University of western Brittany

Final Year Project Defense

Dive into practical Linux debugging

Debugging Linux OS

By :
Jugurtha BELKALEM

Tutor :
David GARRIOU

Supervisor :
Jalil BOUKHOBZA

22 août 2018

Table of contents

1 Acknowledgments

I would like to express my gratitude to all people who helped me throughout this internship.

First, I would start by thanking my tutor « David GARRIOU »for his guidence and advices. I enjoyed working with him.

Next, I want to say big thanks to my manager « Antoine POUSSE »for encouraging, motivating and making me feel comfortable inside the company.

I also want to thank all my colleagues at « SMILE nantes »for making this internship possible and helping me to accomplish my project.

Special thanks to my familly, my friends who were always on my side, sharing my happiness and misfortune.

Lastly, I would like to thank University of western brittany (UBO) and it's staff for making me an embedded system engineer and boosting my potential and capabilities.

2 Introduction

Embedded devices are increasingly popular, they 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 :** Linux kernel sources are 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 such as 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 prone to errors »*), 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 : https://github.com/torvalds/linux

3 General view of internship

3.1 Introducing company

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

FIGURE 1 – SMILE opensource company logo

SMILE advertises 4 different services as shown in **Figure 2**.

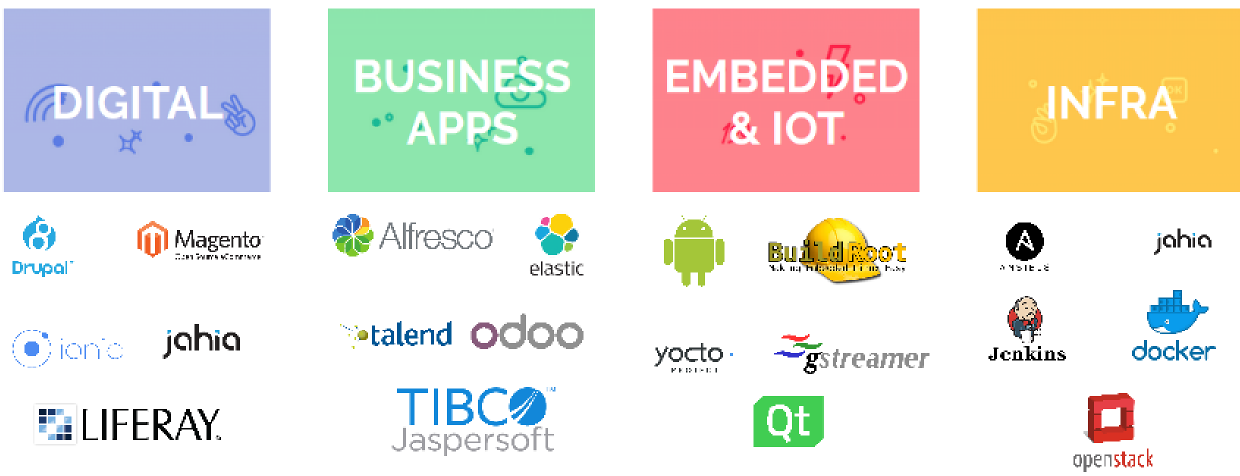


FIGURE 2 – SMILE-Opensource provided services and associated technologies

- **DIGITAL :** a division which creates websites, mobile apps and collaborative software.
- **BUSINESS APPS :** the service collects all business activities of customers allowing them to get better insight into their data and be more efficient.
- **EMBEDDED & IOT :** which builds software for innovative smart objects.
- **OUTSOURCING :** specialized in private cloud computing.

I'm part of **EMBEDDED & IOT** division.

SMILE has over than 1300 employees (**Smilers**) across 7 different countries (*France, Belgium, Switzerland, Luxembourg, Netherlands, Ukraine and Morocco*).

3.2 Internship objectives

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.
- Point-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.

3.3 Internship pre-requests

The pre-requests of the internship are :

* Good skills on C/C++ and Python.
* Working on Linux environment, basic Linux kernel is recommended.
* Background electrical and electronics engineering concepts

3.4 Internship requirements

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

1. **Userspace debugging methodologies :** mainly for C/C++ (*using GDB, strace, ptrace, ltrace, valgrind*).
2. **Kernel-land code debugging :** using KGDB/KDB, kernel oops, magic SysRQ, OpenOCD (with a 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 between each others 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.

4 Available equipements

Debugging Linux is a challenging task which requires a good preparation. In this section We present a global overview of some of the equipement used during the internship.

4.1 Hardware platforms

4.1.1 Beagle Bone Black Wireless

The evolution of beaglebone black which adds wireless support (WIFI, Bluetooth) and fast linux boot (see **Figure 3**).

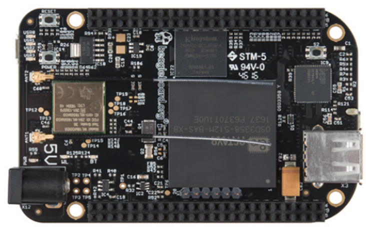


FIGURE 3 – Beaglebone black wireless

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

4.1.2 Raspberry PI 3 B+

The lastest version as this time of writing with enhanced processor and ethernet speed (**Figure 4**).

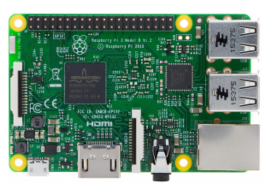


FIGURE 4 – Raspberry PI 3

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

4.1.3 stm32f407 Board :

Used to build high performance applications oriented for audio processing (see **Figure 5**).

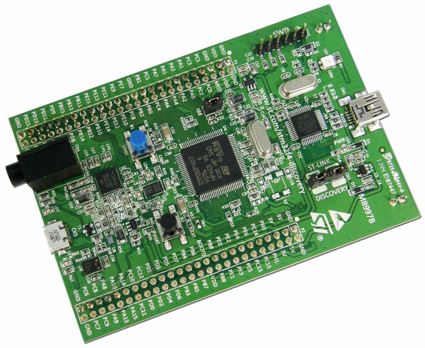


FIGURE 5 – stm32f407 Board

Specifications are available at : can be found 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

4.1.4 AT32UC3C-EK Board :

An old developement kit for Atmel AVR microcontrolers (see **Figure 6**).

FIGURE 6 – AT32UC3C Board

Hardware spefications : available at http://www.farnell.com/datasheets/1511964.pdf

4.1.5 ARM-USB-TINY-H JTAG Adapter :

OpenOCD debugging interface adapter that uses the FTDI protocol (see **Figure 7**).

FIGURE 7 – ARM-USB-TINY-H

Usage is described at : https://www.olimex.com/Products/ARM/JTAG/_resources/ARM-USB-TINY_and_TINY_H_manual.pdf

Note : OpenOCD supports multiple adapter protocols (ftdi, cmsis-dap, amt_jtagaccel, remote_bitbang, ..., etc). We can check the complet list at : http://openocd.org/doc/html/Debug-Adapter-Configuration.html#Debug-Adapter-Configuration.

4.2 Software

4.2.1 Pycharm IDE

Pycharm is a python IDE, which makes developement fast. Examples of projects developed made in Python is an **OpenOCD wrapper** utility « OESdebug »at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/DebugSoftware/OpenOCD-wrapper.

4.2.2 Eclipse C/C++ IDE

Code examples were written maily in C, Eclipse C/C++ was helpful. Code are hosted on github at : https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples.

4.2.3 OpenOCD

Open source software allowing Hardware debugging, sources at maintained at : https://sourceforge.net/projects/openocd/files/openocd/.

5 Internship solutions summary

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

About report

This report gives only some samples of what was made, the 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

5.1 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 8**.

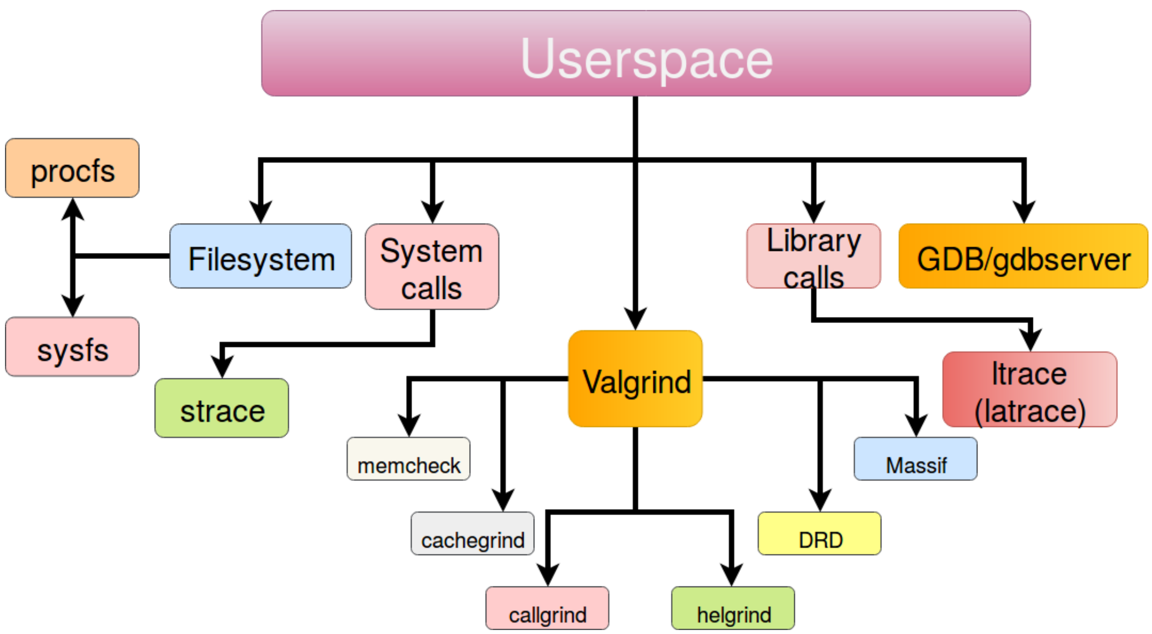


FIGURE 8 – Linux userspace debugging methodologies

Important : We are going to introduce each tool.

