University of western Brittany

Final Year Project Defense

Dive into practical Linux debugging

Debugging Linux OS

By :
Jugurtha BELKALEM

Tutor :
David GARIOU

Supervisor :
Jalil BOUKHOBZA

3 août 2018

Table of contents

1 Introduction

Embedded devices are increasingly popular, devices are becoming smaller, smarter, interactive striving for better user experience.

Such a success was made possible since those tiny devices rely on UNIX-like operating systems (**Linux is the dominant**).

- **Open Source :** Sources are available and maintained by a large community, the latest stable version is available at `https://www.kernel.org/`[1].
- **Not specific to vendor :** Linux is not propriatary operating system. We can point that major big companies are collaborators in it's developement.
- **Architecture support :** Linux supports many architectures (x86, arm, mips, ..., etc).
- **Low developement cost :** Linux is free.

However, such powerful operating systems are complex. Inconsistencies and logic flow errors can raise at any time (As the rule says : « More code, more error prone »), We need mechanisms that can scale efficiently to track issues and bugs during developement and maintenance. A variety of tools have been adopted (some are even built-in) that help developers to write more stable and efficient applications.

More can be said, as Linux is a multitasking and multiuser system. Every piece of code is checked for permissions. It does even distinguish between two distinct spaces : **userspace** and **kernel**. each has it's own operating privileges (**kernel does have all the privileges**) so they must be debugged differently.

1. The lastest version (not stable) is available at Linus github : sfdfd

2 Internship objectives

SMILE (**https ://www.smile.eu/en**) is the 1st integrator and European expert in open source solutions.

I'm part of ECS (*Embedded and Connected Systems*) division which builds software for innovative smart objects.

In order to offer the best experience for **SMILE**'s clients, We require :

- Test our solutions before production to detect faulty code and anticipate bugs.
- Troubleshoot errors that raise during production.
- Pointing-out sources of latencies (*disk, network, scheduler, ..., etc*), memory leaks, kernel panics and many more.
- Handle potential malicious code infections and being able to respond.

The pre-requests of the internship are :

* Good skills on C/C++ and Python.
* Working on Linux environment.
* Background electrical and electronics engineering concepts

The request document of the intership stressed out on experimenting and documenting the following points :

1. **Userspace debugging methodologies :** maily for C/C++ (using GDB, strace, ptrace, ltrace, valgrind)
2. **Kernel-land code debugging :** using KGDB/KDB, kernel oops, magic SysRQ, OpenOCD (with focus on it's syntax).
3. **Tracing and profiling :** to increase software quality, instrumentation must be used with tools like : **Ftrace (trace-cmd)**, **Perf** and **LTTng**.
   Those tracers must be compared to choose the appropriate one for a particular situation.
4. **Testing platforms :** known boards must be used (*Raspberry PI 3*, *Beagle bone black wireless* and *I.MX6*).
5. **Documentation :** providing step by step manual for every tool to be used by engineers at project's developement lifecycle and maintenance.

In short, the goal of the internship is to reduce Linux debugging time.

3 Available equipements

Debugging Linux is a challenging task which requires a good preparation.

3.1 Hardware

3.1.1 Beagle Bone Black Wireless

The evolution of beaglebone black which adds wireless support (WIFI, Bluetooth) and fast linux boot (**Figure 1**).

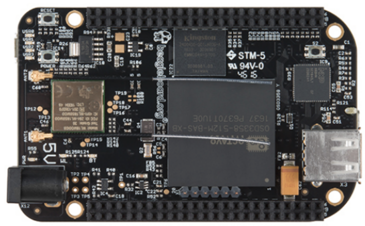


FIGURE 1 – Beaglebone black wireless

Hardware specifications A datasheet is available at : https://www.alliedelec.com/m/d/5505861ee370de1c82065dcc7bc77b0c.PDF.

3.1.2 Raspberry PI 3 B+

The lastest version as this time of writing with enhanced processor and ethernet speed (**Figure 2**).

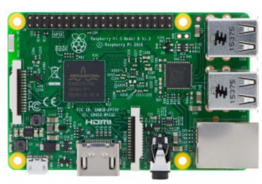


FIGURE 2 – Raspberry PI 3

Hardware specifications A datasheet is available at : https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf.

3.1.3 stm32f407 Board :

(**Figure 3**) specifications are available at : https://www.st.com/content/ccc/resource/technical/document/user_manual/70/fe/4a/3f/e7/e1/4f/7d/DM00039084.pdf/files/DM00039084.pdf/jcr:content/translations/en.DM00039084.pdf

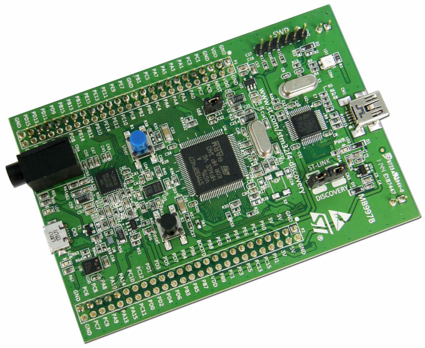


FIGURE 3 – stm32f407 Board

3.1.4 AT32UC3C-EK Board :

(**Figure 4**) see the following link for spefications : http://www.farnell.com/datasheets/1511964.pdf

FIGURE 4 – AT32UC3C Board

3.1.5 ARM-USB-TINY-H JTAG Adapter :

OpenOCD debugging interface adapter (see **Figure 5**).

FIGURE 5 – ARM-USB-TINY-H

https://www.olimex.com/Products/ARM/JTAG/_resources/ARM-USB-TINY_and_TINY_H_manual.pdf

3.2 Software

3.2.1 pycharm

Pycharm is a python IDE, which makes developement fast.

3.2.2 Eclipse C/C++

Code examples were written maily in C, Eclipse C/C++ was helpful.

4 Solutions

The following section discusses results of my intership. We are going to highlight the main points and illustrate with a couple of examples.

About report

This report gives only some samples of what was made, entire project can be accessed at : https://github.com/jugurthab/Linux_kernel_debug.
Full report (over 200 pages) is also available at : https://github.com/jugurthab/Linux_kernel_debug/blob/master/debugging-linux-kernel.pdf

4.1 Userspace

Understading userspace bottlenecks is an everyday's job for every software developer, performance and even security engineer. Most appreciated debugging mechanisms were gathered as shown in **Figure 6**.

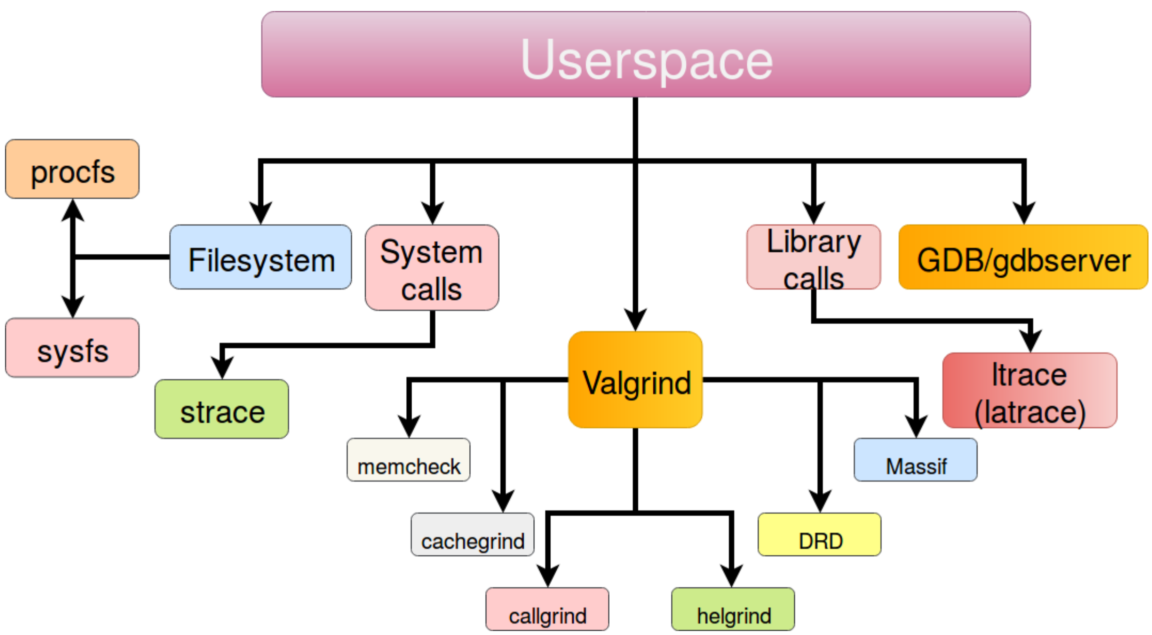


FIGURE 6 – Linux userspace debugging methodologies

Important : We are going to introduce each tool.

4.1.1 Querying the filesystem

Linux is enhanced in terms of security, robust and fault tolerant. It distinguishes between different level of privileges and mainly : a **userspace** and **kernel-land**. This allows the system to correctly handle the resources and prevent unauthorized accesses.

However, there are important data-structures and information that we require even in userspace (*memory allocated*, *available resources*, *state of process*, ..., etc). **Linux** provides us with two pseudo filesystems (*because they do not exist on disk, they are created during system's boot*) that allow the kernel to share some it's knowledge to the userspace.

— **ProcFs :** exposes information related to processes (*from which the name /proc stands for processes*) and system's configuration. Some interesting files for debugging :
  - **/proc/pid/maps :** displays virtual address space layout of a given process (*identified by pid*).
  - **/proc/pid/status :** returns process specific informations (*process status*, *attached debugger*, ..., etc).
  - **/proc/pid/limits :**

  Other files can be also helpful like : **/proc/meminfo** and **/proc/cpuinfo** which returns information associated to memory and processors respectively.

