Hack In Paris 2019

Introduction To IoT Reverse Engineering

with an example on a home router

Valerio Di Giampietro

Linux enthusiast since 1993

http://va.ler.io

v@ler.io

@valerio

©Valerio Di Giampietro - This work is licensed under a Creative Commons Attribution 4.0 International License

Good Afternoon and thank you for coming here to this Introduction to IOT Reverse Engineering with an example on a home router.

I am Valerio Di Giampietro I am passionate about Electronics and Technology since when I was a child, for this reason I studied Electronic Engineering at the University.

I started working as a Digital Hardware Designer, but soon moved to a Unix System Administration position and later to an Infrastructure Manager position, managing servers, both on the cloud and on premise, networking, storage, databases and so on.

I am a Linux enthusiast since 1993 when I installed Linux for the first time on my PC. The year later I run a Linux based, Linux oriented BBS. Sorry if many of you were not born yet and don't know what BBS were and how nerds communicated before the Internet.

My personal website is “Valerio”, written as “va.ler.io”, you will find there this presentation with the links to everything I will talk about today and to my Github repositories with all the details and code about this project.

My email address is “Valerio”, written as “v@ler.io” and my twitter handle is “valerio”.

Being a nerd I am quite proud of these addresses.

What we will talk about …

- Introduction
- Information gathering
- Emulation environment using QEMU
- Analyze how the device works
- Modify the firmware

What we will talk about today?

We will start with a short introduction to reverse engineering and then we will talk about the reverse engineering process based on

- information gathering;
- building an emulation environment based on QEMU, where to run interesting binaries;
- analyzing how the device works
- and modifying the firmware.

Engineering

What is "*engineering*"? "*the science of making things*"

- Define "*product requirements*"
- Design the product
- Build or manufacture the product

Requirements

Design Blueprint

Product

To define "reverse engineering" first we have to define what "engineering" is.

We can say that "engineering" is the science of making things, that starts with product requirements, continues with designing the product and end with the manufacturing of the product.

Reverse Engineering

What is "*reverse engineering*"?

- It is the "*engineering*" process done in reverse order and, usually, with limited scope

What is this example project "*limited scope*"?

- Understand how to modify the router firmware to add features and additional programs

Product

Design Blueprint

Requirements

"Reverse engineering" is the same "engineering" process, but done in reverse order and, usually, with a limited scope.

It starts from the product, manufactured by someone else, analyzing the product, taking measurements; then continues reconstructing the product design and the requirements.

The reverse engineering has a limited scope, in the home router example this scope is to modify the firmware to add additional features and programs to the device; we are not interested to understand and reconstruct other design features.

In other situations we can have different "limited scope". For example if we have an IoT device controlling a production machine may be we are interested in reverse engineering the protocol used by the two devices because we want to replace the IoT device with a more powerful device with additional features.

Information Gathering

- Who makes the device?
- Is there an ODM (*Original Design Manufacturer*)?
- Open the case
- Identify main device components
- Locate UART and possibly JTAG Interfaces
- Get the firmware and the root file system

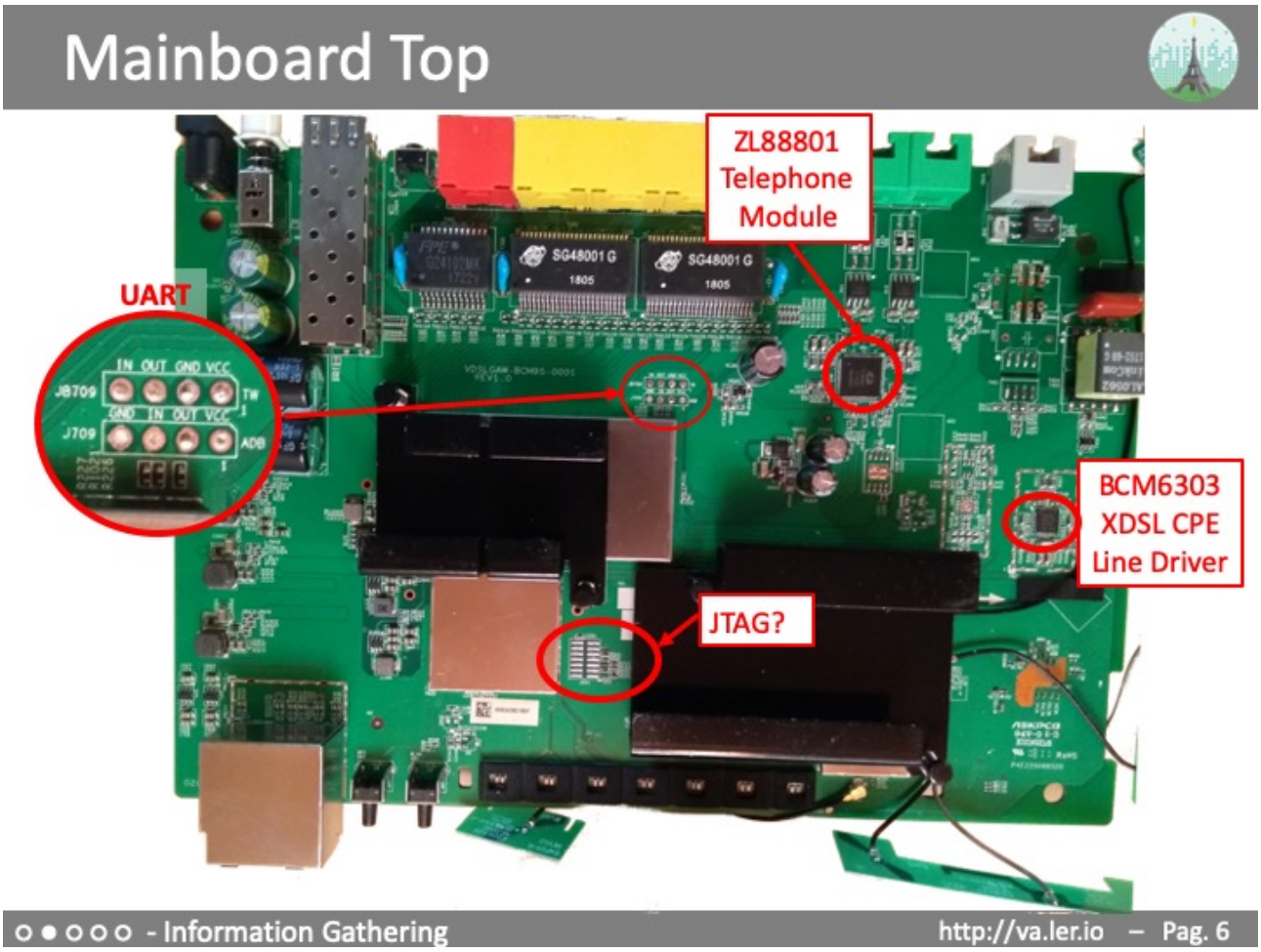
D-Link DVA 5592
example home router

We start the reverse engineering project with information gathering. The first information we need is who makes the device, usually it is easy to find, for example in this case the maker is D-Link.

We are interested also to know if there is an ODM, an Original Design Manufacturer because, sometimes, the device has been manufactured by one company, but the design and the firmware has been developed by another company.

We open the case, we try to identify the main components and to locate the UART interface (it is the serial console) and, possibly, also the JTAG interface. It is very important to identify these 2 interfaces, especially the serial console.

The last step of information gathering is to get the firmware file and the root file system.