5.1.1 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 of it's knowledge to the userspace.

— **ProcFs (/proc) :** 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 :** shows limits (number of files to open, size of core dump file, ..., etc) of a given process.

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

— **SysFs (/sys) :** a more recent filesystem (*more organized than **procfs***), one may be concerned with the folder **/sys/module** as it is required to debug modules (*modules are not part of vmlinux image, We need to know their location*).

5.1.2 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 in order 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
```

Calls are divided into two categories, each using a set of tools to trace them :

— **System calls (Syscalls) :** a request from userspace to kernel in order to provide a service (open file, close socket, allocate memory, ...,etc). A tool called strace allows to have more insight on **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 :** Executables rely on external libraries to provide more functions (*like the C library*). Access to libraries is granted through **library calls**, **ltrace** is used to trace them. « **ltrace** »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

> **About report**
>
> **ltrace** has some limitations as it cannot trace calls amongst libraries. For this purpose, one need to use **latrace** (**Figure 9**).

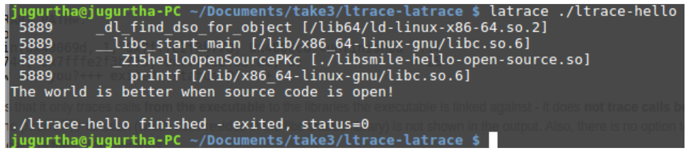


FIGURE 9 – Catching executable to library and library to library calls - latrace

Figure 10 summarizes the differences between : **strace**, **ltrace** and **latrace**.

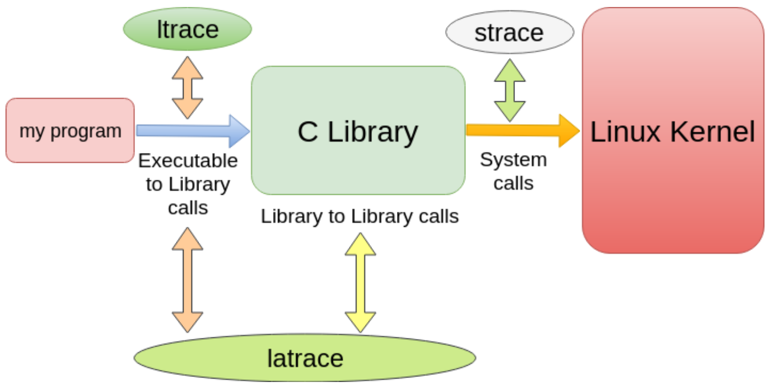


FIGURE 10 – Comparing between strace, ltrace and latrace

5.1.3 Valgrind

Valgrind is one of the most efficient memory debugging, intrumentation and profiling framework for userspace applications. Valgrind ships with 11 tools , We will go through some of them :

— **Memcheck :** the 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** (*a working example is shown at :* *https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/valgrind/helgrind*).

— **Cachegrind :** this tool simulates interactions with the cache hierarchy of the system. Cachegrind 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 available at : *https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap1-userland-debug/valgrind/cachegrind.*

— **callgrind :** this 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.*

5.1.4 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 11** :

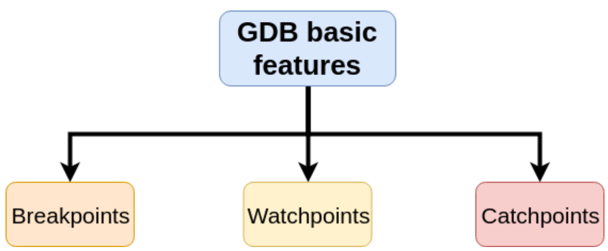


FIGURE 11 – Basic features of GDB

- **Breakpoints :** are predefined points where **GDB** stops when it finds them in a program. They allow us to examine registers status, dumping memory, controlling environment variables, ..., etc (**Figure 12**).

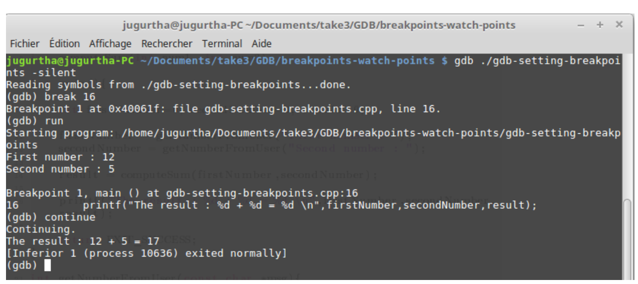


FIGURE 12 – Setting GDB breapoints

- **Watchpoints :** can monitor a variable or memory location (*for read and write operations*) and reports its status (**Figure 13**).

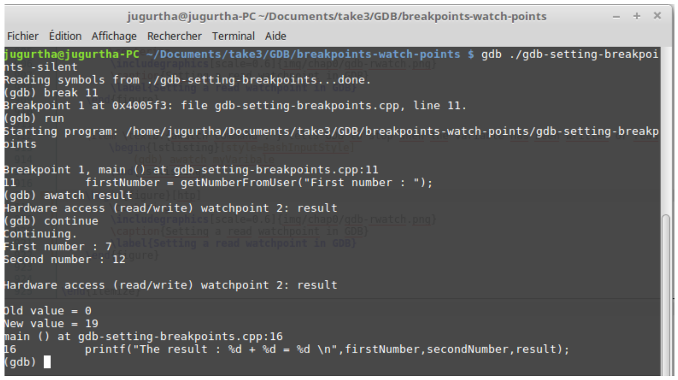


FIGURE 13 – Setting a read watchpoint in GDB

- **Catchpoints :** report events like fork , signal reception (*SIGUSER1, SIGALRM, ..., etc*) and exceptions (**Figure 14**).

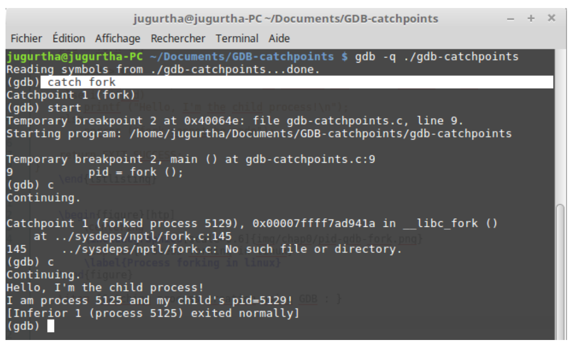


FIGURE 14 – Catching process forking using GDB

— **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 use remote debugging. **GDBserver** allows a program to be debugged remotely (**Figure 15**).

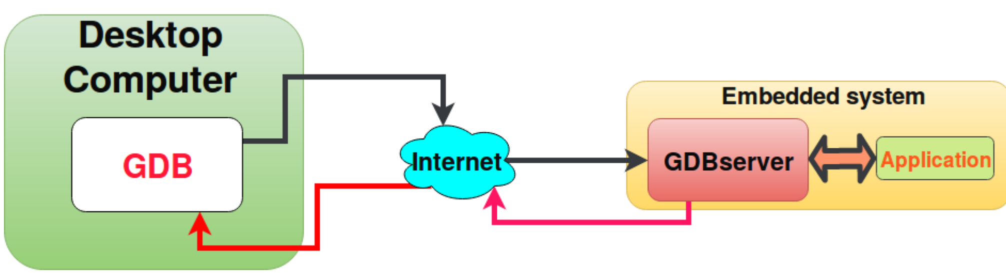


FIGURE 15 – Remote debugging using GDBserver

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 16**.

Figure 16 – Starting gdbServer on Raspberry PI 3

2. **Linux desktop machine :** Launch a **gdb** session on a **Linux** machine and connect to target (*Raspberry PI 3*) as shown in **Figure 17**.

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

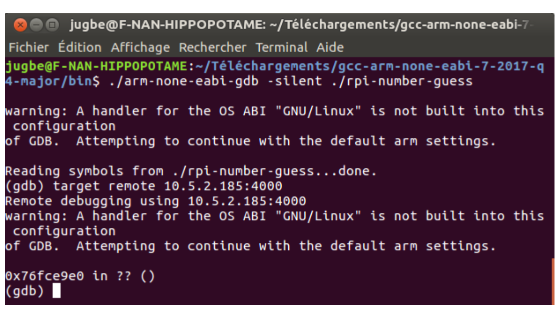
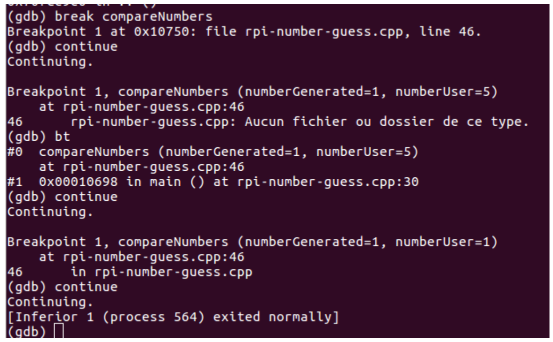


Figure 17 – Rasbperry PI 3 - Remote debugging GDB/GDBserver over Ethernet

At this point, a message sould be displayed on **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 18** (numberGenerated=1).


```
(gdb) break compareNumbers
Breakpoint 1 at 0x10750: file rpi-number-guess.cpp, line 46.
(gdb) continue
Continuing.

Breakpoint 1, compareNumbers (numberGenerated=1, numberUser=5)
    at rpi-number-guess.cpp:46
46      rpi-number-guess.cpp: Aucun fichier ou dossier de ce type.
(gdb) bt
#0  compareNumbers (numberGenerated=1, numberUser=5)
    at rpi-number-guess.cpp:46
#1  0x00010698 in main () at rpi-number-guess.cpp:30
(gdb) continue
Continuing.