— **SysFs :** a more recent filesystem (*more organized than **procfs***), We will be concerned with folder **/sys/module** as it is required to debug modules (*as We will see later*).

4.1.2 System calls and library calls

Ptrace is the most valuable mechanism to debug userspace applications. Most of utilities that are covered later (*strace*, *ltrace* and *GDB*) rely on **ptrace** in the background (*without it they will be useless*).

However, attackers uses it extensively too to escalate privileges. Due to security issues, some distributions like **Ubuntu disables ptrace** by default, We must enable it as follow :

```
$ sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

— **System calls (Syscalls) :** strace is a debugging and diagnostic tool. The « s »stands for « system call », which means that strace can monitor Syscalls and reports them to end users.
An example is provided at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/strace

— **library calls :** Another successful tool is « ltrace »which can record calls made from a binary executable file to shared libraries. *It may save hours of debugging if used correctly.*
We have made an example at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/ltrace

ltrace has some limitations as it cannot trace calls amongst libraries. For this purpose, one need to use latrace (**Figure 7**).

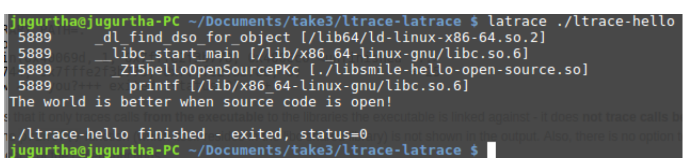


FIGURE 7 – Catching executable to library and library to library calls - latrace

Figure 8 summarizes the differences between : **strace**, **ltrace** and **latrace**.

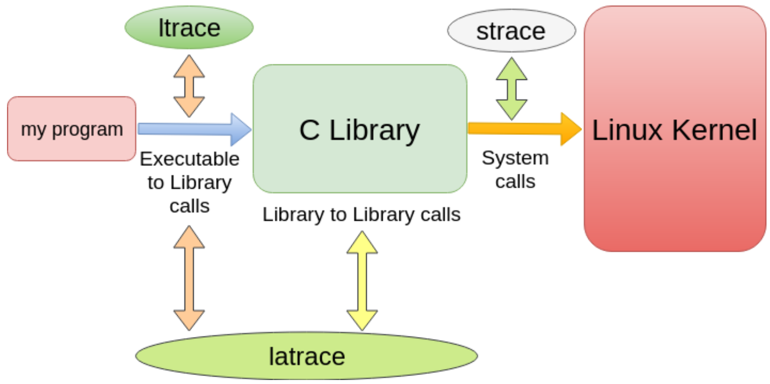


FIGURE 8 – Comparing between strace, ltrace and latrace

4.1.3 Valgrind

Valgrind is one of the most efficient memory debugging, intrumentation and profiling framework for userspace applications.

— **memcheck :** default tool used by valgrind's engine. It can detect memory leaks, uninitialized variables, Mismatch allocation and deallocation functions (using malloc then free), Reading or writing past-off buffer, ..., etc (see `https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/valgrind/memcheck`).

The following record was taken from a report generated by memcheck which locates precisely the memory leak source (40 bytes lost at **memcheck-memory-leak.c :8**) :

```
pi@raspberrypi:~/userspace/valgrind/memcheck$ valgrind --tool=memcheck \
> --leak-check=full ./memcheck-memory-leak
==7369== Memcheck, a memory error detector
........................
==7369== HEAP SUMMARY:
==7369== in use at exit: 40 bytes in 1 blocks
==7369== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==7369==
==7369== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==7369== at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-
amd64-linux.so)
==7369== by 0x40053C: main (memcheck-memory-leak.c:8)
==7369==
```

— **helgrind :** a thread profiler with great support for POSIX pthreads (see `https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/valgrind/helgrind`)

— **cachegrind :** Simulates program's to cache hierarchy interaction in the system. Chachegrind will always simulate two cache levels :

1. **L1 Cache :** Broken down into *L1Data* and *L1Instruction*.
2. **Unified L2 Cache :** Data and instructions are mixed together.

An example is shown at : `https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/valgrind/cachegrind`

— **callgrind :** **Callgrind** is a CPU profiler. The reader is probably familiar with GPROF. Howerver, GPROF is deprecated(it can neither support multithreaded applications nor understand system calls). We have provided an example at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/valgrind/callgrind

4.1.4 GDB and GDBserver

— **GDB :** official build-in debugger from GNU collection. It can start a program for debugging or attach to an already running process. Basically, gdb offers options shown in **Figure 9** :

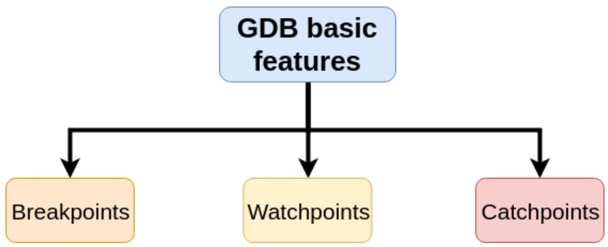


FIGURE 9 – Basic features of GDB

- **Breakpoints :** are predefined points where GDB stops when it finds them in a program. They allow us to examine registers status, memory dumps, environement variables, ..., etc (**Figure 10**).

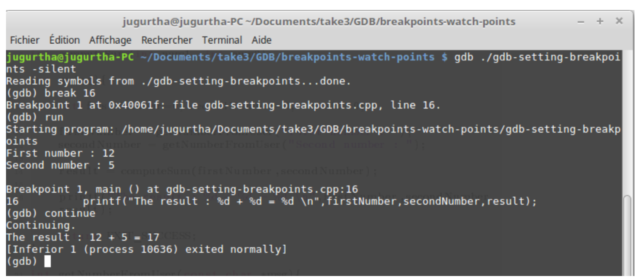


FIGURE 10 – Setting GDB breapoints

- **Watchpoints :** can monitor a variable (read and write) and reports its status (**Figure 11**).

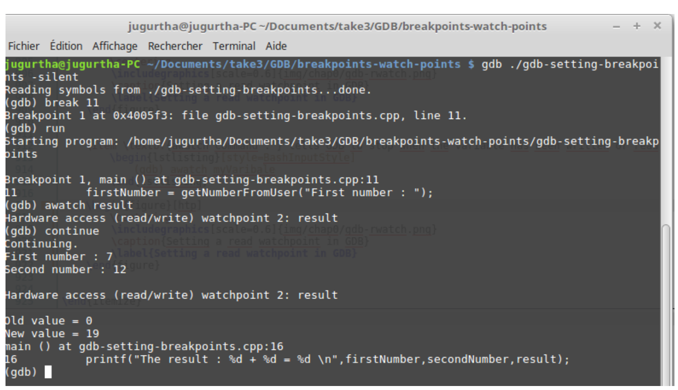


FIGURE 11 – Setting a read watchpoint in GDB

- **Catchpoints :** report events like fork , signal reception (SIGUSER1,SIGALRM, ..., etc) and exceptions (**Figure 12**).

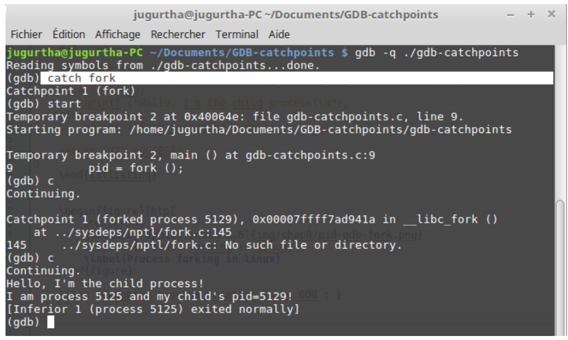


FIGURE 12 – Catching process forking using GDB

— **GDBserver :** Local debugging is not always an option and may not be possible especially for embedded devices. Those systems have fewer capabilities, a reason that leads us to remote debugging. **GDBserver** allows a program to be debugged remotely (**Figure 13**).

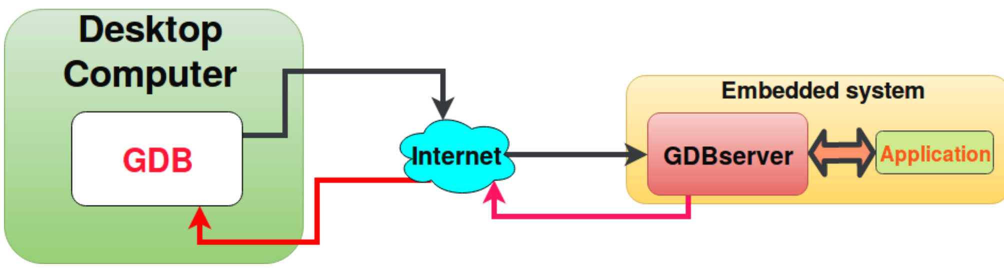


FIGURE 13 – Remote debugging using GDBserver

Remote **GDBserver** accepts connections from both *ethernet* and *serial* communication :

∗ **General settings of ethernet communication :**

1. GDBserver on the target

```
$ gdbserver :<portNumber> ./myProgram
```

2. GDB Client - Linux machine side

```
$ gdb-cross-platform ./myProgram
$ (gdb) target remote ip_address_gdbserver_machine:<portNumber>
```

∗ **General settings of serial communication :**

1. GDBserver on the target

```
$ gdbserver /dev/serial-channel ./myProgram
```

2. GDB Client - Linux machine side

```
$ gdb-cross-platform ./myProgram
$ (gdb) target remote /dev/serial-channel
```

Let's debug a « *Guess number* »program on a **Raspberry PI 3** running **GDBserver** *(sources are available at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-exampl Chap1-userland-debug/gdb/remote-debug/raspberryPI3)* :

1. **Raspberry PI 3 :** We will start Gdbserver on port 4000 (*You can choose any other port*).

