SUIT
Internet-Draft
Intended status: Informational
Expires: April 22, 2020

B. Moran
Arm Limited
M. Meriac
Consultant
H. Tschofenig
Arm Limited
D. Brown
Linaro

October 20, 2019

A Firmware Update Architecture for Internet of Things Devices
draft-ietf-suit-architecture-07

Abstract

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a solid and secure firmware update mechanism that is also suitable for constrained devices. Incorporating such update mechanism to fix vulnerabilities, to update configuration settings as well as adding new functionality is recommended by security experts.

This document lists requirements and describes an architecture for a firmware update mechanism suitable for IoT devices. The architecture is agnostic to the transport of the firmware images and associated meta-data.

This version of the document assumes asymmetric cryptography and a public key infrastructure. Future versions may also describe a symmetric key approach for very constrained devices.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 22, 2020.

Table of Contents

1. Introduction

When developing IoT devices, one of the most difficult problems to solve is how to update the firmware on the device. Once the device is deployed, firmware updates play a critical part in its lifetime, particularly when devices have a long lifetime, are deployed in remote or inaccessible areas where manual intervention is cost prohibitive or otherwise difficult. Updates to the firmware of an IoT device are done to fix bugs in software, to add new functionality, and to re-configure the device to work in new environments or to behave differently in an already deployed context.

The firmware update process, among other goals, has to ensure that

- The firmware image is authenticated and integrity protected. Attempts to flash a modified firmware image or an image from an unknown source are prevented.

- The firmware image can be confidentiality protected so that attempts by an adversary to recover the plaintext binary can be prevented. Obtaining the firmware is often one of the first steps to mount an attack since it gives the adversary valuable insights

into used software libraries, configuration settings and generic functionality (even though reverse engineering the binary can be a tedious process).

More details about the security goals are discussed in Section 5 and requirements are described in Section 3.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and

"OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

This document uses the following terms:

- Manifest: The manifest contains meta-data about the firmware image. The manifest is protected against modification and provides information about the author.

- Firmware Image: The firmware image, or image, is a binary that may contain the complete software of a device or a subset of it. The firmware image may consist of multiple images, if the device contains more than one microcontroller. Often it is also a compressed archive that contains code, configuration data, and even the entire file system. The image may consist of a differential update for performance reasons. Firmware is the more universal term. The terms, firmware image, firmware, and image, are used in this document and are interchangeable.

- Bootloader: A bootloader is a piece of software that is executed once a microcontroller has been reset. It is responsible for deciding whether to boot a firmware image that is present or whether to obtain and verify a new firmware image. Since the bootloader is a security critical component its functionality may be split into separate stages. Such a multi-stage bootloader may offer very basic functionality in the first stage and resides in ROM whereas the second stage may implement more complex functionality and resides in flash memory so that it can be updated in the future (in case bugs have been found). The exact

split of components into the different stages, the number of firmware images stored by an IoT device, and the detailed functionality varies throughout different implementations. A more detailed discussion is provided in Section 8.

- Microcontroller (MCU for microcontroller unit): An MCU is a compact integrated circuit designed for use in embedded systems. A typical microcontroller includes a processor, memory (RAM and flash), input/output (I/O) ports and other features connected via some bus on a single chip. The term 'system on chip (SoC)' is often used for these types of devices.

- System on Chip (SoC): An SoC is an integrated circuit that integrates all components of a computer, such as CPU, memory, input/output ports, secondary storage, etc.

- Homogeneous Storage Architecture (HoSA): A device that stores all firmware components in the same way, for example in a file system or in flash memory.

- Heterogeneous Storage Architecture (HeSA): A device that stores at least one firmware component differently from the rest, for example a device with an external, updatable radio, or a device with internal and external flash memory.

- Trusted Execution Environments (TEEs): An execution environment that runs alongside of, but is isolated from, an REE.

- Rich Execution Environment (REE): An environment that is provided and governed by a typical OS (e.g., Linux, Windows, Android, iOS), potentially in conjunction with other supporting operating systems and hypervisors; it is outside of the TEE. This environment and applications running on it are considered un-trusted.

- Trusted applications (TAs): An application component that runs in a TEE.

For more information about TEEs see [I-D.ietf-teep-architecture].

The following entities are used:

- Author: The author is the entity that creates the firmware image.

There may be multiple authors in a system either when a device consists of multiple micro-controllers or when the the final firmware image consists of software components from multiple companies.

- Firmware Consumer: The firmware consumer is the recipient of the firmware image and the manifest. It is responsible for parsing and verifying the received manifest and for storing the obtained firmware image. The firmware consumer plays the role of the update component on the IoT device typically running in the application firmware. It interacts with the firmware server and with the status tracker, if present.

- (IoT) Device: A device refers to the entire IoT product, which consists of one or many MCUs, sensors and/or actuators. Many IoT devices sold today contain multiple MCUs and therefore a single device may need to obtain more than one firmware image and manifest to succesfully perform an update. The terms device and firmware consumer are used interchangably since the firmware consumer is one software component running on an MCU on the device.

- Status Tracker: The status tracker offers device management functionality to retrieve information about the installed firmware on a device and other device characteristics (including free memory and hardware components), to obtain the state of the

firmware update cycle the device is currently in, and to trigger the update process. The deployment of status trackers is flexible and they may be used as cloud-based servers, on-premise servers, embedded in edge computing device (such as Internet access gateways or protocol translation gateways), or even in smart phones and tablets. While the IoT device itself runs the client-side of the status tracker it will most likely not run a status tracker itself unless it acts as a proxy for other IoT devices in a protocol translation or edge computing device node. How much functionality a status tracker includes depends on the selected configuration of the device management functionality and the communication environment it is used in. In a generic networking environment the protocol used between the client and the server-side of the status tracker need to deal with Internet communication challenges involving firewall and NAT traversal. In

other cases, the communication interaction may be rather simple. This architecture document does not impose requirements on the status tracker.

- Firmware Server: The firmware server stores firmware images and manifests and distributes them to IoT devices. Some deployments may require a store-and-forward concept, which requires storing the firmware images/manifests on more than one entity before they reach the device. There is typically some interaction between the firmware server and the status tracker but those entities are often physically separated on different devices for scalability reasons.

- Device Operator: The actor responsible for the day-to-day operation of a fleet of IoT devices.

- Network Operator: The actor responsible for the operation of a network to which IoT devices connect.

In addition to the entities in the list above there is an orthogonal infrastructure with a Trust Provisioning Authority (TPA) distributing trust anchors and authorization permissions to various entities in the system. The TPA may also delegate rights to install, update, enhance, or delete trust anchors and authorization permissions to other parties in the system. This infrastructure overlaps the communication architecture and different deployments may empower certain entities while other deployments may not. For example, in some cases, the Original Design Manufacturer (ODM), which is a company that designs and manufactures a product, may act as a TPA and may decide to remain in full control over the firmware update process of their products.

The terms 'trust anchor' and 'trust anchor store' are defined in [RFC6024]:

- "A trust anchor represents an authoritative entity via a public key and associated data. The public key is used to verify digital signatures, and the associated data is used to constrain the types of information for which the trust anchor is authoritative."