Breakpoint 1, compareNumbers (numberGenerated=1, numberUser=1)
    at rpi-number-guess.cpp:46
46      in rpi-number-guess.cpp
(gdb) continue
Continuing.
[Inferior 1 (process 564) exited normally]
(gdb)
```

FIGURE 18 – Displaying backtraces on Raspberry PI 3 using GDB/GDBserver over Ethernet

5.1.5 File core dump

When a userspace application has 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 in the system. 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 dumped »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 :

```
1 # 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 :

```
1 # 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.

5.2 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

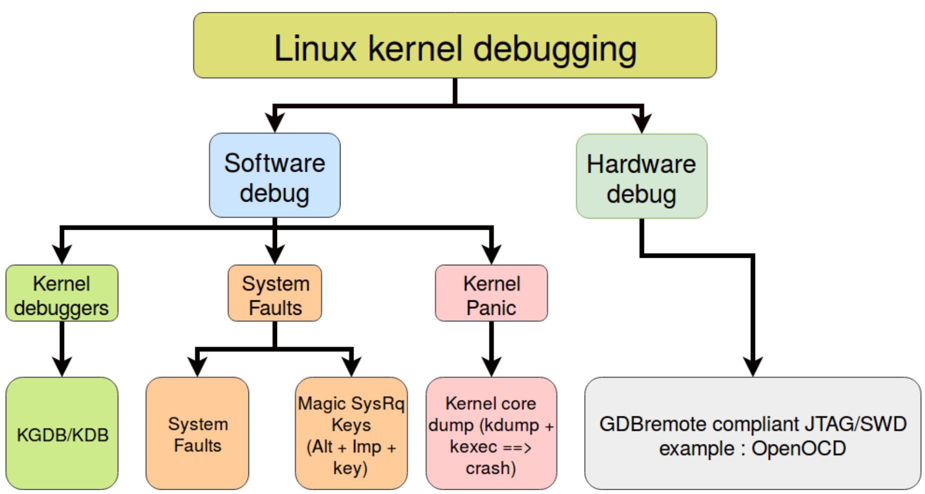


FIGURE 19 – Linux kernel debugging methodologies

5.2.1 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.

Check kernel KGDB/KDB support and enable them

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 20** shows how to check **KGDB/KDB** support on **Raspberry PI 3**.

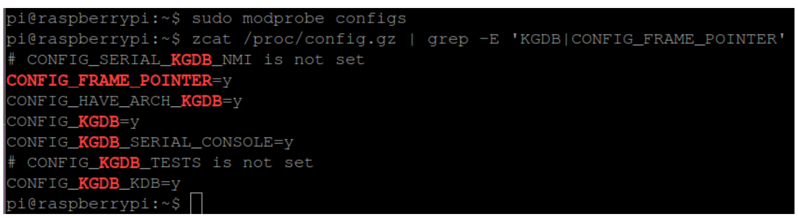


FIGURE 20 – Check for KGDB support on Raspberry PI 3

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

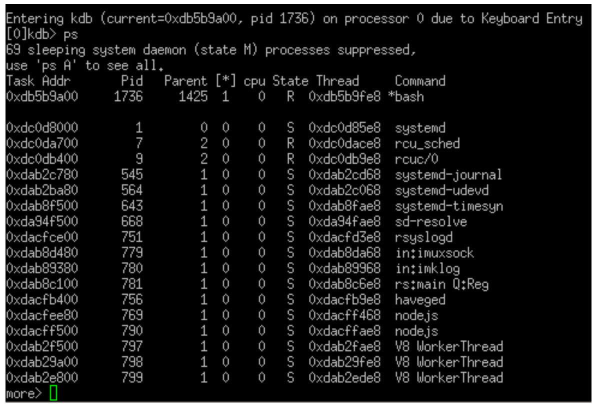
```
# Configure tty console
pi@raspberrypi:~# echo ttyAMA0 > /sys/module/kgdboc/parameters/kgdboc
# Halt the kernel
pi@raspberrypi:~# echo g > /proc/sysrq-trigger
```

Once **KGDB/KDB** is configured on the target, We can establish a debugging session in two different ways :

1. **GDB :** connection will be established with **KGDB** on the target.

```
# Start GDB with kernel image containing debug symbols
remote@machine:~# gdb ./vmlinux
# Connect to remote target (ttyS0 can change depending on configuration)
(gdb) target remote /dev/ttyS0
```

2. **Serial communication :** connection will be received by **KDB** on the target (*an example is shown in **Figure 21***).


```
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 21 – Listing active processes on Beaglebone black wireless - KDB

Note : A serial communication utility (*like putty*) must be configured with the correct serial port and baud rate.

5.2.2 System faults

System faults do 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 system faults 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 Oop

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

We are going to take a look at 2 particular messages from an oops :

1. **Error location and type :** The kernel is really accurate in describing the problem (see **Figure 22**).

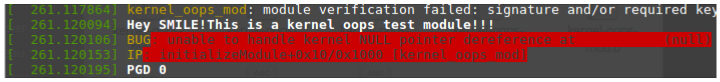


FIGURE 22 – Error type and location of faulty line - kernel oops

- **BUG :** shows the error name, in our case it « dereferencing a NULL pointer ».
- **IP :** Instruction 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 reports us the reason that gave rise to them as shown in **Figure 23**.

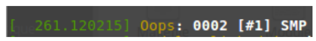


FIGURE 23 – Kernel Oops error code value - kernel oops

The error code « 0002 »must be converted to binary. To understand the interpretation of the code take a look at **Figure 24**.

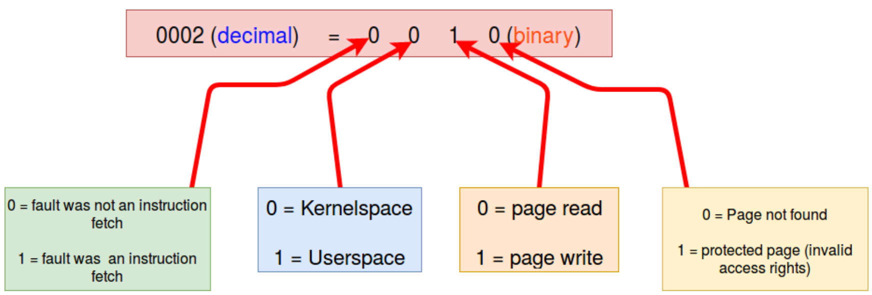


FIGURE 24 – Interpreting kernel oops error code

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 23** 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 execute low level kernel functions. 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 assumes 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 (see **Figure 25**).

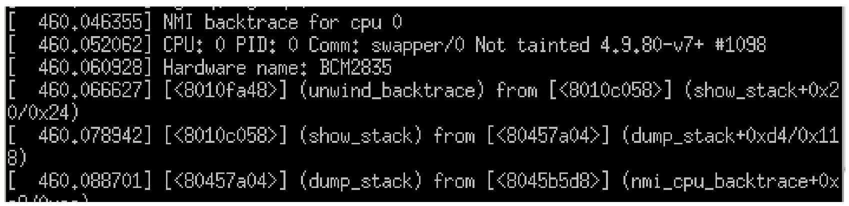


FIGURE 25 – Displaying backtraces for all CPUs using SysRq - Raspberry PI 3

- **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.*

5.2.3 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 at : https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liacf/oprofkernelsymrhel.htm. Howerver, such packages are not always available. **Ryan O'Neill** came with a solution called kdress (but seems to work only on x86_32 and x86_64).

- **Live kernel analysis /proc/kcore :** a file used to explore the Linux adress space.
  1. **Generating vmlinux (optional) :** If the linux image was compiled without debugging symbols, We can try to construct them. « kdress »was written by Ryan O'Neill (*known as elfmaster*) is used for this purpose (*kdress 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 26**).

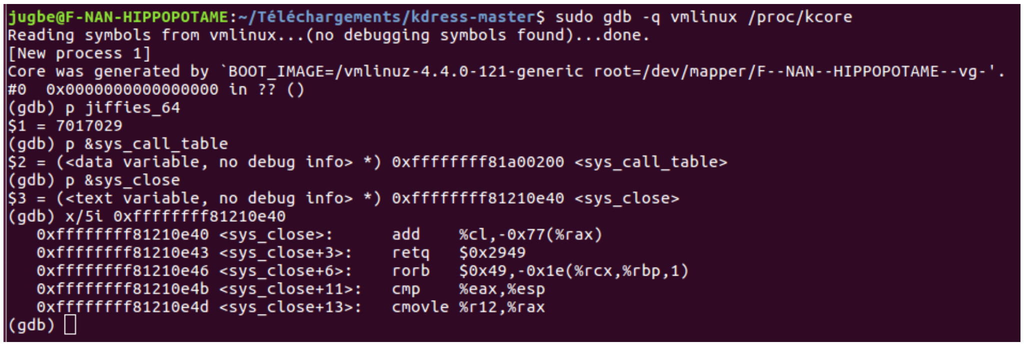


FIGURE 26 – Navigating through /proc/kcore using gdb

As shown in **Figure 26**, We have been displaying various kernel information (like jiffies and location of sys_close). One can even write or place breakpoints on particular functions or instructions.

- **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.
  - **kexec :** which allows to load quickly a new kernel from a running one (*it does not perform any basic setup initialization like those made by BIOS*).
  - **kdump :** uses kexec to start a new kernel when Panic is detected in the current one. Then dumps virtual memory (what can be dumped can be configured) of the crashed kernel from the newly launched one and saves result into a core dump file (can be stored in disk or sent through network which is ideal for embedded devices) as shown in **Figure 27 (taken from Adrien Mahieux presentation- Linux crashdump analysis**).