```
$ gdbserver :4000 ./rpi−number−guess
```

The result of the above command is shown in **Figure 14**

FIGURE 14 – Starting gdbServer on Raspberry PI 3

2. **Linux desktop machine :** Launch a **gdb** session from a **Linux** machine and connect to target as shown in **Figure 15**

```
$ ./arm−none−eabi−gdb −silent ./rpi−number−guess
$ (gdb) target remote ip_address_raspberryPI:4000
```

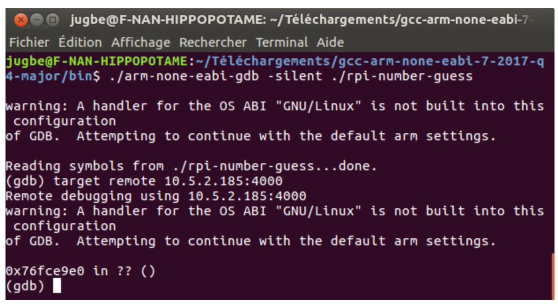


FIGURE 15 – Rasbperry PI 3 - Remote debugging GDB/GDBserver over Ethernet

At this point, a message sould be displayed at **Raspberry PI** side :

```
Remote debugging from host ip_address_host_GDB
```

Now you can place breakpoints, move around (*everything We know from GDB*) or even display generated number as shown in **Figure 16**

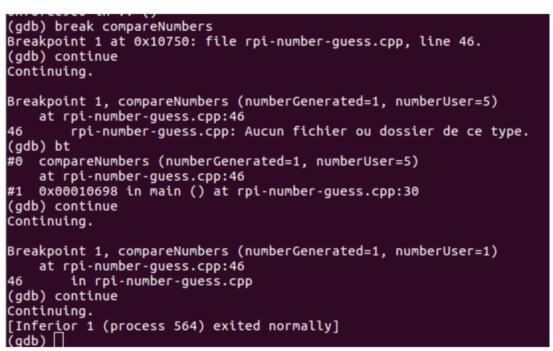


FIGURE 16 – Displaying backtraces on Raspberry PI 3 using GDB/GDBserver over Ethernet

Note : Latest versions of **Raspberry PI** seem to have troubles with Serial communication.

4.1.5 File core dump

When a userspace application was terminated abnormally (*due to a segmentation fault for example*), the system saves the content of program's virtual memory space at the instant of termination for *post analysis*, those files are known as **Core Dumps**.

1. **Enabling file core crash :** core dumping is not available by default and it has to be enabled. Hopefully, we can change this easily as follow :

```
$ ulimit -c unlimited
```

2. **Core crash generation :** Now, dump files are enabled ; We can execute a faulty program :

```
$ ./myProgram
Segmentation fault (core dumped)
```

   Remark : Notice the presence of « core dump »which indicates a generated core dump.

3. **File crash core analysis :** GDB can be used to analyse the userspace crash dump files, all We have to do is to launch **GDB** as follow :

```
$ gdb ./myProgram <coreFile>
```

   Remember : Your binary executable file must have been compilled with -g option, otherwise **GDB** is near to be useless.

4. **Custumizing the name of the core file :** the default name of the core files is « core », but some problems may rise :
   - We may have multiple core files in such a way we cannot differentiate which core dump belongs to a particular application
   - If an application crashes multiple times, then the new core file will overwrite the old one.

   Linux provides two files to custumize the naming convetion of the core dumps :

   (a) **/proc/sys/kernel/core_uses_pid :** generates a core dump file named « core.pid », where pid is the identifier of the process being terminated. We can enable this feature by :

```
# echo 1 > /proc/sys/kernel/core_uses_pid
```

   (b) **/proc/sys/kernel/core_pattern :** allows to set a formated core dump files as shown below :

Specifier	%e	%p	%t	%h
Meaning	Executable filename	process PID	timestamp	hostname

   Example : let's save a core dump file with the naming format :

```
# echo core.%h.%e.%p.%t > /proc/sys/kernel/core_pattern
```

   Which results in a name : « core.hostName.executableFileName.processID.timestamp »
   Other specifiers exit like : %u for real UID. The list is shown at : http://man7.org/linux/man-pages/man5/core.5.html.

4.2 Kernel land

The kernel is more challenging to debug than userspace. Going through a code that changes a variable value (like userspace) is different from a kernel function that handles interrupts, manages memory, migrates tasks between processors, ..., etc.

Weird behaviour should be expected when debugging kernel code

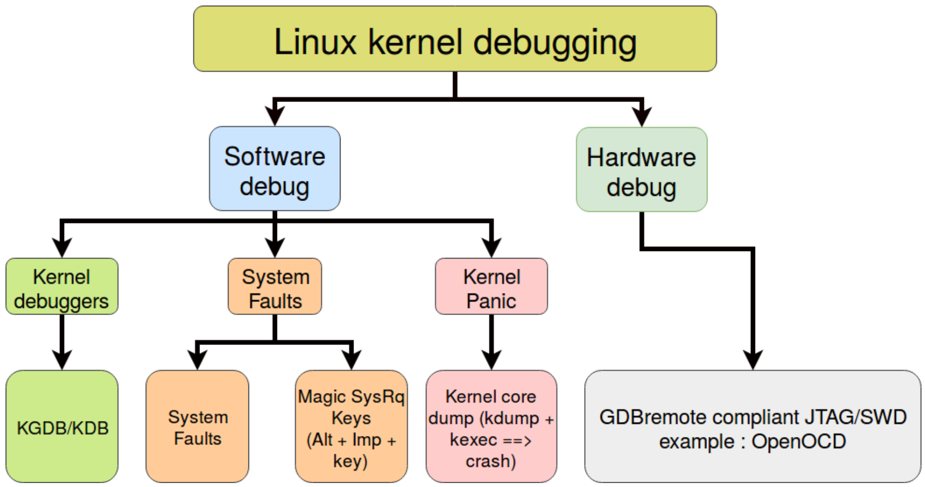


Figure 17 – Linux kernel debugging methodologies

4.2.1 KGDB/KDB

KGDB/KDB are the Linux kernel debuggers. KGDB is source level debugging and KDB is raw level.

The kernel must be built using special parameters in order to support KGDB/KDB as shown below :

— **KGDB :** for KGDB support, the kernel must be compiled with :

```
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
```

— **KDB :** requires the following flags to be enabled :

```
CONFIG_FRAME_POINTER=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
CONFIG_KGDB_KDB=y
CONFIG_KDB_KEYBOARD=y
```

Note : Kernel must be compiled with **debugging symbols**, otherwise **KGDB/KDB** will be almost useless.

Important : We can check for KGDB/KDB support by reading /boot/config file. If this file is missing, one need to look for manufacturer documentation. **Figure 18** shows how to check **KGDB/KDB** support on Raspberry PI 3.

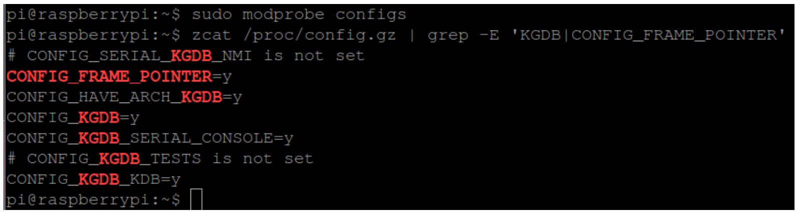

```
pi@raspberrypi:~$ sudo modprobe configs
pi@raspberrypi:~$ zcat /proc/config.gz | grep -E 'KGDB|CONFIG_FRAME_POINTER'
# CONFIG_SERIAL_KGDB_NMI is not set
CONFIG_FRAME_POINTER=y
CONFIG_HAVE_ARCH_KGDB=y
CONFIG_KGDB=y
CONFIG_KGDB_SERIAL_CONSOLE=y
# CONFIG_KGDB_TESTS is not set
CONFIG_KGDB_KDB=y
pi@raspberrypi:~$
```

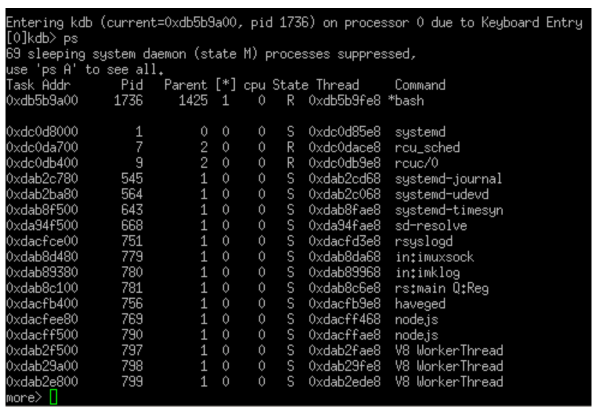
FIGURE 18 – Check for KGDB support on Raspberry PI 3

Both KGDB and KDB can be enabled by writing to the same file as illustrated :

```
pi@raspberrypi:~# echo ttyAMA0 > /sys/module/kgdboc/parameters/kgdboc
pi@raspberrypi:~# echo g > /proc/sysrq −trigger
```

Once **KGDB/KDB** is configured on the target, We can connect to the target in two different ways :

1. **GDB :** connection will be established with **KGDB** on the target.
2. **telnet :** connection will be received by **KDB** on the target (an example is shown in **Figure 19**).


