SUIT
Internet-Draft
Intended status: Informational
Expires: July 31, 2021

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

January 27, 2021

A Firmware Update Architecture for Internet of Things
draft-ietf-suit-architecture-16

Abstract

Vulnerabilities in Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism suitable for devices with resource constraints. Incorporating such an update mechanism is a fundamental requirement for fixing vulnerabilities but it also enables other important capabilities such as updating configuration settings as well as adding new functionality.

In addition to the definition of terminology and an architecture this document motivates the standardization of a manifest format as a transport-agnostic means for describing and protecting firmware updates.

Table of Contents

1. Introduction

Firmware updates can help to fix security vulnerabilities, and performing updates is an important building block in securing IoT devices. Due to rising concerns about insecure IoT devices the Internet Architecture Board (IAB) organized a 'Workshop on Internet of Things (IoT) Software Update (IOTSU)' [RFC8240] to take a look at the bigger picture. The workshop revealed a number of challenges for

developers and led to the formation of the IETF Software Updates for Internet of Things (SUIT) working group.

Developing secure Internet of Things (IoT) devices is not an easy task and supporting a firmware update solution requires skillful engineers. Once devices are deployed, firmware updates play a critical part in their lifecycle management, particularly when devices have a long lifetime, or are deployed in remote or inaccessible areas where manual intervention is cost prohibitive or otherwise difficult. Firmware updates
for IoT devices are expected to work automatically, i.e. without user involvement. Conversely, non-IoT devices are expected to account for user preferences and consent when scheduling updates. Automatic updates that do not require human intervention are key to a scalable solution for fixing software vulnerabilities.

Firmware updates are done not only to fix bugs, but also to add new functionality and to reconfigure the device to work in new environments or to behave differently in an already deployed context.

The manifest specification has to allow that

- The firmware image is authenticated and integrity protected. Attempts to flash a maliciously modified firmware image or an image from an unknown, untrusted source must be prevented. In examples this document uses asymmetric cryptography because it is the preferred approach by many IoT deployments. The use of symmetric credentials is also supported and can be used by very constrained IoT devices.

- The firmware image can be confidentiality protected so that attempts by an adversary to recover the plaintext binary can be mitigated or at least made more difficult. Obtaining the firmware is often one of the first steps to mount an attack since it gives the adversary valuable insights into the software libraries used, configuration settings and generic functionality. Even though reverse engineering the binary can be a tedious process modern reverse engineering frameworks have made this task a lot easier.

Authentication and integrity protection of firmware images must be used in a deployment but the confidential protection of firmware is optional.

While the standardization work has been informed by and optimized for firmware update use cases of Class 1 devices (according to the device class definitions in RFC 7228 [RFC7228]), there is nothing in the architecture that restricts its use to only these constrained IoT devices. Moreover, this architecture is not limited to managing

firmware and software updates, but can also be applied to managing the delivery of arbitrary data, such as configuration information and keys. Unlike higher end devices, like laptops and desktop PCs, many IoT devices do not have user interfaces; and support for unattended updates is, therefore, essential for the design of a practical solution. Constrained IoT devices often use a software engineering model where a developer is responsible for creating and compiling all software running on the device into a single, monolithic firmware image. On higher end devices application software is, on the other hand, often downloaded separately and even obtained from developers different to the developers of the lower level software. The details for how to obtain those application layer software binaries then depends heavily on the platform, programming language used and the sandbox in which the software is executed.

While the IETF standardization work has been focused on the manifest format, a fully interoperable solution needs more than a standardized manifest. For example, protocols for transferring firmware images and manifests to the device need to be available as well as the status tracker functionality. Devices also require a mechanism to discover the status tracker(s) and/or firmware servers, for example using pre-configured hostnames or DNS-SD [RFC6763]. These building blocks have been developed by various organizations under the umbrella of an IoT device management solution. The LwM2M protocol [LwM2M] is one IoT device management protocol.

There are, however, several areas that (partially) fall outside the scope of the IETF and other standards organizations but need to be considered by firmware authors, as well as device and network operators. Here are some of them, as highlighted during the IOTSU workshop:

- Installing firmware updates in a robust fashion so that the update does not break the device functionality of the environment this device operates in. This requires proper testing and offering recovery strategies when a firmware update is unsuccessful.

- Making firmware updates available in a timely fashion considering the complexity of the decision making process for updating devices, potential re-certification requirements, the length of a supply chain an update needs to go through before it reaches the end customer, and the need for user consent to install updates.

