Secure and Robust Firmware Updates of IoT Devices

Jannik Beyerstedt

13.07.2017

Abstract. This paper will summarize the results of my Bachelor's thesis [4], which had to be written in German. Within the thesis a firmware update process was designed, which is robust and secure, but also suitable for low power IoT devices. The update procedure will use a special file format for delivering the firmware, which is secured by a digital signature verifying its origin and encrypted to ensure the confidentiality of the firmware. For conducting experiments and tests, the update infrastructure was implemented and deployed at a small scale. A complete example infrastructure does not only consist of the components on the IoT device, but also the tools and build scripts needed to generate and deploy an update file. The conducted tests should verify the robustness and combined safety and security requirements of the system.
Keywords: security, robustness, firmware update, FOTA, RIOT, CoAP

1 Introduction

The Internet of Things (IoT) is a new and evolving part of the Internet, which will lead to special challenges regarding the security of the whole Internet. Even if IoT devices are mostly powered by microcontrollers, they are still computers, which have the same security problems as every other computer. But the IoT is special, because there will be a big number of devices permanently connected to the Internet.

In the past there already were some IoT-targeted exploits, which had a quite big impact. One example was the "Mirai" malware in October 2016, which lead to massive attacks at the DNS infrastructure in the USA by using Internet-connected surveillance cameras and video recorders.[1]

These attacks show, that there is a need for software updates on IoT devices, so that manufacturers can patch known vulnerabilities. This work will search for an answer to

[1] https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/

the question, how a save and robust firmware update process can be done on IoT devices, especially the ones with little resources.

At first the terms security and robustness must be clarified in this context to defined the basic requirements for the update process. After that, a process can be designed, which fulfills the requirements. For the transportation of the update data, a special file format is needed, which is part of the design as well.

To conduct tests on the update process, a small scale prototype will be implemented. The prototype will not only consist of an example application for the microcontroller, but also of the tools needed to create the update files itself. An ARM Cortex-M4 microcontroller from STMicroelectronics and the embedded OS "RIOT" [2] were choosen for the implementation.

This work will not take care of the migration of the configuration data saved by the application, because this is in the scope of the application itself. Also, we will not cover the details of the data transportation over the Internet and therefore assume, that the data will be transmitted without errors.

2 The Basic Requirements

At first we will evaluate, what is special about an IoT device. This will lead to the first set of requirements. The next sections will define some basic requirements and characterize the attacker to define requirements on security.

2.1 Characterization of an IoT device

The IoT was defined by the International Telecommunications Union (ITU) as a network of physical and virtual things, which enables advanced services for the information society [8]. This definition is quite broad so we cannot get any requirements from this. But this is also a feature of the IoT, that nobody really knows, what the IoT will be exactly.

On the other side it is unquestioned, that the IoT will lead to many devices connected to the Internet. The ITU states, that "the number of devices that need to be managed and that communicate with each other will be at least an order of magnitude larger than the devices connected to the current Internet" [8] . So just by the amount of devices, there will be big challenges regarding the security.

IoT devices will have a quite broad spectrum of capabilities and applications, such as "smart grid", "e-health" and "smart home". But the devices can be divided in groups, two of them being "general devices" and "sensing and actuating devices" [8]. While a general device mainly fulfills a non IoT-related job (such as a smart phone or a fridge), the other group consists of highly specialized devices.

In RFC 7228 [5] these devices are classified as "constrained nodes", which require special communication networks. The devices are constrained, because they only have a limited amount of power and energy to do their job. Also the program complexity is highly limited, because of the limited RAM and ROM space.

[2] https://riot-os.org

Because the sensing devices can be small and have a wireless connection, they could sometimes be installed in places, which will not be easily accessible. Therefore, the updates must be done fully automatically.

These constraints and other considerations lead to the requirements stated in the next sections.

2.2 Basic Requirements

Robustness

The customer probably will not accept updates, if they lead to visible disadvantages, such long down-times or a deleted feature. So in summary, there are these requirements:

- Updates must be done automatically [19].
- A firmware update should only affect the availability of the device by a minimal amount [20, 19].
- A firmware update must not affect the provided functionality of the device [20, 19].
- A firmware update must not lead to an inconsistent or corrupted firmware on the device, even if the communication link is disturbed [1, 2, 18, 9, 11].
- The update process must ensure, that an update will only be installed, if it is designed to work on this device [2, 18, 9, 7].

2.3 Security of the Update Process

The update process has to ensure the integrity, authenticity and confidentiality of the firmware. As seen in presentations of microcontroller manufacturers, confidentiality seems to be a quite important topic [18, 10]. Therefore the first group of attackers are competitors, which want to gain knowledge by getting the firmware data or counterfeits want to create a plagiarism.

Another type of attacker is formed by malicious hackers, which want to make profit from hacked devices for example by selling data or by using the power of a big number of devices for a bot net. A possible attack vector will be the communication between device and update server to achieve, that a manipulated firmware is installed on the device.

The last possible group of attackers are intelligence and other governmental agencies. But this group will not be covered in this analysis, because they have too many and unknown resources to conduct attacks.

All in all there are the following requirements regarding the security of the update process.

Security Requirements

- The update process must ensure, that firmware updates will only be installed, if the original manufacturer created them [1, 2, 18, 9, 12, 19, 20].