- "A trust anchor store is a set of one or more trust anchors stored in a device. A device may have more than one trust anchor store, each of which may be used by one or more applications." A trust anchor store must resist modification against unauthorized insertion, deletion, and modification.

3. Requirements

The firmware update mechanism described in this specification was designed with the following requirements in mind:

- Agnostic to how firmware images are distributed
- Friendly to broadcast delivery
- Use state-of-the-art security mechanisms
- Rollback attacks must be prevented
- High reliability
- Operate with a small bootloader
- Small Parsers
- Minimal impact on existing firmware formats
- Robust permissions
- Diverse modes of operation
- Suitability to software and personalization data

3.1. Agnostic to how firmware images are distributed

Firmware images can be conveyed to devices in a variety of ways, including USB, UART, WiFi, BLE, low-power WAN technologies, etc. and use different protocols (e.g., CoAP, HTTP). The specified mechanism needs to be agnostic to the distribution of the firmware images and manifests.

3.2. Friendly to broadcast delivery

This architecture does not specify any specific broadcast protocol. However, given that broadcast may be desirable for some networks, updates must cause the least disruption possible both in metadata and firmware transmission.

For an update to be broadcast friendly, it cannot rely on link layer, network layer, or transport layer security. A solution has to rely on security protection applied to the manifest and firmware image instead. In addition, the same manifest must be deliverable to many devices, both those to which it applies and those to which it does not, without a chance that the wrong device will accept the update. Considerations that apply to network broadcasts apply equally to the use of third-party content distribution networks for payload distribution.

3.3. Use state-of-the-art security mechanisms

End-to-end security between the author and the device is shown in Section 5.

Authentication ensures that the device can cryptographically identify the author(s) creating firmware images and manifests. Authenticated identities may be used as input to the authorization process.

Integrity protection ensures that no third party can modify the manifest or the firmware image.

For confidentiality protection of the firmware image, it must be done in such a way that every intended recipient can decrypt it. The information that is encrypted individually for each device must maintain friendliness to Content Distribution Networks, bulk storage, and broadcast protocols.

A manifest specification must support different cryptographic algorithms and algorithm extensibility. Because of the nature of unchangeable code in ROM for use with bootloaders the use of post-quantum secure signature mechanisms, such as hash-based signatures [I-D.ietf-cose-hash-sig], are attractive because they maintain security in presence of quantum computers.

A mandatory-to-implement set of algorithms has to be defined offering a key length of 112-bit symmetric key or security or more, as outlined in Section 20 of RFC 7925 [RFC7925]. This corresponds to a 233 bit ECC key or a 2048 bit RSA key.

3.4. Rollback attacks must be prevented

A device presented with an old, but valid manifest and firmware must not be tricked into installing such firmware since a vulnerability in the old firmware image may allow an attacker to gain control of the device.

3.5. High reliability

A power failure at any time must not cause a failure of the device. A failure to validate any part of an update must not cause a failure of the device. One way to achieve this functionality is to provide a minimum of two storage locations for firmware and one bootable location for firmware. An alternative approach is to use a 2nd stage bootloader with build-in full featured firmware update functionality such that it is possible to return to the update process after power down.

Note: This is an implementation requirement rather than a requirement on the manifest format.

3.6. Operate with a small bootloader

Throughout this document we assume that the bootloader itself is distinct from the role of the firmware consumer and therefore does not manage the firmware update process. This may give the impression that the bootloader itself is a completely separate component, which is mainly responsible for selecting a firmware image to boot.

The overlap between the firmware update process and the bootloader functionality comes in two forms, namely

- First, a bootloader must verify the firmware image it boots as part of the secure boot process. Doing so requires meta-data to be stored alongside the firmware image so that the bootloader can cryptographically verify the firmware image before booting it to ensure it has not been tampered with or replaced. This meta-data used by the bootloader may well be the same manifest obtained with the firmware image during the update process (with the severable fields stripped off).

- Second, an IoT device needs a recovery strategy in case the firmware update / boot process fails. The recovery strategy may include storing two or more firmware images on the device or offering the ability to have a second stage bootloader perform the firmware update process again using firmware updates over serial,

USB or even wireless connectivity like a limited version of Bluetooth Smart. In the latter case the firmware consumer

functionality is contained in the second stage bootloader and requires the necessary functionality for executing the firmware update process, including manifest parsing.

In general, it is assumed that the bootloader itself, or a minimal part of it, will not be updated since a failed update of the bootloader poses a risk in reliability.

All information necessary for a device to make a decision about the installation of a firmware update must fit into the available RAM of a constrained IoT device. This prevents flash write exhaustion. This is typically not a difficult requirement to accomplish because there are not other task/processing running while the bootloader is active (unlike it may be the case when running the application firmware).

Note: This is an implementation requirement.

3.7. Small Parsers

Since parsers are known sources of bugs they must be minimal. Additionally, it must be easy to parse only those fields that are required to validate at least one signature or MAC with minimal exposure.

3.8. Minimal impact on existing firmware formats

The design of the firmware update mechanism must not require changes to existing firmware formats.

3.9. Robust permissions

When a device obtains a monolithic firmware image from a single author without any additional approval steps then the authorization flow is relatively simple. There are, however, other cases where more complex policy decisions need to be made before updating a device.

In this architecture the authorization policy is separated from the

underlying communication architecture. This is accomplished by separating the entities from their permissions. For example, an author may not have the authority to install a firmware image on a device in critical infrastructure without the authorization of a device operator. In this case, the device may be programmed to reject firmware updates unless they are signed both by the firmware author and by the device operator.

Alternatively, a device may trust precisely one entity, which does all permission management and coordination. This entity allows the device to offload complex permissions calculations for the device.

3.10. Operating modes

There are three broad classifications of update operating modes.

- Client-initiated Update
- Server-initiated Update
- Hybrid Update

Client-initiated updates take the form of a firmware consumer on a device proactively checking (polling) for new firmware images.

Server-initiated updates are important to consider because timing of updates may need to be tightly controlled in some high- reliability environments. In this case the status tracker determines what devices qualify for a firmware update. Once those devices have been selected the firmware server distributes updates to the firmware consumers.

Note: This assumes that the status tracker is able to reach the device, which may require devices to keep reachability information at the status tracker up-to-date. This may also require keeping state at NATs and stateful packet filtering firewalls alive.

Hybrid updates are those that require an interaction between the firmware consumer and the status tracker. The status tracker pushes notifications of availability of an update to the firmware consumer,

and it then downloads the image from a firmware server as soon as possible.

An alternative view to the operating modes is to consider the steps a device has to go through in the course of an update:

- Notification
- Pre-authorisation
- Dependency resolution
- Download
- Installation

The notification step consists of the status tracker informing the firmware consumer that an update is available. This can be accomplished via polling (client-initiated), push notifications (server-initiated), or more complex mechanisms.

The pre-authorisation step involves verifying whether the entity signing the manifest is indeed authorized to perform an update. The firmware consumer must also determine whether it should fetch and process a firmware image, which is referenced in a manifest.

