Tracing Tools

Michal Sekletár
msekleta@redhat.com

March 21, 2019
whoami

- Senior Software Engineer
- RHEL systemd maintainer
- SW engineer interested in tracing and debugging
Have you ever wanted to answer questions like,

- What files in `/etc` are being accessed on the system?
- How big are memory allocations done by [DAEMON] (insert your favorite)?
- What process is a source of slow filesystem operations (e.g. slower than 50 ms)?
PART I – Introduction
- Tracing vs. Debugging
- Goals
- Methodology

PART II – Tools
- strace and ltrace (ptrace)
- trace-cmd (ftrace)
- SystemTap (kprobes)
- bcc-tools (eBPF)

PART III – Exercises
PART I – Introduction
Debugging

The process of identifying and removing errors from computer software.

Approaches
- Staring into the code
- Debug logging
- Debuggers
  - gdb
  - lldb
Tracing

Non-intrusive observation and monitoring of a software system.

Approaches

- Syscall monitoring
- Gathering execution traces
- Stack sampling
- Debug logging
Goals

- Better understanding of the system behavior
- Tracing should be as non-intrusive as possible
- In-kernel summarization (if possible) and statistical data gathering
Right tool for the job

What tracing tools should I use?

Unfortunately, answer to this question on Linux is not straightforward. We need to understand at least two things,

- Goal
- Tracing target

<table>
<thead>
<tr>
<th>Goal</th>
<th>Target</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good observability</td>
<td>kernel</td>
<td>SystemTap</td>
</tr>
<tr>
<td>Versatility</td>
<td>user-space/kernel</td>
<td>SystemTap</td>
</tr>
<tr>
<td>Non-intrusivness</td>
<td>kernel</td>
<td>trace-cmd(^1)</td>
</tr>
<tr>
<td>In-kernel data aggregation</td>
<td>kernel</td>
<td>bpftrace</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>user-space</td>
<td>strace</td>
</tr>
</tbody>
</table>

\(^1\) depends on the filter setting
Both strace and ltrace leverage ptrace() subsystem

Very old syscall and clunky interface

ptrace first appeared in Version 6 of AT&T UNIX

ptrace allows you to dynamically attach to the process to observe and control its execution

- **request** – type of requested action
  - **pid** – TID of a target process
  - **addr** – User-space address where to write or read from
  - **data** – Data buffer (exact type depends on request type)

- **tracer** – process which calls ptrace()

- **tracee** – process whose TID is specified as the second argument to ptrace()
**ptrace() - Requests**

- **PTRACE_ATTACH** – Attaches tracer to a target
- **PTRACE_DETACH** – Detach tracer from a target
- **PTRACE_SYSCALL** – Resume execution of a target until next syscall enter/exit
- **PTRACE_GETREGS** – Fetches general purpose registers of a target
- **PTRACE_SETREGS** – Sets content of a general purpose registers of a target
- **PTRACE_PEEKTEXT** – Read a word of program text of a target from addr
- **PTRACE_POKETEXT** – Write a word at *addr to a target address-space

---

2Only one tracer per target
3Linux doesn’t have code and data address spaces
ptrace() - Demo

- DEMO
**Properties**

- strace is a well known system call tracer
- Leverages ptrace() subsystem
- Target is stopped at every system call twice (entry, exit)
- Tracing is slow
- Very intrusive and not suitable for diagnosing production issues
- Limited visibility (syscall layer only)
- Very good reporting
- System call tampering (delay, fault injection)

**Example:** strace ls -l 2>&1 | head -n5
execve("/usr/bin/ls", ["ls", "-l"], 0x7ffe0914fff8 /* 34 vars */) = 0
brk(NULL) = 0x564d3e45b000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd875de7e0) = -1 EINVAL
access("/etc/ld.so.preload", R_OK) = -1 ENOENT
openat(AT_FDCWD, "/etc/ld.so.cache", 0_RDONLY|0_CLOEXEC) = 3
Interesting features

- `strace -e trace=%file` - trace filesystem related syscalls
- `strace -e trace='/(l|f)stat'` - trace lstat or fstat
- `strace -k` - stack trace at each syscall
- `strace -y` - print files associated with fd
- `strace -yy` - print network protocols associated with fd
- `strace -C` - system call statistics
- `strace -s 512` - make sure we have buffer big enough to print string arguments into
- `strace -P <path>` - print syscall accessing the path
- `strace -e trace='/(open(at)?' -P /var -e inject=all:error=ENOENT ls -l /var` - fail open or openat for of /var
Tools - ltrace

- Library call tracer
- Tool similar in nature to strace
- Adds breakpoint at PLT trampoline to notify ltrace about library call
- Very slow
- Not suitable for use in production
- Upstream is now unmaintained