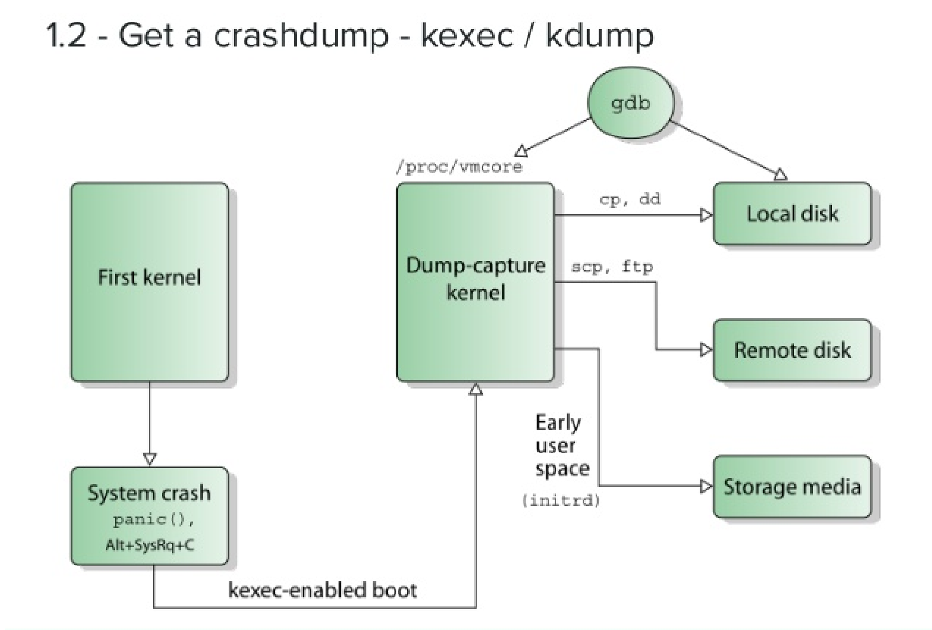


FIGURE 27 – Dumping Linux address space using kdump and kexec

Once a dump file has been generated, it can be analysed using GDB or a more specialized utility like **crash** (*a full course can be found at : https://www.dedoimedo.com/computers/crash-analyze.html*).
Crash can display the list of active processes, memory usage, backtraces, calltraces and may even point to error locations (*can be used to show kernel oops*).

5.2.4 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 and experiment on a Raspberry PI 3 :

1. **General overview of OpenOCD :**

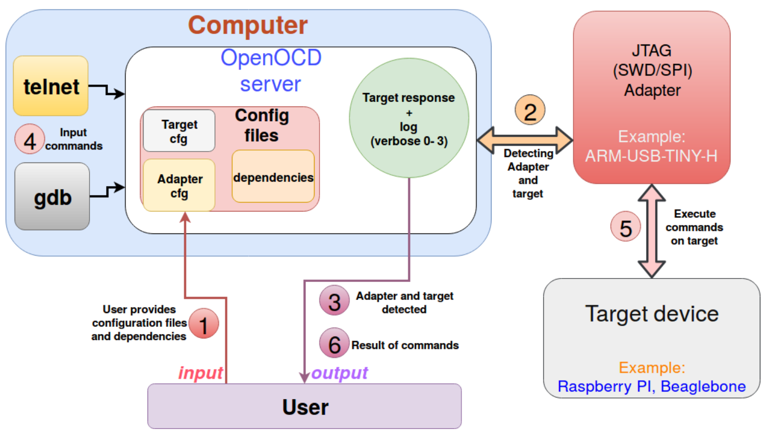


FIGURE 28 – OpenOCD general settings

   (a) User starts OpenOCD with configuration files (*at least adapter and target config files*)
   (b) If OpenOCD succeeds to recognize the target, We can start debugging it by using OpenOCD's commands (*OpenOCD receives commands from GDB or Telnet*).
   (c) OpenOCD executes the commands on the target and returns back the result to the user.

2. **General syntax Of OpenOCD :** OpenOCD requires at least 2 configuration files (one for the target and one for the adapter), file dependencies (if any) must be also included using the option -s :

```
$ 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 29.

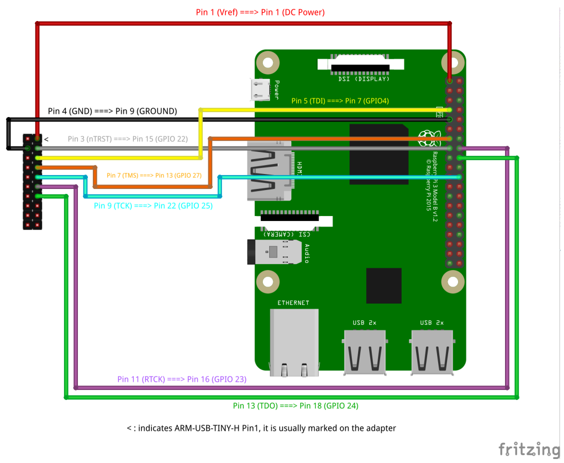


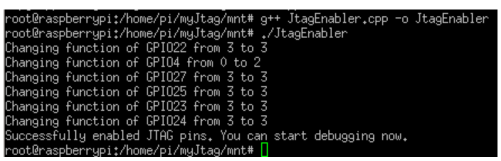
FIGURE 29 – Connecting OpenOCD to Raspberry PI 3

(b) **Enabling JTAG on Raspberry PI 3 :** Jtag is disabled by default on Raspberry.
- **Jtag enabler :** source code is available at : http://sysprogs.com/VisualKernel/legacy_tutorials/raspberry/jtagsetup/JtagEnabler.cpp.
- **Edit JTAG enabler :** JTAG-enabler 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 30**

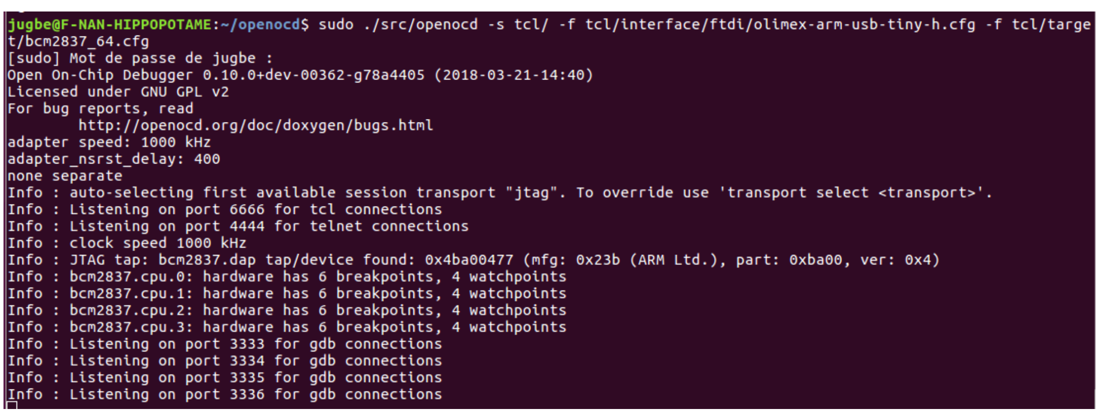


```
root@raspberrypi:/home/pi/myJtag/mnt# g++ JtagEnabler.cpp -o JtagEnabler
root@raspberrypi:/home/pi/myJtag/mnt# ./JtagEnabler
Changing function of GPIO22 from 3 to 3
Changing function of GPIO4 from 0 to 2
Changing function of GPIO27 from 3 to 3
Changing function of GPIO25 from 3 to 3
Changing function of GPIO23 from 3 to 3
Changing function of GPIO24 from 3 to 3
Successfully enabled JTAG pins. You can start debugging now.
root@raspberrypi:/home/pi/myJtag/mnt#
```

FIGURE 30 – 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 31**



