LF LIVE: MENTORSHIP SERIES

TOOLS AND TECHNIQUES TO DEBUG AN EMBEDDED LINUX SYSTEM

Sergio Prado, Embedded Labworks
https://www.linkedin.com/in/sprado
WHOAMI

- Designing and developing embedded software for 25+ years (Embedded Linux, Embedded Android, RTOS, etc).

- Consultant and trainer at Embedded Labworks for 12+ years.
  https://e-labworks.com/en

- Open source software contributor (Buildroot, Yocto Project, Linux kernel, etc).

- Blog and social networks:
  https://sergioprado.blog
  https://www.linkedin.com/in/sprado
  https://twitter.com/sergioprado
AGENDA

- Quick introduction to (software) debugging.
- Debugging tools and techniques (applied to embedded Linux systems).
  - Log/dump analysis.
  - Tracing.
  - Interactive debugging.
  - Debugging frameworks.
- Lot's of hands-on (if we have time)!
THE DEBUGGING PROCESS

• In general, debugging is the process of identifying and removing bugs (errors) from hardware or software.

• Debugging a software problem might involve the following steps:
  ▪ Understand the problem.
  ▪ Reproduce the problem.
  ▪ Identify the root cause.
  ▪ Apply the fix.
  ▪ Test it. If fixed, celebrate! If not, go back to step 1.
THE 5 TYPES OF PROBLEMS

- We might classify software problems in 5 major categories:
  - Crash.
  - Lockup/Hang.
  - Logic/implementation.
  - Resource leakage.
  - (Lack of) performance.
TOOLS AND TECHNIQUES

• We might try to solve those problems using one or more of these 5 tools or techniques:
  ▪ Our brain (aka knowledge).
  ▪ Post mortem analysis (logging analysis, memory dump analysis, etc).
  ▪ Tracing/profiling (specialized logging).
  ▪ Interactive debugging (eg: GDB).
  ▪ Debugging frameworks (eg: Valgrind).
POST MORTEM ANALYSIS

• Post mortem analysis can be done via information exported by the system, including logs and memory dumps.
  
  - *Logs*: any (text or binary) information related to the execution of the system, collected and stored by the operating system (application execution, kernel operation, system errors, etc).
  
  - *Memory dump*: When an application crashes, the kernel is able to generate a special file called *core*, that contains a snapshot of the memory of the offending process and can be used to debug and find the root cause of the crash.