Example: ltrace -e opendir+readdir ls -l /tmp/foo

ls->opendir("/tmp/foo") = { 3 }
ls->readdir({ 3 }) = { 1976910, "bar" }
ls->readdir({ 3 }) = { 1966081, ".." }
ls->readdir({ 3 }) = { 1975851, "." }
ls->readdir({ 3 }) = { 1980802, "baz" }
ls->readdir({ 3 }) = nil

total 0
-rw-rw-r--. 1 msekleta msekleta 0 Jan 25 10:03 bar
-rw-rw-r--. 1 msekleta msekleta 0 Jan 25 10:03 baz
+++ exited (status 0) +++
Itrace oneliners

# count number of memory allocations done by libselinux
ltrace -c -e 'malloc@libselinux.so.1' matchpathcon /etc/fstab >/dev/null

# show usage count for sd_bus APIs
ltrace -c -e 'sd_bus*@libsystemd*' systemctl status cups >/dev/null
• Interacts with Ftrace tracer that is built-in to the kernel
• In order to configure Ftrace tracer it writes files in `/sys/kernel/debug/tracing`.
• Directory contains mount of tracefs filesystem (or `trace-cmd` will mount it for you)
• Ftrace supports both static (predefined trace points) and dynamic (function tracing) tracing
• Ftrace fundamentally works in two phases
  • Collection phase (per-CPU buffers in which selected event traces are recorded)
  • Post-processing phase (content of tracing buffers is concatenated and stored in `trace.dat` file)
trace-cmd oneliners

# trace writes bigger than 10k bytes
trace-cmd record -e syscalls:sys_enter_write -f 'count > 10000'

# record function graph of all kernel function called by echo
trace-cmd record -p function_graph -F /bin/echo foo

# list all networking related events
trace-cmd list -e 'net:.*' 

# trace all ext4 related events
trace-cmd record -e ext4 ls
kProbes

- Linux framework for in-kernel dynamic tracing
- Allows to break into any kernel function
- Specified handlers are run once the breakpoint is hit
- 2 probe types
  - kprobes
  - kretprobes
- kprobes replace first byte at designated address with breakpoint instruction (int3 on x86_64)
- After breakpoint is hit execution state is saved and "pre-handler" is called
- kprobes then single step the trapped instruction
- kprobes then call "post-handler"
- Execution continue with next instruction
kProbes - example

- DEMO
**Tools - SystemTap**

- Complete framework for Linux tracing
- `dnf install -y systemtap && stap-prep`
- SystemTap takes tracing script, translates it to C which gets compiled into kernel module
- Previous point is a major problem for production use cases (partially solved by compilation server)
- kprobes and uprobes, i.e. with SystemTap you can trace both kernel and user-space
- Language is C-like and relatively easy to learn
global odds, evens

probe begin {
    for (i = 0; i < 10; i++) {
        if (i % 2)
            odds [no++] = i
        else
            evens [ne++] = i
    }
    delete odds[2]
    delete evens[3]
    exit()
}

probe end {
    foreach (x+ in odds)
        printf ("odds[%d] = %d", x, odds[x])
    foreach (x in evens-)
        printf ("evens[%d] = %d", x, evens[x])
}
Probes points

- `kernel.function(PATTERN)`
- `kernel.function(PATTERN).call`
- `kernel.function(PATTERN).return`
- `module(MPATTERN).function(PATTERN).call`
- `module(MPATTERN).function(PATTERN).return`
- `process("PATH").function("NAME")`
- `process("PATH").statement("*@FILE.c:123")`
- `process("PATH").library("PATH").function("NAME")`
- `process("PATH").library("PATH").statement("*@FILE.c:123")`

- Probe point context variables,
- `stap -L 'kernel.function("_do_fork")'`
- Tapset—big library of "ready-made" systemtap scripts
- **tid()** – The id of the current thread.
- **pid()** – The process (task group) id of the current thread.
- **uid()** – The id of the current user.
- **execname()** – The name of the current process.
- **cpu()** – The current cpu number.
- **gettimeofday()** – Number of seconds since epoch.
- **get_cycles()** – Snapshot of hardware cycle counter.
- **pp()** – A string describing the probe point being currently handled.
- **ppfunc()** – If known, the function name in which this probe was placed.
- **$vars** – If available, a pretty-printed listing of all local variables in scope.
- **print_backtrace()** – If possible, print a kernel backtrace.
- **print_ubacktrace()** – If possible, print a user-space backtrace.
Berkley Packet Filter (BPF)

- Berkley Packet Filter (BPF) is a technology used for packet filtering in UNIX like operating systems.

Example:

```
# tcpdump -i eno1 -d ip
(000) ldh [12]
(001) jeq #0x800 jt 2 jf 3
(002) ret #262144
(003) ret #0
```
BPF is a special assembly-like language which is specifically tailored for packet filtering.