- The update process must ensure, that the firmware data could not be manipulated or changed during the transmission [1, 2, 18, 9, 12, 19, 20, 11].
- The update process must ensure, that the firmware data could not be read or accessed by third parities [1, 2, 18, 9, 20].
- The update process must offer the opportunity to create updates, which are only valid for a specific device. If this opportunity is used, only the specified device should be able to read and install the update [1, 18, 19].
- The update process must ensure, that the installed firmware could not be manipulated [7].

Requirements for the Microcontroller Hardware

For a save and successful implementation and to minimize the attack surface, the microcontroller must meet a few requirements as well:

- All debugging interfaces should be disabled [9, 10].
- The flash memory of the microcontroller must not be readable over any interface to ensure the confidentiality of the firmware. The only exception of this rule should be the application code itself, which is located on the flash memory. Even the boot loader of the microcontroller-manufacturer, if present, must not read the contents of the flash memory. A modification of the flash memory must result in a mass-erase of the whole flash memory. [9, 10]
- The microcontroller must provide a watchdog timer to the update process [7].
- The microcontroller must have a feature to configure the memory location of the interrupt vector table.

3 Explaining the Design of the Update Process

In this chapter we will explain the basic process chain first and then get into the details of the file format. The third section will describe the details of the resulting update process, which will lead to some additional requirements completing the requirements of the last chapter.

On the device, there will be a boot loader managing the installed firmware. The term “firmware” in this case does not include the boot loader, but only the application which should run on the device. The term “application” will mainly be used to describe the source code, while the term “firmware” will be used for the compiled binary.

3.1 Basic Process Chain

The update process is based on a quite simple chain of data processing steps shown in figure 1. The application source code is compiled first, which will append some metadata

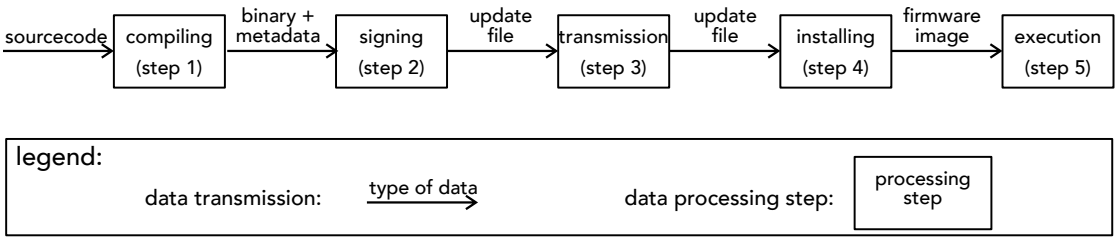

Figure 1: data flow in the process chain

to the binary as well. This data will be signed in the next step to create an update file. The details of the metadata and signing will be described in the next section.

The update file will then be transmitted to the device and installed, which leads to a so-called firmware image on the flash memory of the microcontroller, which can be executed in the last step.

3.2 Update File

The update file has to satisfy several requirements to help ensuring the security of the device and update process. A decision was made to focus on updating the whole firmware instead of allowing incremental updates, because the energy savings were not expected to be relevant (See [13]). Incremental updates are also quite a bit more complex to parse than simply copying the file content to the flash memory.

To ensure the integrity and authenticity of the update data cryptographic signatures are used. But the confidentiality has to be guaranteed as well by encrypting the binary in the update file.

For some other requirements, such as knowing which hardware the update file was designed for, there has to be a metadata section. Because this section has to be read by the boot loader as well to function properly, it has to be part of the installed firmware as well. This results in the schematic file format shown in figure 2.

Because the interrupt vector table of ARM Cortex-M microcontrollers can only be located at a special alignment, the binary section can not begin anywhere in the flash memory. The alignment is 256 byte for the Cortex-M0+ microcontrollers family [15, chapter 2.3.4] and it is 512 byte for the Cortex-M3, -M4 and -M7 families [16, 14, 17]. For this reason it was decided to build the update file around the start of the binary section as a reference point and add padding to the file header if needed to fill multiples of the alignment size.

The inner part of the update file will be called “firmware image” and is the part, which must be stored on the flash memory when installing a firmware update. The firmware image contains the unencrypted binary, the metadata and a signature securing these two sections. Metadata and signature are called firmware header.

To create an update file, the binary section is replaced by an encrypted version leaving the metadata and firmware signature untouched. The encryption can lead to a longer

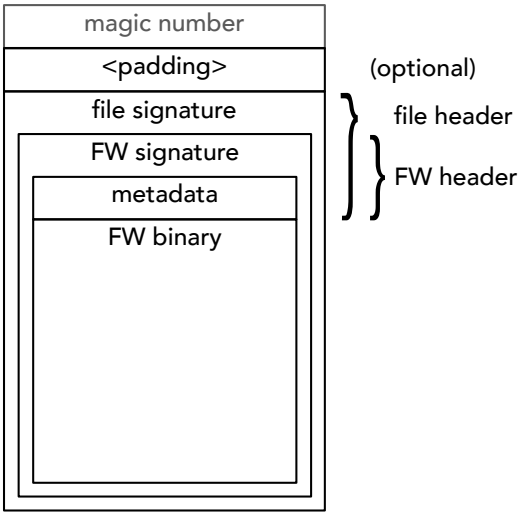

Figure 2: schematic structure of the update file

binary section, which is not relevant. Because the binary section has changed now, the firmware signature is not valid any more. But the update process has to check the integrity before installing or decrypting the binary as well. Therefore, a second signature is added, which secures the firmware header and the encrypted binary. This signature is called file signature and expands the firmware header to a file header.