A dependency resolution phase is needed when more than one component can be updated or when a differential update is used. The necessary dependencies must be available prior to installation.

The download step is the process of acquiring a local copy of the firmware image. When the download is client-initiated, this means that the firmware consumer chooses when a download occurs and initiates the download process. When a download is server-initiated, this means that the status tracker tells the device when to download or that it initiates the transfer directly to the firmware consumer. For example, a download from an HTTP-based firmware server is client-initiated. Pushing a manifest and firmware image to the transfer to the Package resource of the LwM2M Firmware Update object [LwM2M] is server-initiated.

If the firmware consumer has downloaded a new firmware image and is ready to install it, it may need to wait for a trigger from the status tracker to initiate the installation, may trigger the update automatically, or may go through a more complex decision making process to determine the appropriate timing for an update (such as delaying the update process to a later time when end users are less impacted by the update process).

Installation is the act of processing the payload into a format that the IoT device can recognise and the bootloader is responsible for then booting from the newly installed firmware image.

Each of these steps may require different permissions.

3.11. Suitability to software and personalization data

The work on a standardized manifest format initially focused on the most constrained IoT devices and those devices contain code put together by a single author (although that author may obtain code from other developers, some of it only in binary form).

Later it turns out that other use cases may benefit from a standardized manifest format also for conveying software and even

personalization data alongside software. Trusted Execution Environments (TEEs), for example, greatly benefit from a protocol for managing the lifecycle of trusted applications (TAs) running inside a TEE. TEEs may obtain TAs from different authors and those TAs may require personalization data, such as payment information, to be securely conveyed to the TEE.

To support this wider range of use cases the manifest format should therefore be extensible to convey other forms of payloads as well.

4. Claims

Claims in the manifest offer a way to convey instructions to a device that impact the firmware update process. To have any value the manifest containing those claims must be authenticated and integrity protected. The credential used must be directly or indirectly related to the trust anchor installed at the device by the Trust Provisioning Authority.

The baseline claims for all manifests are described in [I-D.ietf-suit-information-model]. For example, there are:

- Do not install firmware with earlier metadata than the current metadata.

- Only install firmware with a matching vendor, model, hardware revision, software version, etc.

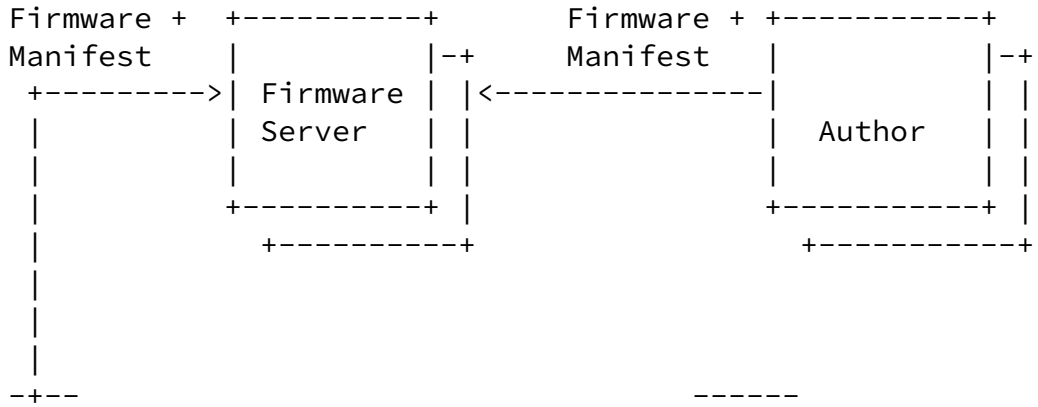
- Only install firmware that is before its best-before timestamp.

- Only allow a firmware installation if dependencies have been met.

- Choose the mechanism to install the firmware, based on the type of firmware it is.

5. Communication Architecture

Figure 1 shows the communication architecture where a firmware image is created by an author, and uploaded to a firmware server. The firmware image/manifest is distributed to the device either in a push or pull manner using the firmware consumer residing on the device. The device operator keeps track of the process using the status tracker. This allows the device operator to know and control what devices have received an update and which of them are still pending an update.



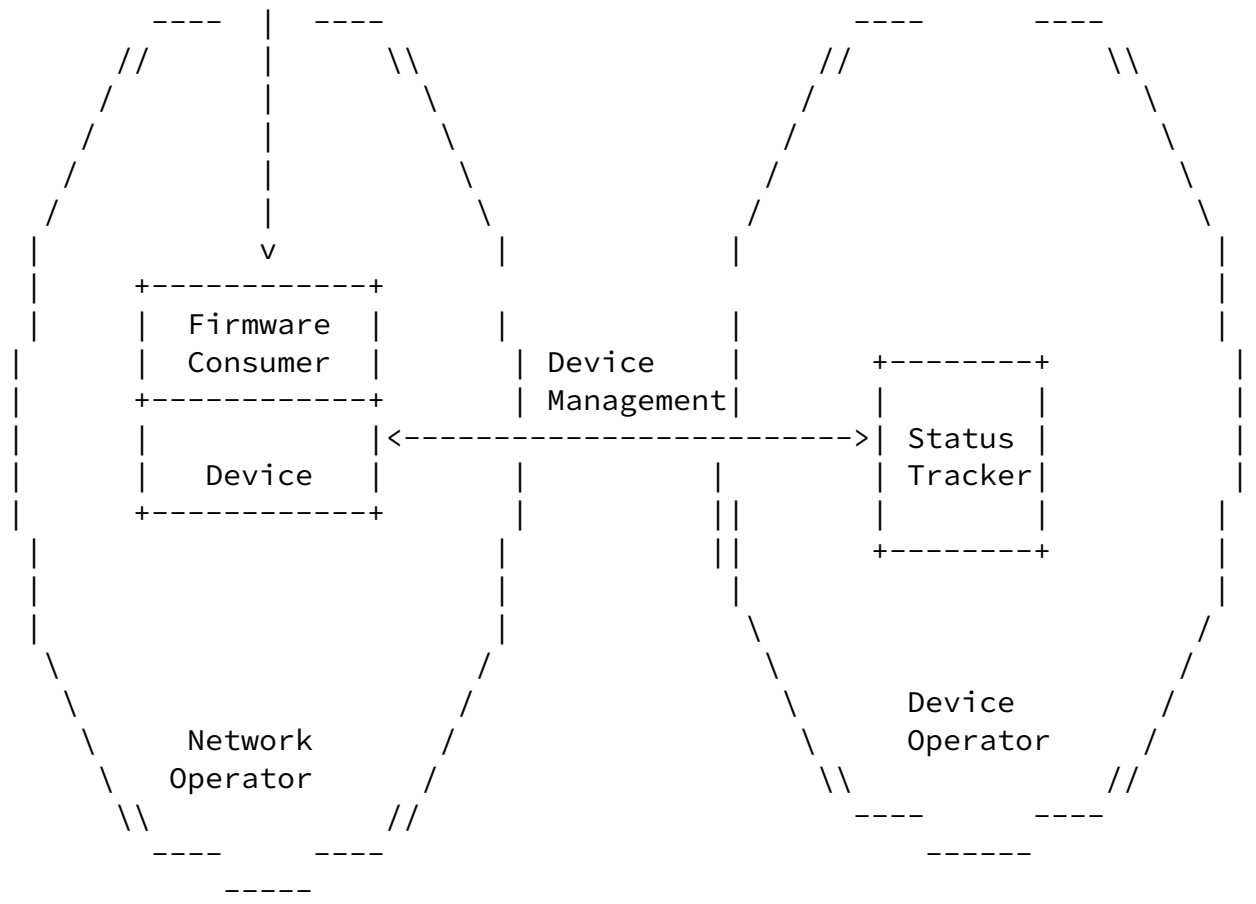
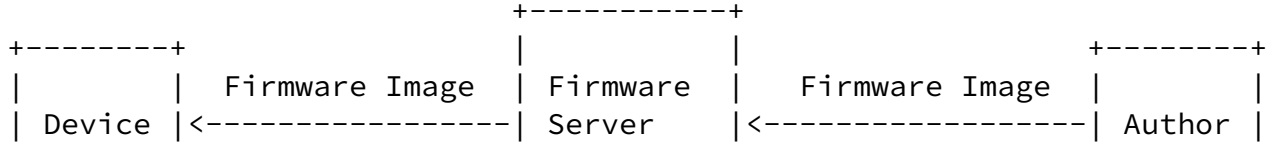


