

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	3
2	Internship objectives	4
3	Available equipments	5
3.1	Hardware	5
3.1.1	Beagle Bone Black Wireless	5
3.1.2	Raspberry PI 3 B+	5
3.1.3	stm32f407 Board :	5
3.1.4	AT32UC3C-EK Board :	6
3.1.5	ARM-USB-TINY-H JTAG Adapter :	6
3.2	Software	6
3.2.1	pycharm	6
3.2.2	Eclipse C/C++	7
4	Solutions	8
4.1	Userspace	8
4.1.1	Querying the filesystem	8
4.1.2	System calls and library calls	9
4.1.3	Valgrind	10
4.1.4	GDB and GDBserver	11
4.1.5	File core dump	14
4.2	Kernel land	15
4.2.1	KGDB/KDB	15
4.2.2	System faults	16
4.2.3	Core dump and Kernel panic	18
4.2.4	Linux hardware debugging with OpenOCD	19
4.3	Linux tracers	23
4.3.1	Ftrace	23
4.3.2	LTtng	24
4.3.3	Perf	26
4.3.4	eBPF	28
4.3.5	Choosing a tracer	30
4.4	Defeating Anti debugging mechanisms	31
4.4.1	Attacking userland	31
4.4.2	targeting Kernel Code	32
5	Encountered difficulties	34
5.1	Hardware issues	34
5.1.1	JTAG	34
5.1.2	OpenOCD hardware interfacing	34
5.2	Software	34
5.2.1	Debugging symbols	34
5.2.2	Yama blocks ptrace	34
5.2.3	JTAG lockers	34
5.2.4	OpenOCD scripts	35

5.2.5	Tracers in embedded systems	35
5.2.6	DebugFs absent	35
6	Conclusion	36
	Appendices	37
A	eBPF	37
A.1	Attaching eBPF kprobe	37
A.2	Enabling eBPF Tracepoint	37

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 proprietary 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 : [sfdld](#)

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 **LTng**.
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 equipments

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**).

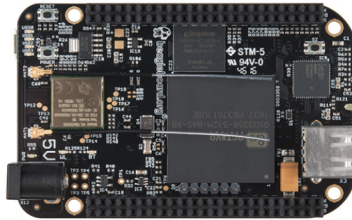


FIGURE 1 – Beaglebone black wireless

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

3.1.2 Raspberry PI 3 B+

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

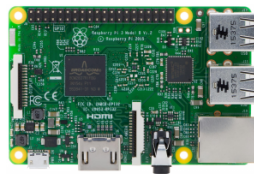


FIGURE 2 – Raspberry PI 3

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

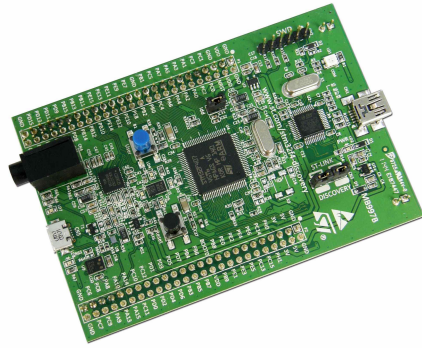


FIGURE 3 – stm32f407 Board

3.1.4 AT32UC3C-EK Board :

(**Figure 4**) see the following link for specifications : <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 mainly 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**.

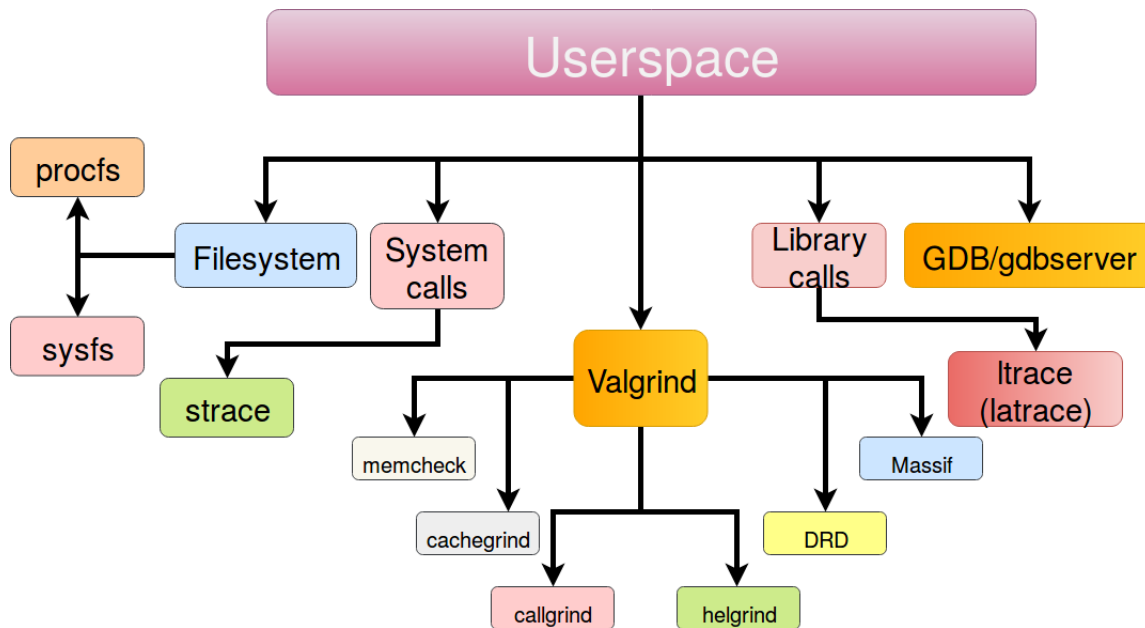


FIGURE 6 – Linux userspace debugging methodologies

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

```
1 $ sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

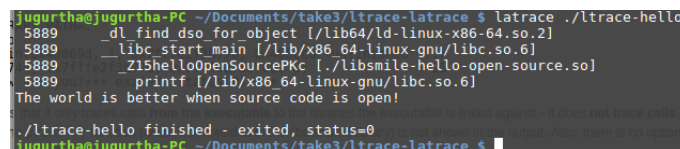
- **System calls (Syscalls)** : *strace* is a debugging and diagnostic tool. The « s » stands for « system call », which means that *strace* can monitor Syscalls and reports them to end users.

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 needs to use *latrace* (**Figure 7**).



