EMBEDDED LINUX CONFERENCE 2022

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 10+ years.
  https://e-labworks.com/en

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

- Blogger at EmbeddedBits.org.
  https://embeddedbits.org/
AGENDA

• Introduction to (software) debugging.
• Debugging tools and techniques (applied to embedded systems based on Linux).
  ▪ Log/dump analysis.
  ▪ Tracing.
  ▪ Interactive debugging.
  ▪ Debugging frameworks.
• Lot's of hands-on (hopefully)!
THE SIX STAGES OF DEBUGGING

- *Denial*: That can't happen.
- *Frustration*: That doesn't happen on my machine.
- *Disbelief*: That shouldn't happen.
- *Testing*: Why does that happen?
- *Confirmation*: Oh, I see.
- *Relief*: How did that ever work?
WHAT IS DEBUGGING?

• In the somewhat distant past, we started using "bug" in the engineering jargon to describe hardware and software errors.

• So debugging is the process of removing bugs from hardware and software designs.

• A wise old man once said: "In the software development process, we spend 50% debugging the software, and the other 50% bugging"!
DEBUGGING STEP-BY-STEP

• Debugging a software problem might involve the following steps:
  ■ Understanding the problem.
  ■ Reproducing the problem.
  ■ Identifying the root cause.
  ■ Fixing the problem.
  ■ Fixed? If so, celebrate! If not, go back to step 1.
THE 5 TYPES OF PROBLEMS

- We might classify software problems in 5 main 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**

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 [ 17.179022] Modules linked in:
6 [ 17.182886] 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 [ 17.204436] 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: 60000013
12 [ 17.216446] sp : c50239c0    ip : c50239fc    fp : c50239fc
13 [ 17.221674] r10: c53e2c00    r9 : c57c9a00    r8 : c0f60b4c
14 [ 17.226902] r7 : c53e2c80    r6 : c0a7df9c    r5 : 0000001   r4 : c57c9a20
15 [ 17.233435] r3 : 00000000    r2 : 1ae1f000    r1 : c0a7df9c    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] [<c06a21b4>]  (storage_probe) from [<c06b2f2c>]  (usb_probe_interface+0xe4/0x29c)
20 [ 17.766488] [<c06b2e48>]  (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.)

```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]
c2ade68ea4e39ca0f11e688a5e9ff002a9b7733, 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 */
7 1114 if (usb_usual_ignore_device(intf))
8 1115 return -ENXIO;
9 1116 /* Print vendor and product name */
10 1117 v = (char *)unusual_dev->vendorName;
11 1118 p = (char *)unusual_dev->productName;
12 1119 if (v && p)
13 1120 dev_dbg(&intf->dev, "vendor=%s product=%s\n", v, p);
```
EXAMPLE: USER SPACE CRASH

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

```bash
$ cd <fping_source_code>
$ ls
aclocal.m4  config.guess  config.status  contrib  INSTALL  Makefile.in  stamp-h1
CHANGELOG.md  config.h  config.sub  COPYING  install-sh  missing
ci  config.h.in  configure  depcomp  Makefile  README.md
compile  config.log  configure.ac  doc  Makefile.am  src
$ file src/fping
src/fping: 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
$ file core
core: ELF 32-bit LSB core file, ARM, version 1 (SYSV), SVR4-style, from 'fping -c 3 192.168.0.1',
real uid: 0, effective uid: 0, real gid: 0, effective gid: 0, execfn: '/usr/sbin/fping',
platform: 'v7l'
```
EXAMPLE: USER SPACE CRASH (CONT.)

```c
1 $ arm-linux-gdb src/fping -c core
2 ...
3 Core was generated by `fping -c 3 192.168.0.1'.
4 Program terminated with signal SIGSEGV, Segmentation fault.
5 #0 optparse_long(options=0xbe8e8914, longopts=0xbe8e89f8, longindex=0x0) at optparse.c:217
6 217 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 analysis.
EXAMPLE: KERNEL TRACING

1 # time echo 1 > /sys/class/leds/ipe:red:ld1/brightness
2 real    0m 4.04s
3 user    0m 0.00s
4 sys     0m 0.00s
5
6 # zcat /proc/config.gz | grep TRACER=y
7 CONFIG_NOP_TRACER=y
8 CONFIG_HAVE_FUNCTION_TRACER=y
9 CONFIG_HAVE_FUNCTION_GRAPH_TRACER=y
10 CONFIG_CONTEXT_SWITCH_TRACER=y
11 CONFIG_GENERIC_TRACER=y
12 CONFIG_FUNCTION_TRACER=y
13 CONFIG_FUNCTION_GRAPH_TRACER=y
14 CONFIG_STACK_TRACER=y
15 CONFIG_IRQSOFF_TRACER=y
16 CONFIG_SCHED_TRACER=y
17 CONFIG_HWLAT_TRACER=y
18 CONFIG_OSN OISE TRACER=y
19 CONFIG_TIMERLAT_TRACER=y
EXAMPLE: KERNEL TRACING (CONT.)