If the file header does not fill a multiple of the vector table alignment, a padding could be added. Otherwise, the installation process must add the required padding itself when installing the update file.

To identify the update file, a magic number is added at the very beginning of the file. This magic number does not affect any alignment of the length of the padding. The value of the magic number was choosen to be the representation of the ASCII string `RIOTFW01` (`0x52 0x49 0x4f 0x54 0x46 0x57 0x30 0x31`). The two last characters could be used to differentiate future incompatible variation of the update file format.

3.2.1 Firmware Metadata

The metadata section will contain the following information:

A magic number was again choosen to identify the metadata block in the flash memory. The last character could be used to differentiate future incompatible versions.

To give the opportunity to create firmware updates, which are only valid for a specific device, the device's serial number can be defined in the metadata. The serial number of the STM32 microcontrollers is only 96 byte long, but 128 byte were choosen for being future proof and compatible to other microcontrollers.

To be able to check, if the update file contains a newer firmware than installed, a firmware version number is stored as well. This number must be an unsigned integer, which will be increased with each released version.

description	ID	length	data type or content
magic number	`magic`	4 byte	ASCII `OTA1` (`0x4f 0x54 0x41 0x31`)
chip's serial number	`chip_id`	16 byte	free data format
firmware version	`fw_vers`	2 byte	16 bit unsigned integer
firmware base address	`fw_base_addr`	4 byte	32 bit unsigned integer
length of the binary	`size`	4 byte	32 bit unsigned integer

Table 1: firmware metadata: data types of the metadata fields

Additionally the base address, for which the firmware was compiled (or being more precise: linked), must be known as well to save the firmware update to the correct location. This also is the address around which all sizes of the update file are calculated.

The last part of the metadata is the size of the unencrypted firmware binary. This number is needed, because there will be no file system on the microcontroller, which can provide the size of the whole file. But the size is needed to successfully calculate the signatures and for decrypting and copying the binary section.

This whole metadata section will use 30 byte, but the allocated space in the file format is 64 byte to being future proof.

3.2.2 Signatures

A cryptographic signature consists of an encrypted hash of the data, which should be signed. For the encryption an asymmetric cipher is used, for which the receiver has the public key and the sender has it's private key (just the other way around than normal asymmetric encryption). To check a signature, the receiver calculates the hash value of the data itself and compares the value with the decrypted signature.

As a hash function SHA 256 was choosen, because it is available in RIOT and could be found on microcontrollers, which contain a hardware module for cryptographic functions. This hash function is also recommended by the German Federal Office for Information Security (BSI) [6].

For creating a signature, an asymmetric cipher is needed, such as RSA. The disadvantage of RSA is, that quite big keys are needed for the recommended level of security. The BSI recommends a key length of more than 2000 byte [6, table 3.1], so the signature will be much longer that the encrypted hash value, which is 32 byte long.

Therefore the NaCl library[3] was found to be available as a RIOT package, which claims to implement fast and efficient ciphers especially for systems with small memory [3]. Because the default version of NaCl will choose between different implementations depending which is the best for your processor, it is not easily portable. So the TweetNaCl library[4] was created by the founders of NaCl, which condenses the NaCl library to just a single header and source file.

[3] https://nacl.cr.yp.to
[4] https://tweetnacl.cr.yp.to

The signatures of the update file will use the "Public-key authenticated encryption" named `crypto_box`, which isn't limited to signing data, but can transport any information securely. A special characteristic of the `crypto_box` is, that keys from both, sender and receiver, are used to encrypt the payload. Also there is just a fixed overhead of 32 byte. If a separate key-pair is generated for each device, a high level of security can be achieved, if it is needed to provide update only for individual devices. This can be the case for certification reasons in safety critical areas.

The keys of the `crypto_box` are 32 byte long and therefore exceed the recommended key length for elliptic curve cryptography by the BSI [6, table 3.1, page 29].

For the firmware signature 64 byte were allocated in the file format, which will be fully used by the SHA 256 hash and 32 byte overhead.

For the file signature 128 byte were allocated, because this signature should be able to transport additional information securely to the receiver. The next section will explain the details of the encryption of the binary section.

3.2.3 Encryption

The binary section will be encrypted with AES 128, because this cipher is well known and therefore tested. This also leads to this cipher being available in RIOT and hardware modules.

Because AES is a block cipher, the data to be encrypted must always be a certain length. To encrypt longer data, there are different operation modes from which Cipher Block Chaining (CBC) was choosen, but this is not relevant for the file format itself.

Because the `crypto_box` has the ability to transport any data, the symmetric key and the initialization vector needed by the CBC mode, will be randomly generated when signing an update file and transported to the receiver in the file signature. So there is no need to store this key on the device and a compromised key has a very little impact, because it is only used for one update file.

The key length of 128 can seem quite low, but it is in the recommendation of the BSI [6] and the secured information is was not classified as highly confidential to justify a longer key.

3.3 Resulting Update Process on the Device

To achieve a robust update process, there will be two firmware slot used on the device, so that one firmware can be executed while the other slot is updated. This also means, that in case of a failed installation, there is always the previous version as a backup.

But there is still the question, whether the boot loader or the running application should install the update. The download of the update file should be done by the application, because this is the only way to minimize downtime and keep the boot loader simple. The application will already have a network stack for it's normal function, but the boot loader does not need one if it doesn't need to download the update.