```
jugurtha@jugurtha-PC ~/Documents/take3/ltrace-latrace $ latrace ./ltrace-hello
5889 _dlopen [./lib64/ld-linux-x86-64.so.2]
5889 _libc_start_main [./lib/x86_64-linux-gnu/libc.so.6]
5889 _Z15helloOpenSourcePKc [./libsmile-hello-open-source.so]
5889 _printf [./lib/x86_64-linux-gnu/libc.so.6]
The world is better when source code is open!
./ltrace-hello finished - exited, status=0
jugurtha@jugurtha-PC ~/Documents/take3/ltrace-latrace $
```

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

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

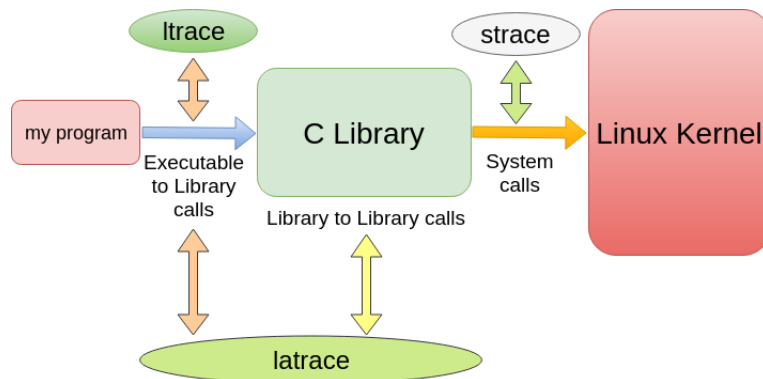


FIGURE 8 – Comparing between strace, ltrace and latrace

4.1.3 Valgrind

Valgrind is one of the most efficient memory debugging, instrumentation 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**) :

```

1 pi@raspberrypi:~/userspace/valgrind/memcheck$ valgrind --tool=memcheck \
2 > --leak-check=full ./memcheck-memory-leak
3 ==7369== Memcheck, a memory error detector
4 .....
5 ==7369== HEAP SUMMARY:
6 ==7369== in use at exit: 40 bytes in 1 blocks
7 ==7369== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
8 ==7369==
9 ==7369== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
10 ==7369== at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-
11 amd64-linux.so)
12 ==7369== by 0x40053C: main (memcheck-memory-leak.c:8)
13 ==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. 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 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. However, 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** :

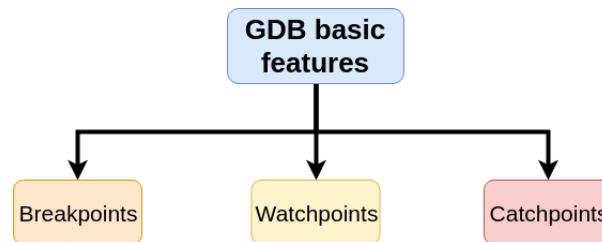


FIGURE 9 – Basic features of GDB

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

```

jugurtha@jugurtha-PC ~/Documents/take3/GDB/breakpoints-watch-points
Fichier Edition Affichage Rechercher Terminal Aide
jugurtha@jugurtha-PC ~/Documents/take3/GDB/breakpoints-watch-points $ gdb ./gdb-setting-breakpoi
nts -silent
Reading symbols from ./gdb-setting-breakpoints...done.
(gdb) break 16
Breakpoint 1 at 0x40061f: file gdb-setting-breakpoints.cpp, line 16.
(gdb) run
Starting program: /home/jugurtha/Documents/take3/GDB/breakpoints-watch-points/gdb-setting-breakp
oints
First number : 12
Second number : 5
Breakpoint 1, main () at gdb-setting-breakpoints.cpp:16
16      printf("The result : %d + %d = %d \n",firstNumber,secondNumber,result);
(gdb) continue
Continuing.
The result : 12 + 5 = 17
[Inferior 1 (process 10636) exited normally]
(gdb)
  
```

FIGURE 10 – Setting GDB breapoints

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

```

jugurtha@jugurtha-PC ~/Documents/take3/GDB/breakpoints-watch-points
Fichier Edition Affichage Rechercher Terminal Aide
jugurtha@jugurtha-PC ~/Documents/take3/GDB/breakpoints-watch-points $ gdb ./gdb-setting-breakpoi
nts -silent
Reading symbols from ./gdb-setting-breakpoints...done.
(gdb) break 11
Breakpoint 1 at 0x4005f3: file gdb-setting-breakpoints.cpp, line 11.
(gdb) run
Starting program: /home/jugurtha/Documents/take3/GDB/breakpoints-watch-points/gdb-setting-breakp
oints
Breakpoint 1, main () at gdb-setting-breakpoints.cpp:11
11      firstNumber = getNumberFromUser("First number : ");
(gdb) awatch result
Hardware access (read/write) watchpoint 2: result
(gdb) continue
Continuing.
First number : 7
Second number : 12
Hardware access (read/write) watchpoint 2: result
Old value = 0
New value = 19
main () at gdb-setting-breakpoints.cpp:16
16      printf("The result : %d + %d = %d \n",firstNumber,secondNumber,result);
(gdb)
  
```

FIGURE 11 – Setting a read watchpoint in GDB

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

```

jugurtha@jugurtha-PC ~/Documents/GDB-catchpoints
Fichier Edition Affichage Recherche Terminal Aide
jugurtha@jugurtha-PC ~/Documents/GDB-catchpoints $ gdb -q ./gdb-catchpoints
Reading symbols from ./gdb-catchpoints...done.
(gdb) catch fork
Catchpoint 1 (fork)
(gdb) start
Temporary breakpoint 2 at 0x40064e: file gdb-catchpoints.c, line 9.
Starting program: /home/jugurtha/Documents/GDB-catchpoints/gdb-catchpoints
Temporary breakpoint 2, main () at gdb-catchpoints.c:9
9      pid = fork ();
(gdb) c
Continuing.
Catchpoint 1 (forked process 5120), 0x0000ffff7ad941a in __libc_fork ()
at ../sysdeps/nptl/fork.c:145
145     ../sysdeps/nptl/fork.c: No such file or directory.
(gdb) c
Continuing.
Hello, I'm the child process!
I am process 5125 and my child's pid=5120!
[Inferior 1 (process 5125) exited normally]
(gdb)

```

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**).