```
Entering kdb (current=0xdb5b9a00, pid 1736) on processor 0 due to Keyboard Entry
[0]kdb> ps
69 sleeping system daemon (state M) processes suppressed,
use 'ps A' to see all.
Task Addr       Pid   Parent [*] cpu State Thread     Command
0xdb5b9a00     1736     1425  1    0   R  0xdb5b9fe8 *bash

0xdc0d8000        1        0  0    0   S  0xdc0d85e8  systemd
0xdc0da700        7        2  0    0   R  0xdc0dace8  rcu_sched
0xdc0db400        9        2  0    0   R  0xdc0db9e8  rcuc/0
0xdab2c780      545        1  0    0   S  0xdab2cd68  systemd-journal
0xdab2ba80      564        1  0    0   S  0xdab2c068  systemd-udevd
0xdab8f500      643        1  0    0   S  0xdab8fae8  systemd-timesyn
0xda94f500      668        1  0    0   S  0xda94fae8  sd-resolve
0xdacfce00      751        1  0    0   S  0xdacfd3e8  rsyslogd
0xdab8d480      779        1  0    0   S  0xdab8da68  in:imuxsock
0xdab89380      780        1  0    0   S  0xdab89968  in:imklog
0xdab8c100      781        1  0    0   S  0xdab8c6e8  rs:main Q:Reg
0xdacfb400      756        1  0    0   S  0xdacfb9e8  haveged
0xdacfee80      769        1  0    0   S  0xdacff468  nodejs
0xdacff500      790        1  0    0   S  0xdacffae8  nodejs
0xdab2f500      797        1  0    0   S  0xdab2fae8  V8 WorkerThread
0xdab29a00      798        1  0    0   S  0xdab29fe8  V8 WorkerThread
0xdab2e800      799        1  0    0   S  0xdab2ede8  V8 WorkerThread
more>
```

FIGURE 19 – Listing active processes on Beaglebone black wireless - KDB

4.2.2 System faults

System faults does not mean « panic ». Kernel Panic is a result of serious fault or a cascading effect of faults that can harm the system.
When a userspace program violates a memory access, a *SIGSEGV* is generated and the faulty process is killed (remember to enable core dump files in order to analyse them). The same is true for the kernel, when a driver tries to dereference an invalid Null pointer or overflows the destination Buffer, it is going to be killed.

Buggy code in a driver or a module may lead to one of the states : **kernel oops** and **System Hang**.

— **Kernel oops :** Sometimes called *Soft panics* (as opposed to hard kernel panic). Generally, they result from dereferencing a NULL pointer, overflowing kernel buffers and others faulty kernel code.

Reading Kernel Oops

Kernel oops can be obtained by reading kernel's ring buffer with : « **dmesg** ».

We are going to take a look at 2 particular messages (We have provided a more detailed page at http://fdeszffffffffff) :

1. **Error location and type :** The kernel is really accurate in describing the problem (see **Figure 20**).

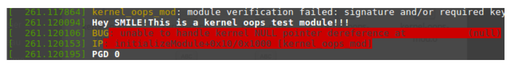


FIGURE 20 – Error type and location of faulty line - kernel oops

   — **BUG :** shows the error name, in our case it « dereferencing an NULL pointer ».
   — **IP :** Instrcution Pointer shows the location of the error (We will come back to it later).

2. **Reason and number of oops :** oops may have cascading effect and lead to chain of oops (maybe even to kernel panic), the kernel identifies them and reports us the reason that gave rise to them as shown in **Figure 21**.

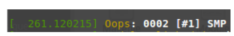


FIGURE 21 – Kernel Oops error code value - kernel oops

The error code « 0002 »must be converted to binary. To understand the interpretation of the code take a look at **Figure 22**.

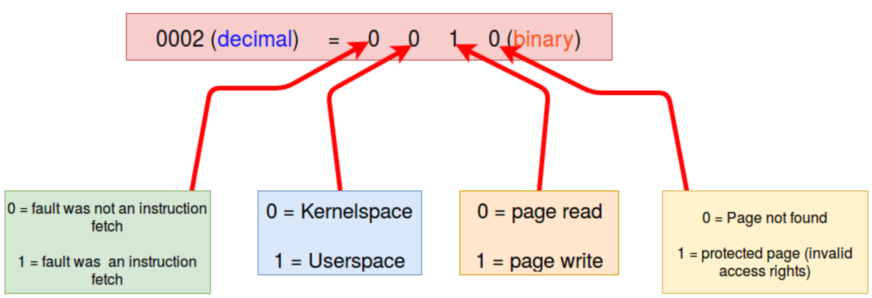


FIGURE 22 – Interpreting kernel oops error code

So finally, We can say that :

0 - 0 - 1 - 0 (binary) = a write request was made to a non existing page from the kernel and the instruction was not a « fetch instruction ».

Remark : **#1** shown in **Figure 21** is the number of oops occurrence (*As We have already said, the oops may happen multiple times and generate others*).

— **Kernel Hang and Magic Sysrq :** Everyone has experienced this situation at least one time. It is the state where a system is not responding anymore and completely frozen (not a KERNEL PANIC). This is called ***Hang state***.
Hopefully, We can use a forgotten feature in linux which is **SysRQ (Magic Keys)**.

SysRQ is combination of keyboard keys that executes a low level function. The kernel will always respond to **SysRQ** whatever the state it is undergoing ; though, the only exception for this is *kernel panic*.

ALT + SysRq + <command key> or ALT + Print Screen + <command key>

SysRq involves QWERTY Keyboard

The kernel pretends a **QWERTY** keyboard when using **SysRq**.

SysRq are not enabled by default on some systems (especially the old ones), they must be activated :

```
# echo 1 > /proc/sys/kernel/sysrq
```

- **ALT + SysRq (Print Screen) + l :** shows the backtraces for all CPUs.

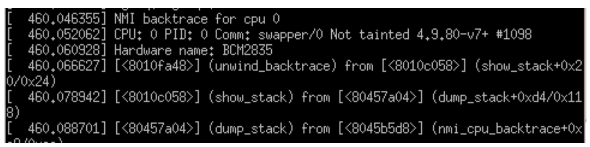


FIGURE 23 – Displaying backtraces for all CPUs using SysRq - Raspberry PI 3

- **ALT + SysRq (Print Screen) + m :** prints memory dump
- **ALT + SysRq (Print Screen) + p :** displays registers related information
- **ALT + SysRq (Print Screen) + c :** Forces a kernel panic, suitable if there is a **crashdump** utility installed on the system (*more in the next section*).

Note : *SysRq do not work on virtual machines (only some VM products support this feature), the combination of the key will be received by the HOST system.*

4.2.3 Core dump and Kernel panic

A kernel dump image can be obtained at any time in multiple ways. But, debugging symbols are mandatory.
If the kernel was not compiled using debugging symbols, one may try to add them as shown : https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liacf/oprofkernelsymrhel.htm. Howerver, such packages are not always available. elfmaster came with a solution called kdress (but seems to work only on x86_32 and x86_64).

- **Live kernel anaylysis /proc/kcore :**
  1. **Generating vmlinux (optional) :** If the linux image was compiled without debugging symbols, We can try to construct them. « kdress »was written by elfmaster is used for this purpose (kress is available at : https://github.com/elfmaster/kdress).
  2. **Accessing the /proc/kcore :** We can play around the kcore using GDB, let's first create a GDB session as follow :

```
# sudo gdb -q vmlinux /proc/kcore
```

3. **Navigating through the /proc/kcore :** technically, We can obtain every information by walking through this file (see **Figure 25**).

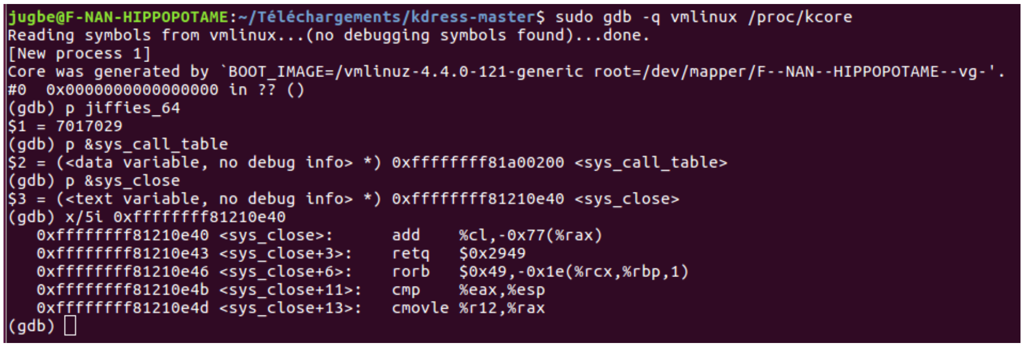


FIGURE 24 – Navigating through /proc/kcore using gdb

- **Post kernel crash analysis :** Kernel panic can be hard to troubleshoot (especially that bugs are almost impossible to reproduce in practice). We can get a kernel dump file in case of panic using Kdump and Kexec.
  — **kdump :**
  — **kexec :**
  Once a dump file was generated, it can be analysed using GDB or a more specialized utlity like : crash.

4.2.4 Linux hardware debugging with OpenOCD

OpenOCD is an open source project created by « **Dominic Rath** ». It is supported by a large community which maintains the source codes at https://sourceforge.net/projects/openocd/. **OpenOCD** provides a high level abstraction to access a debugging hardware interface (*JTAG*, *SWD*, *SPI*). Most today's platforms have built-in JTAG connector which allows them to be inspected, tested and even hacked.

Let's summarize the working internals of OpenOCD :

1. **General overview of OpenOCD :**

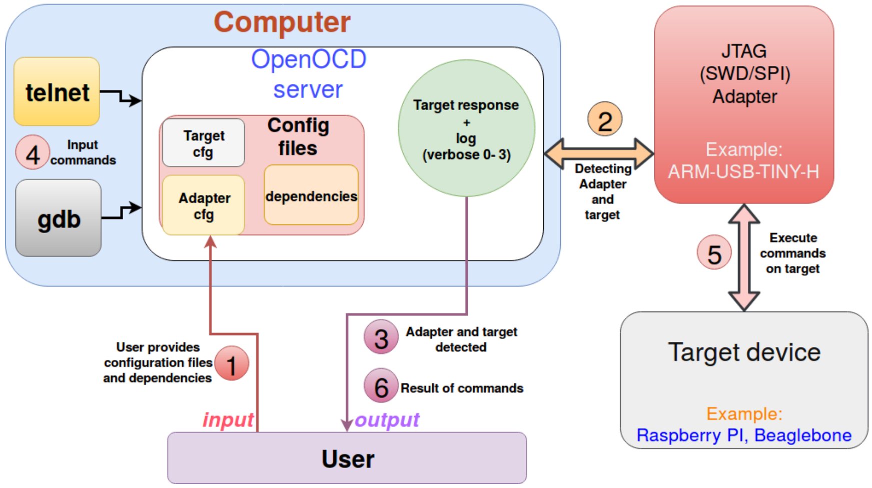


FIGURE 25 – OpenOCD general settings

(a) User starts OpenOCD with configuration files (at least adapter and target config files)

(b) If OpenOCD succeds to recognize the target, We can start debugging the target by using OpenOCD commands (OpenOCD receives commands from GDB or Telnet).

(c) OpenOCD executes executes the commands on the target and returns back the result to the user.

2. **General syntaxe Of OpenOCD :**