- Ensuring an energy efficient design of a battery-powered IoT device because a firmware update, particularly radio communication and writing the firmware image to flash, is an energy-intensive task for a device.

- Creating incentives for device operators to use a firmware update mechanism and to demand the integration of it from IoT device vendors.

- Ensuring that firmware updates addressing critical flaws can be obtained even after a product is discontinued or a vendor goes out of business.

This document starts with a terminology followed by the description of the architecture. We then explain the bootloader and how it integrates with the firmware update mechanism. Subsequently, we offer a categorization of IoT devices in terms of their hardware capabilities relevant for firmware updates. Next, we talk about the manifest structure and how to use it to secure firmware updates. We conclude with a more detailed example.

2. Conventions and Terminology

2.1. Terms

This document uses the following terms:

- Firmware Image: The firmware image, or simply the "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.

  The terms, firmware image, firmware, and image, are used in this document and are interchangeable. We use the term application firmware image to differentiate it from a firmware image that contains the bootloader. An application firmware image, as the name indicates, contains the application program often including all the necessary code to run it (such as protocol stacks, and embedded operating system).

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

- 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 interchangeably with MCU, but MCU tends to imply more limited peripheral functions.

- 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.

- Software: Similar to firmware, but typically dynamically loaded by an Operating System. Used interchangeably with firmware in this document.

- System on Chip (SoC): An SoC is an integrated circuit that contains all components of a computer, such as CPU, memory, input/output ports, secondary storage, a bus to connect the components, and other hardware blocks of logic.

- Trust Anchor: A trust anchor, as defined in [RFC6024], 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.

- Trust Anchor Store: A trust anchor store, as defined in [RFC6024], 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.

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

- Trusted Execution Environments (TEEs): An execution environment that runs alongside of, but is isolated from, an REE. For more information about TEEs see [I-D.ietf-teep-architecture].

2.2. Stakeholders

The following stakeholders are used in this document:

- Author: The author is the entity that creates the firmware image. There may be multiple authors involved in producing firmware running on an IoT device. Section 5 talks about those IoT device deployment cases.

- Device Operator: The device operator is responsible for the day-to-day operation of a fleet of IoT devices. Customers of IoT devices, as the owners of IoT devices - such as enterprise customers or end users - interact with their IoT devices indirectly through the device operator via web or smart phone apps.

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

- Trust Provisioning Authority (TPA): The TPA distributes trust anchors and authorization policies to devices and various stakeholders. The TPA may also delegate rights to stakeholders. Typically, the Original Equipment Manufacturer (OEM) or Original Design Manufacturer (ODM) will act as a TPA, however complex supply chains may require a different design. In some cases, the TPA may decide to remain in full control over the firmware update process of their products.

- User: The end-user of a device. The user may interact with devices via web or smart phone apps, as well as through direct user interfaces.

2.3. Functions

- (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 successfully perform an update.

- Status Tracker: The status tracker has a client and a server component and performs three tasks: 1) It communicates the availability of a new firmware version. This information will flow from the server to the client.
  2) It conveys information about software and hardware characteristics of the device. The information flow is from the client to the server.
  3) It can remotely trigger the firmware update process. The information flow is from the server to the client.

  For example, a device operator may want to read the installed firmware version number running on the device and information about available flash memory. Once an update has been triggered, the device operator may want to obtain information about the state of the firmware update. If errors occurred, the device operator may want to troubleshoot problems by first obtaining diagnostic information (typically using a device management protocol).

We make no assumptions about where the server-side component is deployed. The deployment of status trackers is flexible: they may be found at cloud-based servers or on-premise servers, or they may be embedded in edge computing devices. A status tracker server component may even be deployed on an IoT device. For example, if the IoT device contains multiple MCUs, then the main MCU may act as a status tracker towards the other MCUs. Such deployment is useful when updates have to be synchronized across MCUs.

The status tracker may be operated by any suitable stakeholder; typically the Author, Device Operator, or Network Operator.

- 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 client (locally).

- 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 and these two entities are often physically separated on different devices for scalability reasons.

- Bootloader: A bootloader is a piece of software that is executed once a microcontroller has been reset. It is responsible for deciding what code to execute.

3. Architecture

More devices today than ever before are 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 RS-232. Sending updates over the Internet requires the device to fetch the new firmware image as well as the manifest.