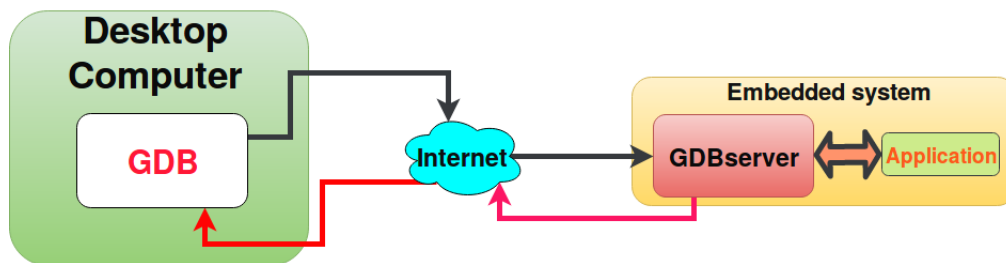


FIGURE 13 – Remote debugging using GDBserver

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

* **General settings of ethernet communication :**

1. GDBserver on the target

```
1 $ gdbserver :<portNumber> ./myProgram
```

2. GDB Client - Linux machine side

```
1 $ gdb-cross-platform ./myProgram
2 $ (gdb) target remote ip_address_gdbserver_machine:<portNumber>
```

* **General settings of serial communication :**

1. GDBserver on the target

```
1 $ gdbserver /dev/serial-channel ./myProgram
```

2. GDB Client - Linux machine side

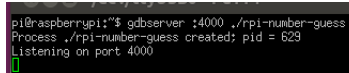
```
1 $ gdb-cross-platform ./myProgram
2 $ (gdb) target remote /dev/serial-channel
```

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-examples/Chap1-userland-debug/gdb/remote-debug/raspberryPI3*) :

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

```
1 $ gdbserver :4000 ./rpi-number-guess
```

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

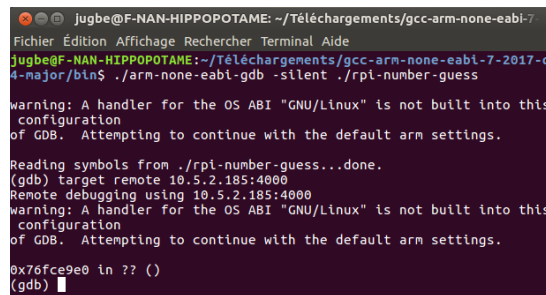


```
pi@raspberrypi:~$ gdbserver :4000 ./rpi-number-guess
Process ./rpi-number-guess created; pid = 629
Listening on port 4000
```

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**

```
1 $ ./arm-none-eabi-gdb -silent ./rpi-number-guess
2 $ (gdb) target remote ip_address_raspberryPI:4000
```



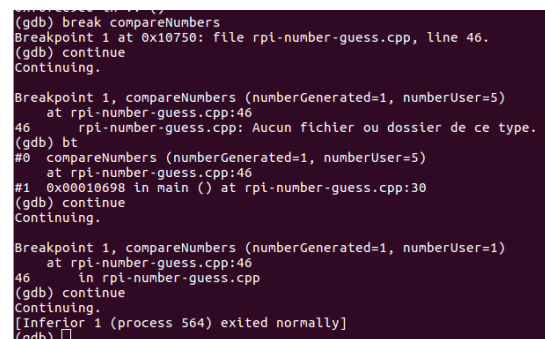
```
jugbe@F-NAN-HIPPOPOTAME: ~/Téléchargements/gcc-arm-none-eabi-7
Fichier Édition Affichage Rechercher Terminal Aide
jugbe@F-NAN-HIPPOPOTAME:~/Téléchargements/gcc-arm-none-eabi-7-2017-q
4-major/bin$ ./arm-none-eabi-gdb -silent ./rpi-number-guess
warning: A handler for the OS ABI "GNU/Linux" is not built into this
configuration
of GDB. Attempting to continue with the default arm settings.
Reading symbols from ./rpi-number-guess...done.
(gdb) target remote 10.5.2.185:4000
Remote debugging using 10.5.2.185:4000
warning: A handler for the OS ABI "GNU/Linux" is not built into this
configuration
of GDB. Attempting to continue with the default arm settings.
0x76fce9e0 in ?? ()
(gdb)
```

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

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

```
1 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**



```
(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 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 :

```
1 $ ulimit -c unlimited
```

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

```
1 $ ./myProgram
2 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 :

```
1 $ gdb ./myProgram <coreFile>
```

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

4. **Customizing 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 customize the naming convention 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 formatted 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 exist 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

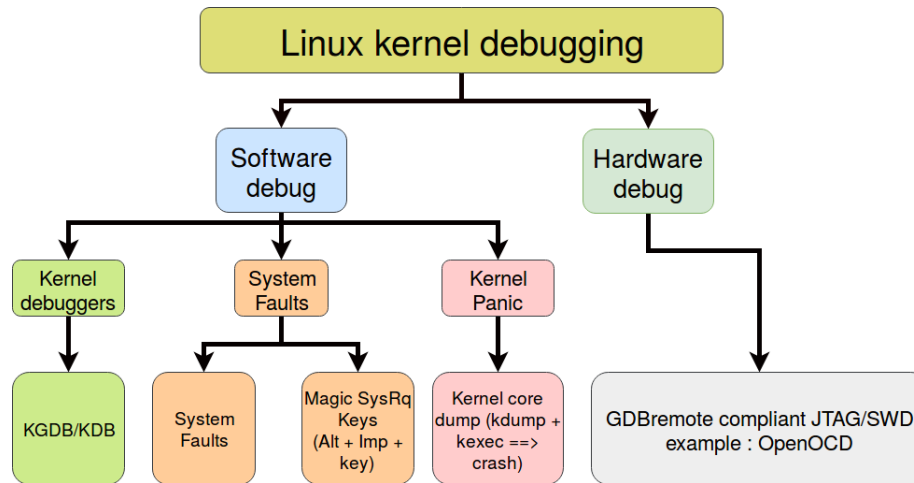


FIGURE 17 – Linux kernel debugging methodologies

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 :