• Post mortem analysis can be very helpful when analyzing crashes and logic problems.
EXAMPLE: KERNEL CRASH

```c
1 [17.160336] Unable to handle kernel NULL pointer dereference at virtual address 00000000
2 [17.168531] pgd = 5df2196d
3 [17.171259] [00000000] *pgd=00000000
4 [17.174990] Internal error:Oops: 5 [#1] SMP ARM
5 [17.179622] Modules linked in:
6 [17.182686] CPU: 0 PID: 83 Comm: kworker/0:2 Not tainted 5.15.17-g85b8fc029a8d-dirty #2
7 [17.190700] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
8 [17.197232] Workqueue: usb_hub_wq hub_event
9 [17.201436] PC is at storage_probe+0x60/0x1a0
10 [17.205810] LR is at storage_probe+0x48/0x1a0
11 [17.210175] pc : [<c06a21cc>]  lr : [<c06a21b4>]  psr: 0000013
12 [17.216446] sp : c50239c0  ip : c50239c0  fp : c50239fc
13 [17.221674] r10: c53e2c00  r9 : c57c9a00  r8 : c0f60b4c
14 [17.226902] r7 : c53e2c00  r6 : c0a7d9fc  r5 : 00000001  r4 : c57c9a20
15 [17.233435] r3 : 00000000  r2 : 1ae1f000  r1 : c0a7d9fc  r0 : 00000000
16 [17.239968] Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment none
17 ...
18 [17.755646] Backtrace:
19 [17.758099] [<c06a21cc>] (storage_probe) from [<c0682f2c>] (usb_probe_interface+0xe4/0x29c)
20 [17.766489] [<c0682e48>] (usb_probe_interface) from [<c05db4f8>] (really_probe.part.0+0xac/0x33c
21 [17.775384] r10:c0f5ff48 r9:00000000 r8:00000008 r7:c57c9a20 r6:c0f60b4c r5:00000000
22 ...
```
EXAMPLE: KERNEL CRASH (CONT.)

```
1 $ cd <linux_source_code>
2 $ ls
3 arch Documentation Kbuild Makefile samples tools
4 block drivers Kconfig mm scripts usr
5 certs fs kernel modules.builtin security virt
6 COPYING include lib modules.builtin.modinfo sound vmlinux
7 CREDITS init LICENSES net System.map vmlinux.o
8 crypto ipc MAINTAINERS README tags vmlinux.symvers
9
10 $ file vmlinux
11 vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]
12 ca2de68ea4e39ca0f11e688a5e9ff0002a9b7733, with debug_info, not stripped
```
EXAMPLE: KERNEL CRASH (CONT.)

```c
1 $ arm-linux-addr2line -f -p -e vmlinux 0xc06a21cc
2 storage_probe at /opt/labs/ex/linux/drivers/usb/storage/usb.c:1118
3 $ arm-linux-gdb vmlinux
4 (gdb) list *(storage_probe+0x60)
5 0xc06a21cc is in storage_probe (drivers/usb/storage/usb.c:1118).
6 $ arm-linux-addr2line -f -p -e vmlinux 0xc06a21cc
7 $ arm-linux-gdb vmlinux
8   
9   (gdb) list *(storage_probe+0x60)
10    /*
11      if (usb_usual_ignore_device(intf))
12         return -ENXIO;
13   */
14 1113  /* Print vendor and product name */
15 1114  v = (char *)unusual_dev->vendorName;
16 1115  p = (char *)unusual_dev->productName;
17 1116  if (v && p)
18 1117     dev_dbg(&intf->dev, "vendor=%s product=%s\n", v, p);
19
```
EXAMPLE: USER SPACE CRASH

```bash
1 # fping -c 3 192.168.0.1
2 Segmentation fault
3 # ulimit -c unlimited
4 # fping -c 3 192.168.0.1
5 Segmentation fault (core dumped)
6
7 # ls -la core
8 -rw------- 1 root root 380928 May 25 2022 core
9
10 # file core
11 core: ELF 32-bit LSB core file, ARM, version 1 (SYSV), SVR4-style, from 'fping -c 3 192.168.0.1',
12 real uid: 0, effective uid: 0, real gid: 0, effective gid: 0, execfn: '/usr/sbin/fping',
13 platform: 'v7l'
14
15 # cat /proc/sys/kernel/core_pattern
16 /root/core
```
EXAMPLE: USER SPACE CRASH (CONT.)

1 $ cd <fping_source_code>
2 $ ls
3 aclocal.m4  config.guess  config.status  contrib  INSTALL  Makefile.in  stamp-h1
4 CHANGELOG.md  config.h  config.sub  COPYING  install-sh  missing
5 ci  config.h.in  configure  depcomp  Makefile  README.md
6 compile  config.log  configure.ac  doc  Makefile.am  src
7
8 $ file src/fping
9 src/fping: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
10 interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 5.15.0, with debug_info, not stripped
11
12 $ file core
13 core: ELF 32-bit LSB core file, ARM, version 1 (SYSV), SVR4-style, from 'fping -c 3 192.168.0.1',
14 real uid: 0, effective uid: 0, real gid: 0, effective gid: 0, execfn: '/usr/sbin/fping',
15 platform: 'v7l'
EXAMPLE: USER SPACE CRASH (CONT.)

```c
$ arm-linux-gdb src/fping -c core
... Core was generated by 'fping -c 3 192.168.0.1'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0 optparse_long (options=0xbe8e8914, longopts=0xbe8e89f8, longindex=0x0) at optparse.c:217
217     char *option = options->argv[options->optind];
7
8 (gdb) list
9 212     int
10 213     optparse_long(struct optparse *options,
11 214         const struct optparse_long *longopts,
12 215             int *longindex)
13 216     {
14 217     char *option = options->argv[options->optind];
15 218     if (option == 0) {
16 219             return -1;
17 220     } else if (is_dashdash(option)) {
18 221             options->optind++; /* consume "--" */
```
EXAMPLE: USER SPACE CRASH (CONT.)

```c
1 (gdb) p options
2 $1 = (struct optparse *) 0xbe8e8914
3
4 (gdb) p options->argv
5 $3 = (char **) 0x0
6
7 (gdb) up
8 #1 0x0042278c in main (argc=4, argv=0xbe8e8e54) at fping.c:509
9 509 while ((c = optparse_long(&optparse_state, longopts, NULL)) != EOF) {
10
11 (gdb) p optparse_state
12 $4 = {
13    argv = 0x0,
14    permute = 1,
15    optind = 1,
16    optopt = 0,
17    optarg = 0x0,
18    errmsg = '\000' <repeats 63 times>,
19    subopt = 0
20 }
```
TRACING

