Toward a generic and secure bootloader for IoT device firmware OTA update

1st Saad EL JAOUHARI
Institut Suprieur d'Electronique de Paris (ISEP)
Issy-les-Moulineaux, France
saad.el-jaouhari@isep.fr

2nd Eric BOUVET
Orange Labs
Cesson Sévigné, France
eric.bouvet@orange.com

2022 International Conference on Information Networking (ICOIN) | 978-1-6654-1332-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICOIN53446.2022.9687242

***Abstract*—The Internet of Things (IoT) devices market has shown strong growth in recent years. Time to market has become essential to be competitive, the faster a competitor develops and integrates his/her product, the more likely he/she is to dominate the market. This competition leads to critical software problems in the systems due to lack of testing or short development times. Lots present some vulnerabilities that can be exploited by attacks via botnets or malwares. Moreover, they are subject to huge number of 0-days that need quick intervention to maintain the security of the environment in which the IoT device is deployed in. For this purpose, the quick update of the firmware of these devices via patches is the most effective solution to counter these attacks. In this process, to operate embedded systems' set-up, control and supervision, an important component called the *bootloader* have to be implemented. This piece of code can manage and execute boot sequence and launch the firmware. However, without any recommendations or references, currently, there is no generic bootloader for all the IoT device, but there are several bootloaders specific for a particular or a group of hardware or kernel. This paper aims to analyze some of these bootloaders and develop a minimal generic bootloader implementing a firmware Over-The-Air update for constrained IoT devices. After analyzing several bootloaders and the OTA update process, a PoC of a bootloaders based on FreeRTOS, has been designed and implemented, and which allows to perform firmware verifications and OTA updates.**

I. Introduction

It is very likely that by the year 2025, the number of connected devices will be more that the double of the humans on earth. These equipment are continuously being produced by many manufacturers. Moreover, in order to catch up to the "Time To Market" race, the priority is given to profitability and performance, to the detriment of robustness and safety. Thus, a large majority of these devices are highly vulnerable to old and new attacks launched by botnets or other malware. Furthermore, most of these devices do not have enough capabilities to deploy the necessary security mechanisms to counter these attacks and do not integrate proper update system that can patch these vulnerabilities.

Currently, new technologies allow more or less secure firmware update to correct any security flaws and to respond to eventual zero-days vulnerabilities, depending on the capabilities of these objects. In this process, the bootloader is an important element which allows the flashing and the installation of the new firmware image or modules. It is the piece of code that will launch the main application. This software can also be used to manage firmware updates remotely. The manufacturers of microprocessors (such as ARM, Intel, Texas Instruments, Atmel, etc.) developed their own bootloaders to better control and debug these updates. Moreover, in the absence of genericity of IoT's hardware modules, bootloaders becomes numerous and most often specific to the same family of microcontrollers. Hence, it becomes necessary to study their mechanisms and to look into the development of a generic module, which is also driven by the presence of very few standards and recommendations for the particular bootloader of the IoT devices. Following the example of universal ports for smartphones, it would be interesting to look into universal processes for managing IoT connected devices. Thus, this work studies the concept of a generic bootloader mechanism and extends it with the conception of secure update mechanism for the IoT that guarantees the integrity and the authenticity during the update process.

The main contributions of this work are: 1) A state of art and a comparison of some popular bootloaders currently used in constrained IoT devices. 2) A Generic bootloading process for typical IoT devices. 3) A Proof of Concept of the firmware over the air process, which uses the generic bootloader on top of ones of the most used OS (i.e., FreeRTOS). 3) The discussion of a secure and generic bootloading process that guarantees the integrity and the authenticity of the received firmware image. The remaining of the paper will be structured as follow: the paper starts with a background overview related to IoT devices, the bootloader and the firmware update process, in sectionII-B. Then, in sectionIII, it presents the related works, followed by the presentation and choices related the generic bootloader, in sectionIV. Next, the sectionV provides the implementation details. Finishes with a discussion in sectionVI and a conclusion in sectionVII.

II. Background

This section provides an overview of some important notions to have a clear understanding of the different challenges related to the firmware update for IoT devices.