Hence, the following components are necessary on a device for a firmware update solution:

- the Internet protocol stack for firmware downloads. Because firmware images are often multiple kilobytes, sometimes exceeding one hundred kilobytes, for low-end IoT devices and even several megabytes for IoT devices running full-fledged 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.

- the capability to write the received firmware image to persistent storage (most likely flash memory).

- a manifest parser with code to verify a digital signature or a message authentication code.

- the ability to unpack, to decompress and/or to decrypt the received firmware image.

- a status tracker.

The features listed above are most likely offered by code in the application firmware image running on the device rather than by the bootloader itself. Note that cryptographic algorithms will likely run in a trusted execution environment, on a separate MCU, in a hardware security module, or in a secure element rather than in the same context with the application code.

Figure 1 shows the architecture where a firmware image is created by an author, and made available to a firmware server. For security reasons, the author will not have the permissions to upload firmware images to the firmware server and to initiate an update directly. Instead, authors will make firmware images available to the device operators. Note that there may be a longer supply chain involved to pass software updates from the author all the way to the party that can then finally make a decision to deploy it with IoT devices.

As a first step in the firmware update process, the status tracker server needs to inform the status tracker client that a new firmware update is available. This can be accomplished via polling (client-initiated), push notifications (server-initiated), or more complex mechanisms (such as a hybrid approach):

- Client-initiated updates take the form of a status tracker client proactively checking (polling) for updates.

- With Server-initiated updates the server-side component of the status tracker learns about a new firmware version and determines which devices qualify for a firmware update. Once the relevant devices have been selected, the status tracker informs these devices and the firmware consumers obtain those images and manifests. Server-initiated updates are important because they allow a quick response time. Note that in this mode the client-

side status tracker needs to be reachable by the server-side component. This may require devices to keep reachability information on the server-side up-to-date and state at NATs and stateful packet filtering firewalls alive.

- Using a hybrid approach the server-side of the status tracker pushes notifications of availability of an update to the client side and requests the firmware consumer to pull the manifest and the firmware image from the firmware server.

Once the device operator triggers an update via the status tracker, it will keep track of the update process on the device. This allows the device operator to know what devices have received an update and which of them are still pending an update.

Firmware images can be conveyed to devices in a variety of ways, including USB, UART, WiFi, BLE, low-power WAN technologies, mesh networks and many more. At the application layer a variety of protocols are also available: MQTT, CoAP, and HTTP are the most popular application layer protocols used by IoT devices. This architecture does not make assumptions about how the firmware images are distributed to the devices and therefore aims to support all these technologies.

In some cases it may be desirable to distribute firmware images using a multicast or broadcast protocol. This architecture does not make recommendations for any such 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.