Figure 1: Architecture.

End-to-end security mechanisms are used to protect the firmware image and the manifest although Figure 2 does not show the manifest itself since it may be distributed independently.



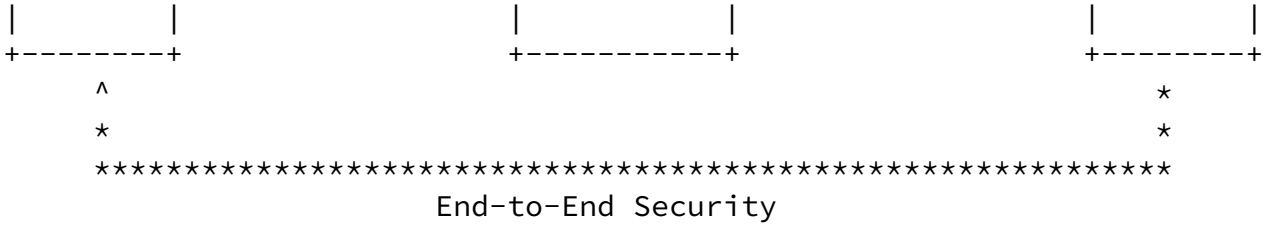


Figure 2: End-to-End Security.

Whether the firmware image and the manifest is pushed to the device or fetched by the device is a deployment specific decision.

The following assumptions are made to allow the firmware consumer to verify the received firmware image and manifest before updating software:

- To accept an update, a device needs to verify the signature covering the manifest. There may be one or multiple manifests that need to be validated, potentially signed by different parties. The device needs to be in possession of the trust anchors to verify those signatures. Installing trust anchors to devices via the Trust Provisioning Authority happens in an out-of-band fashion prior to the firmware update process.

- Not all entities creating and signing manifests have the same permissions. A device needs to determine whether the requested action is indeed covered by the permission of the party that signed the manifest. Informing the device about the permissions of the different parties also happens in an out-of-band fashion and is also a duty of the Trust Provisioning Authority.

- For confidentiality protection of firmware images the author needs to be in possession of the certificate/public key or a pre-shared key of a device. The use of confidentiality protection of firmware images is deployment specific.

There are different types of delivery modes, which are illustrated based on examples below.

There is an option for embedding a firmware image into a manifest. This is a useful approach for deployments where devices are not connected to the Internet and cannot contact a dedicated firmware server for the firmware download. It is also applicable when the

firmware update happens via a USB stick or via Bluetooth Smart. Figure 3 shows this delivery mode graphically.

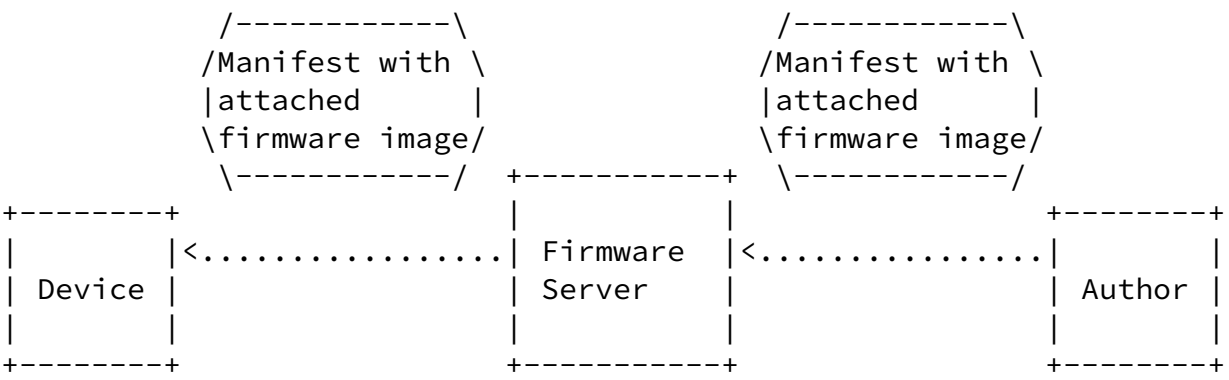


Figure 3: Manifest with attached firmware.

Figure 4 shows an option for remotely updating a device where the device fetches the firmware image from some file server. The manifest itself is delivered independently and provides information about the firmware image(s) to download.