```
1 CONFIG_FRAME_POINTER=y
2 CONFIG_KGDB=y
3 CONFIG_KGDB_SERIAL_CONSOLE=y
```

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

```
1 CONFIG_FRAME_POINTER=y
2 CONFIG_KGDB=y
3 CONFIG_KGDB_SERIAL_CONSOLE=y
4 CONFIG_KGDB_KDB=y
5 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 :

```

1 pi@raspberrypi:~# echo ttyAMA0 > /sys/module/kgdboc/parameters/kgdboc
2 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      1    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   0xdc0db3e8  rcuc/0
0xdab2c780     545     1  0      0  S   0xdab2cd68  systemd-journal
0xdab2e800     554     1  0      0  S   0xdab2c068  systemd-udevd
0xdab9f500     643     1  0      0  S   0xdab9fae8  systemd-timesyn
0xda94f500     658     1  0      0  S   0xda94fae8  sd-resolve
0xdacfee00     751     1  0      0  S   0xdacfd3e8  rsyslogd
0xdab8d480     779     1  0      0  S   0xdab8da68  initmuxsock
0xdab89380     780     1  0      0  S   0xdab89968  initmklog
0xdab8c100     781     1  0      0  S   0xdab8c6e8  rsainit Q:Reg
0xdacfb400     756     1  0      0  S   0xdacfb3e8  haveged
0xdacfee80     763     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://fdeszfffffffff>) :

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

```
001 110000 kernel oops: module verification failed: signature and/or required key
001 110000 Key SMILE!This is a kernel oops test module!!!
001 110000 end
001 110000 PC0 0
```

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** : 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 identifies them and reports us the reason that gave rise to them as shown in **Figure 21**.

```
001 110000 Oop: 0002 [#1] SMP
```

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**.

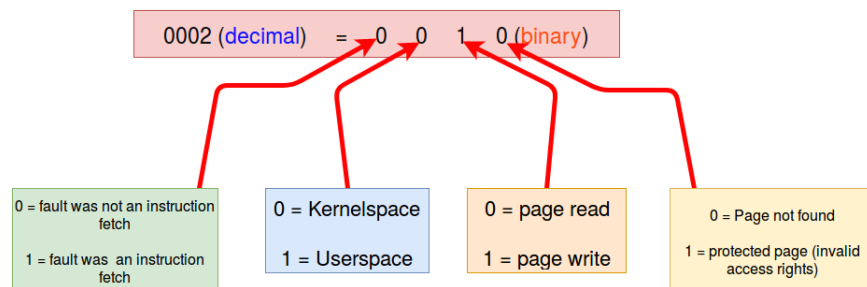


FIGURE 22 – Interpreting kernel oops error code

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>

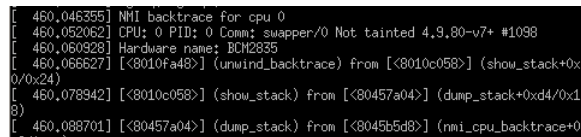
SysRq involves QWERTY Keyboard

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 :

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

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



```
[ 460.046355] NMI backtrace for cpu 0
[ 460.052062] CPU: 0 PID: 0 Comm: swapper/0 Not tainted 4.9.80-v7+ #1098
[ 460.060928] Hardware name: BCM2835
[ 460.066627] [<8010fa48>] (unwind_backtrace) from [<8010c058>] (show_stack+0x20/0x24)
[ 460.078942] [<8010c058>] (show_stack) from [<80457a04>] (dump_stack+0xd4/0x118)
[ 460.088701] [<80457a04>] (dump_stack) from [<8045b5d8>] (nmi_cpu_backtrace+0x0/0x0)
```

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>. However, 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 analysis /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 :

```
1 # 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**).

```
jugbe@F--NAN--HIPPOPOTAME:~/Téléchargements/kdress-master$ sudo gdb -q vmlinux /proc/kcore
Reading symbols from vmlinux...(no debugging symbols found)...done.
[New process 1]
Core was generated by 'BOOT_IMAGE=vmlinuz-4.4.0-121-generic root=/dev/mapper/F--NAN--HIPPOPOTAME--vg-'.
#0  0x0000000000000000 in ?? ()
(gdb) p jiffies_64
$1 = 7017029
(gdb) p &sys_call_table
$2 = (<data variable, no debug info> *) 0xffffffff81a00200 <sys_call_table>
(gdb) p &sys_close
$3 = (<text variable, no debug info> *) 0xffffffff81210e40 <sys_close>
(gdb) x/5i 0xffffffff81210e40
0xffffffff81210e40 <sys_close>:    add    %cl,-0x77(%rax)
0xffffffff81210e43 <sys_close+3>:  retq   $0x2949
0xffffffff81210e46 <sys_close+6>:  rorb   $0x49,-0x1e(%rcx,%rbp,1)
0xffffffff81210e4b <sys_close+11>:  cmp    %eax,%esp
0xffffffff81210e4d <sys_close+13>:  cmovle %r12,%rax
(gdb) □
```

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 utility 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 :

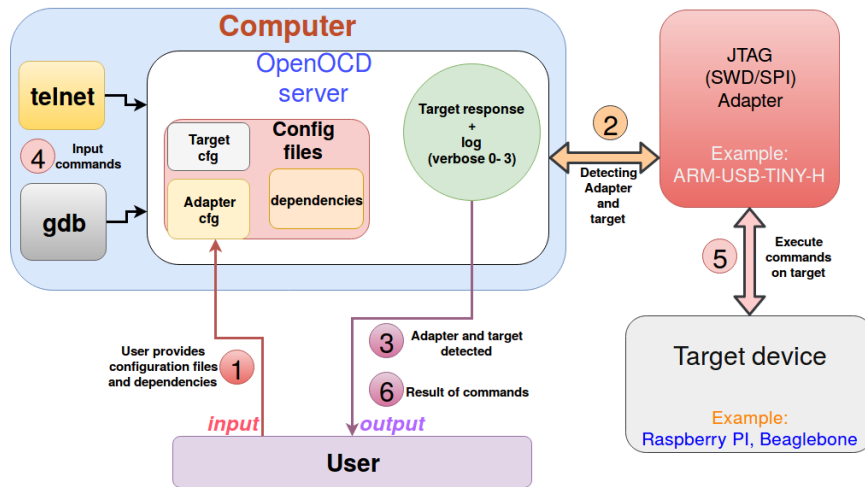


FIGURE 25 – OpenOCD general settings

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

2. General syntaxe Of OpenOCD :

```
1 $ sudo ./src/openocd -s tcl/ -f tcl/interface/adapter_config_file.cfg \
2 > -f tcl/target/target_config_file.cfg
```