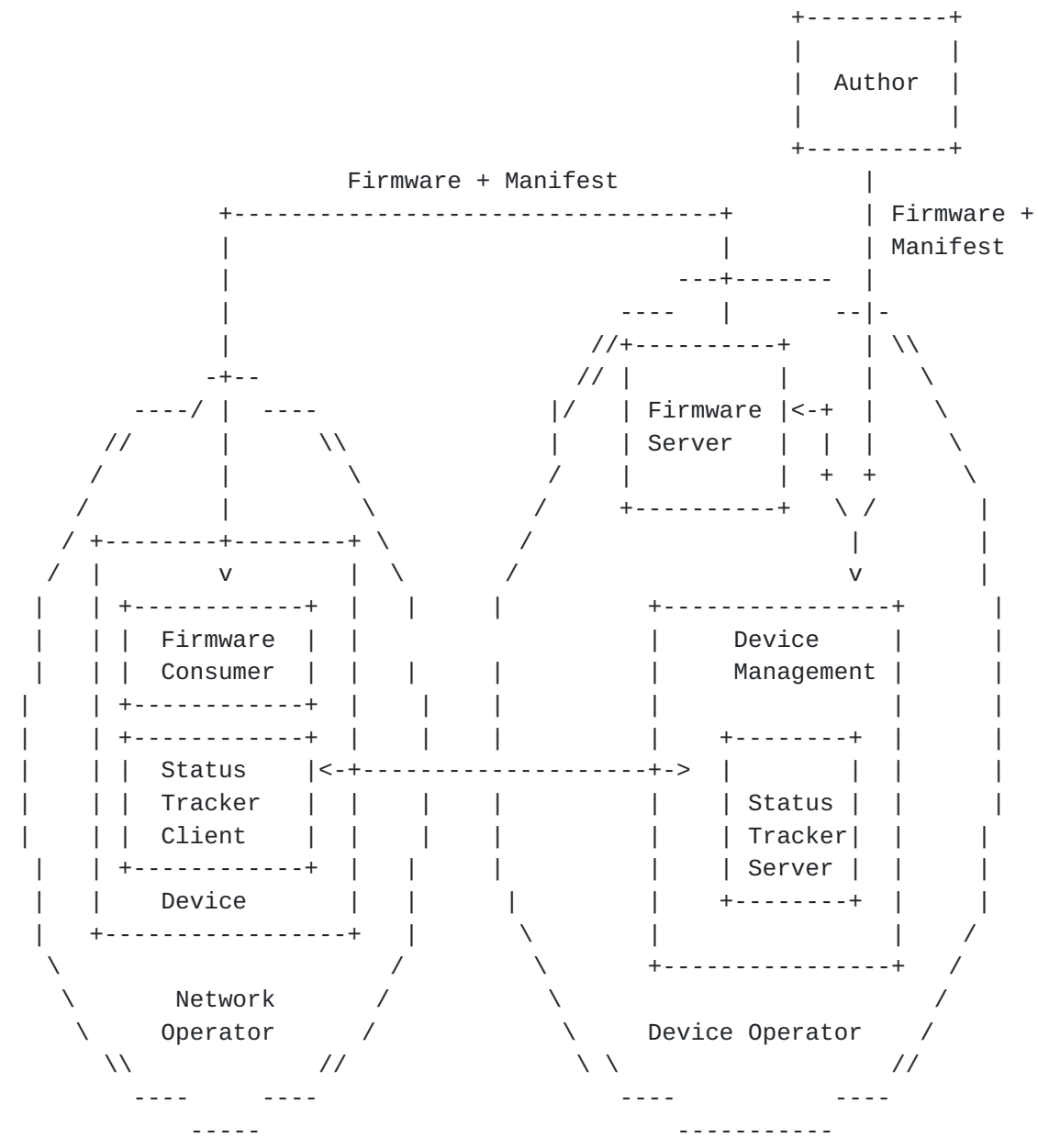


Figure 1: Architecture.

Firmware images and manifests may be conveyed as a bundle or detached. The manifest format must support both approaches.

For distribution as a bundle, the firmware image is embedded into the 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 sticks or short range radio technologies (such as Bluetooth Smart).

Alternatively, the manifest is distributed detached from the firmware image. Using this approach, the firmware consumer is presented with the manifest first and then needs to obtain one or more firmware images as dictated in the manifest.

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/1.1-based firmware server is client-initiated. Pushing a manifest and firmware image to the Package resource of the LwM2M Firmware Update object [LwM2M] is server-initiated update.

If the firmware consumer has downloaded a new firmware image and is ready to install it, to initiate the installation, it may

- either need to wait for a trigger from the status tracker,

- or trigger the update automatically,

- or go through a more complex decision making process to determine

the appropriate timing for an update. Sometimes the final decision may require confirmation of the user of the device for safety reasons.

Installation is the act of processing the payload into a format that the IoT device can recognize and the bootloader is responsible for then booting from the newly installed firmware image. This process is different when a bootloader is not involved. For example, when an application is updated in a full-featured operating system, the updater may halt and restart the application in isolation. Devices must not fail when a disruption, such as a power failure or network interruption, occurs during the update process.

4. Invoking the Firmware

Section 3 describes the steps for getting the firmware image and the manifest from the author to the firmware consumer on the IoT device. Once the firmware consumer has retrieved and successfully processed the manifest and the firmware image it needs to invoke the new firmware image. This is managed in many different ways, depending on the type of device, but it typically involves halting the current version of the firmware, handing control over to a firmware with a higher privilege/trust level (the firmware verifier), verifying the new firmware's authenticity & integrity, and then invoking it.

In an execute-in-place microcontroller, this is often done by rebooting into a bootloader (simultaneously halting the application & handing over to the higher privilege level) then executing a secure boot process (verifying and invoking the new image).

In a rich OS, this may be done by halting one or more processes, then invoking new applications. In some OSs, this implicitly involves the kernel verifying the code signatures on the new applications.

The invocation process is security sensitive. An attacker will typically try to retrieve a firmware image from the device for reverse engineering or will try to get the firmware verifier to execute an attacker-modified firmware image. The firmware verifier will therefore have to perform security checks on the firmware image before it can be invoked. These security checks by the firmware verifier happen in addition to the security checks that took place when the firmware image and the manifest were downloaded by the firmware consumer.