3.3.1 Trust Border Between Boot Loader and Update Module

To choose, if the boot loader is more trustworthy than the update module of the application, a few assumptions must be made:

1. The factory state of the device is trustworthy. This also means, that the firmware, which is written to the device at the factory, is not available to the attackers.
2. The flash memory is protected from read and write operations, which are not executed by the boot loader or application itself.
3. The cryptographic library and the used cryptographic functions have no known vulnerabilities.
4. The application does not manipulate the content of the flash memory or the cryptographic keys by it's own. Obviously the update modules can write to the flash memory.

If these assumptions are true, the boot loader is as trustworthy as the application in the factory state. This also means, that the update module can install the update as secure as the boot loader.

If the update module installs the update, an update of the boot loader could be done as well, but there is no backup in case of failure.

3.4 Additional Requirements

The design of the update process leads to additional requirements to the implementation of the update module, application, boot loader and update server.

3.4.1 Requirements Update Module

- The update module must provide a function to check for a new firmware at the update server.
- The update module must provide a function to download the update file.
- The update module must only install an update file, if the signature is valid and the metadata fits the device.
- The update module must provide a function to install the update file.
- The installation of the update file is done by verifying the update file according to chapter 3.2, decrypting the binary section and saving the data to the firmware slot defined in the update file.
- The update module must only install an update file, if the file was transmitted without errors.

- The update module must only install an update file, if the hardware ID in the metadata is the same as the hardware ID of the executed firmware. Additionally an update must not be installed, if the firmware version of the update file is smaller, than the currently executed firmware version.
- The update module must not write to the own firmware slot.
- The update module must detect an interrupted installation and should skip the download of an update file.
- The update module must provide a function to reboot the device.

3.4.2 Requirements Boot Loader

- The boot loader must try to start the firmware with the highest firmware version. Exceptions will be defined by other requirements
- The boot loader is only allowed to start a firmware, if the signature of the firmware is valid. If none of the installed firmware slots have a valid signature, no firmware should be started.
- The boot loader must detect an interrupted installation and should delete the slot with an incomplete firmware. Only by this action the update module can detect an interrupted installation and continue with a second attempt.
- The boot loader must detect, if a firmware is stuck or does jump back to the boot loader quickly every time it is stated. In both cases, this firmware is not allowed to be further executed and must be deleted.

3.4.3 Requirements Application

- The application should check for updates regularly. After a reboot, an update must be requested first to trigger the integrated check for an interrupted installation.

3.4.4 Requirements Update Server

- The update server must answer to CoAP-GET request to the URI `/update` whether an update is available for the requesting device. To be able to check for updates, the request must contain the currently executed firmware version, the required firmware slot number, the hardware ID and optionally the serial number of the device. The CoAP answer should contain the URI of the update file, if a new firmware is available or nothing, if no update is available.
- The update server must provide a possibility to download the update files.

4 Annotations about the RIOT Implementation

The requirements at the implementation stated in the last chapter are the basis of the prototype, but four of them will not be implemented. Automatic update requests are not necessary in the prototype, because the timing is quite easy and part of the application. Individually signed update files were an optional requirement and will only be needed in special cases and are therefore also not implemented. The update server and update module will not have the ability to actually download the update file, because the CoAP-implementation of RIOT does not include CoAP-block transfer and thus is not able to transmit high amounts of data. Alternatively a TFTP-server could have been implemented on the device, but this was not considered useful for the proof of concept.

This chapter will finally explain the architecture of the update module in it's first section. The following sections will continue with the general structure of the prototype and the architecture of the boot loader.