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

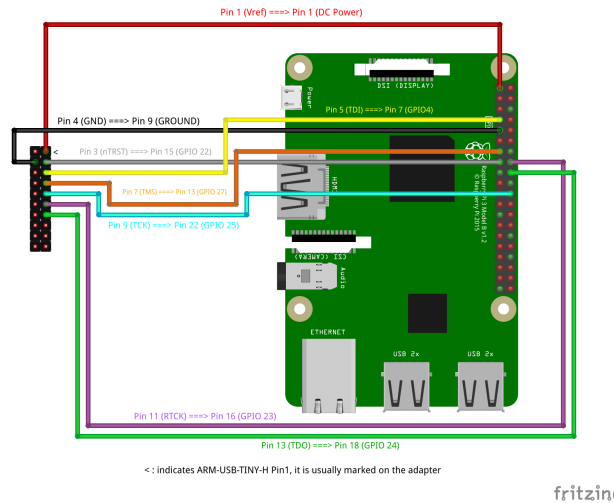


FIGURE 26 – Connecting OpenOCD to Raspberry PI 3

- 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 :

```
1 #define BCM2708_PERI_BASE 0x3F000000
2 #define GPIO_BASE      (BCM2708_PERI_BASE + 0x2000000)
3
```

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

```
root@raspberrypi:/home/pi/w/jtag/wnt# g++ JtagEnabler.cpp -o JtagEnabler
root@raspberrypi:/home/pi/w/jtag/wnt# ./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/w/jtag/wnt#
```

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**

```
jugbe@F-NAN-HIPPOPOATME:~/openocd$ sudo ./src/openocd -s tcl/ -f tcl/interface/ftdi/olimex-arm-usb-tiny-h.cfg -f tcl/target/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 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) :

```
1 # 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.

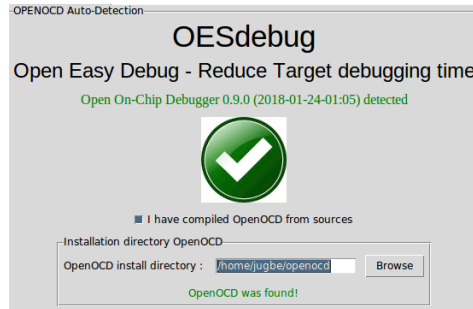


FIGURE 29 – Checking OpenOCD support - OESdebug

- (c) **Adapter Support :** an adapter is the intermediate component that allows OpenOCD (running as a daemon 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 »).

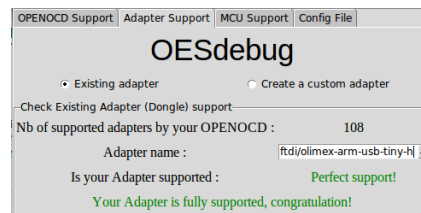


FIGURE 30 – Checking Adapter support - OESdebug

- (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**).

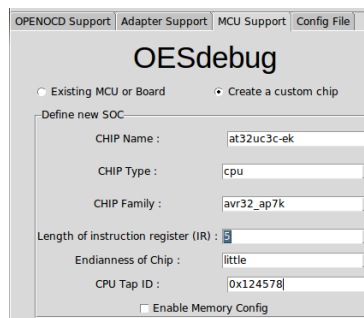


FIGURE 31 – Creating a new target config file - OESdebug

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

```

OPENOCD Support | Adapter Support | MCU Support | Config File
-----
OPENOCD Configuration file
#
# ----- Auto generated file by OESdebug -----
#
# ----- Adapter speed settings -----
adapter_khz 8
# ----- Define the interface -----
source [find interface/ftdi/olimex-arm-usb-tiny-h.cfg]
#
# --- at32uc3c-ek CHIP Settings ---
#
set CHIPNAME at32uc3c-ek
set ENDIAN little
set _CPUTAPID 0x124578
# ----- Create a tap ID controller -----
jtag newtap $CHIPNAME cpu -irlen 5 -expected-id $CPUTAPID
set _TARGETNAME $CHIPNAME.cpu
target create $TARGETNAME avr32 ap7k -chain-position $TARGETNAME
#
# ----- END OF Config File -----
#

```

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 :

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

And for reading events :

```
1 # trace-cmd report
```

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

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

```

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

```

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 »)

```

jugartha-VirtualBox trace-cmd-kernel-module # trace-cmd record -p function_graph -l 'basic_*'
plugin 'function_graph'
hit Ctrl+C to stop recording

```

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**)


```
jugurtha@jugurtha-VirtualBox ~ $ cat /dev/basictestdriver
jugurtha@jugurtha-VirtualBox ~ $
```

FIGURE 35 – Interacting with the kernel device module

4. Reading report : (Figure 36)

```
jugurtha-VirtualBox trace-cmd-kernel-module # trace-cmd report
version = 6
cpus=1
cat-3163 [000] 1456.249613: funcgraph_entry: 4.635 us | basic_open_function();
cat-3163 [000] 1456.249869: funcgraph_entry: 2.711 us | basic_read_function();
cat-3163 [000] 1456.249877: funcgraph_entry: 1.679 us | basic_release_function();
jugurtha-VirtualBox trace-cmd-kernel-module #
```

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 quite 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) :

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

```
root@jugurtha-VirtualBox /home/jugurtha
Fichier Édition Affichage Rechercher Terminal Aide
jugurtha-VirtualBox jugurtha # lttng create my-trace-SMILE
Session my-trace-SMILE created.
Traces will be written in /root/lttng-traces/my-trace-SMILE-20180404-164642
jugurtha-VirtualBox jugurtha #
```