```bash
# 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

...  
5  echo-232 [000] 373.132044: funcgraph_entry:
6  echo-232 [000] 373.132047: funcgraph_entry:
7  echo-232 [000] 373.132050: funcgraph_entry:
8  echo-232 [000] 373.132053: funcgraph_entry:
9  echo-232 [000] 373.132056: funcgraph_entry:  2.667 us
10 echo-232 [000] 373.132062: funcgraph_entry:  2.333 us
11 echo-232 [000] 373.132066: funcgraph_entry:  2.667 us
12 echo-232 [000] 373.132070: funcgraph_entry:  3.334 us
13 echo-232 [000] 373.132074: funcgraph_exit:  8.667 us
14 echo-232 [000] 373.132077: funcgraph_entry:
15 echo-232 [000] 373.132082: funcgraph_entry:
16 echo-232 [000] 373.132086: funcgraph_entry:
17 echo-232 [000] 373.132090: funcgraph_entry:
18 ...  
19 echo-232 [000] 377.194984: funcgraph_entry:  2.666 us
20 echo-232 [000] 377.194999: funcgraph_exit:  + 23.000 us
21 echo-232 [000] 377.194993: funcgraph_exit:  + 4062931 us
22 echo-232 [000] 377.194996: funcgraph_exit:  + 4062943 us
```
EXAMPLE: KERNEL TRACING (CONT.)

![Kernel Shark screenshot](image-url)
EXAMPLE: USER SPACE TRACING

```plaintext
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:
6  read(3, "# /etc/services:
7  read(3, "# /etc/services:
8  read(3, "# /etc/services:
9  read(3, "# /etc/services:
10  close(3) = 0
11  socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
12  setsockopt(3, SOL_SOCKET, SO_LINGER, {l_onoff=1, l linger=0}, 8) = 0
13  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
14  bind(3, NULL, 16) = -1 EFAULT (Bad address)
15  close(3) = 0
16  write(2, "Error: Couldn't setup listening ", 48) = 48
17  exit_group(1) = ?
18  +++ exited with 1 +++
```
EXAMPLE: USER SPACE TRACING (CONT.)

```bash
# ethtool eth0
Settings for eth0:  
<hanging>

# 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
probes: `altera_tse_dump_regs (on altera_tse_dump_regs@build/ethtool-5.12/tse.c in /usr/sbin/ethtool)`
probes: `at76c50x_usb_dump_regs (on at76c50x_usb_dump_regs@build/ethtool-5.12/at76c50x-usb.c in /usr/sbin/ethtool)`