4.1 Architecture of the Update Module

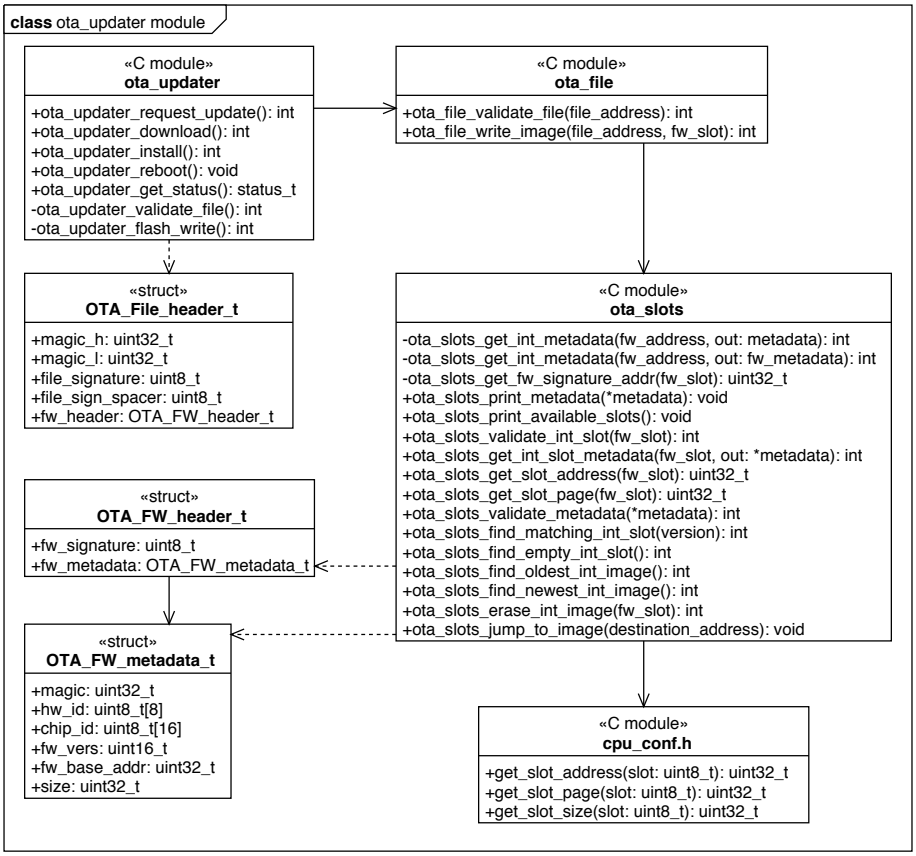

Figure 3: Class Diagram: C modules

The architecture of the update module and it's subcomponents is based upon a pull request issued in the GitHub repository of RIOT, which implements the support for firmware slots.[5] Because the code of this request already has functions to access the firmware slots, these were adapted to the own file format.

While the firmware slots are the base-level of the update module, the top-level functions were defined in the requirements of the last chapter, which will directly lead to the interface of the so-called `ota_update` module.

To achieve maximum flexibility at handling the update file, a third module will be needed to abstract the file layer to the `ota_update` layer. So there are three layers: `ota_slots`, `ota_file` and `ota_update`, which each define private and public functions and data structures.

All of them are displayed in figure 3 including definitions of the data structures and excerpts of some other RIOT modules. The figure uses the UML notation of a class diagram to express C modules, where "public" functions are the ones in the header file and "private" ones only occur in the source file.

Figure 4 displays how the interface of the `ota_updater` module should be used. The application should regularly call `ota_update_request_update()` and continue with the other functions, if there is an update available.

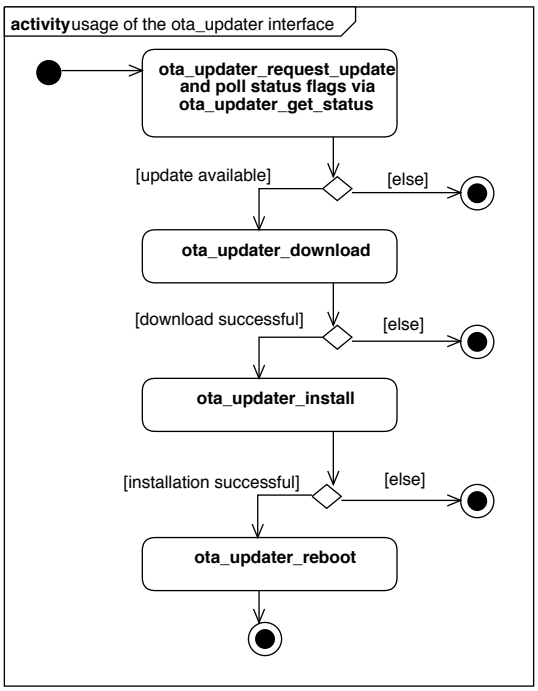

Figure 4: Activity Diagram: Usage of the `ota_updater` Module

Because of the asynchronous nature of the CoAP requests in RIOT, the

[5] https://github.com/RIOT-OS/RIOT/pull/6450

`ota_update_request_update()` function will not directly return, if an update is available. Thus the function `ota_update_get_status()` is provided to check the status of the request. The function for requesting an update will also check, if an interrupted update was detected and will return a special value in this case.

4.2 Structure of the Prototype

The prototype consists of an example application, the boot loader, several tools and the update server.

The example application will be connected to the network interface of the host computer via the Ethernet over Serial (ethos) protocol, which also allows to use the serial console simultaneously. For the prototype the update will not be done automatically, but the update module can be controlled via the RIOT shell.

For controlling the whole update process from a central point, a Makefile is provided in a separate folder of the RIOT-examples.

4.2.1 Example Application

The example application provides access to the interface of the `ota_updater` module via the RIOT shell and can be used for explorative testing. For the demonstration purposes it is sufficient to only connect the example application to the local network of the computer and thus there is no need for routing or global IPv6 addresses.

The serial number check is not implemented in this version, because the individual signing of update files is also not well supported by the signing tool chain and update server.

Another open question is how the cryptographic keys should be stored on the device. An answer to this topic depends quite strongly on the hardware used, so the implementation will simply use static variables to store the keys, which is not very secure, but works well while debugging the process.

On the STM32 microcontrollers, an One-Time-Programmable (OTP) area is available in the flash memory. As the name suggests, this area can only be written and not be erased, but it is not protected from modification until a write protection flag is set, which cannot be erase either. This area sound like a good place to store keys, but it has to be kept in mind, that this area will not be mass erased, if someone tries to save it's own firmware on the device and therefore the keys will be exposed to an attacker.

4.2.2 Tools

The process chain description of chapter 3.1 already mentioned, that some metadata must be generated while compiling and the update file must be signed in a separate step. Additionally there is the central Makefile with several different Makefile targets to control the generation of update files. The details of each tool are described in the corresponding Readme located in the directory of the tool.

Tool: Metadata Generator

The generation of the metadata structure was included in the RIOT build process and utilizes the `ota_update_filemeta` tool. The values of the metadata fields can be defined by the command line parameters.

Tool: File Signer

The file signer (`ota_update_filesign`) will convert the result of the compilation process (including the metadata generation) to an update file. For debugging purposes and to generate the initial flash image for the factory it is necessary to generate firmware images as well. Therefore this tool will compile to two command line applications for these two different purposes.