```
jugbe@F-NAN-HIPPOPOTAME:~/openocd$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/targe
t/bcm2837_64.cfg
[sudo] Mot de passe de jugbe :
Open On-Chip Debugger 0.10.0+dev-00362-g78a4405 (2018-03-21-14:40)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
adapter speed: 1000 kHz
adapter_nsrst_delay: 400
none separate
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : JTAG tap: bcm2837.dap tap/device found: 0x4ba00477 (mfg: 0x23b (ARM Ltd.), part: 0xba00, ver: 0x4)
Info : bcm2837.cpu.0: hardware has 6 breakpoints, 4 watchpoints
Info : bcm2837.cpu.1: hardware has 6 breakpoints, 4 watchpoints
Info : bcm2837.cpu.2: hardware has 6 breakpoints, 4 watchpoints
Info : bcm2837.cpu.3: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
Info : Listening on port 3334 for gdb connections
Info : Listening on port 3335 for gdb connections
Info : Listening on port 3336 for gdb connections
```

FIGURE 31 – 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 quite 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** (*It allows even to generate **OpenOCD** scripts on the fly*).

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 32** shows OESdebug when OpenOCD is detected.

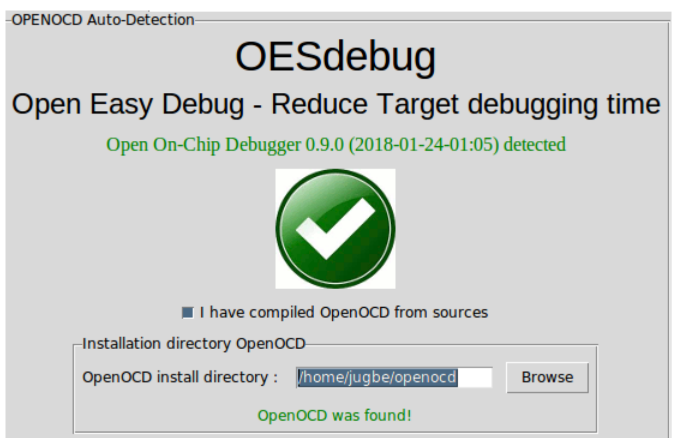


FIGURE 32 – Checking OpenOCD support - OESdebug

(c) **Adapter Support :** an adapter is the intermediate 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 33**) or create one (by checking « create a custom adapter »).

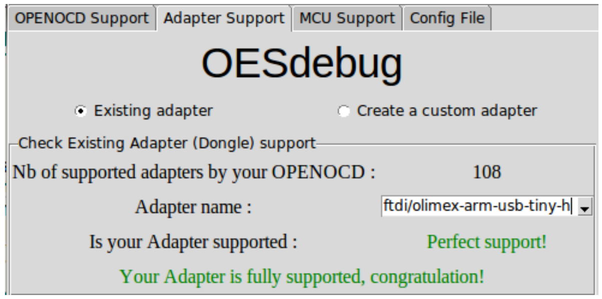


FIGURE 33 – Checking Adapter support - OESdebug

(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 34**).

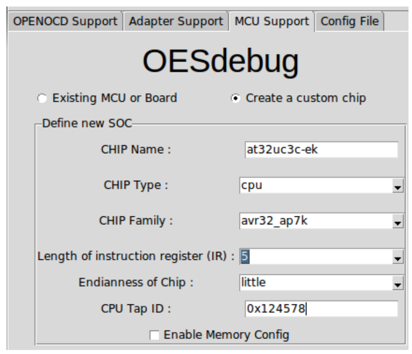


FIGURE 34 – Creating a new target config file - OESdebug

(e) **Generating configuration file :** Now, we can click on "Generate" to get a working OpenOCD script (**Figure 35**).

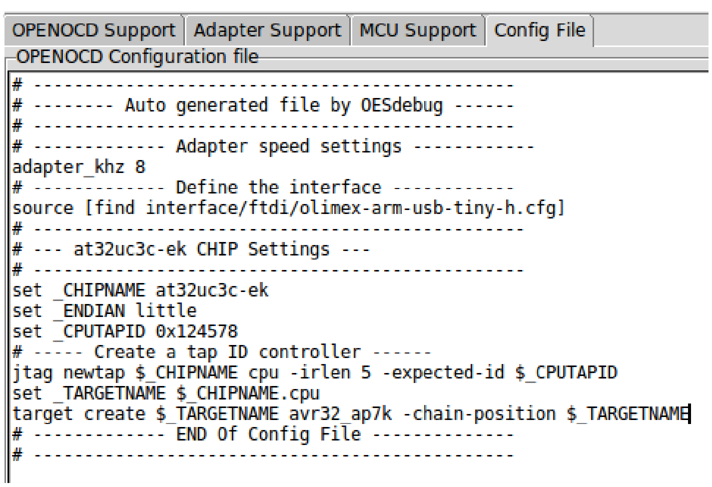


FIGURE 35 – Generating OpenOCD config file - OESdebug

(f) **Launching OpenOCD :** Once script file has been generated, We can start OpenOCD using start OpenOCD button. If the configuration was successful, OpenOCD will recognize the target as shown in **Figure 36**.

```
adapter speed: 8 kHz
Info : auto-selecting first available session transport "jtag". To override use
'transport select <transport>'.
at32uc3c-ek.cpu
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 8 kHz
Info : JTAG tap: at32uc3c-ek.cpu tap/device found: 0x3200003f (mfg: 0x01f (Atmel
), part: 0x2000, ver: 0x3)
Info : Listening on port 3333 for gdb connections
```

FIGURE 36 – Launching OpenOCD from OESdebug to debug AT32UC3C-EK target

The AT32UC3C-EK has been successfully detected by OpenOCD (*because OpenOCD returned tap/device found*).

Note : **OESdebug** defaults the adapter speed to 8 khz (*always use this speed if not sure about adapter's communication speed*).

OESdebug Extra features

OESdebug supports **Auto probing** (to get TAP ID and Intruction register length) and also **saving generated scripts** to share them easily (*see it's Help section*).

5.3 Linux tracers

Tracing is the opposite of security, if security wants to hide what's happening in the kernel then tracing does the complete opposite ; it shows every event.

5.3.1 Ftrace

Ftrace is the official linux tracing tool created by « **Steven Rostedt** »that has been merged to linux mainline since version *2.6.27*.

- **Trace-cmd :** Ftrace is quite tedious 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 37**)

```
jugurtha-VirtualBox module-debug-example # insmod basic-module-debug.ko
jugurtha-VirtualBox module-debug-example # mknod /dev/basictestdriver c 245 0
```

FIGURE 37 – 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 »)

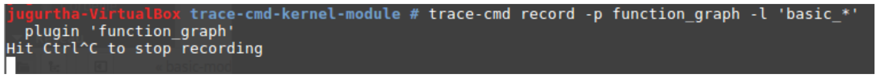


Figure 38 – Tracing functions in a module using trace-cmd

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 39**)

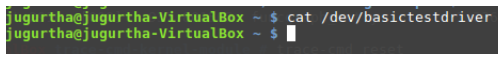


Figure 39 – Interacting with the kernel device module

4. **Reading report :** the trace file can be read as shown in **Figure 40**.

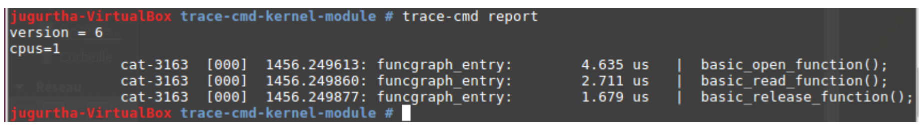


Figure 40 – Reading module trace report with Trace-cmd

- **Kernelshark :** We cannot close the discussion about Ftrace without pointing out an important tool called « Kernelshark ». Reading Ftrace report on a terminal can be quite difficult for interpretation; the third tool released by « **Steven Rostedt** »is KernelShark which is GUI parser for trace-cmd's traces (an example is shown in **Figure 41**).

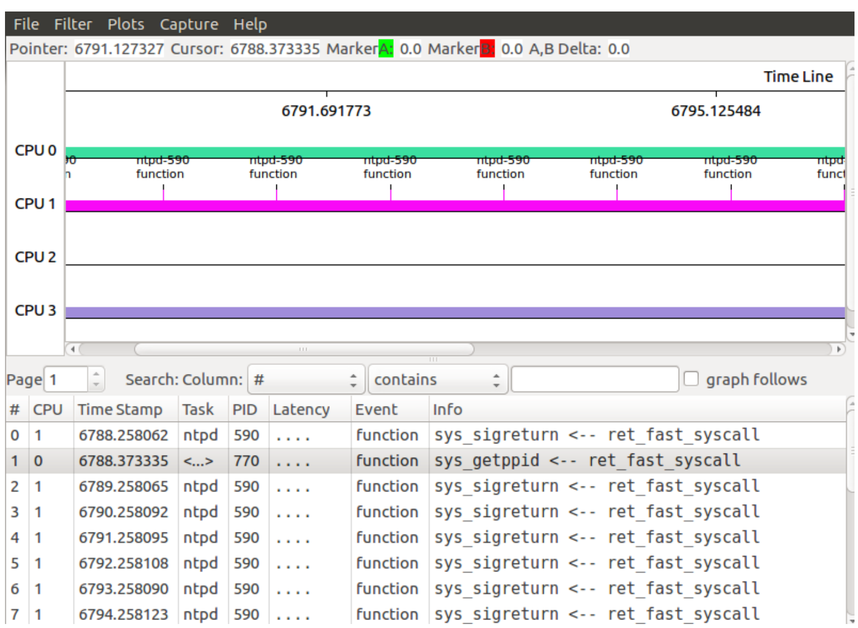


Figure 41 – Kernelshark shows process scheduling after parsing traces of a raspberry PI 3

5.3.2 LTTng

LTTng is an instrumentation toolkit created by **Mathieu Desnoyers** to trace kernel-land and userspace applications efficiently. Unlike Ftrace and Perf, LTTng has not been merged to the Linux mainline but can be easily installed :