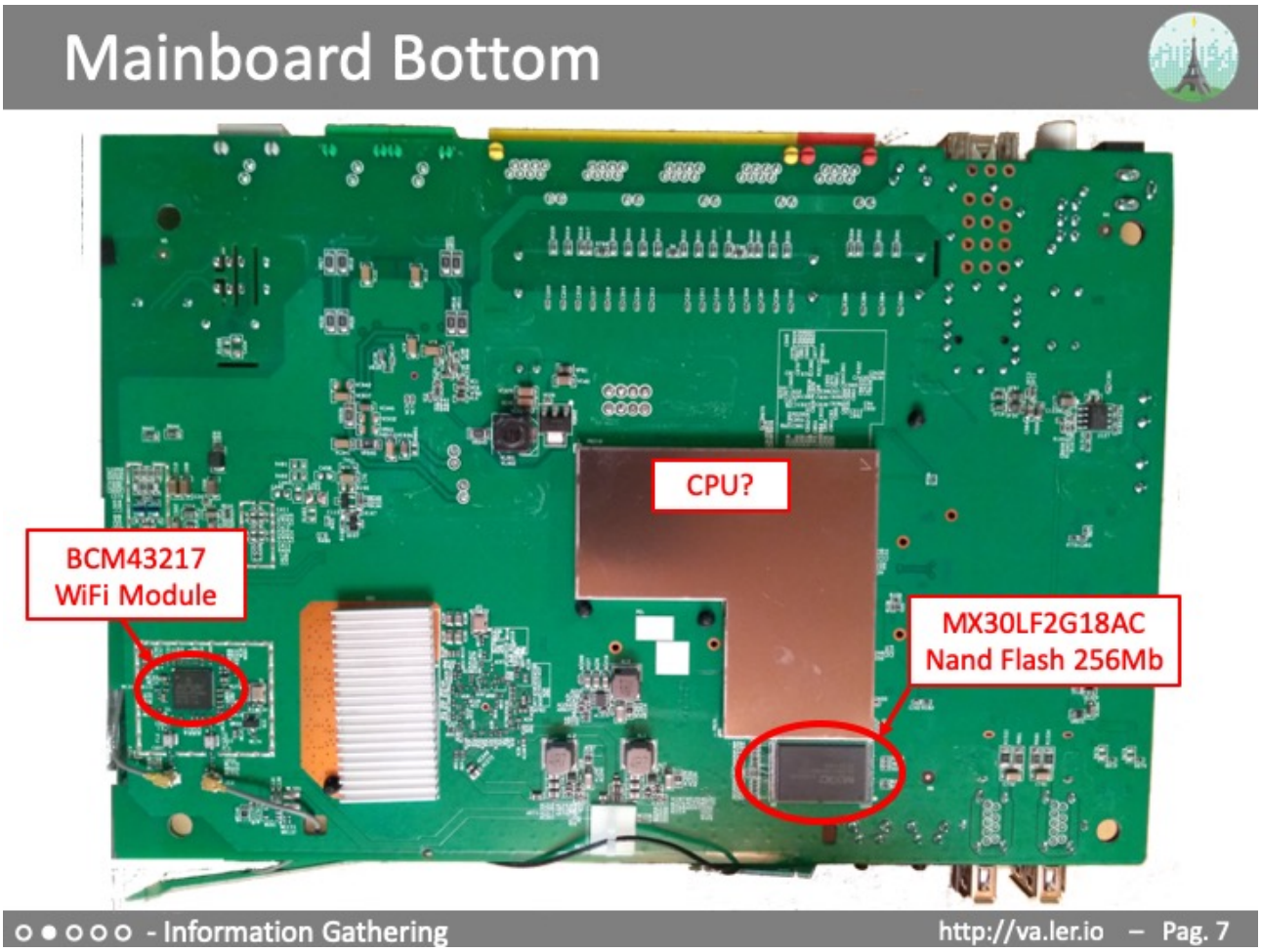


After opening the case, we start looking at the top of the mainboard, trying to identify the main electronic components and to locate the UART and the JTAG interface.

As you can see it is not always easy to identify all components because many components are hidden below these black heat sinks or below thesi metal sheets used to shield radio frequencies.

Anyway we are able to identify some components and we are able to locate the serial console that is clearly labeled on the PCB.

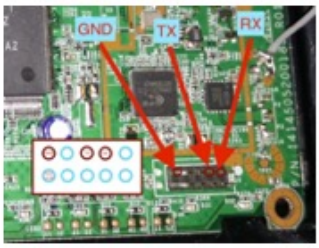
It also important to find the JTAG interface but, in this case, I wasn't able to confirm the position of the JTAG interface.



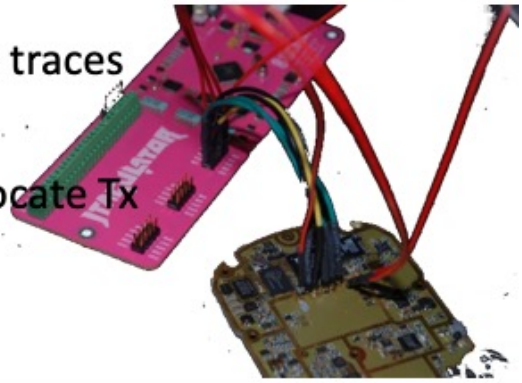
We can also look at the bottom of the mainboard, and identify the NAND flash device and the WiFi module, but we are not able to identify the System On a Chip, probably it is hidden below these metal sheets used to shield radio frequencies.

Locate the UART interface

UART on various boards



- Search on Internet
- Identify potential serial headers candidates
- Sometimes marked in the PCB's silkscreen
- Usually 4 pins: Vcc, Gnd, Tx, Rx
- Use a multimeter to find potential candidates
- Locate pins on SOC and follow PCB traces
- Use tools like Jtagulator
- Oscilloscope or Logic Analyzer to locate Tx (a little overkill)



On the right you can see some UART interface position on some miscellaneous boards.

To locate the serial interface we start searching on the Internet to see if someone else has already done the job for our device.

If this is unsuccessful we have to identify potential serial headers candidates.

Sometimes the serial headers are easy to find because are marked on the PCB silkscreen, as in our case.

If they are not marked we have to search for 4 pins, they are Power, Ground, TX and RX.

We can use a multimeter to find potential candidates, it is easy to identify power and ground, usually on TX there is a pull-up resistor and RX often is an high impedance input.

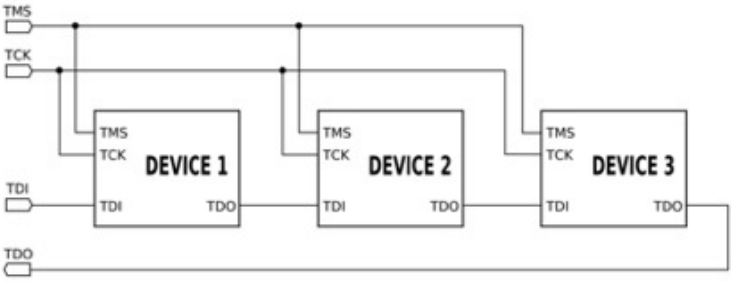
If we have identified the System on a Chip device and if we have the related data sheet, we can identify the Serial Console on the chip pins and then try to follow the PCB traces; it is not easy because the board usually is a multi-layer PCB.

We can also use tools like Jtagulator. Jtagulator is a fantastic tool it has a lot of headers that you can attach to potential UART or JTAG pin candidates, it can run some automatic scanning and identify the UART or the JTAG pinout.

We can also use an oscilloscope or a logic analyzer to identify the TX pin, because, during the boot, something will be written to the serial console and this pin will go up and down. But this method probably is a bit overkill.

The JTAG interface

- JTAG is an industry standard for testing printed circuit boards after manufacture
- Allows access to read/write flash memory contents and can be used as a primary means for an in-circuit emulator
- Multiple devices are daisy-chained together
- Pins:
  TCK *test clock*
  TDI *test data in*
  TDO *test data out*
  TMS *test mode sel.*
  TRST *test reset (opt.)*



We want also to identify the JTAG interface. The JTAG is an industry standard for testing printed circuit boards after manufacture, it provides access to many logic signals of a complex integrated circuits including the device pins.

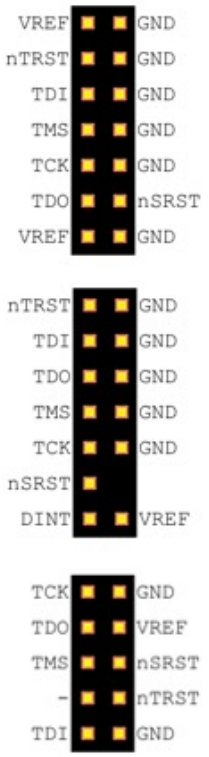
It allows access to read/write flash memory contents and can be used to do in-circuit emulation with tools like GDB, the Gnu Debugger.

Multiple devices on the same board are daisy chained together and each of them can be accessed through the JTAG interface.

The JTAG interface usually has 4 or 5 pins that are the Test Clock, Test Data Input, Test Data Output, Test Mode Select and Test Reset that is optional because we can reset the JTAG interface using the other pins.