The paths to the key files will be set by command line parameters and there is also an option prepared to set the chip's serial number in the metadata.

Tool: Makefile

The central Makefile is the place, where the metadata values and different configuration values can be set. Because the build process for an update file is a bit different than for a normal application, the standard Makefile targets of RIOT are not available, but special ones.

For the prototype it is ok to sign the update files on the same computer as the application was compiled, so there is a Makefile target for that as well. In a real world application this target should submit the compiled application to some signing server instead.

Because the download of the update files was not implemented, the files can be written to the flash memory by this Makefile as well.

4.2.3 Update Server

As stated before, the update server only implements a very small proof of concept to show a CoAP communication between device and server. The server is implemented in Go, because there was a quite well documented library available.

The communication protocol is very simple and was already outlined in the requirements for the update server. The GET request of the client consists only of binary data to save compute power and mainly to keep the datagram small. Without the optional serial number it is only 11 byte long and has the following contents:

- Byte 0:1 – current firmware version (uint16_t, big endian)
- Byte 2 – firmware slot number for which an update is needed (uint8_t)
- Byte 3:10 – hardware id (see chapter 3.2.1)
- Byte 11:26 – optional: serial number (see chapter 3.2.1)

The hardware ID will be written as a 64 bit hexadecimal string in the Makefile to be provided to the metadata generator. Even if the generator and update module might use a 64 bit integer instead of a byte array for easier operation, the number must be transmitted as byte array (big endian 64 bit integer) in this message.

4.3 Boot Loader

The boot loader implements a quite complex activity diagram to satisfy all requirements, which is shown in figure 5.

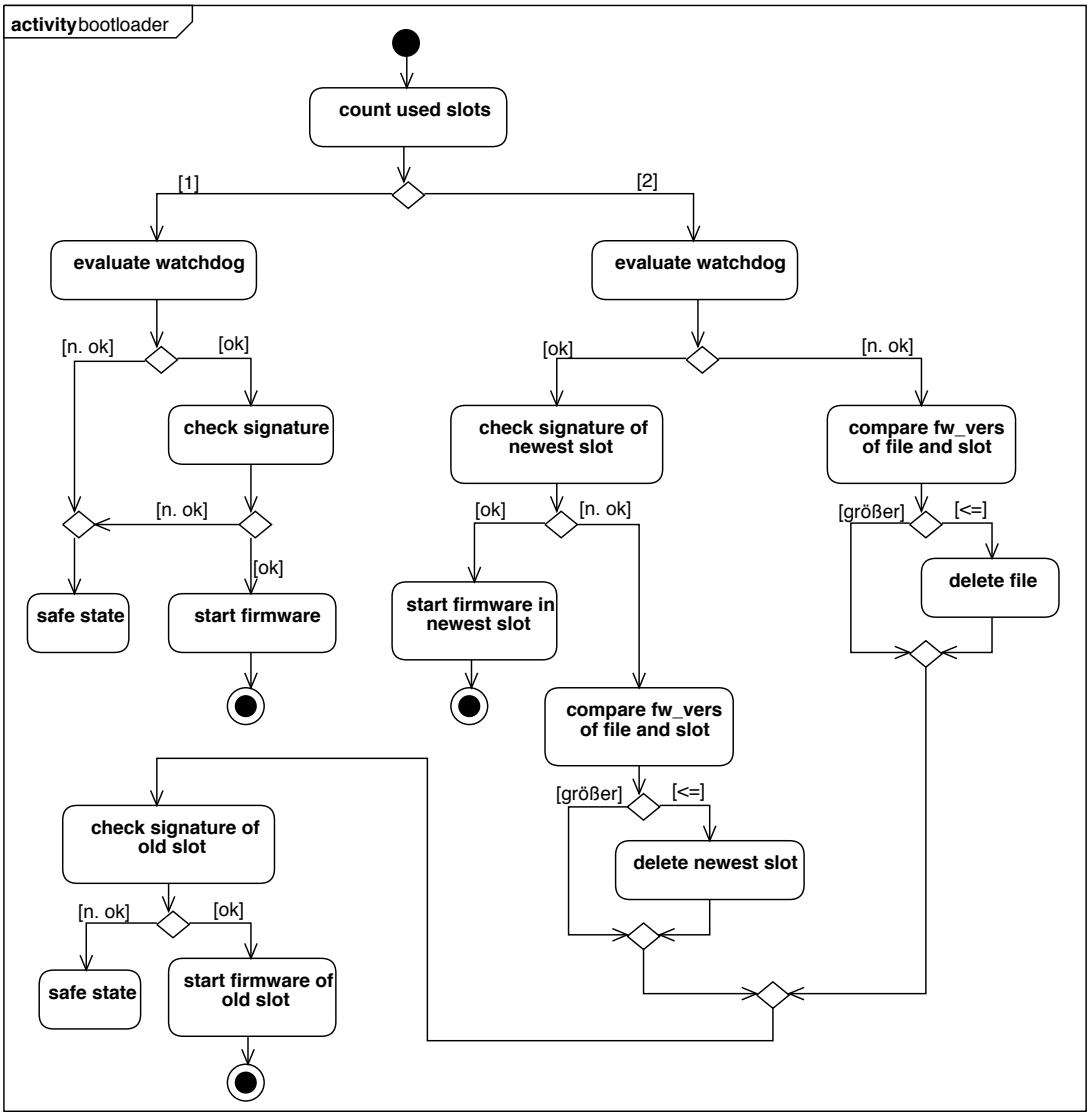

Figure 5: Activity Diagram: Boot Loader

First of all the boot loader actions depend on the number of used slots. On the other end of the decision chain the firmware will eventually be started, but before every start, the signature must be checked. If the signature is not valid and there is no other firmware available, the boot loader should execute a special function, which can place the device in a save state if needed.