A. Generic/Typical IoT Device

With the mass of definitions related to the structure of the IoT device, it is important to have a precise idea about

what is an IoT device. For this purpose, the Fig. 1 provides a generic and typical stack representation of the different hardware and software that can be involved in an IoT device. Within this representation, a constrained devices might have less elements and a reduced stack, for instance a device without an OS or without flash, etc.

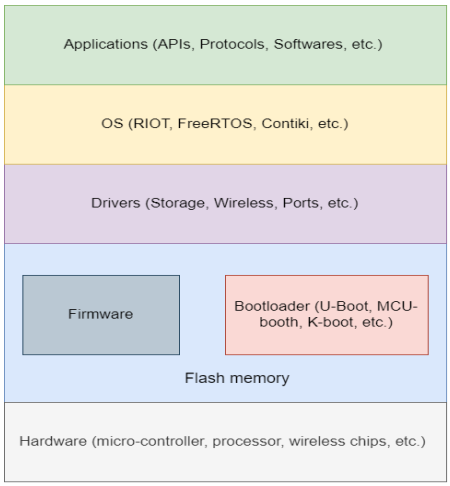

Fig. 1. Typical IoT device stack

B. Bootloader

Generally, a bootloader is a software that launches the kernel or another application during the boot sequence. Implemented in a ROM, NOR, NAND or MMC type microcontroller, this piece of code launches the kernel from a fixed address to the given memory (RAM). The bootloader typically contains minimal operations such as kernel release steps and cryptographic operations to read and verify the integrity of the firmware stored and/or received via an upgrade process. The transition from the bootloader application to the main kernel is achieved by using the partition table which allows the bootloader to know the address of the kernel. This partition table is typically used for a PC bootloader. However, for constrained systems, there is often no partition, just a flash address to target the kernel.

C. Over the Air Updates

In practice, there are multiple ways of updating an IoT devices, since different manufacturers follows different approaches depending on their tools, infrastructures and strategies. The updates the wired updates via a USB port, the wireless Over the Air (OTA) updates, and also they can be either full or partial (e.g., only a piece of the firmware or the OS), they can include signature and integrity verification or not, they can require an attestation of delivery and installation or not, and so on and so fourth. In this work, a special focus if given to the OTA updates of the full firmware. An OTA update refers to the transmission of a piece of software (will be referred as firmware from now onn) to an embedded system (mobile, tablet, connected objects) wirelessly ”over the air”, usually via Wifi, GSM/GPRS, BLE etc. It concerns mainly the new firmware version in order to either add new functionalities or to fix some bugs or vulnerabilities. The bootloader will then receive the firmware, implements checksum operations, verifies the integrity and authenticity if possible, installs it and then reboots.

The OTA update module can be implemented in the IoT device at two levels:

- At the bootloader level: the OTA and firmware verification process are executed at the bootloader stage and is independent of the application. This choice allows a bootloader/application separation but increases the footprint of the first one (bootloader size).
- At the application level: the implementation of the ota process at this level allows to decrease the bootloader footprint. The OTA process is dependent on the application, but independent of the bootloader.

Moreover, the update can be either partial, by updating only the application for instance, or complete, by updating the system file, the bootloader and the application. This update process may also be interrupted due to a loss of connection between the server and the device, the device’s extinction or simply a power cut. It is therefore necessary to schedule update operations in advance in order to avoid blocking the device. In this work, a special interest was given to the OTA update at the application level, to reduce the most possible additional footprint impact on the constrained device.

III. Related work

In the literature there are only few works that studies the IoT device’s bootloader, and even less for having a generic bootloader, OS and hardware independent. In [1], the authors present the concept of the remote programming, configuration, and monitoring system for development and testing which can be applied to resource constrained wireless sensors and IoT nodes, based on a general purpose microcontroller, unit or a field programmable gate array (FPGA) chip. The remote management interface can be run on many different operating systems via to the utilization of the Leshan LWM2M implementation with Web-based GUI. In another work [2], the authors provide an overview of the threats targeting Bootloader and Firmware Update (BFU)s, and existing protections. The paper covers the hardware and software attacks. The authors also provide a generic and typical workflow of a secured firmware update at the bootloader lever, mainly inspired from Atmel architecture [3]. The latter’s security is mainly based on the integrity and the authenticity of the downloaded firmware. However, there is not evaluation nor implementation of this workflow, since the main purpose of the paper is the study of the security aspects and threats of the IoT devices itself. Next, in [4], the authors propose SecFOTA an enhanced bootloader, which decides the kind of update used based on the user’s settings, which a special focus on the security. The concerned firmware in this case is the RIOT-OS application running