# perf record -e probe_ethtool:* -aR -- /usr/sbin/ethtool eth0

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

# ls -l perf.data
-rw------- 1 root root 308153 May 26 2022 perf.data
```

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.084 MB perf.data (185 samples) ]
```
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)
14 ethtool 812 [000] 4908.290452: probe_ethtool:do_ioctl_linksettings: (4ad68)
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

```bash
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 q > /proc/sysrq-trigger
12 [ 6797.741657] sysrq: DEBUG
13 [ 6797.744216] KGDB: Entering KGDB
```
EXAMPLE: KERNEL DEBUGGING WITH GDB (CONT.)

```bash
$ cd <linux_source_code>
$ file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]
ca2de68ea4e39ca0f11e688a5e9ff0002a9b7733, 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

1 $ cd <tree_source_code>
2 $ ls
3   CHANGES  doc  hash.c  html.o  json.o  README  tree  tree.o  xml.c
4 color.c  file.c  hash.o  INSTALL  LICENSE  strverscmp.c  tree.c  unix.c  xml.o
5 color.o  file.o  html.c  json.c  Makefile  TODO  tree.h  unix.o  xml.o
6
7 $ file tree
8 tree: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
9 interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 5.15.0, with debug_info, not stripped
10
11 $ arm-linux-gdb tree -tui
12
13 (gdb) target remote 192.168.0.2:1234
14 Remote debugging using 192.168.0.2:1234
15 Reading symbols from /opt/labs/ex/buildroot/output/host/arm-buildroot-linux-gnueabihf/sysroot/lib
16 0xb6f388c0 in _start () from /opt/labs/ex/buildroot/output/host/arm-buildroot-linux-gnueabihf/sys
17
18 (gdb) cont
19 Continuing
20 <CTRL-C>
EXAMPLE: USER SPACE DEBUGGING WITH GDB
DEBUGGING FRAMEWORKS

• There are a number of support tools and frameworks that can help with debugging Linux systems.

• A classic example is Valgrind, which provides a framework for creating memory debugging tools (memory leak, race condition, profiling, etc).
  https://valgrind.org/

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

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.)

```
1 [ 2604.004290] watchdog: BUG: soft lockup - CPU#1 stuck for 45s! [cat:209]
2 [ 2604.010927] Modules linked in:
3 [ 2604.013991] CPU: 1 PID: 209 Comm: cat Not tainted 5.15.17-g85b8fc029a8d-dirty #2
4 [ 2604.021399] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
5 [ 2604.027931] PC is at uptime_proc_show+0x134/0x15c
6 [ 2604.032651] LR is at vsnprintf+0x28c/0x42c
7 [ 2604.036760] pc: [<c037337c>] lr: [<c0528660>] psr: 600f0013
8 [ 2604.043031] sp: c5103c90 ip: c5103c08 fp: c5103d34
9 [ 2604.048260] r10: f87aa400 r9: 89705f41 r8: 36b4a597
10 [ 2604.053488] r7: 0000027d r6: 4b14b59a r5: 000004a3 r4: 00000000
11 [ 2604.060019] r3: 82889af3 r2: 82889af3 r1: 00000010 r0: 00000010
12 [ 2604.066552] Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment none
13 [ 2604.073696] Control: 10c5387d Table: 158ac04a DAC: 00000051
14 [ 2604.079446] CPU: 1 PID: 209 Comm: cat Not tainted 5.15.17-g85b8fc029a8d-dirty #2
15 [ 2604.086851] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
16 [ 2604.093382] Backtrace:
17 ...
18 [ 2604.285229] [<c0373248>] (uptime_proc_show) from [<c03671b4>] (seq_read_iter+0x9c/0xe4)
19 [ 2604.293433] r10:00400cc0 r9:00000001 r8:c5103db8 r7:c5910018 r6:00000000 r5:00000000
20 [ 2604.301268] r4:c5910000
21 [ 2604.303803] [<c03671b4>] (seq_read_iter) from [<c03671b4>] (proc_reg_read_iter+0x9c/0xe4)
22 ...
```
EXAMPLE: DEBUGGING KERNEL HANGS (CONT.)

$ 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]
c2de68ea439ca0f11e688a5e9ff0002a9b7733, with debug_info, not stripped
EXAMPLE: DEBUGGING KERNEL HANGS (CONT.)

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

$ arm-linux-addr2line -f -p -e vmlinux 0xc037337c
upptime_proc_show at /opt/labs/ex/linux/fs/proc/uptime.c:37
$ arm-linux-gdb vmlinux
(gdb) list *(uptime_proc_show+0x134)
0xc037337c is in uptime_proc_show (fs/proc/uptime.c:37).
(seq_printf(m, "%ld. %02ld %ld. %02ld\n",
(u(unsigned long) uptime.tv_sec,
(u(unsigned long) idle.tv_sec,
(while(1);
) return 0;
static int __init proc_uptime_init(void)
```
EXAMPLE: MEMORY LEAKS IN USER SPACE

```
1  # cpuload
2  Time CPU total nice user system irq softirq iowait steal guest
3  0 CPU  5.9  0.0  0.2  5.2  0.0  0.5  0.3  0.0  0.0
4  1 CPU  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
5  2 CPU  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
6  3 CPU  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
7  ...  
8  <memory is leaking>
9
10 # ls -l /usr/bin/valgrind
11  -rwxr-xr-x  1 root root  25900 May 24  2022 /usr/bin/valgrind
12
13 # file /usr/bin/cpuload
14 /usr/bin/cpuload: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
15  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.)