- Tracing is a specialized form of logging, where data about the state and execution of a program (or the kernel) is collected and stored for runtime (or later) analysis.

- It's implemented via static and dynamic tracepoints (probes) injected in the code to instrument the software at runtime.

- Tracing can be used for debugging purposes and also for latency and performance analysis (profiling).

- Tracing tools can be especially helpful with lockup issues and performance/latency analysis.
EXAMPLE: KERNEL TRACING

```bash
# time echo 1 > /sys/class/leds/ipe:red:ld1/brightness
real    0m 4.04s
user    0m 0.00s
sys     0m 0.00s

# zcat /proc/config.gz | grep TRACER=y
CONFIG_NOP_TRACER=y
CONFIG_HAVE_FUNCTION_TRACER=y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=y
CONFIG_CONTEXT_SWITCH_TRACER=y
CONFIG_GENERIC_TRACER=y
CONFIG_FUNCTION_TRACER=y
CONFIG_FUNCTION_GRAPH_TRACER=y
CONFIG_STACK_TRACER=y
CONFIG_IRQSOFF_TRACER=y
CONFIG_SCHED_TRACER=y
CONFIG_HWLAT_TRACER=y
CONFIG_OSNOISE_TRACER=y
CONFIG_TIMERLAT_TRACER=y
```
EXAMPLE: KERNEL TRACING (CONT.)

```
# mount -t tracefs tracefs /sys/kernel/tracing/

# trace-cmd record -p function_graph -F echo 1 > /sys/class/leds/ipe:red:ld1/brightness
plugin 'function_graph'
CPU0 data recorded at offset=0x2f0000
1421312 bytes in size
CPU1 data recorded at offset=0x44b000
217088 bytes in size

# ls -l trace.dat
-rw-r--r--    1 root     root       4718592 May 26  2022 trace.dat
```
EXAMPLE: KERNEL TRACING (CONT.)

```c
# trace-cmd report > trace.log
# cat trace.log
...  
echo-232 [000] 373.132044: funcgraph_entry:  
  led_set_brightness() {
  led_set_brightness_nosleep() {
    gpio_led_set() {
      msleep() {
        __msecs_to_jiffies();
        schedule_timeout() {
          lock_timer_base();
        }
      }
    }
  }
}
...  
echo-232 [000] 373.132070: funcgraph_entry:  
  _raw_spin_lock_irqsa
...  
echo-232 [000] 373.132044: funcgraph_entry:  
  led_set_brightness() {
    gpio_led_set() {
      msleep() {
        __msecs_to_jiffies();
        schedule_timeout() {
          lock_timer_base();
        }
      }
    }
  }
...  
echo-232 [000] 377.194984: funcgraph_entry:  
  _raw_spin_unlock_irq
```
EXAMPLE: KERNEL TRACING (CONT.)
EXAMPLE: USER SPACE TRACING

```c
1  # netcat -l -p 1234
2  Error: Couldn't setup listening socket (err=-3)
3  # strace netcat -l -p 1234
4  ...
5  read(3, "# /etc/services:\n# $Id: services"
6  ..., 4096) = 4096
7  read(3, "# /etc/services:\n# $Id: services"
8  ... , 4096) = 4096
9  read(3, "inding Protocol\necho		4/ddp				#"
10  ..., 4096) = 2681
11  read(3, "", 4096) = 0
12  close(3) = 0
13  socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
14  setsockopt(3, SOL_SOCKET, SO_LINGER, \{l_onoff=1, l_linger=0\}, 8) = 0
15  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, \{1\}, 4) = 0
