# Kernel debugging approaches 1/2

## Live data

**Vratislav Bendel**
Principal Software Maintenance Engineer

Red Hat

# What we'll cover today:

- Userspace data
  system monitoring and process tracing
- Perf
  The performance analysis swiss-knife
- Kernel tracing
  trace-cmd, perf probe
- Code injections
  systemtap, eBPF

# Mindset

Methodology matters

# Mindset

## How to think when tackling a problem

- 1) **Define the problem** and form a hypothesis
- 2) Determine how to verify the hypothesis - what data are needed
- 3) Use adequate tool to collect the data
- 4) Analyze the data and verify the hypothesis
- .. Finished?
  - Yes -> Well done!
  - No -> Repeat from 1)

# System live data

The bread and butter

# System live data

## Stats and infos

Lot of information are readable directly from kernel via VFS under */proc (procfs), /sys (sysfs), /sys/kernel/debug (debugfs). Userspace monitoring tools tend to get information from these interfaces.*

- /proc/meminfo, ../slabinfo, ../buddyinfo, ../zoneinfo
- /proc/sched_debug
- /sys/devices/...

# System live data

## Stats monitoring

From system administration perspective, it's always recommended to periodically collect system data and save them to permanent storage, in order to inspect historical statistics whenever needed.

There are many tools that collect such information:

- Simple: *sysstat (sar), collectl, nmon, atop*
- More complex: *Performance Co-Pilot, Prometheus*
- Data visualization: *Graphana*

- and more...

# Process tracing

user <-> kernel boundary

# Process tracing

## strace

- Tracing **syscall** invocations
- Significant overhead
- My favourite flags:

```
# strace  -fttTxCy -o <out> <comm>
```

# Process tracing

## ltrace

- Tracing **library** function calls (glibc, ...)
- Works similarly as *strace,* even has similar flags

# Perf

The performance analysis swiss-knife

# Perf

## What is it?

Collects aggregate data points for specific ***"events"*** to form statistics.

- ***# perf list*** – List available events
- Hardware events
  - Depend on specific HW
  - Performance Monitoring Unit [PMU] counters
- Software events
  - Kernel stats & trace events (*)



(*) Kernel trace events are further covered later in this lecture

# Perf

## perf stat

- ***# perf stat [-e event] [program]***
- Collects and outputs simple aggregates of the specified events
- Useful for general performance analysis / troubleshooting

- Mindset hint: *When troubleshooting performance, there should always be a "good" and "bad" example, or at least a "good" target that is realistic, so you may compare which adjustments have positive effect.*

# Perf

## perf record

- ***# perf record [-a] [-g] [-- program]***

Collects “cycles” event: Programs PMU counter(s) to periodically interrupt the CPU and record the RIP and other data.

- Hint: System-wide perf profile

*# perf record -ag -- sleep X*

# Perf

## perf report

- ***# perf report [-i perf.data]***

Perf-report command is used to inspect *perf.data* previously recorded by *perf-record* command. It has various options how to “look” at the recorded data.

- ***[--sort=...]*** - a “group-by” query model
- ***[--time]*** - inspect only specific timeslice
- ***[--no-children]*** - disable “graph aggregation”

# Perf

## perf.data analysis on another machine

To inspect *perf.data* on another machine, you want to have ***symbols*** from the machine where the data were recorded.

- ***# perf archive***
  - Generates a *perf.data.tar[.bz2|.xz]* archive containing symbol map for data that were recorded in *~/.debug* folder.
  - Does **NOT** contain the actual data
  - Unpack this data to local machine:

    *# rm -rf ~/.debug/* && tar xf perf.data.tar.xz -C ~/.debug/*



You may also use *perf-report's **"--kallsyms"** option to supply kernel symbols*

# Perf

## When to use perf?

Useful for:

- “What is the kernel doing?” - analyzing %sys CPU usage

- Performance profiling for applications, workloads or whole systems
  - Tuning
  - Troubleshooting

# Kernel tracing

Almost “breakpoints”

# Kernel tracing

## What is it?

Kernel source code contains defined “trace points”, also known as “events”. These can be dynamically enabled on live kernel to log data whenever execution passes the place in source code where the trace point is defined.

Uses *sync_core()* synchronization mechanism to live-patch kernel execution machine code - replaces designated places with *int3* instruction.

# Kernel tracing

## Trace event definition

- *include/linux/tracepoint.h*
  - Macro API definitions
    - *register_trace_##name(void (*probe)(data_proto), void *data)*

- *kernel/trace/**
  - Various tracepoint definitions

# Kernel tracing

## Control interface (VFS)

Tracing subsystem control built on VFS mounted at

- ***/sys/kernel/debug/tracing***

- echo 1|0 > **.../tracing/events/<group>/<event>/enabled**
- echo 1|0 > **.../tracing/tracing_on**
- cat **.../tracing/trace**

- and much much more...

# Kernel tracing

## trace-cmd

The *trace-cmd* tool is an efficient to use command-line wrapper to control kernel tracing interface via single-line commands.

- ***# trace-cmd list*** - list all events
- ***# trace-cmd record [-e event] [-o output_file]***
  - Default output: "*./trace.dat*"
- ***# trace-cmd report [-i input_file]***

- Hint: Write up post-processing parsers for trace-cmd-report output.

# Kernel tracing

## complex tracers

Kernel tracing subsystem features several complex tracers. Effectively these tracers are simply groups of specific events and can be enabled alongside any other events.