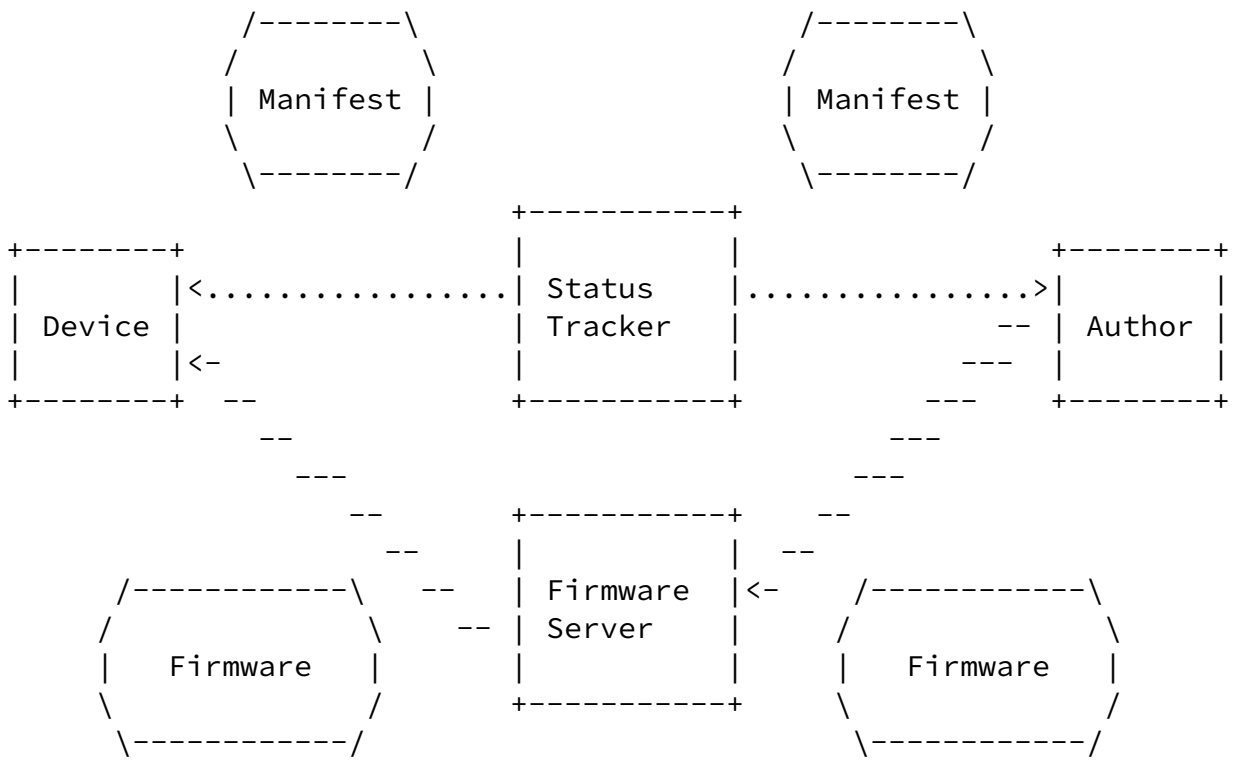


Figure 4: Independent retrieval of the firmware image.

This architecture does not mandate a specific delivery mode but a solution must support both types.

6. Manifest

In order for a device to apply an update, it has to make several decisions about the update:

- Does it trust the author of the update?
- Has the firmware been corrupted?
- Does the firmware update apply to this device?
- Is the update older than the active firmware?
- When should the device apply the update?
- How should the device apply the update?
- What kind of firmware binary is it?
- Where should the update be obtained?
- Where should the firmware be stored?

The manifest encodes the information that devices need in order to make these decisions. It is a data structure that contains the following information:

- information about the device(s) the firmware image is intended to be applied to,
- information about when the firmware update has to be applied,
- information about when the manifest was created,
- dependencies on other manifests,
- pointers to the firmware image and information about the format,
- information about where to store the firmware image,

- cryptographic information, such as digital signatures or message authentication codes (MACs).

The manifest information model is described in [I-D.ietf-suit-information-model].

7. Device Firmware Update Examples

Although these documents attempt to define a firmware update architecture that is applicable to both existing systems, as well as yet-to-be-conceived systems; it is still helpful to consider existing architectures.

7.1. Single CPU SoC

The simplest, and currently most common, architecture consists of a single MCU along with its own peripherals. These SoCs generally contain some amount of flash memory for code and fixed data, as well as RAM for working storage. These systems either have a single firmware image, or an immutable bootloader that runs a single image. A notable characteristic of these SoCs is that the primary code is generally execute in place (XIP). Combined with the non-relocatable nature of the code, firmware updates need to be done in place.

7.2. Single CPU with Secure - Normal Mode Partitioning

Another configuration consists of a similar architecture to the previous, with a single CPU. However, this CPU supports a security partitioning scheme that allows memory (in addition to other things) to be divided into secure and normal mode. There will generally be two images, one for secure mode, and one for normal mode. In this configuration, firmware upgrades will generally be done by the CPU in secure mode, which is able to write to both areas of the flash device. In addition, there are requirements to be able to update either image independently, as well as to update them together atomically, as specified in the associated manifests.

7.3. Dual CPU, shared memory

This configuration has two or more CPUs in a single SoC that share memory (flash and RAM). Generally, they will be a protection mechanism to prevent one CPU from accessing the other's memory. Upgrades in this case will typically be done by one of the CPUs, and is similar to the single CPU with secure mode.

7.4. Dual CPU, other bus

This configuration has two or more CPUs, each having their own memory. There will be a communication channel between them, but it will be used as a peripheral, not via shared memory. In this case, each CPU will have to be responsible for its own firmware upgrade. It is likely that one of the CPUs will be considered a master, and will direct the other CPU to do the upgrade. This configuration is commonly used to offload specific work to other CPUs. Firmware

dependencies are similar to the other solutions above, sometimes allowing only one image to be upgraded, other times requiring several to be upgraded atomically. Because the updates are happening on multiple CPUs, upgrading the two images atomically is challenging.

8. Bootloader

More devices today than ever before are being connected to the Internet, which drives the need for firmware updates to be provided over the Internet rather than through traditional interfaces, such as USB or RS232. Updating a device over the Internet requires the device to fetch not only the firmware image but also the manifest. Hence, the following building blocks are necessary for a firmware update solution:

- the Internet protocol stack for firmware downloads (*),
- the capability to write the received firmware image to persistent storage (most likely flash memory) prior to performing the update,
- the ability to unpack, decompress or otherwise process the received firmware image,
- the features to verify an image and a manifest, including digital signature verification or checking a message authentication code,

- a manifest parsing library, and

- integration of the device into a device management server to perform automatic firmware updates and to track their progress.

(*) Because firmware images are often multiple kilobytes, sometimes exceeding one hundred kilobytes, in size for low end IoT devices and even several megabytes large for IoT devices running full-fletched operating systems like Linux the protocol mechanism for retrieving these images needs to offer features like congestion control, flow control, fragmentation and reassembly, and mechanisms to resume interrupted or corrupted transfers.

All these features are most likely offered by the application, i.e. firmware consumer, running on the device (except for basic security algorithms that may run either on a trusted execution environment or on a separate hardware security MCU/module) rather than by the bootloader itself.

Once manifests have been processed and firmware images successfully downloaded and verified the device needs to hand control over to the bootloader. In most cases this requires the MCU to restart. Once

the MCU has initiated a restart, the bootloader takes over control and determines whether the newly downloaded firmware image should be executed.

The boot process is security sensitive because the firmware images may, for example, be stored in off-chip flash memory giving attackers easy access to the image for reverse engineering and potentially also for modifying the binary. The bootloader will therefore have to perform security checks on the firmware image before it can be booted. These security checks by the bootloader happen in addition to the security checks that happened when the firmware image and the manifest were downloaded.

The manifest may have been stored alongside the firmware image to allow re-verification of the firmware image during every boot attempt. Alternatively, secure boot-specific meta-data may have been created by the application after a successful firmware download and verification process. Whether to re-use the standardized manifest format that was used during the initial firmware retrieval process or

whether it is better to use a different format for the secure boot-specific meta-data depends on the system design. The manifest format does, however, have the capability to serve also as a building block for secure boot with its severable elements that allow shrinking the size of the manifest by stripping elements that are no longer needed.