16  bind(3, NULL, 16) = -1 EFAULT (Bad address)
17  close(3) = 0
18  write(2, "Error: Couldn't setup listening ...", 48)
19  Error: Couldn't setup listening socket (err=-3)
20  ) = 48
21  exit_group(1) = ?
22  +++ exited with 1 +++
```
EXAMPLE: USER SPACE TRACING (CONT.)

```bash
# ethtool eth0
Settings for eth0:

# zcat /proc/config.gz | grep CONFIG_UPROBE
CONFIG_UPROBES=y
CONFIG_UPROBE_EVENTS=y

# file /usr/sbin/ethtool
/usr/sbin/ethtool: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 5.15.0, with debug_info, not
stripped
```
EXAMPLE: USER SPACE TRACING (CONT.)

```bash
# for f in `perf probe -F -x /usr/sbin/ethtool'; \
  do perf probe -q -x /usr/sbin/ethtool $f; done

# perf probe -l | tee
probe_ethtool:altera_tse_dump_regs (on altera_tse_dump_regs@build/ethtool-5.12/tse.c in /usr/sb
probe_ethtool:amd8111e_dump_regs (on amd8111e_dump_regs@build/ethtool-5.12/amd8111e.c in /usr/s
probe_ethtool:at76c50x_usb_dump_regs (on at76c50x_usb_dump_regs@ethtool-5.12/at76c50x-usb.c in
...

# perf record -e probe_ethtool:* -aR -- /usr/sbin/ethtool eth0
Could't synthesize bpf events.
Settings for eth0:
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.084 MB perf.data (185 samples) ]

# ls -l perf.data
-rw------- 1 root root 308153 May 26 2022 perf.data
```
EXAMPLE: USER SPACE TRACING (CONT.)

```bash
# perf script | tee
...
3  ethtool 812 [000] 4908.289466:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
4  ethtool 812 [000] 4908.289493:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
5  ethtool 812 [000] 4908.289528:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
6  ethtool 812 [000] 4908.289546:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
7  ethtool 812 [000] 4908.289573:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
8  ethtool 812 [000] 4908.289600:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
9  ethtool 812 [000] 4908.289626:  probe_ethtool:ethtool_link_mode_set_bit: (4a4bc0)
10 ethtool 812 [000] 4908.289660:  probe_ethtool:find_option: (4b5014)
11 ethtool 812 [000] 4908.289719:  probe_ethtool:netlink_run_handler: (4a4c3c)
12 ethtool 812 [000] 4908.289750:    probe_ethtool:ioctl_init: (4b5e50)
13 ethtool 812 [000] 4908.289849:    probe_ethtool:do_gset: (4ac63c)
15 ethtool 812 [000] 4908.290492:    probe_ethtool:send_ioctl: (4b4cec)
16 ethtool 812 [000] 4908.290544:    probe_ethtool:send_ioctl: (4b4cec)
17 ethtool 812 [000] 4908.290596:    probe_ethtool:dump_link_usettings: (4a6520)
18 ethtool 812 [000] 4908.290628:    probe_ethtool:dump_supported: (4a5f3c)
```
INTERACTIVE DEBUGGING

• An interactive debugging tool allows us to interact with the application at runtime.

• This kind of tool makes it possible to execute the code step-by-step, set breakpoints, display information (variables, stack, etc), list function call history (backtrace), etc.