```
$ apt-get install lttng-tools
$ apt-get install lttng-modules-dkms
$ apt-get install liblttng-ust-dev
```

An exhaustive documentation is available on LTTng's website : https://lttng.org/docs/v2.10/.

LTTng has the advantage of being able to collect events within the kernel (*no context switches are made*) which results in low system's overhead (*this is not the case for Ftrace and Perf*).

The simplest way to use LTTng can be illustrated as follow :

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

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

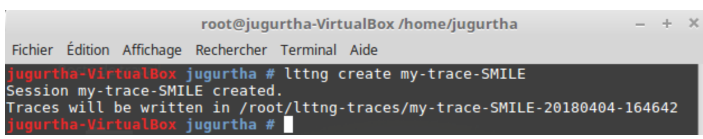


FIGURE 42 – Creating a session in LTTng

   Note : **LTTng** shows the location of where traces will be saved (***/root/lttng-traces/my-trace-SMILE-20180404-164642***).

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 » :

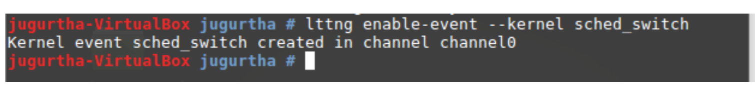


FIGURE 43 – Select kernel tracepoints in LTTng

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

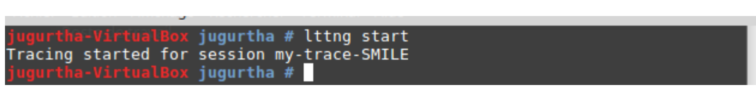


FIGURE 44 – Start tracing using LTTng

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

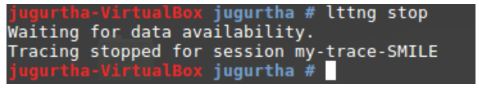


FIGURE 45 – Stop tracing using LTTng

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 as shown in **Figure 46**.

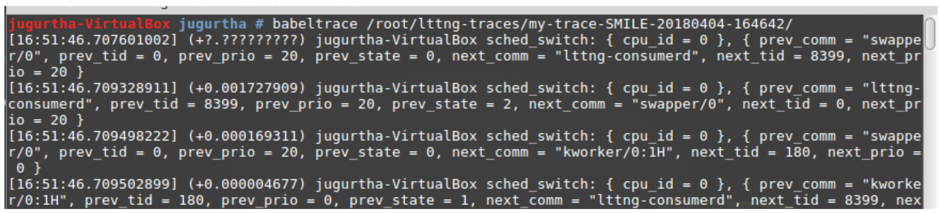


FIGURE 46 – Reading LTTng trace report using babeltrace

   However, when we record a lot of events for a long time, viewing results in text-based mode is far to be easy.
   - **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 (useful for debugging), however it is not reliable as **LTTng USDT**.
— **LTTng toolkit analyses :** LTTng provides a powerful 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 47**

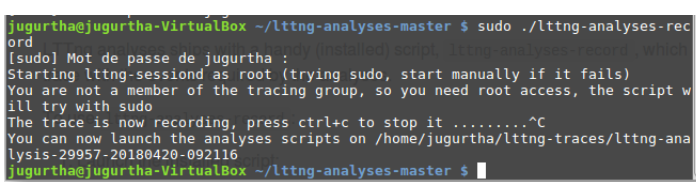


FIGURE 47 – Automatic session recording - LTTng toolkit analyses

— **lttng-schedlog :** shows task scheduling in chronological order (**Figure 48**).

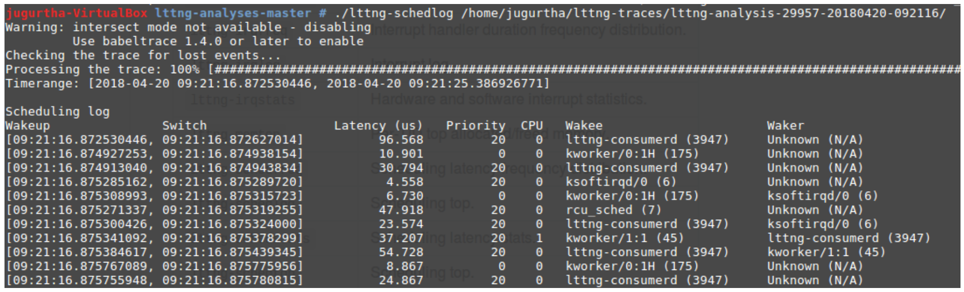


FIGURE 48 – Getting sched_switch logs from traces - LTTng toolkit analyses

5.3.3 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 stat* 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 :** general syntax is illustrated as follow :

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

An example is shown in **Figure 49**.

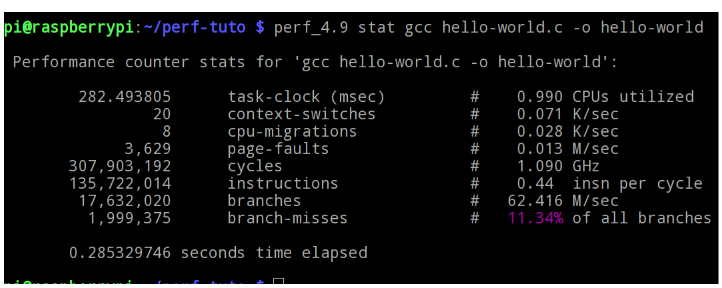


FIGURE 49 – Gather program's statistics - perf

  2. **Filtering returned statistics :** one may choose which statistics to view as shown in **Figure 50**.

```
pi@raspberrypi:~/perf-tuto $ perf_4.9 stat -e page-faults,branches gcc hello-world.c -o hello-world

 Performance counter stats for 'gcc hello-world.c -o hello-world':

             3,628      page-faults
        17,660,326      branches

       0.289320543 seconds time elapsed

pi@raspberrypi:~/perf-tuto $
```

FIGURE 50 – 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 51**.

```
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf/recordHoleSystem$ sudo perf record -
F 99 -ag -- sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.918 MB perf.data (57 samples) ]
jugbe@F-NAN-HIPPOPOTAME:~/Perf/profile-perf/recordHoleSystem$
```

FIGURE 51 – 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 illustrated in **Figure 52**.