If the application image contains the firmware consumer functionality, as described above, then it is necessary that a working image is left on the device. This allows the bootloader to roll back to a working firmware image to execute a firmware download if the bootloader itself does not have enough functionality to fetch a firmware image plus manifest from a firmware server over the Internet. A multi-stage bootloader may soften this requirement at the expense of a more sophisticated boot process.

For a bootloader to offer a secure boot mechanism it needs to provide the following features:

- ability to access security algorithms, such as SHA-256 to compute a fingerprint over the firmware image and a digital signature algorithm.

- access keying material directly or indirectly to utilize the digital signature. The device needs to have a trust anchor store.

- ability to expose boot process-related data to the application firmware (such as to the device management software). This allows a device management server to determine whether the firmware update has been successful and, if not, what errors occurred.

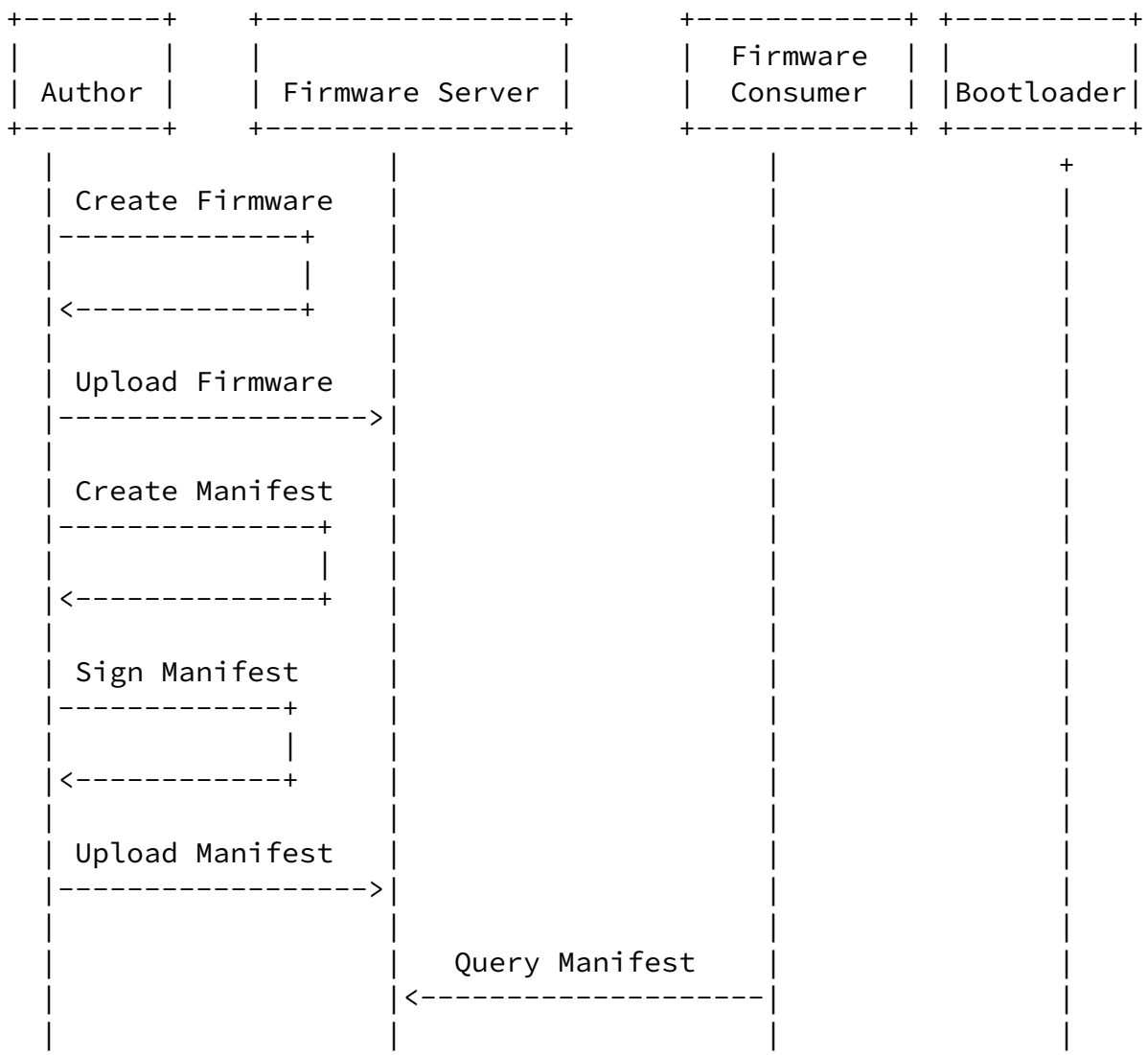
- to (optionally) offer attestation information (such as measurements).

While the software architecture of the bootloader and its security mechanisms are implementation-specific, the manifest can be used to control the firmware download from the Internet in addition to augmenting secure boot process. These building blocks are highly relevant for the design of the manifest.

9. Example

Figure 5 illustrates an example message flow for distributing a

firmware image to a device starting with an author uploading the new firmware to firmware server and creating a manifest. The firmware and manifest are stored on the same firmware server. This setup does not use a status tracker and the firmware consumer component is therefore responsible for periodically checking whether a new firmware image is available for download.