• On Linux systems, the most used interactive debugging tool is GDB. https://www.sourceware.org/gdb/

• An interactive debug tool can especially help with crashes, lockups and logic problems.
EXAMPLE: KERNEL DEBUGGING WITH GDB

```plaintext
1 # echo heartbeat > /sys/class/leds/ipe:red:ld1/trigger
2 # zcat /proc/config.gz | grep ^CONFIG_KGDB
3 CONFIG_KGDB=y
4 CONFIG_KGDB_HONOUR_BLOCKLIST=y
5 CONFIG_KGDB_SERIAL_CONSOLE=y
6 # echo ttymxc0 > /sys/module/kgdboc/parameters/kgdboc
7 [ 6794.040785] KGDB: Registered I/O driver kgdboc
8 # echo g > /proc/sysrq-trigger
9 [ 6797.741657] sysrq: DEBUG
10 [ 6797.744216] KGDB: Entering KGDB
11 # echo g > /proc/sysrq-trigger
12 [ 6797.741657] sysrq: DEBUG
13 [ 6797.744216] KGDB: Entering KGDB
```
EXAMPLE: KERNEL DEBUGGING WITH GDB (CONT.)

```
$ cd <linux_source_code>
$file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]
c2d68ea4e39ca0f11e688a5e9ff0002a9b7733, with debug_info, not stripped
$ arm-linux-gdb vmlinux -tui
(gdb) target remote localhost:5551
Remote debugging using localhost:5551
[Switching to Thread 4294967294]
arch_kgdb_breakpoint () at ./arch/arm/include/asm/kgdb.h:46
(gdb) b led_trigger_write
Breakpoint 1 at 0xc074fbb4: file drivers/leds/led-triggers.c, line 39.
(gdb) cont
```
EXAMPLE: KERNEL DEBUGGING WITH GDB (CONT.)
EXAMPLE: USER SPACE DEBUGGING WITH GDB

```
# tree /var
/var
<hanging>
# gdbserver :1234 tree /var
Process tree created; pid = 834
Listening on port 1234
```
EXAMPLE: USER SPACE DEBUGGING WITH GDB

$ cd <tree_source_code>
$ ls
CHANGES  doc  hash.c  html.o  json.o  README  tree  tree.o  xml.c
color.c  file.c  hash.o  INSTALL  LICENSE  strverscmp.c  tree.c  unix.c  xml.o
color.o  file.o  html.c  json.c  Makefile  TODO  tree.h  unix.o
$ file tree
tree: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 5.15.0, with debug_info, not stripped
$ arm-linux-gdb tree -tui
Remote debugging using 192.168.0.2:1234
Reading symbols from /opt/labs/ex/buildroot/output/host/arm-buildroot-linux-gnueabihf/sysroot/lib/libc.so.6... done.
0xb6f388c0 in _start () from /opt/labs/ex/buildroot/output/host/arm-buildroot-linux-gnueabihf/sysroot/lib/libc.so.6
(gdb) cont
Continuing
<CTRL-C>
EXAMPLE: USER SPACE DEBUGGING WITH GDB
There are a number of support tools and frameworks that can help with debugging Linux systems.

The Linux kernel has several debugging frameworks to identify memory leaks, lockups, etc (see the "Kernel Hacking" configuration menu).

In user space, a very known framework is Valgrind, which provides an infrastructure for creating memory debugging tools (memory leak, race condition, profiling, etc). [https://valgrind.org/](https://valgrind.org/)