```
  Children      Self  Command     Shared Object        Symbol
+   50,00%    50,00%  as          libc-2.23.so         [.] __memset_sse2
-   50,00%     0,00%  cc1         [kernel.kallsyms]    [k] handle_mm_fault
     handle_mm_fault
     alloc_pages_vma
-   50,00%     0,00%  cc1         [kernel.kallsyms]    [k] __do_page_fault
     __do_page_fault
     handle_mm_fault
     alloc_pages_vma
-   50,00%     0,00%  cc1         [kernel.kallsyms]    [k] do_page_fault
     do_page_fault
     __do_page_fault
     handle_mm_fault
     alloc_pages_vma
+   50,00%     0,00%  cc1         [kernel.kallsyms]    [k] page_fault
+   50,00%    50,00%  cc1         [kernel.kallsyms]    [k] alloc_pages_vma
+   50,00%     0,00%  cc1         cc1                  [.] _ZN3gcc12dump_manager13d
+    0,00%     0,00%  gcc         gcc-5                [.] 0xffffffffffc3b61b
```

FIGURE 52 – 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 :** Perf was designed for performance testing, but it has been extended to cover tracing.

1. **Choose a tracepoint :** tracepoints must be supported by perf as shown in **Figure 53**.

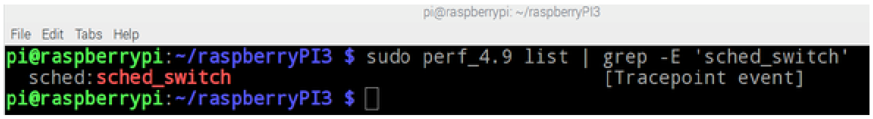


FIGURE 53 – Check tracepoint sched_switch support - perf

2. **Trace selected tracepoint events :** Launch our executable using perf (-e is used to select a tracepoint) as shown in **Figure 54**.

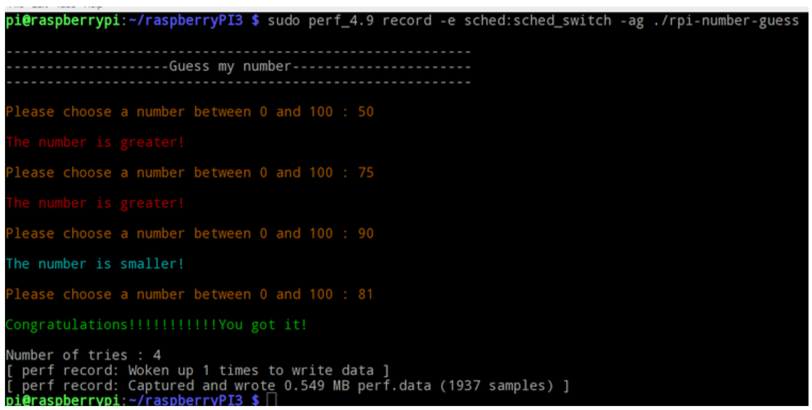


FIGURE 54 – trace sched_switch event - perf

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

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

An example is shown in **Figure 55**.

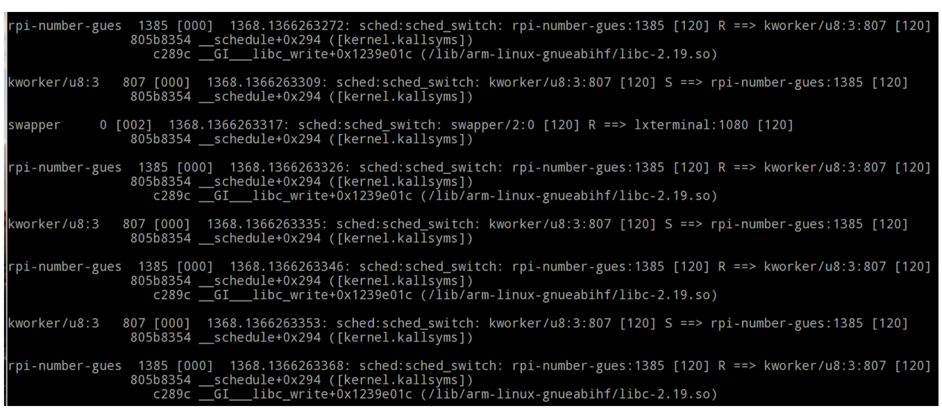


FIGURE 55 – Reading recorded sched_switch event - perf

- Hotspot : A **GUI tool** for **Perf**, able to parse « perf traces »or even make a entire perf session recording. An example is shown in **Figure 56**.

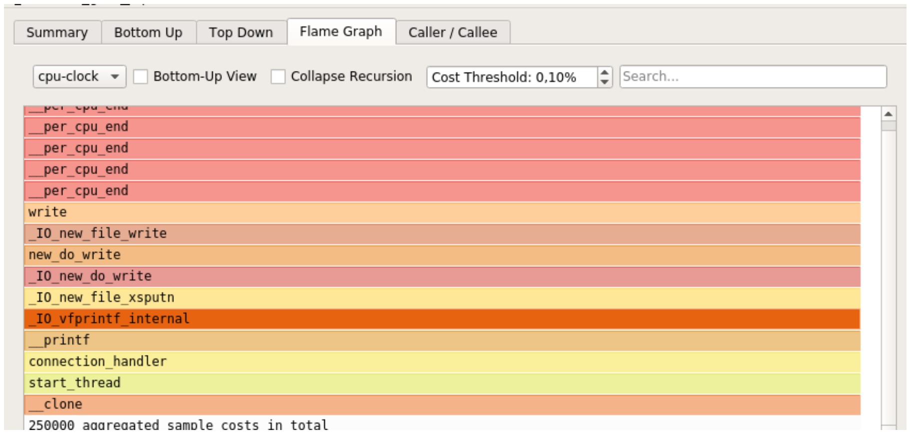


FIGURE 56 – Hotspot (a perf GUI tool) in action - generation of flame graphs

Flame graphs : Shows callgraph of stack frames. for instace, « _clone »frame calls « start_thread »frame in **Figure 56**

5.3.4 eBPF

BPF (*Berkeley Packet Filter*) is the famous virtual machine (*running inside the kernel*) used by **tcpdump** (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 for observability , DDos mitigation , Intrusion detection , Tracing , ..., etc (**Figure 57** - taken from Brenden Gregg's blog).

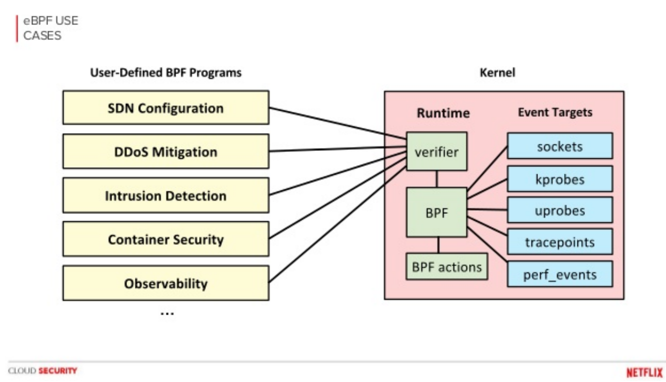


FIGURE 57 – Linux EBPF internal and usage

eBPF is difficult to use (*we must write a flavor 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 link : https://github.com/iovisor/bcc.

1. **Install BCC :** Installing 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 (Formal eBPF/bcc documentation can be found at : `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 58***).

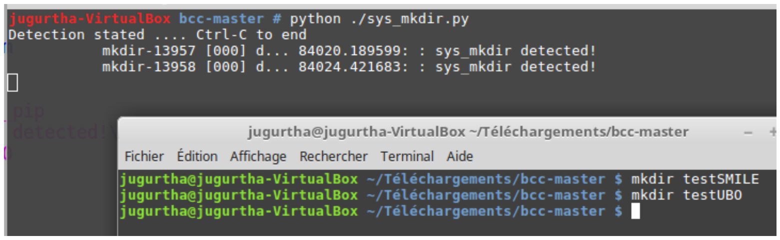


FIGURE 58 – Tapping sys_mkdir using eBPF - kprobe

eBPF is catching « sys_mkdir »events and reports them to end user.

- **Creating tracepoint :** an example is shown in **Appendix A.2** (*see **Figure 59***).

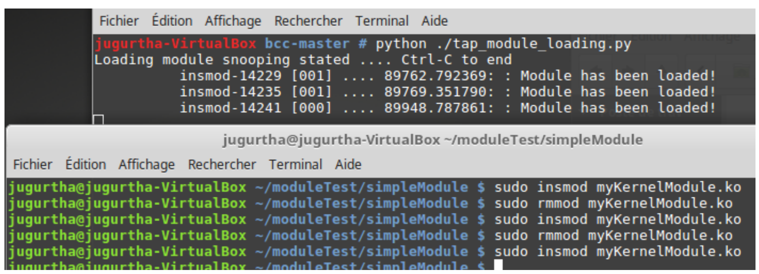


FIGURE 59 – Tapping module loading event using eBPF - tracepoint

eBPF has captured module's loading with success.

eBPF enhanced security

eBPF imposes restrictions on how programs should be written (*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*).

5.3.5 Choosing a tracer

The following table gives a quick summary of important features of some tracers :

Tool	Native support	Front-end tool	Remote tracing	GUI parsing tools	Real time tracing
Ftrace	since linux 2.6.27	Trace-cmd	yes	KernelShark	no
Perf	since linux 2.6.31	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 have made a simple benchmarking tool to help us in choosing the most appropriate. The benchmark measures **memory** (*captures Maximum Resident Set Size Memory*) and **execution time overhead** as well as other metrics (context switches and trace file size).

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 60**

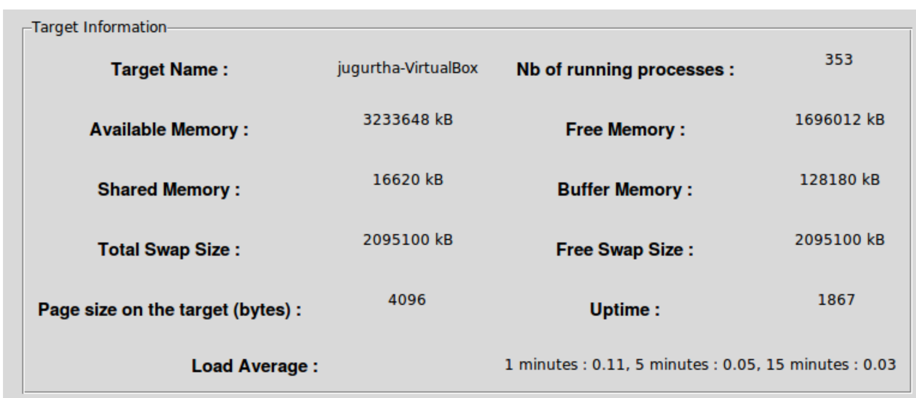


FIGURE 60 – Target's initial state before experiment - Linux Mint

Results are illustrated as follow :

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 **Linux-4.9** to access full functionalities (*or at least Linux4.4 for partial support*).

5.4 Defeating Anti debugging mechanisms

Security is a concern for every modern device, It has become crucial to keep the data safe and avoid them from leaking.

Anti-debugging stops debugging tools

We have covered a set of tools during this internship (gdb, strace, ltrace, ...,etc) ; however, Anti-debugging can make them completely useless. Reversing anti-debugging and changing how tools behave is required to successfully debug a target.

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.

5.4.1 Attacking userland and blocking GDB and strace

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 61**).

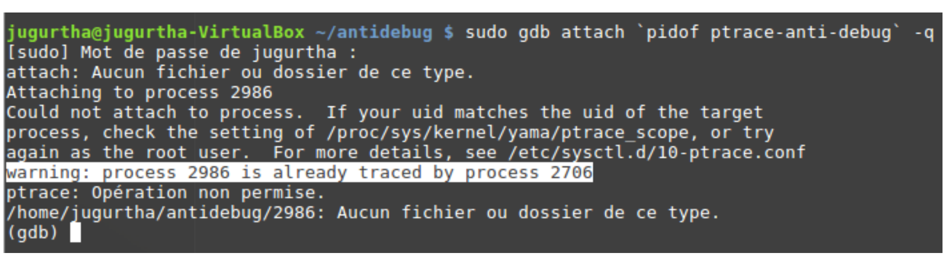


Figure 61 – GDB cannot attach to the program due to Anti-debugging

As one can see from **Figure 61**, GDB was not able to attach to the process (even if launched with root privileges).

Ptrace attacks are very basic and be defeated quickly :

* **Place a breakpoint :** in order to jump to a given location, the program must be running which means that we need at least one breakpoint. Let's place it at the beginning of the *ptrace* function and run the program (**Figure 62**).

```
jugurtha@jugurtha-VirtualBox ~/antidebug $ gdb -q ./ptrace-anti-debug
Reading symbols from ./ptrace-anti-debug...done.
(gdb) break ptrace
Breakpoint 1 at 0x4004d0
(gdb) run
Starting program: /home/jugurtha/antidebug/ptrace-anti-debug

Breakpoint 1, ptrace (request=PTRACE_TRACEME)
    at ../sysdeps/unix/sysv/linux/ptrace.c:36
36      ../sysdeps/unix/sysv/linux/ptrace.c: Aucun fichier ou dossier de ce type
.
(gdb) 
```

FIGURE 62 – Placing a breakpoint at the beginning of ptrace

* **Get the destination address :** retrieve the location of *getchar()* in memory and jump there using GDB (**Figure 63**).

```
(gdb) print getchar
$1 = {int (void)} 0x7ffff7a83160 <getchar>
(gdb) jump *0x7ffff7a83160
Line 34 is not in `ptrace'.  Jump anyway? (y or n) y
Continuing at 0x7ffff7a83160.
```

FIGURE 63 – Jump to getchar function location

Remark : We have forced our program to jump to memory location *0x7ffff7a83160*.

* **Carry on program execution :** At the moment that we made the jump, a blinking cursor was waiting for a character input (*this is the behaviour of getchar()*), We can provide it with a character as shown in **Figure 64**

```
h
No debugger detected
[Inferior 1 (process 2728) exited normally]
(gdb) 
```

FIGURE 64 – Reversing ptrace anti-debug

It is clear that by getting around ptrace can defeat it easily.

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)

5.4.2 Targeting Kernel Code and changing modules behaviour

The kernel can be subjected to many threats (like rootkits). We can change behaviour of almost any instruction in the kernel (*it's parameters 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-securi kernel/kprobes).

We are going to illustrate module tampering (*found at https://github.com/jugurthab/Linux_kernel_debug/tree/master/debug-examples/Chap6-kernel-security/kernel/module-tamp* as shown below :

1. **Merge modules :** ld assembles modules to produce a final one as shown in **Figure 65**.

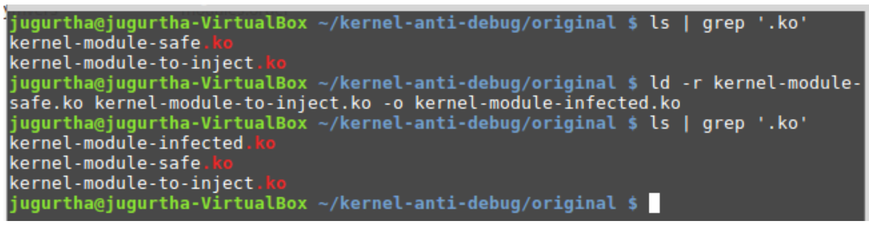


FIGURE 65 – Meging modules using ld

   kernel-module-infected.ko is merly a combination of **kernel-module-safe.ko** and **kernel-module-to-inject.ko**.

2. **Analyse the resulting module :** We can dump module's symbols table (objdump is helpful when debugging ELF files) of as follow :

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

kernel-module-infected.ko: format de fichier elf32-i386

SYMBOL TABLE:
...........
# init function of kernel-module-safe.ko
00000000 l F .init.text 00000014 mon_module_init
00000000 l F .exit.text 00000012 mon_module_cleanup
...........
...........
# init function of kernel-module-to-inject.ko
00000014 l F .init.text 00000014 fak_module_init
...........
...........
00000000 g F .exit.text 00000012 cleanup_module
00000000 g F .init.text 00000014 init_module
00000000 *UND* 00000000 printk
```

   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 location 00000014*).