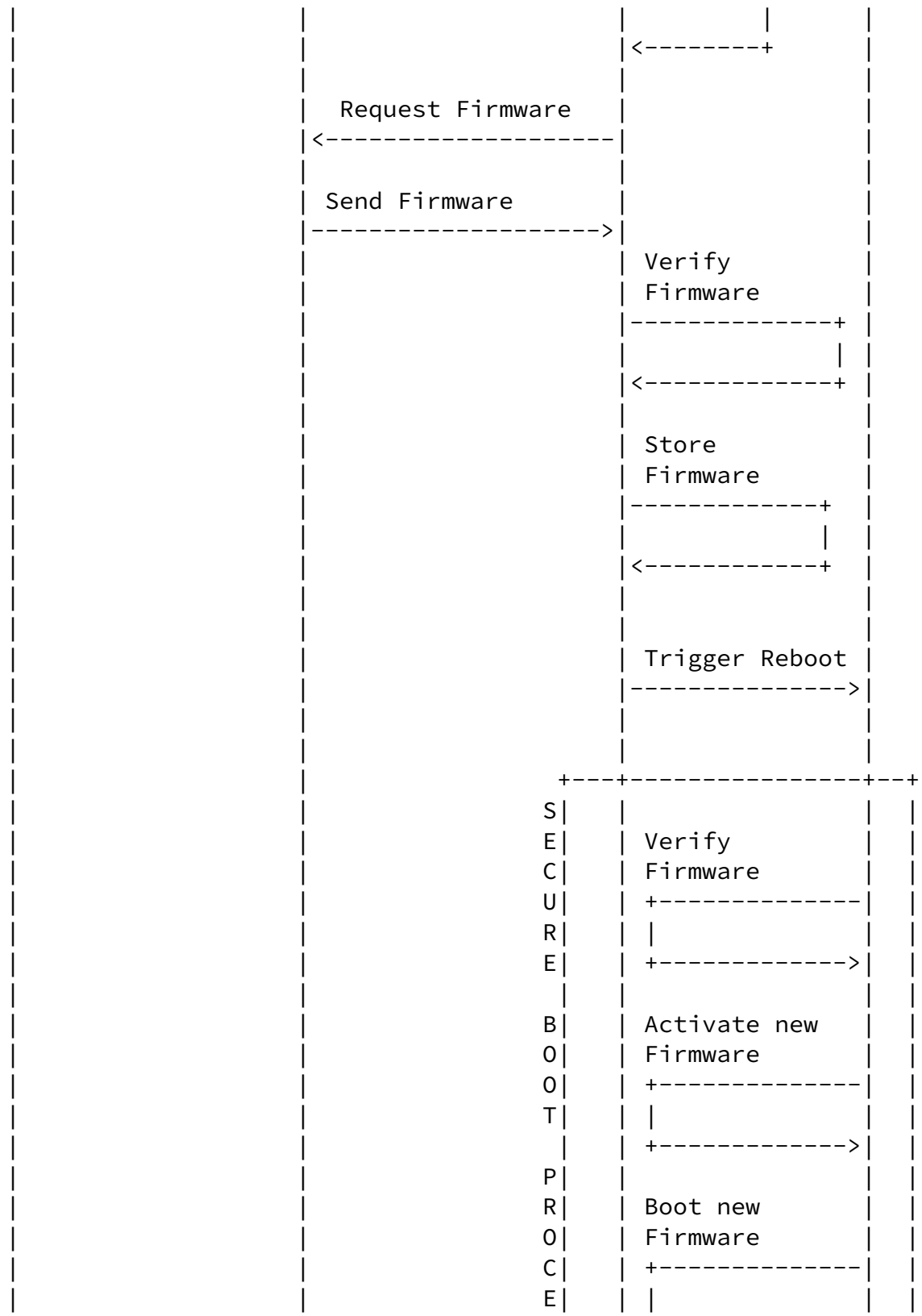
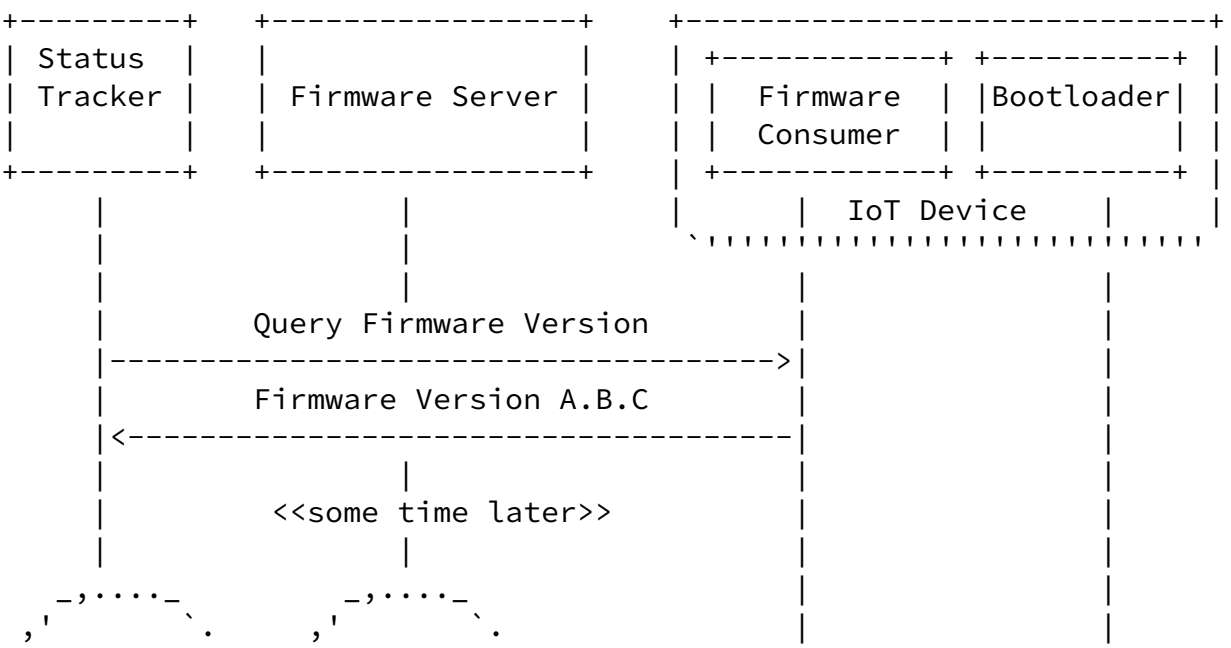

Request Firmware
Send Firmware
Verify Firmware
Store Firmware
Trigger Reboot
SECURE BOOT
Verify Firmware
Activate new Firmware
Boot new Firmware

Figure 5: First Example Flow for a Firmware Upate.

Figure 6 shows an example follow with the device using a status tracker. For editorial reasons the author publishing the manifest at the status tracker and the firmware image at the firmware server is not shown. Also omitted is the secure boot process following the successful firmware update process.

The exchange starts with the device interacting with the status tracker; the details of such exchange will vary with the different device management systems being used. In any case, the status tracker learns about the firmware version of the devices it manages. In our example, the device under management is using firmware version A.B.C. At a later point in time the author uploads a new firmware along with the manifest to the firmware server and the status tracker, respectively. While there is no need to store the manifest and the firmware on different servers this example shows a common pattern used in the industry. The status tracker may then automatically, based on human intervention or based on a more complex policy decide to inform the device about the newly available firmware image. In our example, it does so by pushing the manifest to the firmware consumer. The firmware consumer downloads the firmware image with the newer version X.Y.Z after successful validation of the manifest. Subsequently, a reboot is initiated and the secure boot process starts.