```
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
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.)

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==
5 ==212== HEAP SUMMARY:
6 ==212==    in use at exit: 52,964 bytes in 14 blocks
7 ==212==    total heap usage: 34 allocs, 20 frees, 66,324 bytes allocated
8 ==212==
9 ==212==    36,864 bytes in 9 blocks are definitely lost in loss record 6 of 6
10 ==212==    at 0x484EF68: malloc (vg_replace_malloc.c:381)
11 ==212==    by 0x10A727: print_cpu_load (cpu_load.c:79)
12 ==212==    by 0x10B177: do_stat (cpu_load.c:244)
13 ==212==    by 0x48A888F: ??? (in /lib/libc.so.6)
14 ==212==
15 ==212== LEAK SUMMARY:
16 ==212==    definitely lost: 36,864 bytes in 9 blocks
17 ==212==    indirectly lost: 0 bytes in 0 blocks
18 ==212==    possibly lost: 0 bytes in 0 blocks
19 ==212==    still reachable: 16,100 bytes in 5 blocks
20 ==212==    suppressed: 0 bytes in 0 blocks
21 ==212== Reachable blocks (those to which a pointer was found) are not shown.
22 ==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><img src="image1.png" alt="Smiley" /></td>
<td><img src="image2.png" alt="Smiley" /></td>
<td><img src="image3.png" alt="Smiley" /></td>
<td><img src="image4.png" alt="Smiley" /></td>
<td><img src="image5.png" alt="Smiley" /></td>
</tr>
<tr>
<td>print()</td>
<td><img src="image6.png" alt="Smiley" /></td>
<td><img src="image7.png" alt="Smiley" /></td>
<td><img src="image8.png" alt="Smiley" /></td>
<td><img src="image9.png" alt="Smiley" /></td>
<td><img src="image10.png" alt="Smiley" /></td>
</tr>
<tr>
<td>print()</td>
<td><img src="image11.png" alt="Smiley" /></td>
<td><img src="image12.png" alt="Smiley" /></td>
<td><img src="image13.png" alt="Smiley" /></td>
<td><img src="image14.png" alt="Smiley" /></td>
<td><img src="image15.png" alt="Smiley" /></td>
</tr>
<tr>
<td>print()</td>
<td><img src="image16.png" alt="Smiley" /></td>
<td><img src="image17.png" alt="Smiley" /></td>
<td><img src="image18.png" alt="Smiley" /></td>
<td><img src="image19.png" alt="Smiley" /></td>
<td><img src="image20.png" alt="Smiley" /></td>
</tr>
<tr>
<td>print()</td>
<td><img src="image21.png" alt="Smiley" /></td>
<td><img src="image22.png" alt="Smiley" /></td>
<td><img src="image23.png" alt="Smiley" /></td>
<td><img src="image24.png" alt="Smiley" /></td>
<td><img src="image25.png" alt="Smiley" /></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>
THANK YOU! QUESTIONS?

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