on Atmel SAMR21-Xplained Pro Evaluation Board (Cortex-M0+, 256kB Flash,32kB RAM). The solutions uses different partitions, or areas, in the flash memory run the firmware update. The firmware is loaded into an update area which is used to run integrity and version checks before copying the image into a live partition. Moreover, the last installed version is copied to a backup area, and in case of failure, a rollback to this version is done. Atmel introduces in [3] an in-field firmware upgrading and describes various aspects regarding the implementation of a safe and secure bootloader for Atmel SAM3 and SAM4 families of microcontrollers. The document discusses several design considerations in developing this kind of software, the most important ones are: the bootloader sequence diagram for firmware upgrade, which includes firmware integrity verification and code encryption; the memory partitioning (i.e., single or dual banked memory), which makes it possible to have at least one working version of the firmware in the device at anytime, in order to avoid firmware corruption in case of an issues such as power or connection loss; safety solutions so as to prevent safety related errors from happening (i.e., transmission error, transmission failure, information loss); Security solution, by enforcing the privacy, integrity and authenticity features, via hash functions, digital signatures, message authentication codes (MACs) and encryption in order to prevent attacks (i.e., unauthorized device, third party firmware, firmware alteration, reverse-engineering).

IV. Presentation and choice of generic bootloaders for the IoT

A. Hardware environment

For the PoC, the following hardware have been used:

- STM32F411re Nucleo DIscovery and BLE Nucleo extension: this development kit, developed by STMicroelectronics, has a cortex M4 processor. The presence of an st-link, which is a programmer and debugger, allows the firmware to be flashed via USB. The X-NUCLEO-IDB05A1 BLE expansion board allows the board to be controlled via BLE.
- Hexiwear: Based on a Cortex M4 processor, the Hexiwear development kit is the result of a collaboration between NXP and MikroElektronika.
- ESP32 WROVER KIT: Cortex M4 development module integrating a WiFi SOC and Bluetooth. The presence of an integrated LCD screen allows the development of graphical user interfaces.
- Espressif ESP8266: is a microcontroller integrated circuit allowing a WiFi connection. Widely used for Arduino projects, it is also possible to use it under FreeRTOS or Lua OS. Moreover, NodeMCU, which is an open source firmware for IoT and uses only few Lua script lines, is deployed inside the ESP8266.

Each cards has specific tool-chains. The chosen bootloaders for the PoC was influenced by the available environment.

B. State of the Art of some popular bootloaders

The sequential analysis of the bootloader was necessary in order to be able to master the bootloading process. The next sections present some popular bootloaders for embedded systems and the chosen one for our PoC.

1) U-boot: Now known as Das U-boot is an open source primary bootloader for embedded devices. It is a very generic solution and used by a large community. However, It requires lots of resources for a constrained memory SsC. Moreover, it is currently available for a number of well-known computer architectures such as ARM, PowerPC, x86 and MIPS.

2) MCUBboot: Bootloader is developed by NXP and previously named Kboot (Kitenis Bootloaders). MCUBboot is intended for 32-bit microprocessors. It is operational with the Apache Mynewt, RIOT and Zephyr operating systems. Zephyr is an OS with a small footprint especially for embedded and constrained systems. It supports a large number of architectures such as ARM Cortex-M, AVR MIPS etc. Moreover, it is supported by many architectures and several OSs (Zephyr, Mynewt, RIOT) and Open source. However, it is too large for constrained memory SoCs.

3) FreeRTOS: Is an open-source kernel developed in 2003 under the MIT license. FreeRTOS is known as covering a large environment of architectures (TI, Microchip, Atmel, NXP, Intel, etc.) with 18 compilation strings and having a very small footprint (4Ko to 9Ko). In 2014, FreeRTOS is one of the leading RTOS on the embedded market. Furthermore, in addition to be supported by numerous architectures, it provides real-time processing, and very active by it large community. However, it is characterized by it high complexity of implementation if using multi-tasking.