Debugging frameworks can be very useful when analysing resource leaks and lockups.
EXAMPLE: DEBUGGING KERNEL HANGS

```bash
1 # cat /proc/uptime
2 <hanging>
3
4 # zcat /proc/config.gz | grep "CONFIG_SOFTLOCKUP_DETECTOR|CONFIG_DETECT_HUNG_TASK"
5 CONFIG_SOFTLOCKUP_DETECTOR=y
6 CONFIG_DETECT_HUNG_TASK=y
7
8 # cat /proc/uptime
9 <wait for a few seconds>
```
EXAMPLE: DEBUGGING KERNEL HANGS (CONT.)

```
watchdog: BUG: soft lockup - CPU#1 stuck for 45s! [cat:209]
Modules linked in:
CPU: 1 PID: 209 Comm: cat Not tainted 5.15.17-g85b8fc029a8d-dirty #2
Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
PC is at uptime_proc_show+0x134/0x15c
LR is at vsnprintf+0x28c/0x42c
Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment none
Watchdog: BUG: soft lockup - CPU #1 stuck for 45s! [cat:209]
Backtrace:
```

```c
EXAMPLE: DEBUGGING KERNEL HANGS (CONT.)
```

```
Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
CPU: 1 PID: 209 Comm: cat Not tainted 5.15.17-g85b8fc029a8d-dirty #2
CPU: 1 PID: 209 Comm: cat Not tainted 5.15.17-g85b8fc029a8d-dirty #2
Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
Backtrace:
```

```
```
EXAMPLE: DEBUGGING KERNEL HANGS (CONT.)

```bash
$ cd <linux_source_code>
$ ls
arch     Documentation  Kbuild       Makefile                 samples     tools
block    drivers        Kconfig      mm                       scripts     usr
certs    fs             kernel       modules.builtin          security    virt
COPYING  include        lib          modules.builtin.modinfo  sound       vmlinux
CREDITS  init           LICENSES     net                      System.map  vmlinux.o
crypto   ipc            MAINTAINERS  README                   tags        vmlinux.symvers
$ file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]
  c2de68ea4e39ca0f11e688a5e9ff0002a9b7733, with debug_info, not stripped
```
EXAMPLE: DEBUGGING KERNEL HANGS (CONT.)

```c
static int __init proc_uptime_init(void)
{
    while(1);

    seq_printf(m, "%lu.%02lu %lu.%02lu\n",
        (unsigned long) idle.tv_sec, (unsigned long) idle.tv_nsec / (NSEC_PER_SEC / 100)),
        (unsigned long) uptime.tv_sec, (uptime.tv_nsec / (NSEC_PER_SEC / 100)));

    return 0;
}
```

1 $ arm-linux-addr2line -f -p -e vmlinux 0xc037337c
2 uptime_proc_show at /opt/labs/ex/linux/fs/proc/uptime.c:37
3
4 $ arm-linux-gdb vmlinux
5
6 (gdb) list *(uptime_proc_show+0x134)
7 0xc037337c is in uptime_proc_show (fs/proc/uptime.c:37).
8 32               seq_printf(m, "%lu.%02lu %lu.%02lu\n",
9 33                               (unsigned long) uptime.tv_sec, (uptime.tv_nsec / (NSEC_PER_SEC / 100)),
10 34                               (unsigned long) idle.tv_sec, (idle.tv_nsec / (NSEC_PER_SEC / 100)));
11 35               while(1);
12 36               return 0;
13 37       }
14 38
15 39
16 40
17 41 static int __init proc_uptime_init(void)
```
EXAMPLE: MEMORY LEAKS IN USER SPACE

```bash
# cpuload

Time   CPU    total   nice   user   system   irq   softirq   iowait   steal   guest  
0      CPU   5.9    0.0    0.2    5.2     0.0    0.5     0.3     0.0    0.0
1      CPU   0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.0    0.0
2      CPU   0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.0    0.0
3      CPU   0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.0    0.0
4      CPU   0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.0    0.0
5      CPU   0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.0    0.0
6      CPU   0.0    0.0    0.0    0.0     0.0    0.0     0.0     0.0    0.0
7      <memory is leaking>
8      
9      # ls -l /usr/bin/valgrind
10     -rwxr-xr-x  1 root  root  25900 May 24  2022 /usr/bin/valgrind
11     
12     # file /usr/bin/cpuload
13     /usr/bin/cpuload: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 5.15.0, with debug_info, not stripped
```
EXAMPLE: MEMORY LEAKS IN USER SPACE (CONT.)