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 » :

```
jugurtha-VirtualBox jugurtha # lttng enable-event --kernel sched_switch
Kernel event sched_switch created in channel channel0
jugurtha-VirtualBox jugurtha #
```

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

```
jugurtha-VirtualBox jugurtha # lttng start
Tracing started for session my-trace-SMILE
jugurtha-VirtualBox jugurtha #
```

FIGURE 39 – Start tracing using LTTng

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

```
jugurtha-VirtualBox jugurtha # lttng stop
Waiting for data availability.
Tracing stopped for session my-trace-SMILE
jugurtha-VirtualBox jugurtha #
```

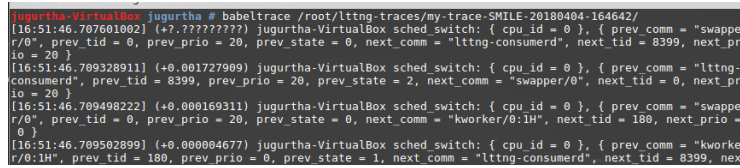
FIGURE 40 – Stop tracing using LTTng

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

```
1 # 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



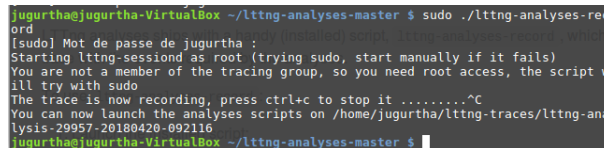
```
jugurtha-VirtualBox jugurtha # babeltrace /root/lttng-traces/my-trace-SMILE-20180404-164642/
[16:51:46.707601002] (+7.777777777) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "swapper/0", prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "lttng-consumerd", next_tid = 8399, next_prio = 20 }
[16:51:46.709328911] (+0.001727909) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "lttng-consumerd", prev_tid = 8399, prev_prio = 20, prev_state = 2, next_comm = "swapper/0", next_tid = 0, next_prio = 20 }
[16:51:46.709498222] (+0.000169311) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "swapper/0", prev_tid = 0, prev_prio = 20, prev_state = 0, next_comm = "kworker/0:1H", next_tid = 180, next_prio = 0 }
[16:51:46.709502899] (+0.000004677) jugurtha-VirtualBox sched_switch: { cpu_id = 0 }, { prev_comm = "kworker/0:1H", prev_tid = 180, prev_prio = 0, prev_state = 1, next_comm = "lttng-consumerd", next_tid = 8399, next_prio = 20 }
```

FIGURE 41 – Reading LTTng trace report using babeltrace

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

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** daemon 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**



```
jugurtha@jugurtha-VirtualBox ~/lttng-analyses-master $ sudo ./lttng-analyses-record
[sudo] Mot de passe de jugurtha :
Starting lttng-sessiond as root (trying sudo, start manually if it fails)
You are not a member of the tracing group, so you need root access, the script will try with sudo
The trace is now recording, press ctrl+c to stop it .....^C
You can now launch the analyses scripts on /home/jugurtha/lttng-traces/lttng-analyses-29957-20180420-092116
jugurtha@jugurtha-VirtualBox ~/lttng-analyses-master $
```

FIGURE 42 – Automatic session recording - LTTng toolkit analyses

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

```

jugurtha@VirtualBox: ltng-analyses-master # ./ltng-schedlog /home/jugurtha/ltng-traces/ltng-analysis-29957-20180420-092116/
Warning: intersect mode not available - disabling sample hardware and/or frequency distribution
Use babeltrace 1.4.0 or later to enable
Checking the trace for lost events...
Processing the trace: 100% [#####]
Timerange: [2018-04-20 09:21:16.872530446, 2018-04-20 09:21:25.386926771]

ltng-traces: Hardware and software interrupt statistics
Scheduling log
Wakeup      Switch      Latency (us)  Priority  CPU  Wakee      Waker
[09:21:16.872530446, 09:21:16.872627014] 96.568      20      0      ltng-consumerd (3947) Unknown (N/A)
[09:21:16.874927253, 09:21:16.874938154] 10.901      0      0      kworker/0:1H (175) Unknown (N/A)
[09:21:16.874913040, 09:21:16.874943834] 30.794      20      0      ltng-consumerd (3947) Unknown (N/A)
[09:21:16.875285162, 09:21:16.875289720] 4.558      20      0      ksoftirqd/0 (6) Unknown (N/A)
[09:21:16.875308993, 09:21:16.875315723] 6.730      0      0      kworker/0:1H (175) ksoftirqd/0 (6)
[09:21:16.875271337, 09:21:16.875319255] 47.918      20      0      rcu_sched (7) Unknown (N/A)
[09:21:16.875300426, 09:21:16.875324000] 23.574      20      0      ltng-consumerd (3947) ksoftirqd/0 (6)
[09:21:16.875341092, 09:21:16.875378299] 37.207      20      1      kworker/1:1 (45) ltng-consumerd (3947)
[09:21:16.875384617, 09:21:16.875439345] 54.728      20      0      ltng-consumerd (3947) kworker/1:1 (45)
[09:21:16.875767089, 09:21:16.875775956] 8.867      0      0      kworker/0:1H (175) Unknown (N/A)
[09:21:16.875755948, 09:21:16.875780815] 24.867      20      0      ltng-consumerd (3947) Unknown (N/A)

```

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 :

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

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

```

pi@raspberrypi:~/perf-tuto $ perf_4.9 stat gcc hello-world.c -o hello-world
Performance counter stats for 'gcc hello-world.c -o hello-world':

      282.493805      task-clock (msec)      #    0.990 CPUs utilized
           20          context-switches      #    0.071 K/sec
           8          cpu-migrations        #    0.028 K/sec
          3,629        page-faults          #    0.013 M/sec
    307,903,192        cycles                #    1.090 GHz
    135,722,014        instructions         #    0.44 insn per cycle
     17,632,020        branches             #   62.416 M/sec
       1,999,375       branch-misses        #   11.34% of all branches

0.285329746 seconds time elapsed

```

FIGURE 44 – Gather program's statistics - perf

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

```

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 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 :

```

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

```

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

```

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

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

```

1 $ 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**.