```
$ sudo ./src/openocd -s tcl/ -f tcl/interface/adapter_config_file.cfg \
> -f tcl/target/target_config_file.cfg
```

(a) **Hard wiring ARM-USB-TINY-H with raspeberry PI 3 :** connect Raspberry PI 3 with olimex JTAG adapter as shown in Figure 26.

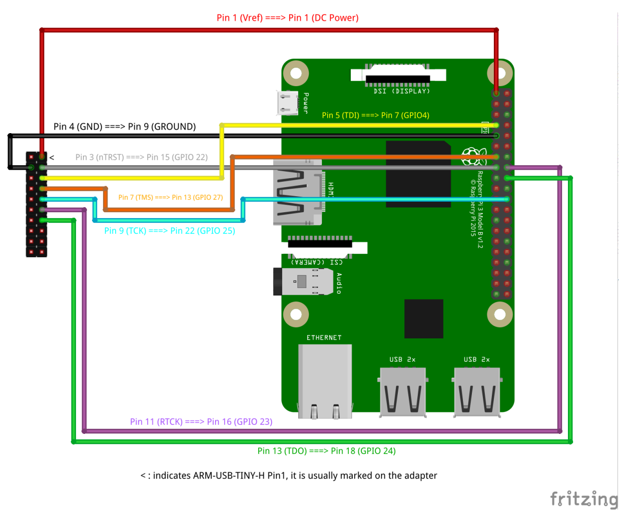


FIGURE 26 – Connecting OpenOCD to Raspberry PI 3

(b) **Enabling JTAG on Raspberry PI 3 :**

— **Jtag enabler :** source code is available at : http://sysprogs.com/VisualKernel/legacy_tutorials/raspberry/jtagsetup/JtagEnabler.cpp.

— **Edit Jtag enabler :** Jtag enablers seems to work only for Raspberry PI 1, the following lines should be changed as shown below :

```
#define BCM2708_PERI_BASE 0x3F000000
#define GPIO_BASE         (BCM2708_PERI_BASE + 0x200000)

```

— **Execute Jtag enabler :** as shown **Figure 27**

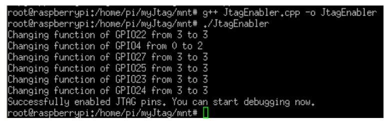


FIGURE 27 – Enable JTAG Debugging on Raspberry PI 3

Important : JTAG is enabled on Raspberry PI 3.

(c) **Debugging with OpenOCD :** We are ready to start **OpenOCD** as illustrated in **Figure 28**

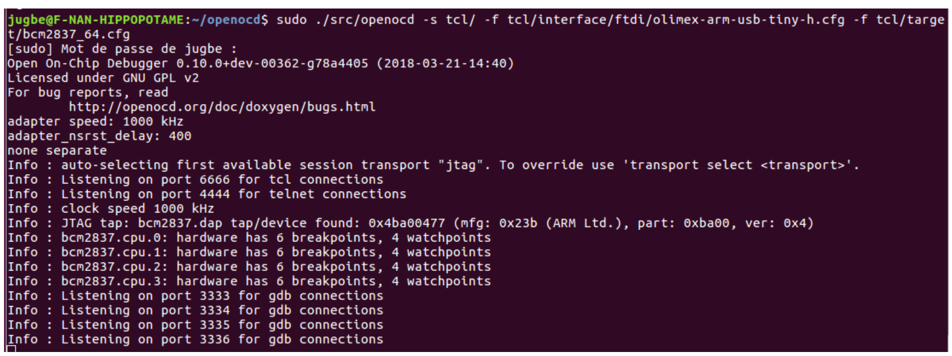


FIGURE 28 – Hardwiring ARM-USB-TINY-H to raspberry PI 3

Note : the line « *Info : JTAG tap : bcm2837.dap tap/device found : 0x4ba00477 (mfg : 0x23b (ARM Ltd.), part : 0xba00, ver : 0x4)* »means that OpenOCD was able to detect the Raspberry PI 3. We also see the breakpoints which indicates highly that the connection was a success.

3. **OpenOCD made easy with OESdebug :** OpenOCD is quiet difficult and complex to setup. We have provided a tool called « OpenOCD-wrapper »(OpenEasy Debug is written in python3) as a high level wrapper around **OpenOCD**.

OESdebug sources

Sources are available at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/DebugSoftware/OpenOCD-wrapper.

(a) **Start OESdebug :** We only need python3 interpreter and python-tk (graphic's library to be installed) :

```
# python3 main.py
```

(b) **OpenOCD support : OESdebug** is a wrapper program which intends to use OpenOCD easily. OESdebug checks for OpenOCD presence at start-up (*one can pinpoint OpenOCD's location if compiled from sources*). **Figure 29** shows OESdebug when OpenOCD is detected.

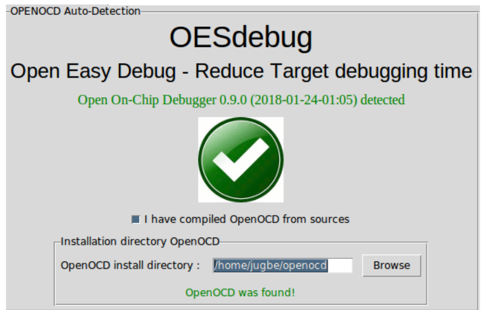


FIGURE 29 – Checking OpenOCD support - OESdebug

(c) **Adapter Support :** an adapter is the intermidiate component that allows OpenOCD (running as a deamon in the host) to access the target's JTAG TAP controller. We must choose a supported adapter that ships with OpenOCD (as shown in **Figure 30**) or create one (by checking « create a custom adapter »).

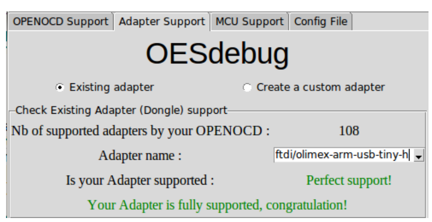


FIGURE 30 – Checking Adapter support - OESdebug

(d) **MCU support :** OpenOCD cannot support every target that exists (*We can add our own configuration file but it's a bit more enhanced* as shown in **Figure 31**).

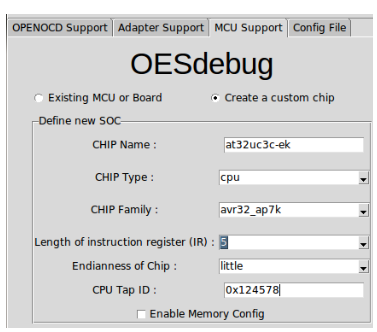


FIGURE 31 – Creating a new target config file - OESdebug

(e) **Generating configuration file :** Now, We can click on "Generate" to get a working OpenOCD script (**Figure 32**).

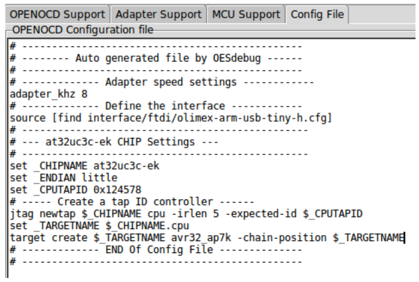


FIGURE 32 – Generating OpenOCD config file - OESdebug

4.3 Linux tracers

Tracing is the opposite of security, if security wants to hide what's happening in the kernel than tracing does the complete opposite.

4.3.1 Ftrace

Ftrace is the official linux tracing tool created by « **Steven Rostedt** »that has been merged to linux mainline since version *2.6.31*.

- **Trace-cmd :** Ftrace is quite tedious, boring and requires a long setup before we can get a trace. The creator of Ftrace « **Steven Rostedt** »released a Front-end tool for Ftrace called Trace-cmd.
  The general syntax used to record events using trace-cmd is :

  ```
  # trace-cmd record -p <tracer> -e <event1> -e <event2> -e <eventN> <program>
  ```

  And for reading events :

  ```
  # trace-cmd report
  ```

  As a working example, We will are going to trace a module :

  1. **Loading module :** neverthless to say that before tracing the module, it must be running (**Figure 33**)

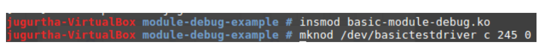
  

  FIGURE 33 – Insertion of module to kernel before tracing

  2. **Tracing module function :** We can launch trace-cmd, and set a filter on the functions to trace (in our case all function names that begin with « basic »)

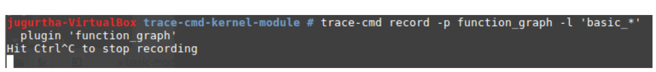
  

  FIGURE 34 – Tracing functions in a module using trace-cmd

  3. **Interact with the module :** after starting Ftrace, We must call one of the functions of our module. let's make a simple read on it (**Figure 35**)

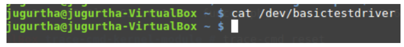


FIGURE 35 – Interacting with the kernel device module

4. **Reading report :** (**Figure 36**)

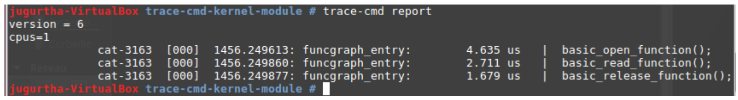


FIGURE 36 – Reading module trace report with Trace-cmd

- **Kernelshark :** We cannot close the discussion about Ftrace without pointing out an important tool called « Kernelshark ». Reading Ftrace report can be quiet difficult ; the third tool released by « **Steven Rostedt** »is KernelShark which is GUI based.

4.3.2 LTTng

1. **Create a session :** Every LTTng record must be made within a session (the session name can be anything We want) :