4) Expressif Rboot: Is designed to be a flexible open source boot loader, only for Espressif devices, a replacement for the binary blob supplied with the SDK. This bootloader is supported by multiple Hardware architecture Microcontrollers: CC3220, CC3200, ESP32, ESP8266, STM32F4 and multiple hardware platforms: TI CC3200, TI MSP432, NRF52, STM32, PIC32, ESP8266, ESP32 and it is Open source. However, there is no generic aspect.

5) Tiva: The Texas Instruments Tiva bootloader is a small piece of code that can be programmed at the beginning of flash to act as an application loader and an update mechanism for applications running on a Tiva ARM Cortex-M4-based microcontroller. The bootloader can be built to use either the UART, SSI, I2C, CAN, Ethernet, or USB ports to update the code on the microcontroller. The bootloader is customizable via source code modifications, or simply deciding at compile time which routines to include. Since full source code is provided, the bootloader can be completely customized.

6) LK (Little Kernel): Is a tiny operating system suited for small embedded devices, bootloaders, and other environments where OS primitives like threads, mutexes, and timers are needed, but theres a desire to keep things small and lightweight. On embedded ARM platforms the core of LK is typically 15-20 KB. LK is the Android bootloader and

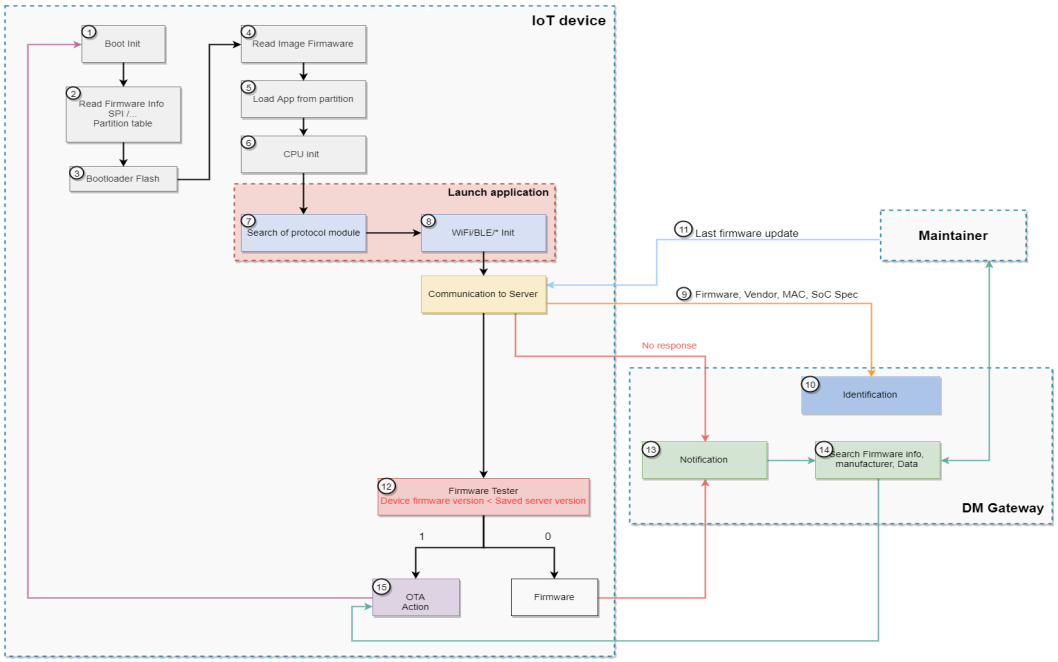

Fig. 2. Generic bootload process at the application level

is also used in Android Trusted Execution Environment - "Trusty TEE" Operating System.

For demonstration, we decided to use FreeRTOS which implements its own bootloader. The technical achievements using this OS and the PoC will be exposed next.

V. Implementation

In this section, technical details about the PoC of the generic bootloader for update IoT devices are provided. For this purpose, three entities interact to update an IoT device: Maintainer, Device Management gateway, and IoT device.