	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 47 – Displaying Perf records in Basic mode

Remark : We can display a tree view report using :

```

1 $ 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**.

```

pi@raspberrypi:~/raspberrypi3
File Edit Tabs Help
pi@raspberrypi:~/raspberrypi3 $ sudo perf_4.9 list | grep -E 'sched_switch'
sched:sched_switch [Tracepoint event]
pi@raspberrypi:~/raspberrypi3 $

```

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**.

```

pi@raspberrypi:~/raspberrypi3 $ sudo perf_4.9 record -e sched:sched_switch -ag ./rpi-number-guess
-----Guess my number-----
Please choose a number between 0 and 100 : 50
The number is greater!
Please choose a number between 0 and 100 : 75
The number is greater!
Please choose a number between 0 and 100 : 90
The number is smaller!
Please choose a number between 0 and 100 : 81
Congratulations!!!!!!You got it!
Number of tries : 4
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.549 MB perf.data (1937 samples) ]
pi@raspberrypi:~/raspberrypi3 $

```

FIGURE 49 – trace sched_switch event - perf

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

```
1 $ sudo perf script [-i Path_to_perf.data]
```

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

```

rpi-number-gues 1385 [000] 1368.1366263272: sched:sched_switch: rpi-number-gues:1385 [120] R ==> kworker/u8:3:807 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
c289c __GI___libc_write+0x1239e01c (/lib/arm-linux-gnueabi/libc-2.19.so)
kworker/u8:3 807 [000] 1368.1366263309: sched:sched_switch: kworker/u8:3:807 [120] S ==> rpi-number-gues:1385 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
swapper 0 [002] 1368.1366263317: sched:sched_switch: swapper/2:0 [120] R ==> lxterminal:1080 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
rpi-number-gues 1385 [000] 1368.1366263326: sched:sched_switch: rpi-number-gues:1385 [120] R ==> kworker/u8:3:807 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
c289c __GI___libc_write+0x1239e01c (/lib/arm-linux-gnueabi/libc-2.19.so)
kworker/u8:3 807 [000] 1368.1366263335: sched:sched_switch: kworker/u8:3:807 [120] S ==> rpi-number-gues:1385 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
rpi-number-gues 1385 [000] 1368.1366263346: sched:sched_switch: rpi-number-gues:1385 [120] R ==> kworker/u8:3:807 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
c289c __GI___libc_write+0x1239e01c (/lib/arm-linux-gnueabi/libc-2.19.so)
kworker/u8:3 807 [000] 1368.1366263353: sched:sched_switch: kworker/u8:3:807 [120] S ==> rpi-number-gues:1385 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
rpi-number-gues 1385 [000] 1368.1366263368: sched:sched_switch: rpi-number-gues:1385 [120] R ==> kworker/u8:3:807 [120]
805b8354 __schedule+0x294 ([kernel.kallsyms])
c289c __GI___libc_write+0x1239e01c (/lib/arm-linux-gnueabi/libc-2.19.so)

```

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 observability, DDos mitigation, Intrusion detection, Tracing, ..., etc (**Figure 51** - taken from Brenden Gregg's blog)

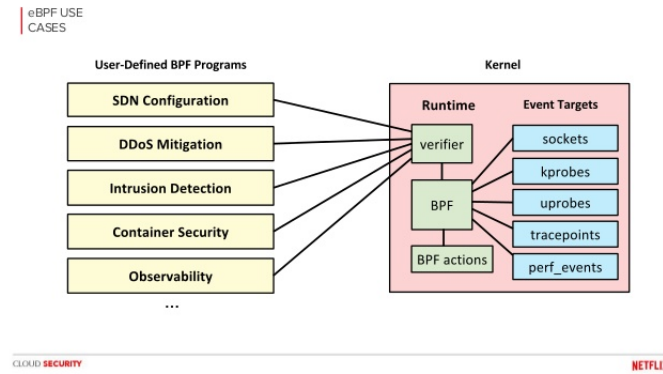


FIGURE 51 – Linux eBPF internal and usage

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**).

```
jugurtha-VirtualBox bcc-master # python ./sys_mkdir.py
Detection stated .... Ctrl-C to end
mkdir-13957 [000] d... 84020.189599: : sys_mkdir detected!
mkdir-13958 [000] d... 84024.421683: : sys_mkdir detected!
```

```
jugurtha@jugurtha-VirtualBox ~/Téléchargements/bcc-master
Fichier Édition Affichage Rechercher Terminal Aide
jugurtha@jugurtha-VirtualBox ~/Téléchargements/bcc-master $ mkdir testSMILE
jugurtha@jugurtha-VirtualBox ~/Téléchargements/bcc-master $ mkdir testUB0
jugurtha@jugurtha-VirtualBox ~/Téléchargements/bcc-master $
```

FIGURE 52 – Tapping sys_mkdir using eBPF - kprobe

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

```
Fichier Édition Affichage Rechercher Terminal Aide
jugurtha-VirtualBox bcc-master # python ./tap_module_loading.py
Loading module snooping stated .... Ctrl-C to end
insmod-14229 [001] .... 89762.792369: : Module has been loaded!
insmod-14235 [001] .... 89769.351790: : Module has been loaded!
insmod-14241 [000] .... 89948.787861: : Module has been loaded!
```

```
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule
Fichier Édition Affichage Rechercher Terminal Aide
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule $ sudo insmod myKernelModule.ko
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule $ sudo rmmod myKernelModule.ko
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule $ sudo insmod myKernelModule.ko
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule $ sudo rmmod myKernelModule.ko
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule $ sudo insmod myKernelModule.ko
jugurtha@jugurtha-VirtualBox ~/moduleTest/simpleModule $
```

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**

Target Information			
Target Name :	jugurtha-VirtualBox	Nb of running processes :	353
Available Memory :	3233648 kB	Free Memory :	1696012 kB
Shared Memory :	16620 kB	Buffer Memory :	128180 kB
Total Swap Size :	2095100 kB	Free Swap Size :	2095100 kB
Page size on the target (bytes) :	4096	Uptime :	1867
Load Average :	1 minutes : 0.11, 5 minutes : 0.05, 15 minutes : 0.03		

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 fonctionnalités (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 :

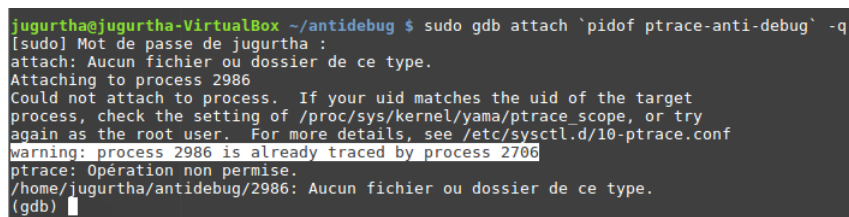
```

1 if (ptrace(PTRACE_TRACEME , 0) < 0 ) {
2     printf("You cannot debug me!\n");
3     exit(EXIT_FAILURE);
4 }
5

```

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**).