```
# lttng create <mySessionName>
```

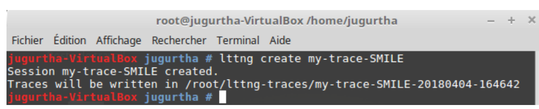


FIGURE 37 – Creating a session in LTTng

2. **Select a tracepoint (instrumentation point) :** We may select one or multiple (or even all) tracepoints.
We will choose for example to trace « sched_switch » :

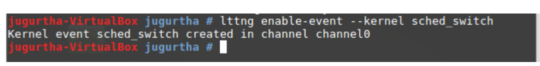


FIGURE 38 – Select kernel tracepoints in LTTng

3. **Start the tracing session :** We can start tracing at this point, LTTng will record all « sched_switch »events

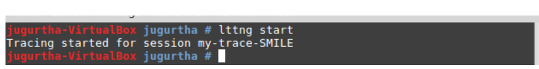


FIGURE 39 – Start tracing using LTTng

4. **Stop tracing session :** stop subcommand will halt recording and saves the tracing report.

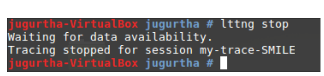


FIGURE 40 – Stop tracing using LTTng

5. **Destroy LTTng session :** We need to stop and destroy the current session.

```
# lttng destroy
```

6. **Visualize the trace report :**
   - **babeltrace :** We can view LTTng report in the console, however, when We record a lot of events for a long time, viewing the result in text-based mode is far to be easy

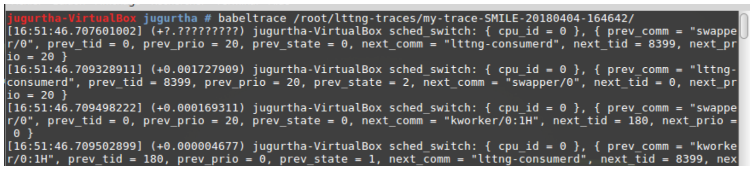


FIGURE 41 – Reading LTTng trace report using babeltrace

   - **trace compass :** This is a visual GUI to display the LTTng traces in a more convinient way. *Trace compass is an Eclipse C/C++ pluggin*.

We can say even more on LTTng :

— **LTTng USDT :** LTTng enables to attach User Statically Defined Tracepoints to userspace applications (*something not possible using Ftrace or perf*). It can trace **C/C++** code (as shown at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap3-tracers/Lttng-examples/Tracing-Userspace-C-App), **Python** scripts (https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap3-tracers/Lttng-examples/Tracing-Userspace-Python-App) and even **Java**.

— LTTng Logger file : when **LTTng** deamon is running (lttngd), it creates a special file in **ProFs** : `/proc/lttng-logger`. Applications can log their messages to this file (usefull for debugging), however it is not reliable as **LTTng USDT**.

— **LTTng toolkit analyses :** LTTng provides a powerfull toolkit called « LTTng analyses »to extract most relevant data from recorded traces (https://github.com/lttng/lttng-analyses). We are going to show two examples :

  — **lttng-analyses-record :** which record an automatic LTTng session (instead of manual recording as We did) as shown in **Figure 42**

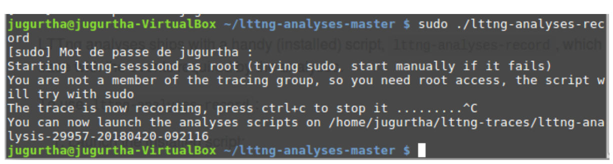


FIGURE 42 – Automatic session recording - LTTng toolkit analyses

  — **lttng-schedlog :** shows task scheduling in chronogical order (**Figure 43**).

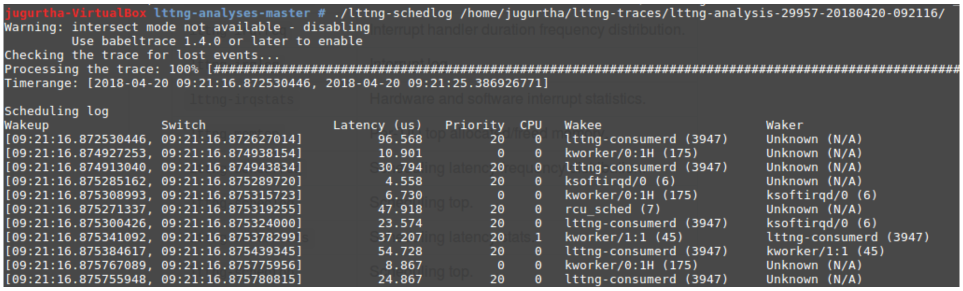


FIGURE 43 – Getting sched_switch logs from traces - LTTng toolkit analyses

4.3.3 Perf

Perf is a linux official profiler, tracer and benchmarker tool that has been merged to the linux mainline since version 2.6.31.

The most perf's used commands are :

— **list :** lists the events supported by perf (HW/SW events, tracepoints).

— **stat :** counts the number of occurrence of an event (group of events or all the events) in the system or particular program.

— **record :** samples an application (or the entire system) and shows the callgraph of functions.

— **report :** parses and displays the report generated by perf (perf list or perf record).

— **script :** Prints trace as text so that it can be parsed by other tools.

Perf can be used in different ways :

- **Perf to gather statistics :** perf count statistics related to programs (or system).

  1. **Collecting statistics :** genarl syntax is illustrated as follow :

```
# perf stat ./program [arguments_program]
```

  An example is shown in **Figure 44**. **Figure 44**

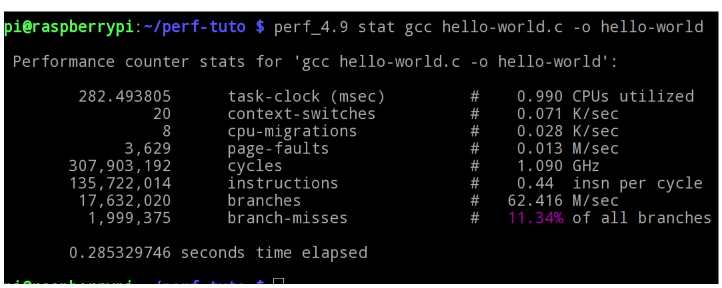


FIGURE 44 – Gather program's statistics - perf

  2. **Filtering returned statistics :** one may choose which statistics to view as shown in **Figure 45**.

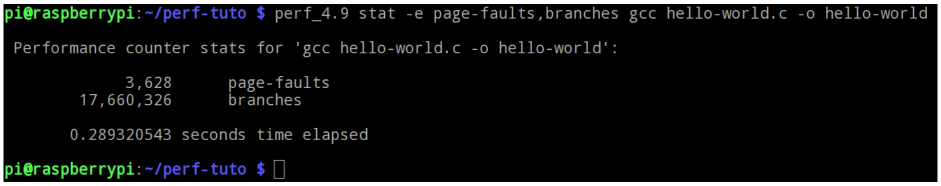


FIGURE 45 – Get specific program's statistics - perf

- **Perf as a profiling tool :** perf can sample and record applications callgraphs (or entire system).
  — **record phase :** general syntax is shown below :

```
# perf record -F <frequency_rate> [optional perf arguments] ./program [arguments_program]
```

  An example is shown in **Figure 46**.

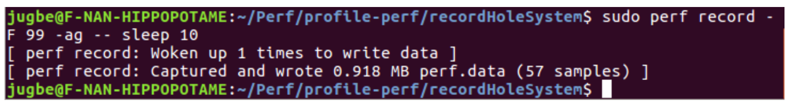


FIGURE 46 – Sampling function calls and stack traces on the entire system

  — **Reading report :** reports can be read using :

```
$ sudo perf report -g
```

  Reports are displayed with functions sorted according to their execution time (*time exhaustive functions are on the top and shown in red*) as shown in **Figure 47**.

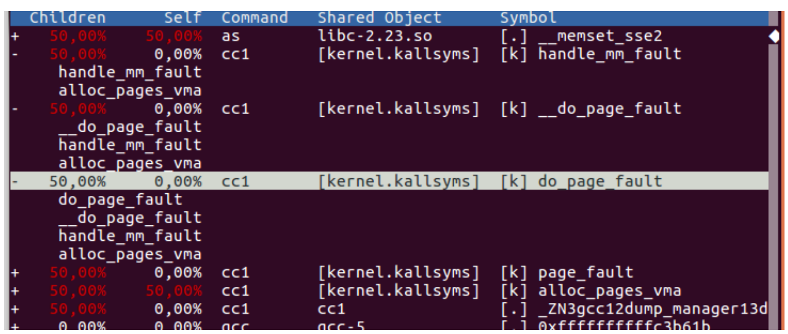


FIGURE 47 – Displaying Perf records in Basic mode

  Remark : We can display a tree view report using :