Locate the JTAG interface

- No standard pinout, but few popular pinouts: *http://www.jtagtest.com/pinouts/*
- Search on the Internet
- Look for headers labeled TCK, TDI, TDO, TMS
- Look for 1x5/6, 2x5, 2x7, 2x10 pin headers
  - Look for GND and VCC with a multimeter and compare to popular pinouts
  - Often there are pullups (1-100k) for TMS, TDI and TRST, TRST can also be pulled low
  - TDO should be high impedance
- Locate pins on SOC and follow PCB traces
- Use tools like Jtagulator



It is not easy to locate the JTAG interface because there are no standard pinouts, but there are anyway few popular pinouts that can be found at the web address shown on the slide. On the right side of the slide you will find some examples of popular pinouts.

The first step is to search on the Internet to see if someone else has already found where the JTAG interface is located in our IoT device.

If the Internet search is unsuccessful we start looking on the PCB to find headers labeled TCK, TDI, TDO and TMS.

If no label is found, we start searching on the PCB pin headers arranged in a single row of 5 or 6 pins or in a double row of 10, 14, or 20 pins.

We can use a multimeter to locate ground and VCC, we know that often there are pull-up resistors for TMS and TDI, Test Reset can be pulled-up or pulled-down.

TDO should be high impedance. With a multimeter we can try to identify these pins and compare what we have found with popular pinouts finding a possible JTAG interface identification.

If we know the System on a Chip and have its datasheet, we can identify the JTAG interface on his pinout and can try to follow the PCB traces, but this can be difficult due to the multilayer board.

We can also use tools like Jtagulator that, as seen before, can be attached to potential headers candidate and can run some automatic scanning logic to identify each pin of the JTAG interface.

Repopulate the interfaces

Connect the serial cable

After identifying the interfaces we have to repopulate them, this means attaching headers to the interfaces using a de-soldering pump and a soldering iron.

Then we have to connect the interface to the PC. For the UART we need a USB TTL serial cable as the one shown on the slide.

Why repopulate the interfaces?

- UART (Serial Interface)
  - Watch what is printed on the serial console during the boot cycle and find bootloader and OS version
  - Watch the firmware upgrade cycle
  - Use a USB TTL serial adapter and a terminal emulator on the PC
- JTAG
  - Be able to read the firmware out of the flash eeprom
  - Be able to break into the boot cycle and use JTAG as a means to do "*in circuit debugging*"
  - Attach an interface board, like *Bus Pirate*, to the JTAG interface
  - Use software, as *OpenOCD*, to dump flash eeprom and to do "*in circuit debugging*"

Why we want to repopulate the interfaces and connect the UART and the JTAG interface to the PC?

The UART interface is very important, because it allows us to watch what is printed on the serial console during the boot cycle, allowing the identification of boot loader and Operating System version.

It, also, allows us to watch what is printed on the serial console during the upgrade cycle and to get some information on how the firmware upgrade is done.

To connect the serial interface we have to use a USB TTL serial adapter connected to the PC and a terminal emulator like Putty.

We are also interested in the JTAG interface because it allows us to read the firmware out of the flash EEPROM.

It allows us also to break into the boot cycle and use the JTAG as a means to do "in circuit debugging" with tools like GDB, the Gnu Debugger.

To do these things we need to attach an interface board, like "Bus Pirate" to the JTAG interface and we have to use software, like OpenOCD, to dump flash EEPROM and to do "in circuit debugging".

To do this kind of things is quite difficult especially if nobody else have already done and documented this for our own IoT device because we have to do a lot of trial and error and have to define some device specific configuration files. We can find a lot of documentation on the Internet but it's not an easy task.

Getting the firmware file

- Follow the easiest path first
- If the supplier has a website with firmware updates go and download the firmware file
- If the firmware update can be downloaded directly only by the device, sniff the communication with *wireshark* and get the firmware file
- If the above steps are not available, download the eeprom image through the JTAG connector using Bus Pirate and OpenOCD

Now we have to get the firmware out of the IoT device.

A general principle for every reverse engineering task is always to follow the easiest path first. This means that, if the IoT device maker has a website with firmware updates, we download the firmware file form there.

If the firmware file can be downloaded directly only by the IoT device we can sniff the communication with Wireshark and get the firmware file.

If the above steps are not available we have to download the EEPROM image through the JTAG interface using Bus Pirate and OpenOCD. Unfortunately this can be quite complicated and can require a lot of time

Get info from the firmware

- Get basic info from the firmware file

```
$ file DVA-5592_A1_WI_20180405.sig
DVA-5592_A1_WI_20180405.sig: data

$ binwalk DVA-5592_A1_WI_20180405.sig
DECIMAL HEXADECIMAL DESCRIPTION
--------------------------------------------------
512       0x200     JFFS2 filesystem, little endian
24379992  0x1740258 gzip compressed data, from
                    Unix, last modified:
                    2018-04-11 10:40:16
```


Once we have the firmware file, in our example we have downloaded it from the maker website, we can have basic information about the firmware file using, on a Linux system, the "file" command that tells us that this file is just data.

But we can have a lot of meaningful information using a fantastic piece of software that is "binwalk". Binwalk is able to read inside a data file and search for signatures of many different file system images, of compressed data segments, of image pictures, of digital certificates and so on. Binwalk is able to identify what is inside a data file.

For example in our case it tells us that we have a JFFS2 file system image. JFFS2 stands for Journaling Flash File System version 2 and is quite popular on embedded systems with NAND flash EEPROMs. It tells us that we have also a data segment of gzip compressed data.

Extract content from firmware

- Install Jefferson to extract files from JFFS2 file system
- Use binwalk to extract content from firmware

```
$ binwalk -e DVA-5592_A1_WI_20180405.sig

$ ls -lh _DVA-5592_A1_WI_20180405.sig.extracted
-rw-rw-r-- 1 val val  30K ott 21 13:28 1740258
-rw-rw-r-- 1 val val  24M ott 21 13:27 200.jffs2
drwxrwxr-x 5 val val 4,0K ott 21 13:28 jffs2-root

$ file 1740258
1740258: POSIX tar archive (GNU)

$ tar -tvf 1740258
drwxr-xr-x l.fornalczyk/adb  boards/
drwxr-xr-x l.fornalczyk/adb  boards/963138_VD5….ipk
```


The JFFS2 file system cannot be loop mounted because it works on top of the MTD driver, the Nand Flash device driver, that works differently from the block device driver.

For this reason we need to install the Jefferson software to extract files from the JFFS2 file system. Once Jefferson is installed we can use binwalk to recursively extract what is inside the firmware file, including the JFFS2 file system; to do so we have to run the binwalk command with the “-e” option.

We can see that the first file extracted by binwalk is a gzipped compressed tar archive containing some board specific packages.

The second file, 200.jffs2, is the JFFS2 file system image that is recursively extracted in the folder jffs2-root.

Looking at the extracted files

- Looking at the extracted files

```
$ ls jffs2-root/
fs_1  fs_2  fs_3
```

- It seems we have 3 file systems here: "/boot" and "/" splitted in two parts

```
$ ls -lh fs_1
-rw-r--r-- 1 val val    0 ott 21 13:28 a
-rw-r--r-- 1 val val 260K ott 21 13:28 cferam.000
-rw-r--r-- 1 val val 1,2M ott 21 13:28 vmlinux.lz
```

- cferam.000 is the boot loader image based on Broadcom CFE (Common Firmware Environment)
- vmlinux.lz is the kernel, in an unusual CFE compressed format

If we look at the extracted files it seems that we have 3 file systems, in reality we have only two. This is a glitch of the Jefferson program related to a not standard feature in the file system.

The file systems are the boot file system and the root file system splitted in two folders.

In the boot file system we find two files: cferam.000 is the boot loader, it is based on the Broadcom Common Firmware Environment toolkit provided by the chip manufacturer.

vmlinux.lz is the kernel in an unusual Broadcom CFE compressed format. It can be decompressed using file layout information available on the Internet.

Looking at other files

- /sbin/init is missing (but it's not true), busybox is there

```
$ ls -lh fs_2/bin/busybox
-rwsr-sr-x 1 val val 382K fs_2/bin/busybox
$ strings fs_2/bin/busybox
…
BusyBox v1.17.3 (2018-04-11 12:29:54 CEST)
…

$ arm-linux-readelf -a fs_2/bin/busybox
…
… program interpreter: /lib/ld-uClibc.so.0]

$ ls -lh fs_2/lib/ld-uClibc*
-rwxr-xr-x ld-uClibc-0.9.33.2.so
lrwxrwxrwx ld-uClibc.so.0 -> ld-uClibc-0.9.33.2.so

$ ls -l fs_3/lib/libgcrypt.so.11*
lrwxrwxrwx libgcrypt.so.11 -> libgcrypt.so.11.5.3
-rwxr-xr-x libgcrypt.so.11.5.3
```


If we do some file system analysis we can discover that the system is based on Busybox, using the string command we know that the Busybox version is 1.17.3; busybox is very popular in embedded systems and, in a small single binary, includes a lot of functions like init, bash, ls and many other linux commands.