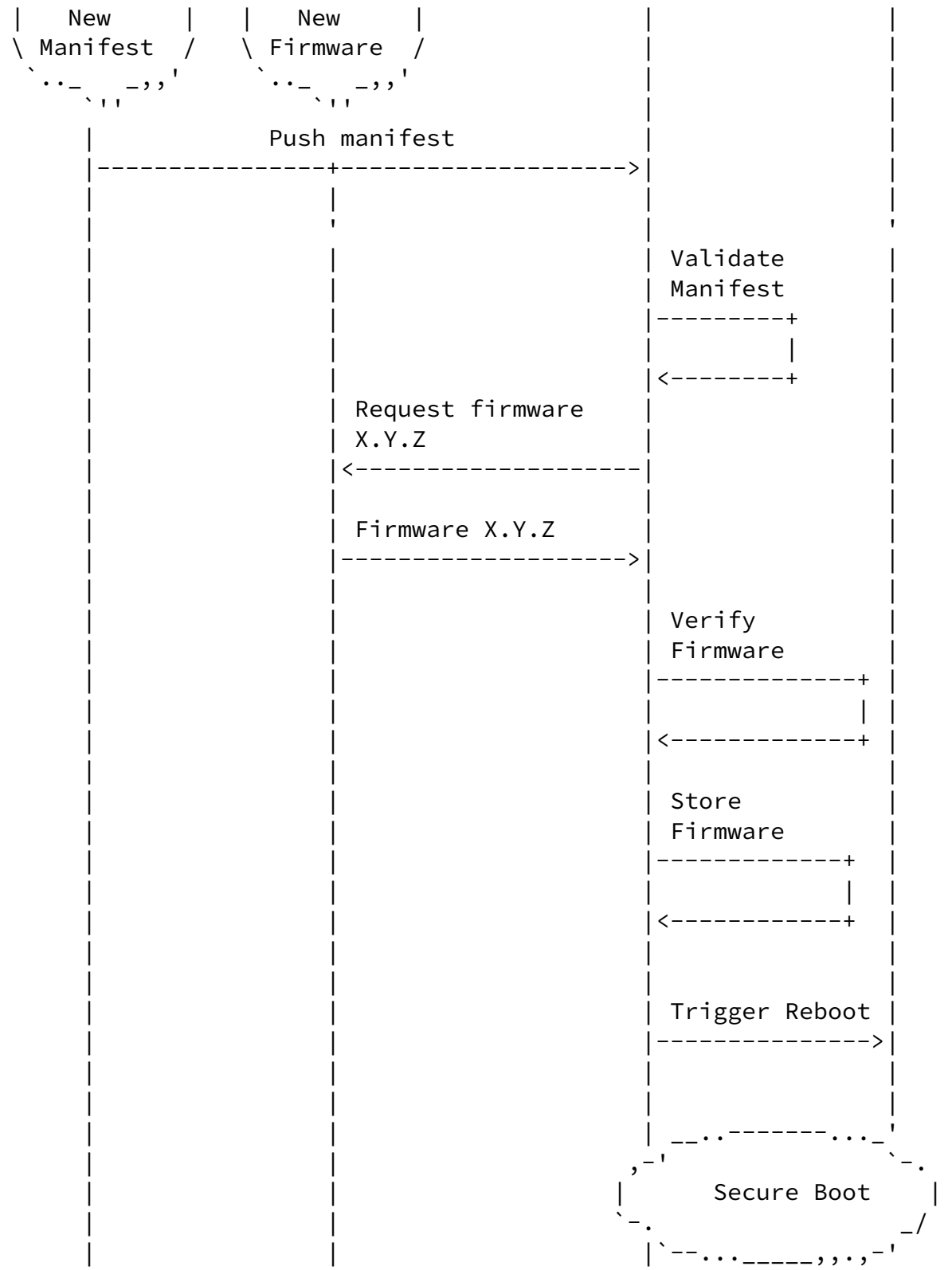

New Manifest
New Firmware
Push manifest
Validate Manifest
Request firmware X.Y.Z
Firmware X.Y.Z
Verify Firmware
Store Firmware
Trigger Reboot
Secure Boot

Figure 6: Second Example Flow for a Firmware Upate.

10. IANA Considerations

This document does not require any actions by IANA.

11. Security Considerations

Firmware updates fix security vulnerabilities and are considered to be an important building block in securing IoT devices. Due to the importance of firmware updates for IoT devices the Internet Architecture Board (IAB) organized a 'Workshop on Internet of Things (IoT) Software Update (IOTSU)', which took place at Trinity College Dublin, Ireland on the 13th and 14th of June, 2016 to take a look at the big picture. A report about this workshop can be found at [RFC8240]. A standardized firmware manifest format providing end-to-end security from the author to the device will be specified in a separate document.

There are, however, many other considerations raised during the workshop. Many of them are outside the scope of standardization organizations since they fall into the realm of product engineering, regulatory frameworks, and business models. The following considerations are outside the scope of this document, namely

- installing firmware updates in a robust fashion so that the update does not break the device functionality of the environment this device operates in.

- installing firmware updates in a timely fashion considering the complexity of the decision making process of updating devices, potential re-certification requirements, and the need for user consent to install updates.

- the distribution of the actual firmware update, potentially in an efficient manner to a large number of devices without human involvement.

- energy efficiency and battery lifetime considerations.

- key management required for verifying the digital signature protecting the manifest.

- incentives for manufacturers to offer a firmware update mechanism as part of their IoT products.

12. Mailing List Information

The discussion list for this document is located at the e-mail address suit@ietf.org [1]. Information on the group and information on how to subscribe to the list is at https://www1.ietf.org/mailman/listinfo/suit

Archives of the list can be found at: https://www.ietf.org/mail-archive/web/suit/current/index.html

13. Acknowledgements

We would like to thank the following persons for their feedback:

- Geraint Luff

- Amyas Phillips

- Dan Ros

- Thomas Eichinger

- Michael Richardson

- Emmanuel Baccelli

- Ned Smith

- Jim Schaad
- Carsten Bormann
- Cullen Jennings
- Olaf Bergmann
- Suhas Nandakumar
- Phillip Hallam-Baker
- Marti Bolivar
- Andrzej Puzdrowski
- Markus Gueller
- Henk Birkholz

- Jintao Zhu
- Takeshi Takahashi
- Jacob Beningo
- Kathleen Moriarty

We would also like to thank the WG chairs, Russ Housley, David Waltermire, Dave Thaler for their support and their reviews.

14. References

14.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC7925] Tschofenig, H., Ed. and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things", RFC 7925, DOI

10.17487/RFC7925, July 2016, <https://www.rfc-editor.org/info/rfc7925>.

14.2. Informative References

[I-D.ietf-cose-hash-sig]
Housley, R., "Use of the HSS/LMS Hash-based Signature Algorithm with CBOR Object Signing and Encryption (COSE)", draft-ietf-cose-hash-sig-04 (work in progress), October 2019.

[I-D.ietf-suit-information-model]
Moran, B., Tschofenig, H., and H. Birkholz, "Firmware Updates for Internet of Things Devices - An Information Model for Manifests", draft-ietf-suit-information-model-03 (work in progress), July 2019.

[I-D.ietf-teep-architecture]
Pei, M., Tschofenig, H., Wheeler, D., Atyeo, A., and D. Liu, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-03 (work in progress), July 2019.

[LwM2M] OMA, ., "Lightweight Machine to Machine Technical Specification, Version 1.0.2", February 2018, <http://www.openmobilealliance.org/release/LightweightM2M/V1_0_2-20180209-A/OMA-TS-LightweightM2M-V1_0_2-20180209-A.pdf>.

[RFC5649] Housley, R. and M. Dworkin, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm", RFC 5649, DOI 10.17487/RFC5649, September 2009, <https://www.rfc-editor.org/info/rfc5649>.

[RFC6024] Reddy, R. and C. Wallace, "Trust Anchor Management Requirements", RFC 6024, DOI 10.17487/RFC6024, October 2010, <https://www.rfc-editor.org/info/rfc6024>.

[RFC8240] Tschofenig, H. and S. Farrell, "Report from the Internet of Things Software Update (IoTSU) Workshop 2016", RFC 8240, DOI 10.17487/RFC8240, September 2017, <https://www.rfc-editor.org/info/rfc8240>.

14.3. URIs

[1] mailto:suit@ietf.org

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Milosch Meriac
Consultant

EMail: milosch@meriac.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

David Brown
Linaro

EMail: david.brown@linaro.org