```
$ sudo perf report -g --stdio
```

- **Perf as a tracing tool :**
  1. **Choose a tracepoint :** tracepoints must be supported by perf as shown in **Figure 48**.

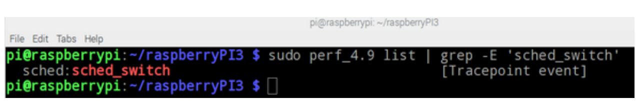


FIGURE 48 – Check tracepoint sched_switch support - perf

2. **Trace selected tracepoint events :** Launch our executable using perf (-e is used to select a tracepoint) as shown in **Figure 49**.

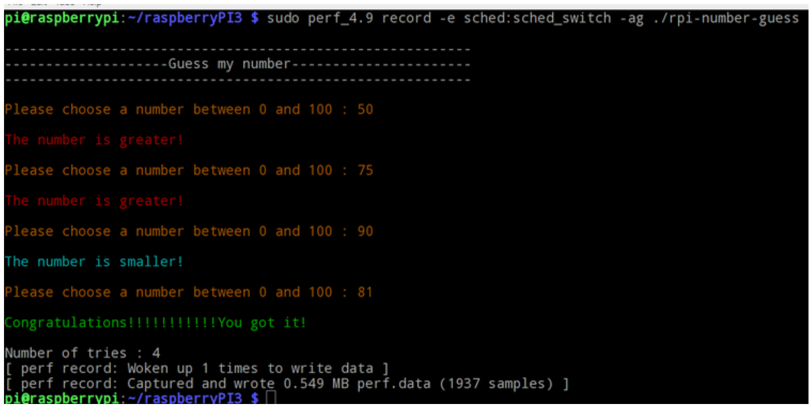


Figure 49 – trace sched_switch event - perf

3. **Read tracepoint report :** We can read the report using :

```
$ sudo perf script [-i Path_to_perf.data]
```

An example is shown in **Figure 50**.

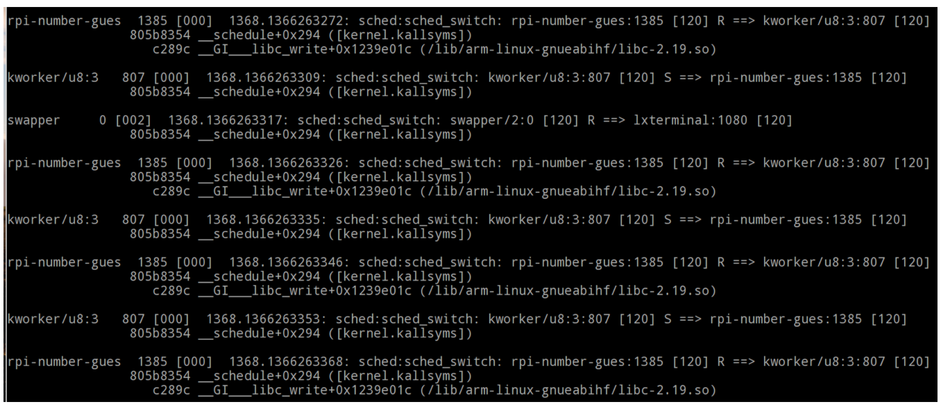


Figure 50 – Reading recorded sched_switch event - perf

4.3.4 eBPF

BPF (*Berkeley Packet Filter*) is the famous virtual machine (*running inside the kernel*) which is used by https://www.tcpdump.org/. eBPF (**Extended Berkeley Packet Filter**) is the extension of **BPF**. Hopefully, it does much more than handling packets, it can serve as an observality , DDos mitigation , Intrusion detection , Tracing , ..., etc (**Figure 51** - taken from Brenden Gregg's blog)

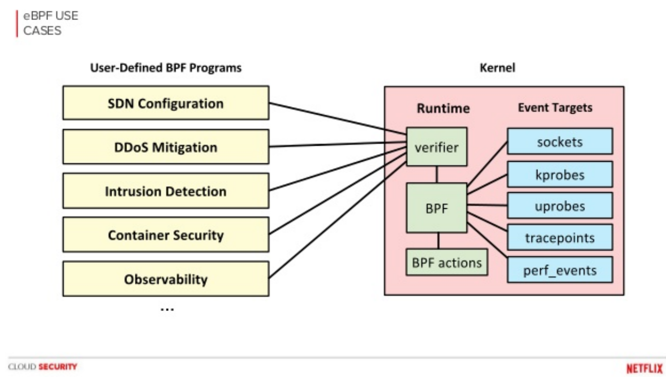


FIGURE 51 – Linux EBPF internal and usage

eBPF is difficult to use (We must write C codes), BCC (BPF Compiler Collection) was made to make it easier. BCC is a front-end toolkit of eBPF which can be found at the following page : https://github.com/iovisor/bcc.

1. **Install bcc :** BCC became easy to install, instructions are provided at : https://github.com/iovisor/bcc/blob/master/INSTALL.md
2. **Running eBPF scripts :**
   — **bcc provided scripts :** bcc ships with tools that handles everyday's common tasks. One can try them as shown in : https ://github.com/iovisor/bcc
   — **Creating scripts from scratch :** We can write custom eBPF scripts https://github.com/iovisor/bcc/blob/master/docs/tutorial_bcc_python_developer.md

Basic syntax of eBPF :

- **Creating kprobe :** sample code is shown in **Appendix A.1** (*output result is illustrated in* ***Figure 52***).

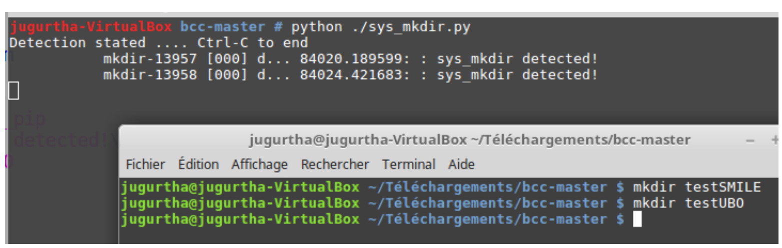


FIGURE 52 – Tapping sys_mkdir using eBPF - kprobe

- **Creating tracepoint :** an example is shown in **Appendix A.2** (*see* ***Figure 53***).

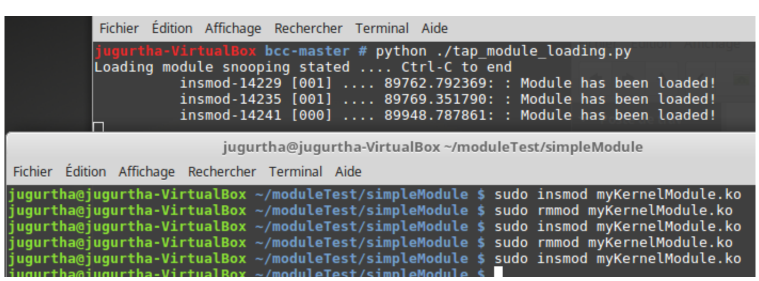


FIGURE 53 – Tapping module loading event using eBPF - tracepoint

eBPF enhanced security

eBPF programs imposes restrictions (*no infinite loops, kprobes cannot be attached to all functions, ..., etc*). It ensures that a script will never crash or hang kernel code (https://lkml.org/lkml/2015/4/14/232).

Important : Tracepoints are highly encouraged to be used than kprobes as they are more stable and portable (*function names and prototype can change so kprobes will be incorrect*).

4.3.5 Choosing a tracer

The following table summarizes quickly important features of some tracers :

Tool	Native support	Front-end tool	Remote tracing	GUI parsing tools	Real time tracing
Ftrace	since linux 2.7	Trace-cmd	yes	KernelShark	no
Perf_event	since linux 2.8	perf	no	Hotspot	no
LTTng	no	lttng	yes	Trace compass	no
eBPF	since linux 4.4	bcc	no	no	no

Tracers may be selected depending on requirements, We made a simple benchmarking tool to help us in choosing the most appropriate. The benchmark measures memory and execution time overhead as well as other metrics.

Benchmarking sources

Sources are located at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/DebugSoftware/tracers-cmp-benchmark.

A python3 utility (available at : **https://github.com/jugurthab/Linux_kernel_debug/tree/master/DebugSoftware/cmpTracer-GUI**) visualizes the results in GUI form.

Some results are shown below : Tests were made 10 times (then average was taken) on a machine with initial conditions are shown in **Figure 54**

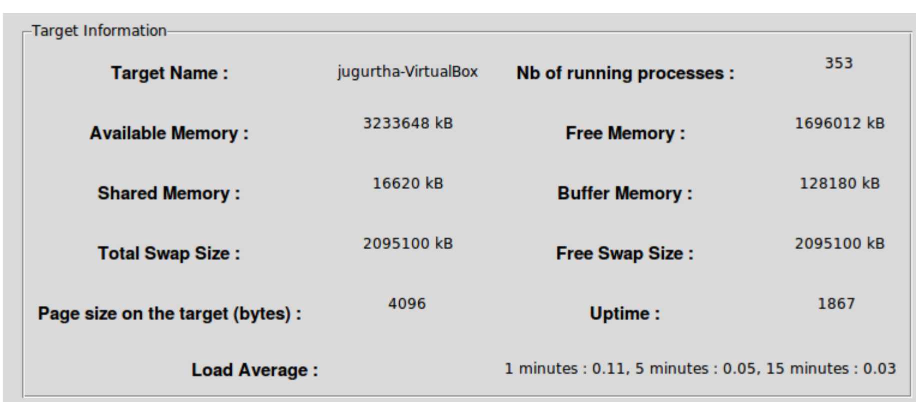


FIGURE 54 – Target's initial state before experiment - Linux Mint

Tool	Execution time (s)	Max RSS	V.C Switches	Inv.C Switches	Minor page faults	Size of file (KB)
qsort	0.19	8818	1	79	1194	0
Ftrace	4.04	8848	170	261	1323	29418
Perf	0.53	10227	27	118	3170	23
LTTng	0.21	8834	1	69	1213	2723

Important : eBPF requires Linux4.9 to access full functionnalities (or at least Linux4.4 for partial support).

4.4 Defeating Anti debugging mechanisms

Security is a concern for every modern device, It became crucial to keep the data safe and avoid them from leaking.
Debugging is not only meant to troubleshoot a slow or faulty system. A good security analyst requires skills in debugging.
Bugs are not only introduced as a result of programming mistakes (No one writes perfect code), they can be caused by malicious code injected on purpose by attackers.

4.4.1 Attacking userland

Attacking the userland is a wide spread practice and requires only few setup to achieve the desired result.

The simplest example is the use of ptrace as shown below :