Using the cross tool chain readelf command we can discover that the system use uses uClibc version 0.9.33.2. uClibc is a sort of stripped down libc library that implements almost everything that glibc provides but with a much smaller footprint.

Looking around the file system we can identify other components and other library versions, like the libgcrypt library.

Boot output on serial console

- Output on serial console during boot

```
...
CFE version 1.0.38-118.3-S for BCM963138 (32bit,SP,LE)
generic
...
Chip ID: BCM63136B0, ARM Cortex A9 Dual Core: 1000MHz
Total Memory: 268435456 bytes (256MB)
NAND ECC BCH-4, page size 0x800 bytes, spare size 64 bytes
NAND flash device: , id 0xc2da block 128KB size 262144KB
...
Linux version 3.4.11-rt19 (l.fornalczyk@quelo) (gcc version
4.5.4 20120306 (prerelease) (Linaro GCC 4.5-2012.03) )
...
CPU: ARMv7 Processor [414fc091] revision 1 (ARMv7)
...
jffs2: version 2.2 (NAND) (SUMMARY) (ZLIB) (LZMA) (RTIME)
...
```


Having done some file system analysis we are now ready to attach the serial cable to the UART, start the putty terminal emulator on the PC, power-up the device and start looking at what is printed on the serial console during boot.

We can identify a lot of useful information, in this case we can identify the boot loader and his version, the bootloader is the Broadcom CFE Common Firmware Environment.

We can identify the System on a Chip id, it is a Broadcom chip but there is little information on the internet about this chip, probably because it is a custom made SOC. Broadcom allows his customers to build a custom made chip assembling a processor core and some other devices in a single chip.

We can see that the processor core is an ARM Cortex A9 dual core CPU running at 1 Ghz, that we have 256 Mb of RAM and a NAND flash device with 256 Mb of storage.

We can understand that the Linux version is 3.4.11 with real time patches version 19.

The processor has an ARMv7 instruction set and the JFFS2 file system, among other things, use a not standard LZMA compression.

Boot output on serial console

- Output on serial console during boot (part 2)

```
...
[2.502000] Found YAPS PartitionSplit Marker at 0x080FFF00
[2.503000] Creating 8 MTD partitions on "brcmnand.0":
[2.504000] 0x000000000000-0x000000020000 : "CFE"
[2.505000] 0x000000000000-0x000008100000 : "bootfs_1"
[2.506000] 0x000000000000-0x000000fbc0000 : "rootfs_1"
[2.507000] 0x000000000000-0x000007ee0000 : "upgrade"
[2.508000] 0x000000000000-0x00000fdc0000 : "conf_fs"
[2.509000] 0x000000000000-0x00000fe00000 : "conf_factory"
[2.510000] 0x000000000000-0x000010000000 : "bbt"
[2.517000] 0x000000000000-0x000010000000 : "flash"
...
Init started: BusyBox v1.17.3 (2018-04-11 12:29:54 CEST)
starting pid 235, tty '': '/etc/init.d/rcS S boot'
Starting boot.sh ...
Restore passwd ....
Restore group ....
Starting /etc/rc.d/S11services.sh ...
Starting Configuration Manager (B)
```

```
/etc/inittab
 ::sysinit:/etc/init.d/rcS S boot
 ::shutdown:/etc/init.d/rcS K shutdown
 tts/0::askfirst:/bin/login
 ttyS0::askfirst:/bin/login
```


Continuing to look at the serial console output during boot we can understand also how the NAND flash EEPROM is partitioned.

In this case we have a partition for the boot loader CFE, a partition for the boot file system, another one for the root file system. A partition dedicated to the upgrade process; two configuration partitions one for the running system and another one with factory default configuration.

We have also a Bad Block Table partition and a pseudo partition that allows to see the entire NAND flash device as a single partition.

We can see the “init” process is BusyBox that starts executing the first shell script that is rcS with “S” and “boot” as parameters. This is according to the /etc/inittab configuration file.

Boot output on serial console

- Output on serial console during boot (part 3)

```
...
CM TR-181 ready
CM TR-98 ready
Epicentro Software Version: DVA-5592_A1_WI_20180405
Epicentro Platform Version: 6.0.0.0028
...
Starting /etc/rc.d/S13acsd.sh ...
Starting /etc/rc.d/S20voip.sh ...
Starting /etc/rc.d/S60ipsec.sh ...
Starting /etc/rc.d/S70vpn.sh ...
Starting /etc/rc.d/S94printkd.sh ...
```

Searching «Epicentro Software» on Internet gives the ODM (Original Design Manufacturer): ADB www.adbglobal.com

Continuing to look at the serial console we can see the text "Epicentro Software Version"; if we search on the internet this string we are able to identify the Original Design Manufacturer that it is a swiss company called ADB with the adbglobal.com website.

It is very important to identify the Original Design Manufacturer because, when we search for information on the internet about our device, if we don't find anything searching for D-Link router, may be we can find something searching for Epicentro Software.

What was found

- Processor ARMv7 Cortex-A9 Multicore
- 256Mb NAND Flash
- Linux version 3.4.11-rt19 (September 2012)
- uClibc version 0.9.33.2 (May 2012)
- BusyBox version 1.17.3 (October 2010)
- Libgcrypt version 1.4.5 (December 2009)
- Epicentro software by ADB (adbglobal.com)

Analyzing the file system and analyzing what was printed on the serial console we have been able to gather the following information:

- We have an ARMv7 Cortex-A9 Multicore processor
- We have 256 Megabytes of NAND Flash
- We have a Linux kernel version 3.4.11 with real time patches version 19, it was released on September 2012. This home router started shipping on April 2018, but it uses a six year old linux kernel.
- The uC-libc version is 0.9.33.2, it was released on May 2012, again a very old version.
- BusyBox version is 1.17.3, released in October 2010, older than the other components.
- Libgcrypt version is 1.4.5, released in December 2009 and many other libraries have, more or less, the same age.
- We were also able to identify that the firmware Epicentro Software was produced by the Swiss company ADB.

We can see that we have old software on a new device; Why this happens? This happens quite often, especially in the consumer space, because the manufacturer tend to save some money re-using what he already has or directly using some old SDK provided by the Sistem on a Chip maker.

This can be very dangerous from a security point of view, because the IoT device can have a lot of known and unpatched vulnerabilities discovered and published during the years.

QEMU Emulation

- Choosing Board and CPU emulation in QEMU

```
$ qemu-system-arm -M help|egrep Cortex-A9
realview-pbx-a9 ARM RealView Platform for Cortex-A9
vexpress-a9     ARM Versatile Express for Cortex-A9
xilinx-zynq-a9  Xilinx Zynq Platform  for Cortex-A9

$ qemu-system-arm -M vexpress-a9 -cpu help
...
cortex-a9
...
```


After the information gathering process we have to build an emulation environment where to run interesting binaries to analyze what these binaries do.

We have to choose how to build the emulation environment, we have to emulate the same processor that we have on the IoT device and a board that is reasonably similar to our IoT device board.

We choose QEMU that is a fantastic piece of software that is able to emulate, with good performance, a very large number of different processors and a lot of different boards.

In this case we want to emulate a Cortex-A9 ARM processor and we choose the vexpress-a9 board with the cortex-A9 CPU.

Choosing a Build System

- **The Yocto Project**
  very powerful, builds a root file system and create a custom Linux distribution. It's main drawback is that it has a steep learning curve

- **Buildroot**
  builds the root file system and the kernel, easy and fast to learn, very good user manual

- **Openwrt/LEDE Build System**
  tailored to build a replacement router firmware, documentation scattered in the web site, requires more time to learn compared to Buildroot

To run interesting binaries in the emulated environment we need also a linux kernel and a root file system reasonably similar to our IoT device and with debugging tools.

Theoretically we could build the kernel and the root filesystem cross-compiling everything by ourselves, but due to the inter-dependencies of all the packages versions, drivers and libraries we need, this would be an almost impossible task.

But we are lucky and we can chose among different free and open source build systems that allow us to easy configure and build the kernel and the root file system.

The most popular build system is the Yocto Project, it is very powerful and allows us not only to build a kernel and a root file system, but can create a custom Linux distribution with an included package management system. The main drawback is that it has a steep learning curve. If you don't already know the Yocto Project it would be quite difficult to start with this build system.