The watchdog is used in cooperation with the application to detect a firmware, which is not responsive or which crashes at start-up and therefore leads to a boot loop. In this case, the firmware should not be executed any more. If an update file is available in this case and the file has the same version number, a firmware update must have been installed successfully, but the firmware is not suitable for this device. This can only occur, if something was not right in the quality assurance process before releasing an update. To provide the option to the application to detect and report such a mishap, the update file should be deleted by the boot loader.

If the watchdog is ok and two slots are occupied, the signature of the newest slot is checked first. If the signature is not valid, an interrupted installation could have occurred, the firmware was tampered with or a bit in the flash memory accidentally flipped. The last two cases cannot be distinguished and therefore will lead to the same action: The firmware will not be executed. If the firmware version number of the update file, if existent, is the same as the one of the newest slot, an interrupted update must have occurred. In this case, the slot is deleted as a hint for the update module and to provide the update module with the possibility to reinstall the update.

Any of the cases, where the newest slot cannot be executed will lead to the boot loader trying to start the old firmware in the other slot. The same rules for checking the signature and the eventual save state apply.

5 Automated Testing Procedure

For checking the correct behavior of the implementation, five test cases were developed and implemented in a python script. Tests should always be automated, so all tests will be run automatically by this script.

5.1 Test Cases

These test cases test every implemented requirement except for the update server, because the server is only a very basic example, which could not be used in a production environment, while the boot loader and update module could be used in a product.

All tests rely on the serial console interface between the application or boot loader and the computer to trigger functions and get information about the state of the application.

Case 1: Normal Behavior

The device is provisioned with the factory firmware and will be updated three times to test all possible update conditions.

Case 2: Faulty Update Files

The device is provisioned with a test-hex which will occupy both firmware slots. There are three similar steps in the main part of this test, which follow the same principle: An update file will be flashed to the device, the update will be tried to be installed and the installation function should return an invalid update file.

The faulty update files are: A file with an invalid signature, a file with an incompatible hardware ID and a file with a lower version number than the installed version.

Case 3: Power Failure

The device is provisioned with the factory firmware and an update will be started. Right after the installation begins to write the update data to the slot, the power is cut and the communication interface for the boot loader is opened. After reconnecting the power, the boot loader should state, that an interrupted update was detected. After that, the boot loader communication is closed and the ethos communication is opened again.

For the sake of completeness, the installation of the update is tried another time but without cutting the power.

Case 4: Manipulation of an Installed Firmware

The device is provisioned with the boot loader and a firmware image which contains an invalid signature and a valid backup firmware image. For this test it is sufficient to only have a communication link with the boot loader, which should state, that a corrupted firmware image was detected and therefore the old version is started.

Case 5: Detection of a Boot Loop

For this test, a firmware image will be used, which does not reset the watchdog. This simulates a non-responsive firmware or a firmware, which does not start correctly and therefore jumps back to the boot loader (boot loop). These two cases cannot be distinguished by the boot loader.

For the test to be successful, the boot loader messages must first state, that the newest firmware will be executed. After the timeout period of the watchdog, the boot loader must have been started another time stating, that there was a watchdog timeout and therefore the old firmware will be started.

5.2 Realization of the Tests

Because there are many different variants of the firmware and update files needed, a run of the text script takes a very long time to compile all the versions. Therefore the script provides a command line option to skip the compilation of all resources. For debugging purposes it is also possible to run the tests individually instead of beginning with test 1 every time.

The test script is located in the same directory as the central Makefile, because the Makefile is used by the test scripts.

The power failure can be automated quite easily on the ST Nucleo boards, because they have a jumper connector for the power source. This jumper will be replaced by an Arduino microcontroller board with a relay shield, which is controlled by a simple custom command protocol via the serial interface of the Arduino. With this solution, the power of the Nucleo board can be switched by the test script.

6 Conclusion

The question, how a save and robust update procedure could be implemented on an IoT device, has been answered in big parts in this work. But some aspects had to be ignored or simplified, such as the communication via the Internet.

The prototype has shown, that the update process can be realized well. But during the implementation it also has shown, that the flash memory's layout of the STM32F4 series by STMicroelectronics does not fit very well to the update procedure. Many microcontrollers have a flash memory, which is divided into several equally sized flash pages, but the STM32F4 has differently sized flash sectors instead. Therefore the firmware slots can only be placed in certain locations and it will be difficult to create an unified flash driver in RIOT.

The tests were implemented as a fully automatic procedure and they will test all implemented requirements. It has shown, that the tests take a quite long time, because the hardware has to be flashed several times, which is a quite slot process. But this will be no big problem, because the is no human interaction needed.

Besides ignoring the pitfalls of a communication via the Internet, there were some other decisions made, which might lead to different result in other circumstances. Every manufacturer should evaluate, if the memory use of multiple firmware slots is justified by the added robustness. A memory saving alternative would be to only use a minimal backup system in a second slot, but in this case the boot loader must install the update. Another decisions was, that the update process will not be especially hardened against attacks, which need access to the hardware, because only a few measures can be taken by the update process without detailed knowledge about the hardware.

