tracing cat uprobe_events
p:uprobes/zfree_entry /bin/zsh:0x000000000005de10 arg1=%ip arg2=%ax
➜  tracing echo 1 > events/uprobes/enable

➜  tracing echo 0 > events/uprobes/enable
```

```
n-4283  [003] d...   405.794974: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x40
n-4283  [003] d...   405.794980: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x7f39b6e80
n-4283  [003] d...   405.794982: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x0
n-4283  [003] d...   405.794985: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x40
n-4283  [003] d...   405.795002: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x105
n-4283  [003] d...   405.795060: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x7f39b6e7c
n-4283  [003] d...   405.795070: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x7f39b6e7c
n-4283  [003] d...   405.795072: zfree_entry: (0x562034863e10) arg1=0x562034863e10 arg2=0x0
```

eBPF

- Pretty sure everyone here has used BPF likely with out knowing
  - tcpdump uses BPF
- eBPF is enhanced BPF
  - sandboxed byte code executed b kernel which is safe and user defined
  - attach eBPF to kprobes and uprobes
  - certain restrictions in abilities



BCC

- BPF Compiler Collection, compiles code for the in kernel VM to be executed
- Several high level wrappers for Python, lua and GO
- Code is still written in C however

```
const std::string BPF_PROGRAM = R"(
int on_sys_clone(void *ctx) {
  bpf_trace_printk("Hello, World! Here I did a sys_clone call!\n");
  return 0;
}
)";
```

Recap

- Several back-end tracing capabilities in the kernel
  - Tracepoints, kprobes, jprobes, kretprobes and uprobes
  - eBPF allows attachment to kprobe, uprobes and tracepoints for **safe** execution
- Linux tracing world can use better generic frontends for adhoc tracing
  - Best today are perf and systemtap (IMHO)
  - Who wants to write C when you want to print a member of a complex struct? (ply)

Systemtap

- High level scripting language to work with the aforementioned tracing capabilities of Linux
- Flexible as it allows for writing scripts that can trace specific lines within a file (debug symbols)
- Next to tracing, it can also make changes to running programs when run in “guru mode”
- Resulting scripts from systemtap are kernel modules that are loaded in to the kernel (kprobe and uprobes)
- Adding a eBPF target is in the works as currently, systemtap may result in unremovable modules or sudden death of traced processes

stp files

- Example script oneliner:
  - stap -e ‘probe syscall.open { printf(“exec %s, file%s, execname(), filename) }’
- stap -L ‘syscall.open'
  - syscall.open: __nr:long name:string filename:string flags:long flags_str:string mode:long argstr:string
- List user space functions in process “trace”
  - stap -L ‘process(“./trace").function("*")'
  - .call and .return probes for each function

List probes

```
➜ talk stap -L 'process("./trace").function("*")'

process("/code/talk/trace").function("__do_global_dtors_aux")
process("/code/talk/trace").function("__libc_csu_fini")
process("/code/talk/trace").function("__libc_csu_init")
process("/code/talk/trace").function("_fini")
process("/code/talk/trace").function("_init")
process("/code/talk/trace").function("_start")
process("/code/talk/trace").function("deregister_tm_clones")
process("/code/talk/trace").function("frame_dummy")
process("/code/talk/trace").function("larger@/code/talk/trace.c:10")
process("/code/talk/trace").function("main@/code/talk/trace.c:33") $ret:int
process("/code/talk/trace").function("register_tm_clones")
process("/code/talk/trace").function("smaller@/code/talk/trace.c:5")
process("/code/talk/trace").function("traceme2@/code/talk/trace.c:21") $x:int $y:int $number:int
process("/code/talk/trace").function("traceme@/code/talk/trace.c:15") $a:int $b:int
➜ talk
```

Tracing line numbers

- What's the value of ret after line 35?
- Could be done by tracing ret values, but that is not the purpose of this exercise
- gcc -g -O0
  - full debug info

```
14 int
15 traceme(int a, int b) {
16
17     return a + b;
18 }
19
20 int
21 traceme2(int x, int y) {
22
23     int number =  traceme(x, y);
24
25     if (number < 10)
26         smaller();
27     else
28         larger();
29
30     return 0;
31 }
32
33 int main(void) {
34
35     int ret = traceme(1,2);
36     ret = traceme2(5, 5);
37     ret = traceme2(1, 1);
38     ret = traceme2(traceme(3,3), 4);
39     return 0;
40 }
```

Tracing line number