3. **Make init_module as an alias of fak_module_evil :** init_module should point to our malicious function. One may do it by tampering the sections of **kernel-module-infected.ko** but a knowledge of ELF file format is required. A program called « elfchger »can

edit ELF files quickly and easily (https://github.com/jugurthab/Linux_kernel_debug/blob/master/debug-examples/Chap6-kernel-security/kernel/module-tampering/elfchger.c) 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 demonstrated in **Figure 66**

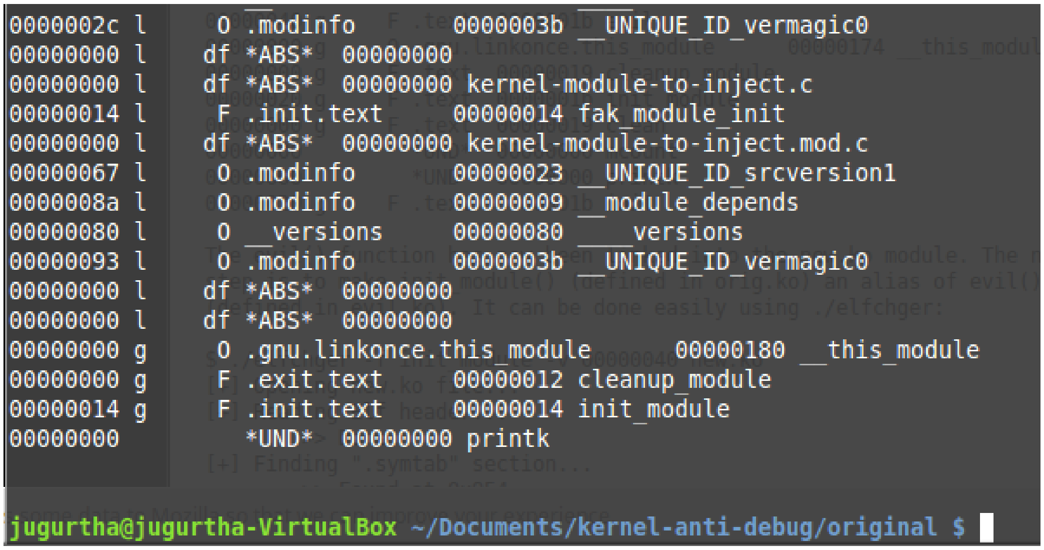


FIGURE 66 – Forcing init_module to become an alias of a malicious function

4. **Insert infected module into kernel :** After loading the buffer into the kernel, one can see that malicious function has been executed (see **Figure 67**).

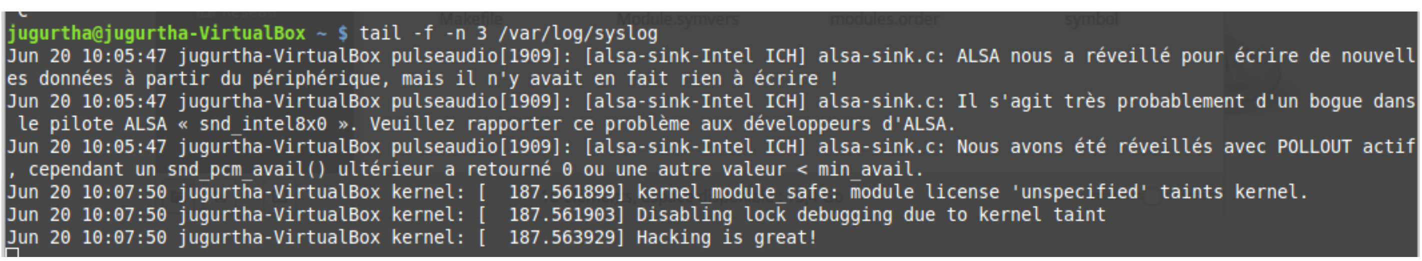


FIGURE 67 – Infected module executing malicious function

Once again, this has been detected easily using basic debugging commands tools like "objdump", "readelf" or even "objcopy".

6 Encountered difficulties

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

6.1 Hardware issues

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

6.1.1 JTAG tampering

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 (solutions like **JTAGulator** at : https://hackaday.com/2013/10/02/jtagulator-finds-debug-interfaces/ may be helpful).

6.1.2 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 (**Figure 68**).

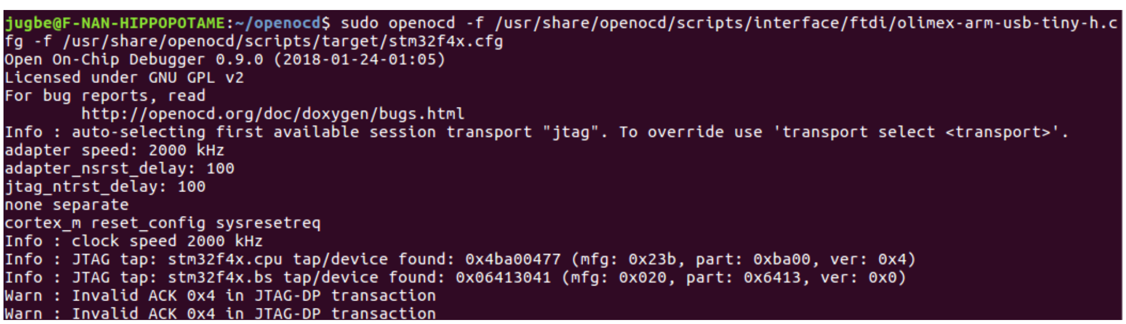


FIGURE 68 – OpenOCD ACK error due to incorrect TDI signal connection

6.1.3 OpenOCD's compliant adapter

Adapters are expensive, and those that are compatible with OpenOCD are difficult to find.
Solution : We used ARM-USB-TINY-H (https://www.olimex.com/Products/ARM/JTAG/ARM-USB-TINY-H
from Olimex.

6.2 Software

6.2.1 Debugging symbols

Most kernels in production are compiled removing this option. The advantage is to reduce kernel's image size, however, tools like : GDB becomes practically useless as they require debugging symbols (see **Figure 69**).

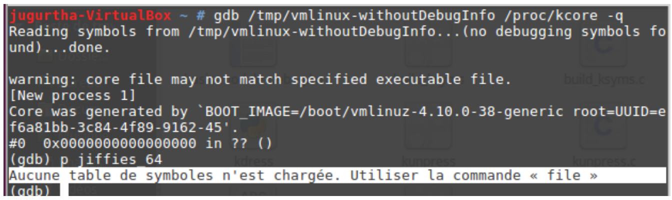


FIGURE 69 – GDB is practically useless without debugging symbols

Some solutions exist to reconstruct it (without recompiling the kernel) and 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].

6.2.2 Yama blocks ptrace

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

Solution : enable ptrace as shown in subsection 5.1.2

6.2.3 JTAG lockers

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

Solution : enable JTAG as shown in subsection 5.2.4

6.2.4 Disabled serial communication

Serial communication can be disabled on some devices, **Raspberry PI** is an example of those. It took me 3 hours to figure out the reason of unsuccessful connection (*even if hardware setup was correct* as shown in **Figure 70**).

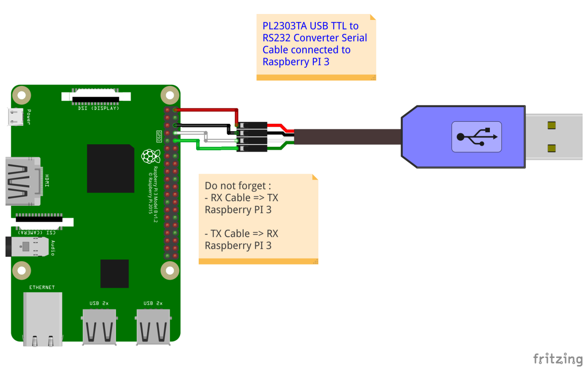


FIGURE 70 – Hardware setup for serial communication - Raspberry PI 3

Solution : To enable serial communication on Raspberry PI, follow the steps presented at : https://hallard.me/enable-serial-port-on-raspberry-pi/.

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

6.2.5 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`).

Solution : see subsection 5.2.4 (go to « *OpenOCD made easy with OESdebug* »).

6.2.6 DebugFs absent

Security engineers drop down **DebugFs** support as it allows anyone to get insight into the Kernel. Only hardware debugging can help in such cases.
We point the fact that tracers will be difficult to port and used as some rely heavily on **DebugFS**.

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

A solution like printf (*or printk*) works great with small codes, however can overwhelm a system with messages, making it slow with strange behaviour side effect (*or completely unresponsive*). Industrial projects goes beyond millions lines of code.

Personally, I have 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 developer. I had a lot of fun debugging Linux ; gathering performances, stack traces and I loved getting full control over a target using OpenOCD.

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()
```