Another popular build system is Buildroot; Buildroot builds the root file system and the kernel, it is easy and fast to learn and has a very good user manual the it is not too long and not too short.

Another popular build system is Openwrt Build system, lately renamed LEDE. It is tailored to build a replacement router firmware, it was forked from Buildroot about 10 or 15 years ago, but the documentation is scattered over the web site and requires much more time to learn compared to Buildroot.

For the home router example project I choose the Buildroot system.

Choosing a Buildroot version

- Based on uClibc 0.9.33.2
- Gnu libgcrypt crypto library compatible with version 1.5 (with library file: libgcrypt.so.11)
- With custom kernel version 3.4.1-rt19
- Other libraries with compatible versions
- The version to use is: buildroot-2014.02
- This version doesn't run on Ubuntu 16.04 or Ubuntu 18.04
- Use Debian Wheezy (released in 2013) in a docker Container

Choosing the build root system is not enough we have also to choose a Buildroot version, because we want to build a root file system that is reasonably similar to the one we have on the IoT device and that is able to run IoT device interesting binaries.

It has to be based on the same uClibc version as on the IoT device, on a compatible libgcrypt library, with the same custom kernel version and real time patch and with other libraries with compatible versions.

This means that we have to use an old Buildroot version. Searching through buildroot documentation we found that what we need was available in Buildroot version 2014.02 that was released on February 2014.

The problem is that this old Buildroot version, due to the dependency hell, doesn't run on a recent Linux distribution like Ubuntu 16.04 or Ubuntu 18.04.

To solve the problem we have to use an older Linux distribution like Debian Wheezy that was released in 2013 and has a similar age as the Linux kernel and other software components that we need.

We will use Debian Wheezy in a Docker Container. Usually Docker Containers are used to implement scalable and isolated micro-services based infrastructures, we, instead, use Docker as a sort of light virtualization environment not isolated from the host machine.

The Dockerfile

- The Dockerfile builds a minimal Debian Wheezy container to run Buildroot

```
FROM debian:wheezy
...
RUN apt-get update
RUN apt-get upgrade
RUN apt-get install -y -q \
    bash \
...
RUN mkdir -p       /src/misc
ADD startup.sh     /src/misc/startup.sh
RUN chmod a+x      /src/misc/startup.sh
ENTRYPOINT cd      /src/misc ; ./startup.sh
```

- Available at: https://github.com/digiampietro/buildroot-armv7

This Dockerfile builds a minimal Debian Wheezy container that includes only what is needed to run Buildroot, it is available on my GitHub repository and on my Docker Hub.

You will find the link on this presentation that is available on my website.

The Docker run command

- The *Docker run* maps user and user's home directory in the Docker Container

```
docker run      -h BRHOST                          \
                --rm                               \
                -v /tmp/.X11-unix:/tmp/.X11-unix   \
                -v $HOME:$HOME                     \
                -e DISPLAY=$GDISPLAY               \
                -e GUSERNAME=$GUSERNAME            \
                -e GUID=$GUID                      \
                -e GGROUP=$GGROUP                  \
                -e GGID=$GGID                      \
                -e GHOME=$GHOME                    \
                -e GSHELL=$SHELL                   \
                -e GRUNXTERM=$GRUNXTERM            \
                -e GPWD=$GPWD                      \
                -it digiampietro/buildroot-armv7
```


To run Docker we use 3 important choices:

- we map the temporary X11 folder in the host machine at the same position in the docker machine, in this way we are able to run X11 applications in the docker machine, like, for example, the "make xconfig" command;
- we map the home directory in the host machine at the same position inside the docker machine, in this way the home directory is shared between the docker machine and the host machine;
- we map the userid and username in the host machine with the same userid and username in the docker machine.

Docker run in action

```
valerio@ubuntu-hp:~$ ls -ld br
drwxrwxr-x 6 valerio valerio 4096 ott 26 22:30 br

valerio@ubuntu-hp:~$ grep VERSION /etc/os-release
VERSION="18.04.1 LTS (Bionic Beaver)"
VERSION_ID="18.04"
VERSION_CODENAME=bionic
valerio@ubuntu-hp:~$ br/..../docker/dockrun.sh
```

```
valerio@BRHOST:~$ ls -ld br
drwxrwxr-x 6 valerio valerio 4096 Oct 26 20:30 br

valerio@BRHOST:~$ grep VERSION /etc/os-release
VERSION_ID="7"
VERSION="7 (wheezy)"
```


In this way we can have two windows on our Linux machine, on one window we have Ubuntu 18.04 and on another window we have the running docker machine with Debian wheezy and both windows share exactly the same home directory and exactly the same userid and username.

On the Debian Wheezy window we can run Buildroot and on the Ubuntu 18.04 window we can run everything else.

Buildroot configuration

- Based on "*qemu_arm_vexpress_defconfig*"
- With the following main modifications
  - Build packages and libraries with debugging symbols, don't strip binaries, no gcc optimization
  - Build gdb, gdbserver, ltrace, strace and cross gdb for the host
  - Include mtd and jffs2 file system and tools also for the host
  - Include main libraries used in the router (libgcrypt, expat, roxml, libxml2, Mini-XML)

Our Buildroot needs to be configured.

Configuration is based on the Buildroot shipped configuration for our vexpress-a9 emulated board, but we have to do some modification to the default configuration:

first of all we want to build packages and libraries with debugging symbols and other debugging configurations because we want to run interesting binaries with the Gnu Debugger and be able to trace library function calls and to understand what is going on.

We want also to build debugging tools on both the target system and on the host system like, for example, the gnu debugger, the gnu debugger server on the target, ltrace strace and so on.

We have to include the NAND flash driver, the jffs2 file system and tools for both the target and the host.

We have also to include the main libraries used on our IoT device.

Linux kernel configuration

- With the following additional settings
  - Versatile Express platform type with Device Tree support
  - Preemptible Kernel
  - NAND Device Support and Support for NAND Flash Simulator
  - JFFS2 file system with LZMA compression

We have also to configure the kernel.

First of all we choose the same kernel version as the one running on our IoT device, then configure it to run on our emulated board with Device Tree support.

We select the preemptible kernel as in our IoT device

We enable the NAND Device support and the NAND Flash Simulator support and the JFFS2 file system with LZMA compression support.

uClibc configuration

- Minor modifications to be compatible with the router's binaries (like native POSIX threading etc.)
- To include debugging symbols, uClibc don't obey to the general option included in the Buildroot configuration, has his own flag for this purpose; the problem is that enabling his own flag the compilation gives impossible to fix errors
- in Compiler Warnings add the string "-ggdb", this is the work around to compile the uClibc with debugging symbols

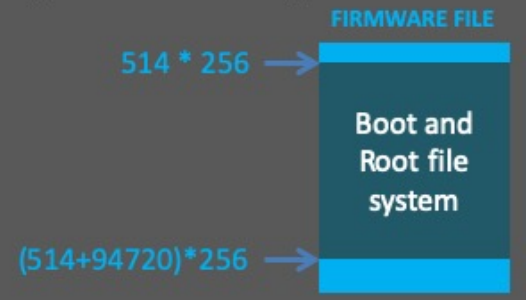
We have also to do some minor configuration modification to the uClibc, in the Buildroot system, to be compatible with router's binaries (like native POSIX threading etc.), this configuration changes are the result of trial and error.

Other modifications are related to the inclusion of debugging symbols that requires some quirks in the configuration.

The upgrade process

- Analyzing the upgrade process on the UART output (1 of 2)

```
Start pid 4 /usr/sbin/upgrade-prepare.sh cwmp
...
/usr/sbin/upgrade.sh
Signature OK
...
/usr/sbin/flash_eraseall \
    -j -p 0 -l 8 /dev/mtd3
...
Writing boot & root filesystems...
dd if=/tmp/upgrade/fw.bin bs=256 skip=514 count=94720
| /usr/sbin/nandwrite -s 524288 /dev/mtd3 -
...
[…] Found YAPS PartitionSplit Marker at 0x080FFF00
[…] Creating 2 MTD partitions on "brcmnand.0":
[…] 0x000007f00000-0x000008100000 : "bootfs_2"
[…] 0x000008100000-0x00000fcc0000 : "rootfs_2"
```



After building the emulation environment we can start analyzing the device upgrade cycle; because we want to modify the firmware the idea is to understand how the upgrade works to modify the upgrade file with our own modifications.