```plaintext
1 # valgrind --leak-check=full /usr/bin/cpuload
2 ==212== Memcheck, a memory error detector
3 ==212== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
4 ==212== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
5 ==212== Command: /usr/bin/cpuload
6 ==212==
7 Time   CPU   total   nice   user   system   irq   softirq   iowait   steal   guest
8 0     CPU   5.9    0.0    0.2    5.2     0.0   0.5      0.3     0.0    0.0
9 1     CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
10 2    CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
11 3    CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
12 4    CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
13 5    CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
14 6    CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
15 7    CPU   0.0    0.0    0.0    0.0     0.0   0.0      0.0     0.0    0.0
16 <CTRL-C>
```
EXAMPLE: MEMORY LEAKS IN USER SPACE (CONT.)

```bash
1 ==212== Process terminating with default action of signal 2 (SIGINT)
2 ==212== at 0x492491C: pause (in /lib/libc.so.6)
3 ==212== by 0x10ACFB: main (cpu_load.c:193)
4 ==212== HEAP SUMMARY:
5 ==212== in use at exit: 52,964 bytes in 14 blocks
6 ==212== total heap usage: 34 allocs, 20 frees, 66,324 bytes allocated
7 ==212== 36,864 bytes in 9 blocks are definitely lost in loss record 6 of 6
8 ==212== at 0x484EF68: malloc (vg_replace_malloc.c:381)
9 ==212== by 0x10A727: print_cpu_load (cpu_load.c:79)
10 ==212== by 0x10B177: do_stat (cpu_load.c:244)
11 ==212== by 0x48A888F: ??? (in /lib/libc.so.6)
12 ==212== LEAK SUMMARY:
13 ==212== definitely lost: 36,864 bytes in 9 blocks
14 ==212== indirectly lost: 0 bytes in 0 blocks
15 ==212== possibly lost: 0 bytes in 0 blocks
16 ==212== still reachable: 16,100 bytes in 5 blocks
17 ==212== suppressed: 0 bytes in 0 blocks
18 ==212== Reachable blocks (those to which a pointer was found) are not shown.
19 ==212== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```
# Problems vs Techniques (1)

<table>
<thead>
<tr>
<th></th>
<th>Crash</th>
<th>Lockup</th>
<th>Logic</th>
<th>Leak</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
<td>😊😊😊😊😊</td>
</tr>
</tbody>
</table>
# PROBLEMS VS TECHNIQUES (2)

<table>
<thead>
<tr>
<th></th>
<th>Crash</th>
<th>Lockup</th>
<th>Logic</th>
<th>Leak</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>print()</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
</tr>
<tr>
<td>print()</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
<td>😊😊😊</td>
</tr>
</tbody>
</table>
### PROBLEMS VS TECHNIQUES (3)

<table>
<thead>
<tr>
<th></th>
<th>Crash</th>
<th>Lockup</th>
<th>Logic</th>
<th>Leak</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge</td>
<td>😊</td>
<td>😊</td>
<td>😊</td>
<td>😊</td>
<td>😊</td>
</tr>
<tr>
<td>Post mortem</td>
<td>😊</td>
<td>😞</td>
<td>😊</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td>Tracing</td>
<td>😞</td>
<td>😊</td>
<td>😞</td>
<td>😞</td>
<td>😊</td>
</tr>
<tr>
<td>Interactive debugging</td>
<td>😊</td>
<td>😊</td>
<td>😊</td>
<td>😞</td>
<td>😞</td>
</tr>
<tr>
<td>Debugging frameworks</td>
<td>😞</td>
<td>😊</td>
<td>😞</td>
<td>😊</td>
<td>😞</td>
</tr>
</tbody>
</table>
LF LIVE: MENTORSHIP SERIES

THANK YOU! QUESTIONS?

Sergio Prado, Embedded Labworks
sergio.prado@e-labworks.com
https://www.linkedin.com/in/sprado
https://sergioprado.blog