Notice special addressing mode on previous slide ([12] is an offset to L2 PDU).

Filter expression is compiled (usually using `libpcap`) into the BPF program which is then loaded to the kernel.

```c
setsockopt(s, SOL_SOCKET, SO_ATTACH_FILTER, prog, sp)
```

Kernel contains virtual machine (VM) able to interpret BPF byte code on every received packet.
Extended Berkley Packet Filter (eBPF)

- eBPF is Linux only (so far) extension of classic BPF (cBPF)
- Instruction set is much richer
- Virtual Machine has now 10, 64 bit registers (cBPF only had 2)
- In-kernel JIT compiler
- Possibility to hook into various kernel subsystems
  - probe points
  - kprobes
  - cgroups
  - sockets
  - packet forwarding
  - system calls
- Subsystem is manipulated by bpf syscall (man 2 bpf)
- Possibility to share data structures (e.g. arrays, hash-maps) with kernel
- In-kernel aggregations (useful for histograms and stack-trace counting)
Extended Berkley Packet Filter (eBPF)

**bpf system call**

```c
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

- **cmd** – Requested action (e.g. BPF_PROG_LOAD, BPF_MAP_CREATE, BPF_MAP_ELEM_LOOKUP)
- **attr** – Precise value stored in attr union depends on action
- **size** – Size of object pointed by attr
Before BPF program is loaded the kernel will run static analyzer (verifier) to determine if it is safe to load it.

Verification consists of:

- Checking for loops
- Depth-first-search (DFS) of program’s Control Flow Graph (loops, unreachable instruction analysis)
- Program run simulation
- Checks for accesses or jumps based on uninitialized data
- Checks for out-of-bound accesses
- Checks for pointer arithmetics (in Secure Mode, i.e. processes w/o CAP_SYS_ADMIN)
BCC - Installation

- Option 1 – Install tools on your workstation
  - # dnf -y install bcc-tools kernel-devel-$(uname -r)
  - # export PATH=/usr/share/bcc/tools:/$PATH

- Option 2 – Try the tools in virtual machine (Vagrantfile)

  $ cat Vagrantfile
  Vagrant.configure("2") do |config|
  config.vm.box = "fedora/28-cloud-base"

  config.vm.provider "libvirt" do |lv|
  lv.memory = "1024"
  lv.cpus = "2"
  end

  config.vm.provision "shell", inline: <<-SHELL
  sudo dnf -y install bcc-tools kernel-devel-$(uname -r)
  sudo echo 'export PATH=/usr/share/bcc/tools:$PATH' >> /root/.bashrc
  SHELL
  end
execsnoop - trace execve() syscalls
opensnoop - trace open() syscalls
mountsnoop - trace mount() syscalls
statsnoop - trace stat() syscalls
ext4slower - trace ext4 operations slower than threshold (10ms by default)
gethostlatency - trace latency of DNS resolution
tcpaccept, tcpconnect - TCP related tracing tools
runqlen, runqlat - scheduler related tracing tools
filetop - identify "hot files" on the system
# Count all malloc() calls done by PID 1
funccount -p 1 c:malloc

# Collect and count all stack traces leading tcp_sendmsg()
stackcount tcp_sendmsg

# Count the libc write() calls for PID 1 by file descriptor
argdist -p 181 -C 'p:c:write(int fd):int:fd'

# Trace all malloc calls and print the size of the requested allocation
trace ':c:malloc "size = \%d", arg1'

# List all tracepoints
tplist

# List information about tracepoint cgroup:cgroup_attach_task
tplist -v cgroup:cgroup_attach_task

# Trace cgroup migrations system wide
trace 't:cgroup:cgroup_attach_task "\%d", args->pid'
PART III – Exercises
**Excercises**

- Print all getsockopt() and setsockopt() syscalls done by systemctl on AF_UNIX sockets
- What is the ratio of successful to failed socket option calls for systemctl?
- What files systemd opens (PID1) during restart of cups.service?
- Is there a memory leak in cups server during the dispatch of scheduler status request (lpstat -r)?
- Can you trace one process with strace and ltrace at the same time?
- What is the ICMP packet size sent by ping with default settings?
Exercises

- How many calls to kmalloc() is triggered by send+recv of single ICMP packet?
- How many allocation requests are bigger than 64 bytes?
- How many times does systemd call unit_load() on cups restart? In addition, print user-space backtrace on each call.
- What is protocol overhead of HTTP download of google.com (curl www.google.com)?
- What tool would you use in order to trace IPC via signals?
References

- https://strace.io
- https://www.kernel.org/doc/Documentation/trace/ftrace.txt
- https://sourceware.org/systemtap/langref/
- https://www.kernel.org/doc/Documentation/bpf/bpf_design_QA.txt