The overlap between the firmware consumer and the firmware verifier functionality comes in two forms, namely

- A firmware verifier 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 firmware verifier 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 firmware verifier may well be the same manifest obtained with the firmware image during the update process.

- An IoT device needs a recovery strategy in case the firmware update / invocation process fails. The recovery strategy may include storing two or more application firmware images on the device or offering the ability to invoke a recovery image to perform the firmware update process again using firmware updates

over serial, USB or even wireless connectivity like Bluetooth Smart. In the latter case the firmware consumer functionality is contained in the recovery image and requires the necessary functionality for executing the firmware update process, including manifest parsing.

While this document assumes that the firmware verifier itself is distinct from the role of the firmware consumer and therefore does not manage the firmware update process, this is not a requirement and these roles may be combined in practice.

Using a bootloader as the firmware verifier requires some special considerations, particularly when the bootloader implements the robustness requirements identified by the IOTSU workshop [RFC8240].

4.1. The Bootloader

In most cases the MCU must restart in order to hand over control to the bootloader. Once the MCU has initiated a restart, the bootloader determines whether a newly available firmware image should be executed. If the bootloader concludes that the newly available firmware image is invalid, a recovery strategy is necessary. There are only two approaches for recovering from an invalid firmware: either the bootloader must be able to select a different, valid firmware, or it must be able to obtain a new, valid firmware. Both of these approaches have implications for the architecture of the update system.

Assuming the first approach, there are (at least) three firmware images available on the device:

- First, the bootloader is also firmware. If a bootloader is updatable then its firmware image is treated like any other application firmware image.

- Second, the firmware image that has to be replaced is still available on the device as a backup in case the freshly downloaded firmware image does not boot or operate correctly.

- Third, there is the newly downloaded firmware image.

Therefore, the firmware consumer must know where to store the new firmware. In some cases, this may be implicit, for example replacing the least-recently-used firmware image. In other cases, the storage location of the new firmware must be explicit, for example when a device has one or more application firmware images and a recovery image with limited functionality, sufficient only to perform an update.

Since many low end IoT devices do not use position-independent code, either the bootloader needs to copy the newly downloaded application firmware image into the location of the old application firmware image and vice versa or multiple versions of the firmware need to be prepared for different locations.

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 reliability risk.

For a bootloader to offer a secure boot functionality it needs to implement the following functionality:

- The bootloader needs to fetch the manifest from nonvolatile storage and parse its contents for subsequent cryptographic verification.

- Cryptographic libraries with hash functions, digital signatures (for asymmetric crypto), message authentication codes (for symmetric crypto) need to be accessible.

- The device needs to have a trust anchor store to verify the digital signature. (Alternatively, access to a key store for use with the message authentication code.)

- There must be an ability to expose boot process-related data to the application firmware (such as to the status tracker). This allows sharing information about the current firmware version, and the status of the firmware update process and whether errors have occurred.

- Produce boot measurements as part of an attestation solution. See [I-D.ietf-rats-architecture] for more information. (optional)

- The bootloader must be able to decrypt firmware images, in case confidentiality protection was applied. This requires a solution for key management. (optional)

5. Types of IoT Devices

There are billions of MCUs used in devices today produced by a large number of silicon manufacturers. While MCUs can vary significantly in their characteristics, there are a number of similiaries allowing us to categorize in groups.

The firmware update architecture, and the manifest format in particular, needs to offer enough flexibility to cover these common deployment cases.

5.1. Single MCU

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. A notable characteristic of these SoCs is that the primary code is generally execute in place (XIP). Due to the non-relocatable nature of the code, the firmware image needs to be placed in a specific location in flash since the code cannot be executed from an arbitrary location in flash. Hence, when the firmware image is updated it is necessary to swap the old and the new image.

5.2. 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.

5.3. Symmetric Multiple CPUs

In more complex SoCs with symmetric multi-processing support, advanced operating systems, such as Linux, are often used. These SoCs frequently use an external storage medium, such as raw NAND flash or eMMC. Due to the higher quantity of resources, these devices are often capable of storing multiple copies of their firmware images and selecting the most appropriate one to boot. Many SoCs also support bootloaders that are capable of updating the firmware image, however this is typically a last resort because it requires the device to be held in the bootloader while the new firmware is downloaded and installed, which results in down-time for the device. Firmware updates in this class of device are typically not done in-place.