A. Maintainer

As a first step, a Maintainer server was developed. It intends to simulate multiple manufacturers providing multiple updates of firmware for various IoT devices. In this case, and to simplify the PoC, a simple HTTP server was implemented and enhanced with a JSON API. The latter allows a device to know the status of the IoT device (i.e, the firmware version) to make an update request decision. If so, it provides the IoT device with the URL to the new firmware image. Basically, the server stores the card id, firmware version and manufacturer in the form of a JSON Object. The design and the implementation of the firmware update process including the security consideration and the use of firmware manifests [5], are currently in progress, and will be exposed in future works. Furthermore, since this is only a preliminary work, a more secure PoC including end-to-end security from this Maintainer to the IoT devices is under investigation. Solutions including pre-shared asymmetric keys, CoAP-DTLS, or via a key establishment protocol are considered.

B. Generic Bootloader

To realize the PoC, we decided to implement additional blocks over the base codes of FreeRTOS. Then, a simple and generic bootloader that manages also the firmware updates was implemented. Thus, it is necessary to explain first the different steps of FreeRTOS from the hardware initialization to kernel launch, which will be explained later in the update process. FreeRTOS allows the initialization of the different partitions like the partition table which has the firmware address. Hardware initialization is illustrated in Fig. 2, from step 1 to 6. These different steps are unmodified libraries of FreeRTOS. After these steps, the application is launched.

C. OTA firmware update

To completely separate the OTA bootloader from the application, additional processes to perform updates during the boot process were implemented. In the case of the ESP32-WiFi, the bootloader will first initialize and configure the WiFi (i.e., steps 7 and 8 shows the connection steps performed by the bootloader). Next, in the step 9, the bootloader (the communication server inside the IoT device) connects to the Device Management (DM) Gateway and will initially send some information to declare its identity (identification process, step 10): name, communication protocol, vendor, firmware version. Since FreeRTOS allows programming in the form of tasks that run in parallel and that can communicate, the bootloader will also query via an HTTP GET request, the Maintainer that contains the firmware information, in the step 11, in parallel with the execution of the identification task. In order to offload some of the computation from the constrained devices, these information are then relayed to

the DM gateway. Another alternative is to delegate all the interaction with the Maintainer to the DM gateway since the latter already knows the device's firmware version. If the firmware registered on Maintainer is different from the one in flash memory (i.e., step 12), the bootloader will start an update process (step 15). Via an HTTP GET request, the bootloader will retrieve the firmware new_firmware.bin, which is stored in a directory of the Maintainer server. If the binary file new_firmware.bin is not present in the directory or the connection to the remote Maintainer has failed, the bootloader notifies the DM gateway of this error, as in step 13. The DM gateway will then query the Maintainer via an HTTP GET request and retrieve the firmware to send it directly to the module, as in step 14. This principle takes example on web services to find driver updates for computers.

Similar steps can be applied to different communication protocols such as BLE, Zigbee, Z-wave and so one, by adding simply some adapters. In case of failure, the bootloader launches the present firmware which will be executed until the next attempt of the system. In addition, an alert to the DM gateway is sent in the background. The complete diagram is shown in Fig. 2. Moreover, in order to demonstrate the previous processes, a PoC has been implemented on top of the FreeRTOS OS for embedded systems, via two use cases in order to apply the firmware update: 1) a first one by using only WiFi and 2) a second use case of using BLE. This will be the main purpose of the next sections. Note that, since HTTP+JSON download can be heavy for a light loader. It is especially suitable for wifi only. The delegation of these tasks to a DM gateway for instance is a better alternative in order to offload and lighten the update process.

D. OTA firmware update via BLE / WiFi

Updating via BLE/Bluetooth is more complex than updating via WiFi. Indeed, the speed of Bluetooth (3 Mbit/s) and even more for BLE (1 Mbit/s) is much lower than that of WiFi (6-30 Mbit/s). Thus, the transfer time is extended and errors are more frequent. For some IoT devices, the manufacturer uses WiFi in order to transfer updated firmware, applications or just some module to an intermediary, which can be a smartphone or a gateway, and then the latter connects to constrained device via a more constrained communication protocols (e.g., BLE, Zigbee, Lora, etc.) and transfer the updated piece of code. In our use cases, the smartphone connects to the IoT device via BLE. The connected object is in server position and is waiting for a connection from a client (here the smartphone or other host). An android application was also implemented and used in order to scan the BLE/Bluetooth environment, to connect to the device, to send data via a GATT service. Using this application, data in text or binary form can be sent. Depending on the requests sent via the GATT protocol, the IoT will perform an actions.