With the serial console connected to our PC running Putty, we start the firmware upgrade on the web interface and look at what is printed on the serial console.

We can see that the first script running is “upgrade-prepare.sh”, then the script “upgrade.sh” is started and it prints "Signature OK", this means that the firmware file is signed and that inside this script the signature is checked.

Then the “flash_eraseall” command is executed to erase the third partition in the flash EEROM, this is the partition dedicated to the upgrade.

Later the boot and root file systems are written in a single “nandwrite” operation. The firmware file is read with the “dd” command.

After the “nandwrite” operation, some kernel module detectS the Partition Splitter Marker and split the upgrade partition in two parts: bootfs_2 and rootfs_2.

The “dd” command parameters let us understand where the boot and root file systems are located inside the firmware file.

The upgrade process

- Analyzing the upgrade process on the UART output (2 of 2)

```
Installing packages...
opkg … -f /tmp/new_rootfs/etc/opkg.conf -o
/tmp/new_rootfs install …
Umount /tmp/new_rootfs
...
Writing first block of cferam...
dd if=/tmp/upgrade/fw.bin bs=256 skip=2 count=512 |
/usr/sbin/nandwrite /dev/mtd8 -
...
rebooting...
```


After the boot and root file systems are written on the flash EEPROM, some other packages are installed on the new root file system. This packages were available inside the gzipped tar file that we extracted from the firmware file using Binwalk and Jefferson.

At the end, the boot loader is written with another “nandwrite” operation. Also in this case, looking at the “dd” parameters, we can understand where the boot loader is located inside the firmware file.

After the upgrade cycle is completed the IoT device is rebooted.

The upgrade process - summary

- The upgrade script to analyze is `/usr/sbin/upgrade.sh`
- The firmware is signed, signature is checked with `sig_verify $file 2> /dev/null`
- Boot and root file systems are written in a single nandwrite operation
- A JFFS2 partition splitter proprietary kernel module is used to create the two partitions on the fly
- A JFFS2 end marker open source kernel module is used to delimit the end of the root file system partition
- Some additional packages are added, based on board type
- The cferam boot loader is written with another nandwrite operation

In summary, about the upgrade process we have found that:

- the main script to analyze to understand how the upgrade process works is “upgrade.sh”. Inside this file the signature is checked with the command "sig_verify";
- Boot and root file systems are written in a single “nandwrite” operation;
- A JFFS2 partition splitter proprietary module is used to create two partitions on the fly;
- A JFFS2 end marker open source kernel module is used to delimit the end of the root file system partition;
- Some additional packages are added, based on the board type;
- The “cferam” boot loader is written with another “nandwrite” operation and then the system is rebooted.

Reverse engineering sig_verify

sig_verify is a stripped binary, but calls library functions. We put breakpoints on these calls in the emulation environment

```
$ arm-linux-readelf --sym -D sig_verify
Symbol table for image:
  Num Buc: Value   Size Type Bind   Vis      Ndx Name
   16   0: 00008928  0 FUNC GLOBAL DEFAULT UND fseek
   29   1: 00008994  0 FUNC GLOBAL DEFAULT UND strcmp
   40   3: 000089dc  0 FUNC GLOBAL DEFAULT UND gcry_md_ctl
   38  11: 000089d0  0 FUNC GLOBAL DEFAULT UND fputs
   23  14: 00008964  0 FUNC GLOBAL DEFAULT UND fread
    1  15: 00008898  0 FUNC GLOBAL DEFAULT UND printf
   44  16: 00008a0c  0 FUNC GLOBAL DEFAULT UND gcry_md_get_algo…
   41  16: 000089e8  0 FUNC GLOBAL DEFAULT UND close
    9  17: 000088e0  0 FUNC GLOBAL DEFAULT UND lseek
    7  19: 000088c8  0 FUNC GLOBAL DEFAULT UND gcry_md_open
    6  19: 000088bc  0 FUNC GLOBAL DEFAULT UND gcry_md_write
    3  20: 000088a4  0 FUNC GLOBAL DEFAULT UND gcry_check_version
...
```


What we want to do if we want to modify the firmware file? We want to analyze what the “sig_verify” binary does, in our emulated environment, to understand if we can fake the signature to load our own version of the firmware file.

“sig_verify” is a stripped binary, without debugging symbols or any other debugging information, but it calls library functions.

This means that it is difficult to understand, with a debugger, what it does, but we can put breakpoints on library functions entry to understand what it does in the emulation environment.

Using the cross toolchain command "readelf", built by the buildroot build system, we can identify all the library calls that “sig_verify” does.

In the emulation environment we can put a breakpoint on each library function entry; We compiled the libraries with debugging information, and so we can understand what “sig_verify” does looking at what library calls it does and what parameter it pass to each library function and what value it get back.

Running sig_verify in GDB

Start gdb server in the emulation environment

```
# gdbserver :9000 sig_verify --readonly \
   DVA-5592_A1_WI_20180405.sig
```

Start gdb in the host machine

```
$ arm-linux-gdb --ex="target remote :9000" \
                --ex="set sysroot $SYSROOT"\
                --ex="directory $MYDIR"    \
                --ex="directory $TOOLBIN"  \
                  -x  sv.gdb
```


We run the “sig_verify” binary with the “gdbserver” in the emulated environment and run the cross toolchain gnu debugger in the host machine.

The gnu debugger in the host machine does the hard work of displaying breakpoints, variable values, source code and so on and the “gdbserver”, in the emulated target device, controls the execution of the binary and communicates with the gnu debugger in the host machine.

Running sig_verify in GDB

- Reads the last 256 bytes from the file (signature)
- Calls `gcry_md_open`,`gcry_md_write`, `gcry_md_ctl` to calculate SHA1 checksum
- Calls `gcry_sexp_build` 3 times to build the 3 s-expressions and then `gcry_pk_verify` to verify the signature with the following parameters
  - **SHA1 message digest**
  - **Signature** (the firmware file last 256 bytes)
  - **The public key** (embedded in the sig_verify binary)

During this type of analysis we can clearly understand what “sig_verify” does:

- it reads the last 256 bytes from the firmware file, it is the signature;
- then it calculates the SHA1 checksum of the file, excluding the last 256 bytes, using libgcrypt functions;
- at the end it verify the signature using:
  - the calculated SHA1 checksum
  - the signature stored in the last 256 bytes of the file that contains the SHA1 checksum encrypted with the private key
  - the public key, embedded in the binary itself

Using the public key, the libgcrypt function, decrypts the signature of the firmware file and compare it with the calculated SHA1 checksum, if the comparison matches it returns OK, otherway it returns an error.

This means that without knowing the private key is impossible to fake the signature.

Running sig_verify in GDB

- The public key (MPI modulus and exponent) can be dumped from memory to recover the public key in the standard .pem format
- Unfortunately the private key remains unknown, it is not included in router's certificates files in the folder /etc/certs

The public key is composed of 2 big integers, or Multi Precision Integers, the modulus and the exponent; these 2 big integers can be dumped from memory, with the debugger, to recover the public key in the standard ".pem" format.

Unfortunately the private key remains unknown, as expected it is not included in the router's certificate folder in /etc/certs.

Restricted shell

- Firmware modification through the upgrade process seems impossible
- The router allows telnet/ssh but present a Cisco-like restricted shell

```
$ telnet 192.168.1.1
Connected to 192.168.1.1.
Escape character is '^]'.
Login: admin
Password:
*****************************************
*                 D-Link                *
*      WARNING: Authorised Access Only  *
*****************************************
Welcome
DLINK#
```


The firmware modification through the upgrade process seems impossible because we are not able to fake the signature.

So we have to find another way.

The router allows a telnet and an ssh connection, but presents a Cisco like restricted shell that means that we can type help to have a list of all available commands; if we start typing some command we can press tab to auto-complete it, we can type the question mark to show the available command options.

We can execute only these commands and not others.

Escaping the restricted shell

- */etc/shells* suggests that the restricted shell is a Clish (or Klish), open source, shell
- /bin/clish is a script:

```
#!/bin/sh
...
exec /bin/clish.elf -l -x /tmp/clish
```

- In */etc/init.d/services.sh:*

```
#in factory mode
ln -s /etc/clish/prod /tmp/clish
#in normal mode
ln -s /etc/clish /tmp/clash
```

- clish xml startup files are:
  /etc/clish/prod/startup.xml
  /etc/clish/startup.xml

If we do some file system analysis we can discover that this Cisco-like interface is based on an open source project called Klish;

the /bin/clish shell is a script that executes clish with startup file in the /tmp/clish folder;

continuing with file system analysis we can discover that /tmp/clish is a link that points to a directory in "normal mode" and to a different directory in "factory mode";

this means that we have two startup.xml configuration files and we can analyze them to see if we can find something interesting.