```

jugurtha@jugurtha-VirtualBox ~/antidebug $ sudo gdb attach `pidof ptrace-anti-debug` -q
[sudo] Mot de passe de jugurtha :
attach: Aucun fichier ou dossier de ce type.
Attaching to process 2986
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 2986 is already traced by process 2706
ptrace: Opération non permise.
/home/jugurtha/antidebug/2986: Aucun fichier ou dossier de ce type.
(gdb)

```

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 parameters and return value*), and cause serious system issues that goes from simple **Denial of Services to stealing 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**.

```
jugurtha@jugurtha-VirtualBox ~/kernel-anti-debug/original $ ls | grep '.ko'
kernel-module-safe.ko
kernel-module-to-inject.ko
jugurtha@jugurtha-VirtualBox ~/kernel-anti-debug/original $ ld -r kernel-module-
safe.ko kernel-module-to-inject.ko -o kernel-module-infected.ko
jugurtha@jugurtha-VirtualBox ~/kernel-anti-debug/original $ ls | grep '.ko'
kernel-module-infected.ko
kernel-module-safe.ko
kernel-module-to-inject.ko
jugurtha@jugurtha-VirtualBox ~/kernel-anti-debug/original $
```

FIGURE 56 – Merging modules using ld

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

```
1 $ 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.

```
1 ./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**

```

0000002c l 0 .modinfo 0000003b __UNIQUE_ID_vermagic0
00000000 l df *ABS* 00000000 this module
00000000 l df *ABS* 00000000 kernel-module-to-inject.c
00000014 l F .init.text 00000014 fak_module_init
00000000 l df *ABS* 00000000 kernel-module-to-inject.mod.c
00000067 l 0 .modinfo 00000023 __UNIQUE_ID_srcversion1
0000008a l 0 .modinfo 00000009 module_depends
00000080 l 0 .versions 00000080 versions
00000093 l 0 .modinfo 0000003b __UNIQUE_ID_vermagic0
00000000 l df *ABS* 00000000 module.The
00000000 l df *ABS* 00000000 line of evils
00000000 l df *ABS* 00000000 disabling /etc/passwd
00000000 l 0 .gnu.linkonce.this_module 00000180 __this_module
00000000 g F .exit.text 00000012 cleanup_module
00000014 g F .init.text 00000014 init_module
00000000 *UND* 00000000 printk
jugurtha@jugurtha-VirtualBox ~/Documents/kernel-anti-debug/original $

```

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

4. Insert infected module into kernel : see Figure 58

```

jugurtha@jugurtha-VirtualBox ~$ tail -f -n 3 /var/log/syslog
Jun 20 10:05:47 jugurtha-VirtualBox pulseaudio[1909]: [alsa-sink-Intel ICH] alsa-sink.c: ALSA nous a réveillé pour écrire de nouvelles données à partir du périphérique, mais il n'y avait en fait rien à écrire !
Jun 20 10:05:47 jugurtha-VirtualBox pulseaudio[1909]: [alsa-sink-Intel ICH] alsa-sink.c: Il s'agit très probablement d'un bogue dans le pilote ALSA « snd_intel8x0 ». Veuillez rapporter ce problème aux développeurs d'ALSA.
Jun 20 10:05:47 jugurtha-VirtualBox pulseaudio[1909]: [alsa-sink-Intel ICH] alsa-sink.c: Nous avons été réveillés avec POLLOUT actif, cependant un snd_pcm_avail() ultérieur a retourné 0 ou une autre valeur < min avail.
Jun 20 10:07:50 jugurtha-VirtualBox kernel: [ 187.561899] kernel module safe: module license 'unspecified' taints kernel.
Jun 20 10:07:50 jugurtha-VirtualBox kernel: [ 187.561903] Disabling lock debugging due to kernel taint
Jun 20 10:07:50 jugurtha-VirtualBox kernel: [ 187.563929] Hacking is great!

```

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 taught me that we need more than coding skills to be a good developer. 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

```
1 from bcc import BPF
2
3 # prog will store the eBPF C program
4 prog = """
5 int detect(void *ctx){
6     // write message into trace_pip
7     bpf_trace_printk("sys_mkdir detected!\n");
8     return 0; // always return 0
9 }
10 """
11
12 # Loads eBPF program
13 b = BPF(text=prog)
14
15 # Attach kprobe to kernel function and sets ... as jprobe handler
16 b.attach_kprobe(event="sys_mkdir", fn_name="detect")
17
18 # Show message when eBPF stats
19 print("Detection started... Ctrl-C to end")
20
21 # print result to user
22 while 1:
23     # read messages from trace_pip and display them to user
24     b.trace_print()
```

A.2 Enabling eBPF Tracepoint

```
1 from bcc import BPF
2
3 # prog will store the eBPF C program
4 prog = """
5 TRACEPOINT_PROBE(module, module_load){
6     // events are from /sys/kernel/debug/tracing/events/module/module_load/fo
7     bpf_trace_printk("Module has been loaded!\n");
8     return 0; // always return 0
9 };
10 """
11
12 # Loads eBPF program
13 b = BPF(text=prog)
```

```
14
15 # Show message when ePBF stats
16 print("Loading module snooping stated.... Ctrl-C to end")
17
18 # print result to user
19 while 1:
20     # read messages from trace_pip and display them to user
21     b.trace_print()
```