The Fig. 4 summarizes the two update techniques that uses the loader in order to perform the firmware over the air update. The first technique in red, allows you to perform updates without using WiFi. This technique, limited by the problems related to bluetooth bitrate, makes it possible to update the modules containing only BLE or Bluetooth. This is interesting for devices such as smart watches which generally do not integrate an update process and Wi-Fi. The second, represented by the blue arrows, illustrates the method of sending WiFi passwords via BLE. Then the module will connect to the Gateway (e.g., smartphone) and the WiFi OTA will be engaged. This method, very often used, is only applicable if the module has a WiFi SoC and BLE. The last one in green, represents the update by WiFi without the need of a gateway, as is the case for a large segment of IoT devices directly connected to Internet such as IP camera.

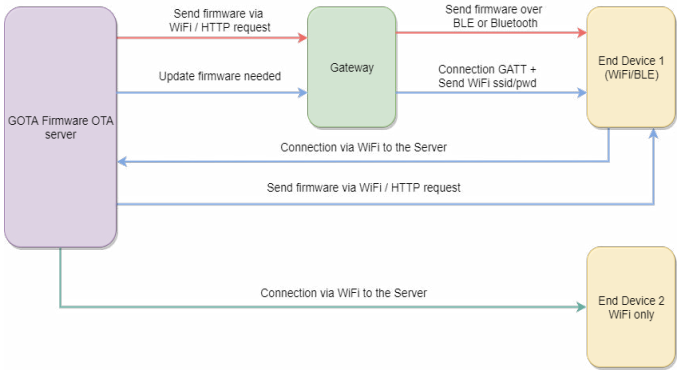

Fig. 4. Over the Air firmware update architecture via WiFi and BLE

VI. Discussion

This work is a starting line toward a generic and secure firmware OTA updates for IoT devices, and in particular the most constrained ones. It consists of a first and important brick, since it determines the way the bootloader is conceived in order to receive the new firmware image, how it is downloaded and how it is installed. However, we argue that there are several drawback in the current proposition that requires further and deeper investigations in order to provide a generic, portable, robust and secure bootloader for IoT and in particular for the constrained ones. In this section, some of there drawbacks are discussed in particular the ones related to the security and the evaluations.

An important aspect that was not addressed in this paper, since it requires additional capabilities, is the security all along the bootloading process. This challenge is tightly related to the hardware capabilities in the SoC. We argue that it is very important to be able to at least perform some integrity and authenticity verification in order to make sure that the received firmware image was not altered and that it is coming from the right benign origin. The Fig. 3, shows the ideal bootloading process with these verifications in mind. In this process, it is important to have an integrity and authenticity verification right after the version verification in the IoT device, if the latter downloads directly the firmware from the maintainer. Furthermore, if the IoT device relay on a DM gateway in order to recover the firmware image from