```
➜ talk stap -e 'probe process("./trace").statement("main@/code/talk/trace.c:36") { printf("ret val here is %d\n", $ret)}'
ret val here is 3
^C#
➜ talk
```

- .statement("main@code/talk/trace.c:36") { … }

Understanding code flow

```
probe process("./trace").function("*@/code/talk/trace.c").call
{
    printf ("%s -> %s\n", thread_indent(1), probefunc())
}

probe process("./trace").function("*@/code/talk/trace.c").return
{
    if (@defined($return)) {
        printf ("%s <- %s return %x\n", thread_indent(-1), probefunc(), $return)
    } else {
        printf ("%s <- %s return 0/NULL\n", thread_indent(-1), probefunc())
    }


}

probe begin {

    printf("Press CTRL+C to exit\n")
}
```

Understanding code flow

```
➜  talk stap indent.stp
Press CTRL+C to exit
     0 trace(58698): -> main
    10 trace(58698):  -> traceme
    15 trace(58698):  <- main return 3
    19 trace(58698):  -> traceme2
    24 trace(58698):   -> traceme
    27 trace(58698):   <- traceme2 return a
    31 trace(58698):   -> larger
   108 trace(58698):   <- traceme2 return 0/NULL
   111 trace(58698):  <- main return 0
   116 trace(58698):  -> traceme2
   121 trace(58698):   -> traceme
   124 trace(58698):   <- traceme2 return 2
   128 trace(58698):   -> smaller
   134 trace(58698):   <- traceme2 return 0/NULL
   135 trace(58698):  <- main return 0
   139 trace(58698):  -> traceme
   142 trace(58698):  <- main return 6
   145 trace(58698):  -> traceme2
   151 trace(58698):   -> traceme
   153 trace(58698):   <- traceme2 return a
   157 trace(58698):   -> larger
   161 trace(58698):   <- traceme2 return 0/NULL
   163 trace(58698):  <- main return 0
   164 trace(58698): <- 0x7fcd94e893f1 return 0
```

Downstack

- All functions being called by a function

```
global trace = 0;

probe process("./trace").function("traceme2").call

{
    trace = 1;
}


probe process("./trace").function("traceme2").return

{
    trace = 0;
}


probe process("./trace").function("*@/code/talk/trace.c").return
{
        if (trace == 1 && ppfunc() != "traceme2")
            printf("%s called from traceme2\n", ppfunc())

}

probe begin {

     printf("Press CTRL+C to exit\n")
}
```

```
➜ talk stap downstack.stp
Press CTRL+C to exit
traceme called from traceme2
larger called from traceme2
traceme called from traceme2
smaller called from traceme2
traceme called from traceme2
larger called from traceme2
```

Tracing go

```
➜ go cat main.go
package main

import "fmt"

func traceme(a int, b int) (ret int) {
        return a + b

}

func main() {

        ret := traceme(1, 2)
        fmt.Printf("ret %d\n", ret)
}
➜ go stap -e 'probe process("./stap").function("main.traceme") { printf("%d and %d\n",$a, $b)}' -c ./stap

ret 3
1 and 2
➜ go
```

Cant trace return values

```
➜  go go build -gcflags "-N -l"  -o stap
➜  go stap -e 'probe process("./stap").function("main.traceme").return { printf("%d \n", $return)}' -c ./stap

semantic error: while processing probe process("/code/go/stap").function("main.traceme@/code/go/main.go:5").return f
rom: process("./stap").function("main.traceme").return

semantic error: function main.traceme (go) has no return value: identifier '$return' at <input>:1:75
        source: probe process("./stap").function("main.traceme").return { printf("%d \n", $return)}
                                                                                          ^

Pass 2: analysis failed.  [man error::pass2]
➜  go
```

Calling convention

- AMD64 calling conventions
  - RDI, RSI, RDX, RCX, R8 and R9
- Go is based on PLAN9 which uses a different approach therefore tracing does not work as well as one would like it to be (yet)
  - This also goes for debuggers
- Perhaps Go will start using the X86_64 ABI as it moves forward or all tools and debuggers will add specific PLAN9 support
  - https://go-review.googlesource.com/#/c/28832/ (ABI change?)
- GO bindings to the BCC tool chain
  - Allows for creating eBPF tracing tools written in go
  - but still requires writing the actual trace logic in C

Summary

- Dynamic tracing is an invaluable tool for understanding code flow
- To verify hypotheses around software bugs or understanding
- Ability to make changes to code on the fly with out recompiling (guru mode)
- Under constant development most noticeable the eBPF/BCC work