- ***# trace-cmd record [-p tracer]***
  - Example: *function_graph, osnoise*

- **Beware** of trace-buffer **overflow**, especially with function_graph tracer!
  - Hint: Control buffer size or write-out frequency via '-b' and '-s'

# Kernel tracing

## perf probe

Custom tracepoints can be defined via *.../tracing/kprobe_events*

The *perf-probe* command can be used as a wrap-up:

- ***# perf probe [-m module] [-a probe_definition] [-d probe_name]***
  - Probe definition syntax: ***# man perf probe***

- Then you can enable the probe as any other tracing event

# Code injections

Absolute control

# Code injections

## Concept

Tracing has the benefit of being relatively lightweight, but generally provides only read-only capabilities without any logic. Dynamic code patching can be used for even more complex adjustments - you can add logic or even modify live data.

Primary tools to achieve this are:

- Systemtap
- eBPF (extended Berkeley Packet Filter)

# Code injections

## Systemtap

- Scripting language to create custom complex probes
- Many pre-implemented library functions and probes (ref. *tapsets*)
- Can embed direct C-lang code (you can pretty much modify kernel :))
- Compiles a kernel module
- Relatively heavy-weight

- ***# man stap*** - manual entrypoint

# Code injections

## Systemtap

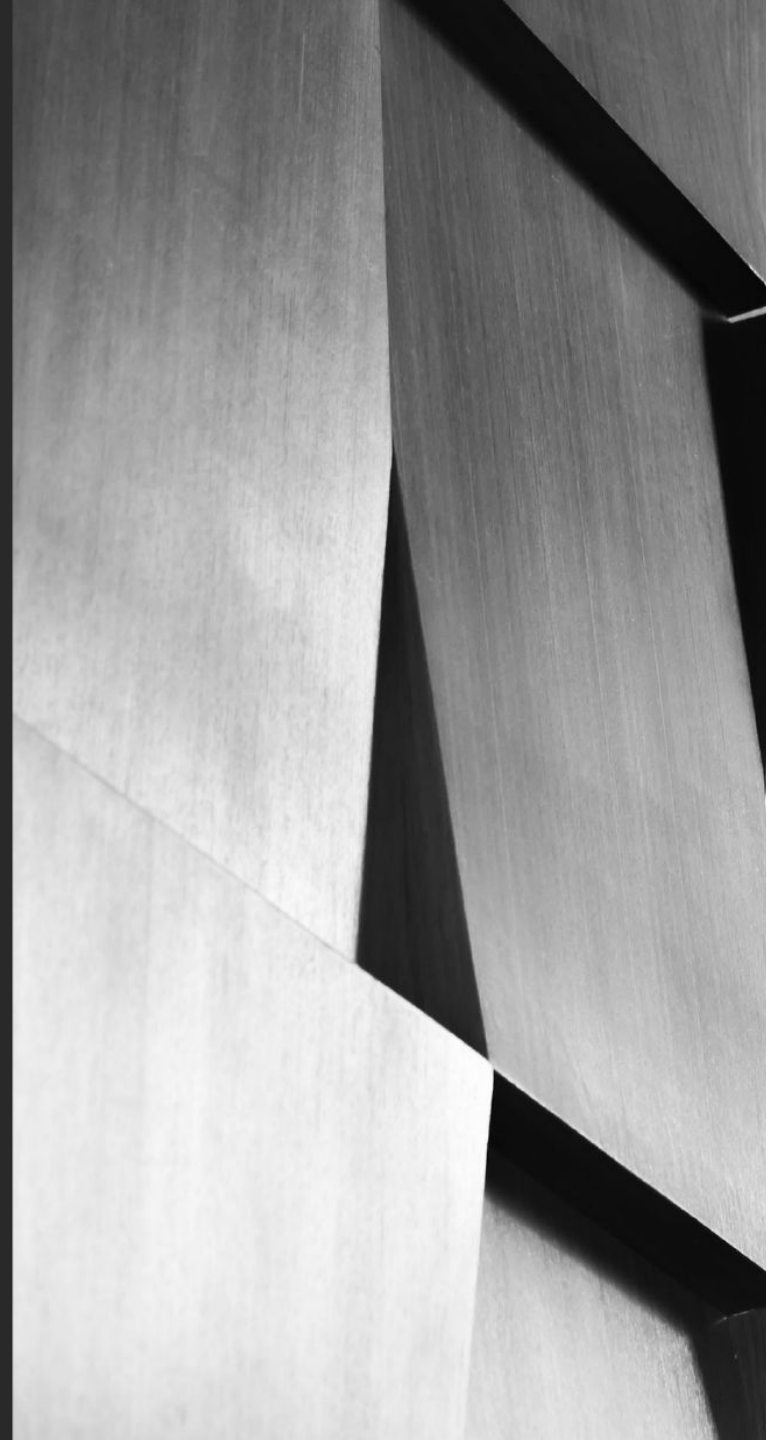
- Scripting language to create custom complex probes
- Many pre-implemented library functions and probes (ref. *tapsets*)
- Can embed direct C-lang code (you can pretty much modify kernel :))
- Compiles a kernel module
- Relatively heavy-weight

- ***# man stap*** - manual entrypoint

# Code injections

## eBPF

- Works on kprobes, similar as kernel tracing
- More lightweight than systemtap (no kernel module)
- Existing pre-implemented eBPF scripts - */usr/share/bpftrace/tools*
- Commonly used nowadays in kernel-bypass mechanisms (ex. DPDK)



# Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

facebook.com/redhatinc

youtube.com/user/RedHatVideos

twitter.com/RedHat

Red Hat