Escaping the restricted shell

- In */etc/clish/startup.xml:*

```
<COMMAND name="factory-mode" help="hidden">
  <ACTION>cmclient DUMPDM FactoryData
         /tmp/cfg/FactoryData.xml > /dev/null
          nvramUpdate Feature 0x2 > /dev/null
          cmclient REBOOT > /dev/null
  </ACTION>
</COMMAND>
```

- ”factory-mode” is an hidden, not auto-completed command: it is a command to try:

```
DLINK# factory
DLINK(factory)# factory-mode
DLINK(factory)#
DLINK(factory)# Connection closed by foreign
host.
```


Analyzing the normal mode configuration file we can see that there is the "factory-mode" command that is an hidden command, hidden means that when you type help this command is not shown and if you start typing this command and press "tab" it is not auto-completed.

Just because this is an hidden command, means that we have to execute it to see what it does, it switch the rooter to factory mode and does a reboot.

Escaping the restricted shell

- Factory mode is special mode: no normal WiFi, no Internet connection, no DHCP server, but it allows a non privileged shell login:

```
Login: admin
Password:
*****************************************
*                 D-Link                *
*                                       *
*      WARNING: Authorised Access Only  *
*****************************************
Welcome
DLINK# system shell
BusyBox v1.17.3 built-in shell (ash)
Enter 'help' for a list of built-in commands.
/root $
```


Factory mode is a special mode, we have no normal WiFi, no Internet connection, no DHCP server, but it has a "system shell" command that gives a not privileged shell login.

So we can enter into factory mode and have access to the shell, it is not a root shell so we cannot modify any system configuration but we can explore the system as a normal user.

The Quest for Root

- Looking for processes running with root privileges

```
/root $ ps -ef
PID USER       VSZ STAT COMMAND
   1 0         1184 S    init
 261 0          724 S <  /sbin/udevd --daemon
 274 1001      1328 S    /bin/clish.elf -l -x /tmp/clish
 326 0         2332 S    cm
 365 0         1800 S    logd
 368 0          704 S    ec
2383 0          820 S    dns
2630 0         2480 S    cwmp
2631 0         1204 S    inetd -f
2633 0          736 S    yamp -c /tmp/yamp.conf -p /tmp/…
2658 0          664 S    wpspbc
3089 0         2316 S    hostapd -B /tmp/wlan/config/ho…
3090 65534     3560 S    httpd -u nobody
3647 0         1068 S    chronyd -n -f /tmp/chrony.conf
4191 0          696 S    /sbin/rngd -r /dev/urandom -W 4000
4211 0         7136 S    voip
4404 1001      1176 S    /bin/ash
```


We have to find a way to become root, so we start looking at processes running with root privileges because we want to hack some process running as root.

With the “ps” command we can identify processes and executables running as root.

The Quest for Root - 1

- Identify each process and executable version using “strings” and/or running the executable with parameters “`-v -V -version -h -h -help`”
- Identify open source executables
- Search the internet for known vulnerabilities for the specific executable version
- Check if the vulnerability is exploitable in the specific IoT device configuration

We can identify and get version information of each executable using the "strings" command, to look at what strings are embedded inside the related binary file. We can also run the related binary in our emulated environment, or in the target itself, with parameters like -v for version or -h for help to identify its version.

In this way we can identify open source executables and their version and search the Internet for known vulnerabilities for the specific version running on our router. Because our router uses very old software we have good probability to find some documented vulnerability.

If we have found some vulnerability we have to confirm that this vulnerability is exploitable in our specific IoT device configuration, because often some vulnerability is exploitable only under certain circumstances.

The Quest for Root - 2

- If no exploitable vulnerability has found select a process candidate to reverse engineer to find vulnerabilities
- Operating system binaries with no known vulnerabilities are hard to crack
- Lower level binaries (dns, voip ...) are more difficult to crack
- Higher level executables with bigger configuration files are less difficult to crack

If, as in this case, no exploitable vulnerability was found we have to select a process candidate to reverse engineer, in the emulated environment, to find vulnerabilities by ourselves;

We want to follow the easiest path first, so we want to start with processes that have higher probabilities of having a vulnerability;

Operating system binaries with no known vulnerabilities are hard to crack, because if nobody else has already found vulnerabilities on these open source binaries it means that it will be very difficult, for us, to find one.

Some lower level stuff, like binaries related to dns, voip and other specific hardware devices, usually are more difficult to crack because they do one specific thing and so they have a smaller attack surface.

Higher level executables with bigger configuration files often are less difficult to crack because, usually, they have a larger attack surface.

The Quest for Root - 3

- The most interesting process is “cm”: router configuration with root privileges is done by the “cm” process (add users, configure dhcp server, set ip address etc.)
- “cm” uses shell scripts to carry out his duties
- The “cmclient” command, running as normal user, is used by restricted shell and web interface to talk to the “cm” process to configure the router
- “cmclient” is used, in startup scripts, to configure the “cm” process

In our case the most interesting process is "cm", it is the configuration manager process, it runs with root privileges and does router configuration like changing router IP address, configuring the DHCP server and so on.

Analyzing startup scripts and running this process in the emulation environment and with the gnu debugger we can discover that it uses shell scripts to carry out his duties;

In the router there is the "cmclient" command, that runs as normal user, it is used by the restricted shell and by the web interface to talk to the "cm" process to configure the router.

So it seems that the system is well designed because we have a process running as root that does all the configuration change and we have a process running as normal user that asks to the cm process to do the configuration change. In this way only permitted changes are allowed.

The Quest for Root - 4

– In a startup script there is:

```
cmclient DOM Device /etc/cm/tr181/dom/
```

– This loads all the xml file in that directory to configure the cm process, including

```
/etc/cm/tr181/dom/Management.xml
```

– That has the following snippet

```
<object name="Users.User.{i}."
        access="readOnly"
        minEntries="0"
        maxEntries="unbounded"
        numEntriesParameter="UserNumberOfEntries"
        enableParameter="Enable"
        set="Users.sh"
        add="Users.sh"
        del="Users.sh"
  >
```

○○○●○ - Analyze how the device works http://va.ler.io – Pag. 46

Analyzing the startup scripts and reverse engineering the cm process in the emulated environment, it is possible to discover that the cmclient command is used also to configure the cm process.

For example in a startup script there is the cmclient command shown on the slide, that tells to the cm process to load all the xml files in the folder passed as parameter.

Among these files there is the Management.xml file that has the shown snippet that tells to the cm process that to add a user to the router it has to use the “Users.sh” helper bash script. This file is loaded from a specific folder.

The Quest for Root - 5

– To trigger an execution of our script:

```
cmclient DOM Device /tmp/fakeManagement.xml
```

– It has the following snippet

```
<object name="Users.User.{i}."
      access="readOnly"
      minEntries="0"
      maxEntries="unbounded"
      numEntriesParameter="UserNumberOfEntries"
      enableParameter="Enable"
      set="../../tmp/fakeUsers.sh"
      add="../../tmp/fakeUsers.sh"
      del="../../tmp/fakeUsers.sh"
>
```

– Trigger the execution of the «fakeUsers.sh» script with

```
cmclient ADD Device.Users.User
```

○○○●○ - Analyze how the device works http://va.ler.io – Pag. 47

We can try to copy the Management.xml file in a temporary folder, modify it to execute our own version of Users.sh that we call fakeUsers.sh.

fakeUsers.sh is a modification of the original Users.sh in which we have added a command to modify the /etc/passwd file to remove the password for the root user, in this way we can do an “su” to root without password.

Finally we can use the cmclient command to trigger the execution of our fakeUsers.sh. Because our script is executed by the cm process, running as root, it will run as root and will succeed in modifying the /etc/passwd file.

We can include all the above steps in a shell script, I have done this and the script is available on my Github repository.