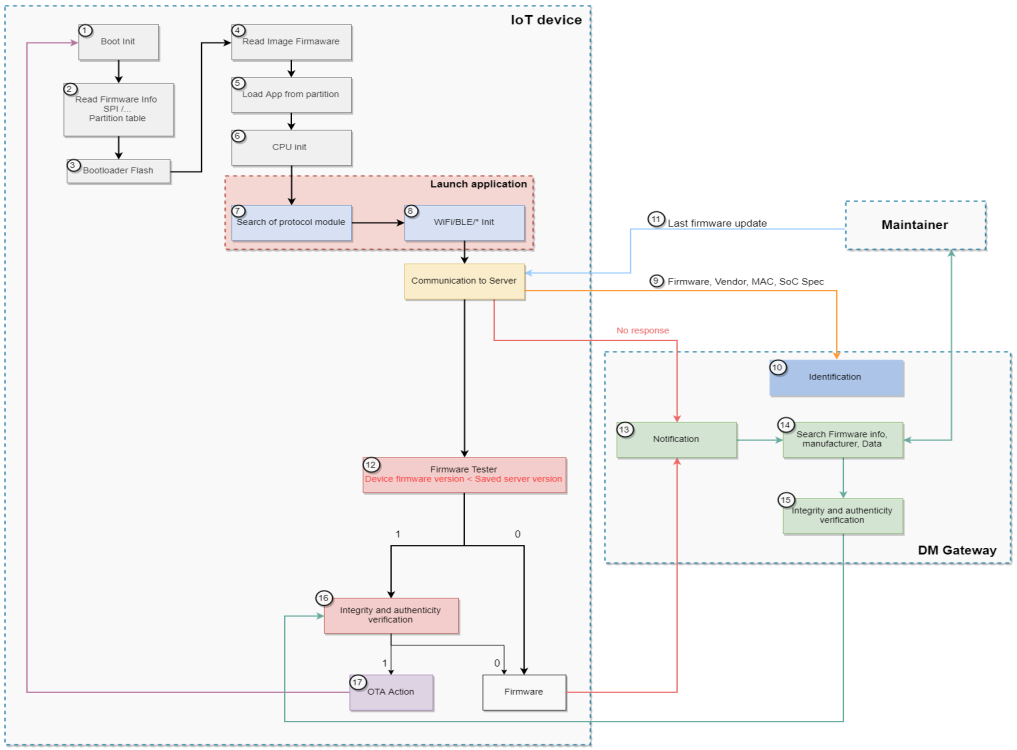

Fig. 3. Ideal secure and generic bootloading process at the application level

the maintainer, the gateway needs first to verify the version of the firmware image and then performing a first integrity and authenticity checks before sending them using the underlying communication protocol to the IoT device.

The IoT device, will again perform the integrity and authenticity checks and then install the firmware image if the checks passed. The second check need to be done in order to make sure that a MITM cannot alter the firmware since it is usually sent in clear via a short range protocol such as BLE. Finally, it is important to study the threats against the bootloader such as the Rollback attack, where the attacker can resend a valid but old firmware version to the devices [6], and the Firmware reverse engineering, by reverse engineering the binaries into assembly in order to analyze the functionality and to get access to secret data [7] [6], and to prepare some countermeasures in order to counter them. Moreover, in order to make sure that the proposed solution is generic and secure enough, multiple tests and evaluations need to be done in order to study the impact of the additional footprint on the different classes of IoT devices, in particular on the most constrained one (e.g., with Cortex M0).

VII. Conclusion

This work presents a first step toward developing a generic bootloader for the different type of IoT devices, and in particular for the most constrained ones. The main objective is to be able to update different IoT devices from different sources in a simple way, and to simplify the task for developer. The solution has been also extended with a design of a secure update mechanism for the IoT that guarantees the integrity and the authenticity during the update process. However, multiple challenges still need to be solved such as the overhead of adding the security mechanism, the presence of different bootloader in the same IoT device, the scheduling of the firmware update (including the interruption of the devices tasks and the devices in the sleep mode).

References

[1] T. Michalec, M. Wojczuk, R. Brzoza-Woch, T. Szydo, Remote programming and reconfiguration system for embedded devices, in: 2019 Federated Conference on Computer Science and Information Systems (FedCSIS), 2019, pp. 467–470. doi:10.15439/2019F170.

[2] L. Morel, D. Courouss, Idols with feet of clay: On the security of bootloaders and firmware updaters for the iot, in: 2019 17th IEEE International New Circuits and Systems Conference (NEWCAS), 2019, pp. 1–4. doi:10.1109/NEWCAS44328.2019.8961216.

[3] Atmel, Atmel AT02333: Safe and Secure Bootloader Implementation for SAM3/4, 2013.

[4] S. e. a. Schmidt, Secure Firmware Update Over the Air in the Internet of Things Focusing on Flexibility and Feasibility, Internet of Things Software Update Workshop (IoTSU), At Dublin (June).

[5] B. Moran, H. Tschofenig, H. Birkholz, K. Zandberg, A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest, Internet-Draft draft-ietf-suit-manifest-04, Internet Engineering Task Force, work in Progress (Mar. 2020).
URL https://datatracker.ietf.org/doc/html/draft-ietf-suit-manifest-04

[6] B. M. et al., A Firmware Update Architecture for Internet of Things, Internet-Draft draft-ietf-suit-architecture-08, Internet Engineering Task Force, work in Progress (Nov. 2019).
URL https://datatracker.ietf.org/doc/html/draft-ietf-suit-architecture-08

[7] J. Park, A. Tyagi, Using power clues to hack iot devices: The power side channel provides for instruction-level disassembly., IEEE Consumer Electronics Magazinedoi:10.1109/MCE.2017.2684982.