```
if(ptrace(PTRACE_TRACEME , 0) < 0 ){
    printf("You cannot debug me!\n");
    exit(EXIT_FAILURE);
}

```

The code snippet means that the process will be traced by it's father.

Problem : only one debugger can be attached to a running process at time t (see **Figure 55**).

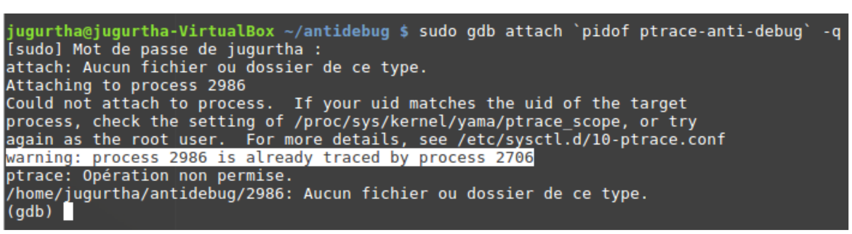


FIGURE 55 – GDB cannot attach to the program due to Anti-debugging

As one can see from **Figure 55**, GDB was not able to attach to the process (even if launched with root privileges).

More attacks are possible

Other methods have been experimented like : LD_PRELOAD and hijacking C library (https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap6-kernel-security/userspace/LD_preload)

4.4.2 targeting Kernel Code

The kernel can be subjected to many threats (like rootkits). We can change behaviour of almost any instruction in the kernel (*it's paramters and return value*), and cause serious system issues that goes from simple **Denial of Services to steeling private data**.

Some basic attacks like : **Jprobes** (https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap6-kernel-security/kernel/jprobes) and **Kprobes** (https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap6-kernel-security/kernel/kprobes).

More advanced attacks like module tampering : (https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap6-kernel-security/kernel/module-tampering)

1. **Merge modules :** ld can assemble modules to produce a final one as shown in **Figure 56**.

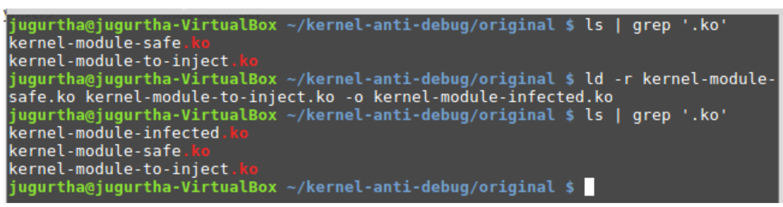


FIGURE 56 – Meging modules using ld

2. **Analyse the resulting module :** We can dump module's symbol table of as follow :

```
$ objdump -t kernel-module-infected.ko
```

The reader can notice that « fak_module_init »has been linked correctly.
All what is left is forcing « init_module »to point to our malicious symbol « fak_module_init »(*at relative location 00000014*).

3. **Make init_module as an alias of fak_module_evil :** We must change the relative address of init_module to execute our malicious function as shown below.

```
./elfchger -s init_module -v 00000014 kernel-module-infected.ko
```

Dumping the infected module using « objdump -t kernel-module-infected.ko »is shown in **Figure 57**

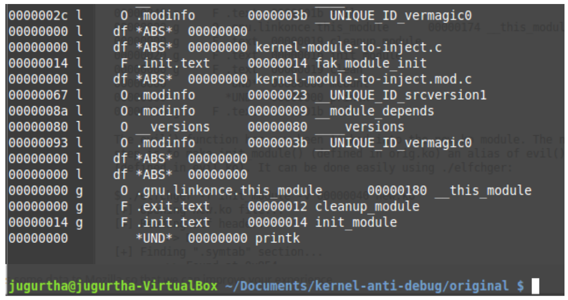


FIGURE 57 – Forcing init_module to become an alias of a malicious function

4. **Insert infected module into kernel :** see **Figure 58**

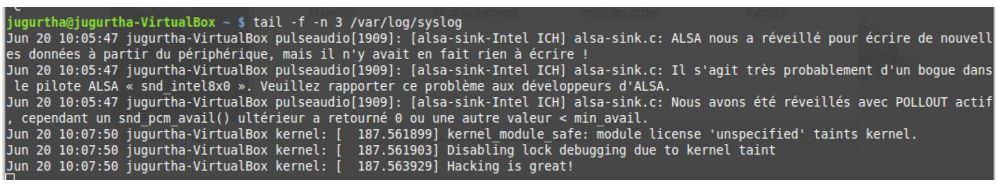


FIGURE 58 – Infected module executing malicious function

The module is executing the evil function

5 Encountered difficulties

Debugging is a rare skill, only few resources are available. We can point out some difficulties that We have seen during internship :

5.1 Hardware issues

Hardware problems were a real bottlenecks, as they are more difficult to locate.

5.1.1 JTAG

Some manufacturers try to hide **JTAG connectors** to make it difficult to access (*due to security reasons*). Beaglebone black wireless is an example of those boards. Soldering a JTAG connector was mandatory (*it is not easy on those tiny devices*).

Note : sometimes JTAG connection is encrypted or even damaged by manufacturers (*but this is rare*). More can be said about **JTAG** as connectors are different and pinout definition is not always easy to find.

5.1.2 OpenOCD hardware interfacing

As mentionned previously, **OpenOCD** is a hardware debugging solution (it is complicated). It took me 1.5 week to understand how to make a correct hardware setup. Interfacing OpenOCD compliant adapter with the target is far to be easy.

5.2 Software

5.2.1 Debugging symbols

Most kernels in production are compiled stripping this option. The advantage is to reduce kernel's image size, however, tools like : GDB becomes practically useless as they require debugging symbols. some solutions exist to reconstruct it (without recompiling the kernel) but works only fine on x86 (see https://github.com/elfmaster/kdress).

Even worse, /proc/kcore does not exist on most embedded systems (like ARM) [2].

5.2.2 Yama blocks ptrace

Yama is a security module that disables ptrace. GDB, strace and ltrace make use of ptrace which must be enabled.

5.2.3 JTAG lockers

Even if OpenOCD hardware interfacing is correct, some boards have software protections to disable JTAG. Raspberry PI is an example. The firmware refuses any JTAG connection by default. Workarounds were made to disable such mechanisms.

2. More details about /proc/kcore are available at : https://lwn.net/Articles/45315/

5.2.4 OpenOCD scripts

As We have already mentionned, OpenOCD does not support every board. Custom configuration files must be written to include new platforms. We have made scripts generation easier with OESdebug, provided step by step documentation of OpenOCD and an animation that helps to understand more (https://jugurthab.github.io/debug_linux_kernel/zero-to-hero-openocd.html)

5.2.5 Tracers in embedded systems

Tracers are not always easy to port on embedded systems, especially if DebugFS does not exist (kernel has not been compiled with it).

Some troubles were noticed with trace-cmd and perf.

LTTng on the otherside is not supported by every kernel.

5.2.6 DebugFs absent

Security engineers drops down DebugFs support as it is allows anyone to get insight into the Kernel. Only Hardware debugging can help in such case.

6 Conclusion

A long journey was made with Linux debugging, testing tools and documenting results. We have crossed through the userspace, then went exploring various tools like : *GDB*, *Valgrind*, *strace* and *ltrace*.

We moved to Kernel-land and learnt to solve it's issues through debuggers (KGDB/KDB, Kernel oops and Magic SysRq). We provided a step by step guide for writing custom OpenOCD scripts for JTAG debugging (We also have made a wrapper tool to make it easier).

We also made a big step in understanding Linux working internals with tracers (Ftrace, Perf, LTTng). We have seen their usages, front-end tools and compared them to help us choosing the most appropriate for a given situation. We have also discovered eBPF which is the most prominent Linux tracer. We must keep in mind that debugging is not only made to trace bugs, but also reverse malicious code (another reason to sharpen our debugging skills). Such scenarios are quite common today, and being able to detect them is a crucial requirement.

Once again, We should stress out that debugging can save hours of trying to troubleshoot a problem. We must keep in mind that developpers work in team, each has it's coding style and not everyone checks for return values, null pointers, buffer overflows, ..., etc.

Reader must keep in mind that printf(printk) works great with small codes, however can overwhelm a system with messages, making it slow and even unresponsive. Industrial projects can goes beyond of million lines of code.

Personally, I enjoyed SMILE's internship, It prepared me for real world industry and tought me that we need more than coding skills to be a good developper. I had a lot of fun debugging Linux, gathering performances and stack traces and I loved OpenOCD as Hardware JTAG debugging allows a complet control over target.

Appendices

A eBPF

A.1 Attaching eBPF kprobe

```
from bcc import BPF

# prog will store the eBPF C program
prog = """
int detect(void *ctx){
  // write message into trace_pip
  bpf_trace_printk("sys_mkdir detected!\\n");
  return 0; // always return 0
}
"""

# Loads eBPF program
b = BPF(text=prog)

# Attach kprobe to kernel function and sets  ... as jprobe handler
b.attach_kprobe(event="sys_mkdir", fn_name="detect")

# Show message when ePBF stats
print("Detection␣stated␣....␣Ctrl-C␣to␣end")

# print result to user
while 1:
  # read messages from trace_pip and display them to user
  b.trace_print()
```

A.2 Enabling eBPF Tracepoint

```
from bcc import BPF

# prog will store the eBPF C program
prog = """
TRACEPOINT_PROBE(module, module_load){
  // events are from /sys/kernel/debug/tracing/events/module/module_load/fo
  bpf_trace_printk("Module has been loaded!\\n");
  return 0; // always return 0
};
"""

# Loads eBPF program
b = BPF(text=prog)
```

```

# Show message when ePBF stats
print("Loading module snooping stated .... Ctrl-C to end")

# print result to user
while 1:
  # read messages from trace_pip and display them to user
  b.trace_print()
```