The Quest for Root - 7

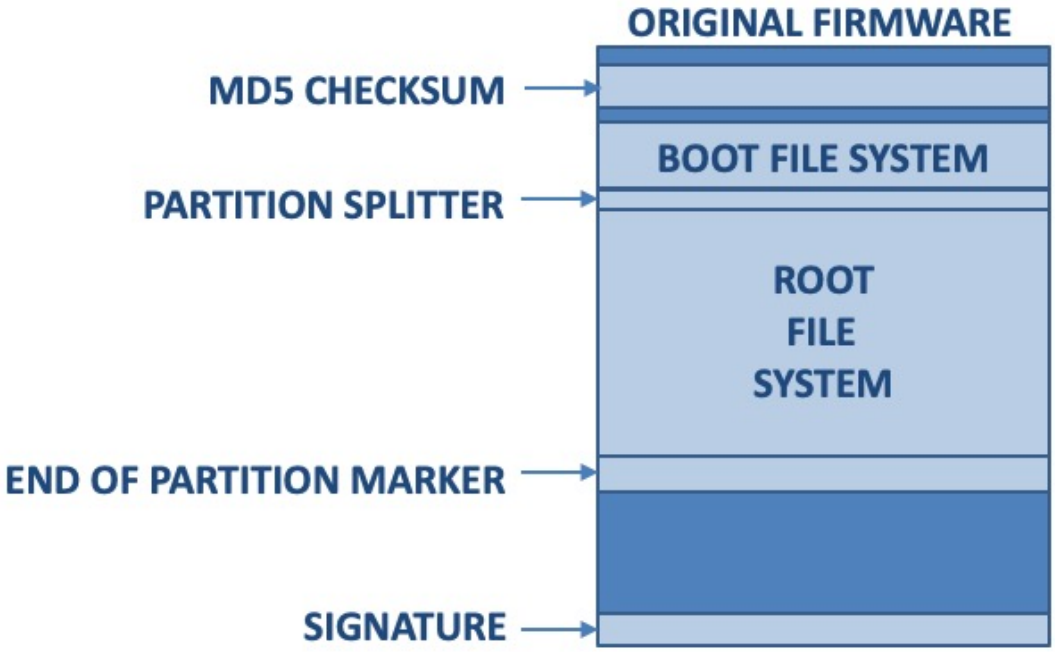
```
/root $ cat > /tmp/hack-script.sh
   do a copy and paste of the script
   press CTRL-D to terminate the copy
/root $ chmod a+x /tmp/hack-script.sh
/root $ /tmp/hack-script.sh
...
/root $ su -
BusyBox v1.17.3 (2018-04-11) built-in shell (ash)
Enter 'help' for a list of built-in commands.
r41358.07b1b3a7
.......................................................
yet another solution by  Advanced Digital Broadcast SA
.......................................................
root@localhost:~# id
uid=0(root) gid=0(root)
groups=0(root),19(remoteaccess),20(localaccess)
```


We can copy this script on the IoT device, make it executable, execute it and we can use “su” to become root.

As root we can now do everything on our router.

Firmware Modification Kit - 1

- Based on firmware upgrade script analysis the firmware file has the following structure:



Returning back to our analysis of the firmware upgrade script we can see that the firmware file has the following structure:

- we have an MD5 checksum, so if we modify the firmware file we have to recalculate this checksum;
- we have the boot file system, a partition splitter, the root file system, the end of partition marker;
- at the end we have the signature;

in between these discovered fields we have some other unknown fields whose purpose we don't know and don't know how they are calculated.

Firmware Modification Kit - 2

- Extract the root file system, modify it
- Create the new root file system image
- Pad the file system image to the same size as the original root file system image (the USB key will be used for additional software)
- Reassemble the firmware file putting together all the pieces, excluding the signature, using the ”dd” command
- The unsigned firmware file is ready

To modify the firmware the idea is:

- extract the root file system, modify it;
- create the new modified root file system image;
- pad the file system image to the same size as the original root file system image; we will be able to add additional software using an USB key attached to the router;

we want to have the new file system image with the same size as the old file system image because we don’t know the meaning of many fields in the firmware file and we don’t know how to change them to reflect a different size in the file system image;

We can reassemble the firmware file putting together all the pieces with the “dd” command, excluding the signature;

In this way the unsigned firmware file is ready. The problem is that this file cannot be loaded into the router with the standard upgrade procedure because it fails the signature test.

Loading the unsigned firmware

- The upgrade script checks the firmware signature:

```
sig_verify $file 2> /dev/null
ret_code=$?
```

- As root copy the upgrade script in /tmp
- Modify it:

```
sig_verify $file 2> /dev/null
ret_code=0
```

- Temporary replace it with mount:

```
mount --bind /tmp/upgrade.sh \
             /usr/sbin/upgrade.sh
```

- Do the upgrade through the web interface

The problem is that the upgrade script checks the firmware signature with the “sig_verify” command an then returns the result of this command

As root we can copy the upgrade script in /tmp;

Modify it to return always zero, that means always OK also if the signature check fails;

We can temporary replace the original upgrade script with our own modified version using the “mount –bind” command.

The “—bind” option in the mount command is usually used to mount an existing folder over another folder, but it can be also used, as in this case, to mount a single file over another. In this way it is possible to temporary replace files also in read only file systems.

We can now start the firmware upgrade using the web interface and, in this case, the upgrade will succeed.

In this way we have achieved our original goal that was to modify the firmware of our router.

In my Github repository you will find all the documentation and scripts for this project, including the firmware modification kit.

Summary

- Reverse engineering can be really challenging
- Clearly define the limited scope of your reverse engineering project
- Start gathering information following the easiest path first
- If some information is missing or difficult to get move forward, go back only if absolutely needed
- Search on Internet for known vulnerabilities
- Select to hack processes running as root and with a large attack surface

Summarizing we can say that:

- Reverse engineering can be really, really challenging;
- It is very important to clearly define the limited scope of the reverse engineering project to remain always focused on the target;
- When we gather information we have to follow the easiest path first principle; this means that if some information is missing or difficult to get we move forward anyway and we return back only if absolutely needed. For example I wasn't able to identify the JTAG interface, but I moved forward anyway and I succeeded;
- Always search on the Internet for everything, including vulnerabilities; It is always the first easiest step, because if someone else has already done the job for us, this will save us a lot of time and troubles;
- When we have to select to hack a process running as root, we have to start from the process with the larger attack surface.

Useful Links & Documentation

GitHub repositories related to the Home Router Example

- **Adbtools2**, Tools for hacking ADB Epicentro routers, including firmware modification: https://github.com/digiampietro/adbtools2
- **Buildroot-armv7**: a set of scripts, configuration files and Buildroot external tree to setup a Qemu emulation environment to run and reverse engineer the Netgear DVA 5592 executables: https://github.com/digiampietro/buildroot-armv7

Reverse engineering and physical disassembly

- Introduction to reverse engineering, Mike Anderson, Embedded Linux Conference 2018, slides and videos:
  https://elinux.org/images/c/c5/IntroductionToReverseEngineering_Anderson.pdf
  https://www.youtube.com/watch?v=7v7UaMsgg_c

Recommended Books

- *Chris Simmonds* - Mastering Embedded Linux Programming - Second Edition - *Packt Publishing 2017*
- *Norman Matloff , Peter Jay Salzman* - The Art of Debugging with GDB , DDD and Eclipse - *NO STARCH PRESS 2008*

At the end of this presentation, that you can download from my personal website, you will find useful and interesting links.

In particular you will find link to my Github repository containing what I have done on this router including the “firmware modification kit”;

You will find also a link to another Github repository with everything needed to setup the emulation environment.

Useful Links & Documentation

Hardware tools

- Bus Pirate: *http://dangerousprototypes.com/docs/Bus_Pirate*
- Jtagulator: *http://www.grandideastudio.com/jtagulator/*
- J-Link debug probes: *https://www.segger.com/products/debug-probes/j-link/*

JTAG and UART interfaces

- Popular pinouts: *http://www.jtagtest.com/pinouts/*

Software

- Buildroot: https://buildroot.org/
- Putty terminal emulator: *https://www.putty.org/*
- OpenOCD (Open On-Chip Debugger) provides debugging, in-system programming and boundary-scan testing for embedded target devices: *http://openocd.org/*
- Wireshark, network protocol analyzer: *https://www.wireshark.org/*
- Binwalk, firmware analysis tool: *https://github.com/ReFirmLabs/binwalk*
- Jefferson, JFFS2 filesystem extraction tool: *https://github.com/sviehb/jefferson*

You will find also links to everything I talked about today including Bus Pirate, Jtagulator, JTAG pinouts, OpenOCD, Binwalk and so on.

If you have some questions I will be happy to answer, you can contact me at the email address v@ler.io citing this presentation.

Thank you very much for reading this presentation, thank you for your time and your patience, I know that has not been easy to read till the end.