The update process also has no possibility to be secure, if the application itself opens new attack vectors. But this can only be controlled by the manufacturer as well as it's in the manufacturers responsibility to have a good quality assurance for the software. The update process will rely on the metadata, such as the hardware ID, so this information must be correct for the device to work properly.

Software and device manufacturers must be aware, that they are responsible for their software and the clients must be willing to pay for software quality. Because the IoT will have a vast amount of devices connected, there will be big problems when safety and security critical software errors cannot be fixed.

An update procedure, as every added bit of complexity, will always lead to possible new attack vectors and will not by any protection against attacks. But it can be assumed, that devices will be more secure, if the manufacturer has the possibility to patch known security vulnerabilities. But this also means, that this possibility must be used.

References

[1] Atmel, San Jose, CA, USA. *Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers*, 2006. Application Note doc6253, online: http://www.atmel.com/Images/doc6253.pdf.

[2] Atmel, San Jose, CA, USA. *Safe and Secure Firmware Upgrade via Ethernet*, 2015. Application Note AT11787, online: http://www.atmel.com/Images/Atmel-42492-Safe-and-Secure-Firmware-Upgrade-via-Ethernet_ApplicationNote_AT11787.pdf.

[3] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library, 2012. online: https://cr.yp.to/highspeed/coolnacl-20120725.pdf.

[4] Jannik Beyerstedt. Sichere und robuste Firmware-Udpates von IoT-Geräten, 2017. online: https://jannikbeyerstedt.de/publications.

[5] Carsten Bormann, Mehmet Ersue, and Ari Keranen. Terminology for Constrained-Node Networks. RFC 7228, 2014. online: https://rfc-editor.org/rfc/rfc7228.txt.

[6] Kryptographische Verfahren: Empfehlungen und Schlüssellängen. Technische Richtlinie BSI TR-02102-1, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Deutschland, 2017. Version 2017-01, online: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf.

[7] Essensium NV, Mind - Embedded Software Division. *Safe upgrade of embedded systems*, 2012. Präsentationsfolien, online: http://mind.be/content/Presentation_Safe-Upgrade.pdf.

[8] Overview of the Internet of things. Recommendation ITU-T Y.2060, International Telecommunication Union, 2012. online: http://www.itu.int/ITU-T/recommendations/rec.aspx?rec=y.2060.

[9] Lukas Kvarda, Pavel Hnyk, Lukas Vojtech, Zdenk Lokaj, Marek Neruda, and Tomas Zitta. Software implementation of a secure firmware update solution in an IoT context. *Advances in Electrical and Electronic Engineering*, 14(4):389–396, 2016. online: http://advances.utc.sk/index.php/AEEE/article/download/1858/1170.

[10] Renesas Electronics America Inc. *Secure Firmware Update Lab Session*, 2012. Präsentationsfolien, online: https://elearning.renesas.com/file.php/1/CoursePDFs/DevCon_2012/Security/BL02I_Secure_Firmware_update_LabSession_92012_BL02I_Final.pdf.

[11] Silvie Schmidt, Mathias Tausig, Matthias Hudler, and Georg Simhandl. Secure firmware update over the air in the internet of things focusing on flexibility and feasibility. In *Internet of Things Software Update Workshop (IoTSU). Proceeding*, 2016. online: https://down.dsg.cs.tcd.ie/iotsu/subs/IoTSU_2016_paper_13.pdf.

[12] Loren K. Shade. Implementing secure remote firmware updates. Conference paper, Allegro Software Development Corporation, 2011. online: https://www.allegrosoft.com/wp-content/uploads/Secure-Firmware-Updates-Paper.pdf.

[13] Leonardo Steinfeld and Luigi Carro. The case for interpreted languages in wireless sensor networks. In *IESS 09 - International Embedded Systems Symposium. Langenargen, Germany*, pages 279–289. Springer, 2009.

[14] STMicroelectronics, Genf, Schweiz. *STM32F10xxx/20xxx/21xxx/L1xxxx Cortex-M3 programming manual*, 2013. Programming Manual PM0056, DocID15491 Rev 5.

[15] STMicroelectronics, Genf, Schweiz. *STM32L0 Series Cortex-M0+ programming manual*, 2014. Programming Manual PM0223, DocID025763 Rev 1.

[16] STMicroelectronics, Genf, Schweiz. *STM32F3, STM32F4 and STM32L4 Series Cortex-M4 programming manual*, 2016. Programming Manual PM0214, DocID022708 Rev 5.

[17] STMicroelectronics, Genf, Schweiz. *STM32F7 Series Cortex-M7 processor programming manual*, 2017. Programming Manual PM0253, DocID028474 Rev 3.

[18] Texas Instruments, Dallas, TX, USA. *Secure In-Field Firmware Updates for MSP MCUs*, 2015. Application Report SLAA682, online: http://www.ti.com/lit/an/slaa682/slaa682.pdf.

[19] Shivani Tomar. White paper – secure firmware upgrade system. White paper, HCL Technologies, 2014. online: https://www.hcltech.com/sites/default/files/resources/whitepaper/files/2014/06/19/design_of_secure_firmware_upgrade_system.pdf.

[20] Hannes Tschofenig and Stephen Farrell. Report from the Internet of Things (IoT) Software Update (IoTSU) Workshop 2016. Internet-Draft draft-farrell-iotsu-workshop-01, Internet Engineering Task Force, 2016. work in progress, online: https://datatracker.ietf.org/doc/html/draft-farrell-iotsu-workshop-01.