5.4. Dual CPU, shared memory

This configuration has two or more heterogeneous CPUs in a single SoC that share memory (flash and RAM). Generally, there will be a mechanism to prevent one CPU from unintentionally accessing memory currently allocated to the other. Upgrades in this case will

typically be done by one of the CPUs, and is similar to the single CPU with secure mode.

5.5. Dual CPU, other bus

This configuration has two or more heterogeneous 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 the primary CPU, 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.

6. Manifests

In order for a firmware consumer to apply an update, it has to make several decisions using manifest-provided information and data available on the device itself. For more detailed information and a longer list of information elements in the manifest consult the information model specification [I-D.ietf-suit-information-model], which offers justifications for each element, and the manifest specification [I-D.ietf-suit-manifest] for details about how this information is included in the manifest.

Table 1 provides examples of decisions to be made.

Decision	Information Elements
Should I trust the author of the firmware?	Trust anchors and authorization policies on the device
Has the firmware been corrupted?	Digital signature and MAC covering the firmware image
Does the firmware update apply to this device?	Conditions with Vendor ID, Class ID and Device ID
Is the update older than the active firmware?	Sequence number in the manifest (1)
When should the device apply the update?	Wait directive
How should the device apply the update?	Manifest commands
What kind of firmware binary is it?	Unpack algorithms to interpret a format.
Where should the update be obtained?	Dependencies on other manifests and firmware image URI in Manifest
Where should the firmware be stored?	Storage Location and Component Identifier

Table 1: Firmware Update Decisions.

(1): 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.

Keeping the code size and complexity of a manifest parsers small is important for constrained IoT devices. Since the manifest parsing code may also be used by the bootloader it can be part of the trusted computing base.

A manifest may be used to protect not only firmware images but also configuration data such as network credentials or personalization data related to firmware or software. Personalization data demonstrates the need for confidentiality to be maintained between two or more stakeholders that both deliver images to the same device.

Personalization data is used with Trusted Execution Environments (TEEs), which 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. The TA's author does not want to expose the TA's code to any other stakeholder or third party. The user does not want to expose the payment information to any other stakeholder or third party.

7. Securing Firmware Updates

Using firmware updates to fix vulnerabilities in devices is important but securing this update mechanism is equally important since security problems are exacerbated by the update mechanism: update is essentially authorized remote code execution, so any security problems in the update process expose that remote code execution system. Failure to secure the firmware update process will help attackers to take control over devices.

End-to-end security mechanisms are used to protect the firmware image and the manifest. The following assumptions are made to allow the firmware consumer to verify the received firmware image and manifest before updating software:

- 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. 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 a duty of the Trust Provisioning Authority.

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

- For confidentiality protection of the firmware image, it must be done in such a way that the intended firmware consumer(s), other authorized parties, and no one else can decrypt it. The information that is encrypted individually for each device/ recipient must be done in a way that is usable with Content

Distribution Networks, bulk storage, and broadcast protocols. 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 optional.

A manifest specification must support different cryptographic algorithms and algorithm extensibility. Moreover, since RSA- and ECC-based signature schemes may become vulnerable to quantum-accelerated key extraction in the future, unchangeable bootloader code in ROM is recommended to use post-quantum secure signature schemes such as hash-based signatures [RFC8778]. A bootloader author must carefully consider the service lifetime of their product and the time horizon for quantum-accelerated key extraction. The worst-case estimate, at time of writing, for the time horizon to key extraction with quantum acceleration is approximately 2030, based on current research [quantum-factorization].

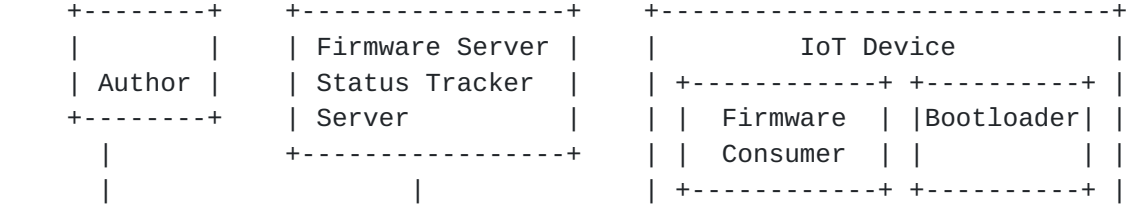
When a device obtains a monolithic firmware image from a single author without any additional approval steps, the authorization flow is relatively simple. There are, however, other cases where more complex policy decisions need to be made before updating a device.

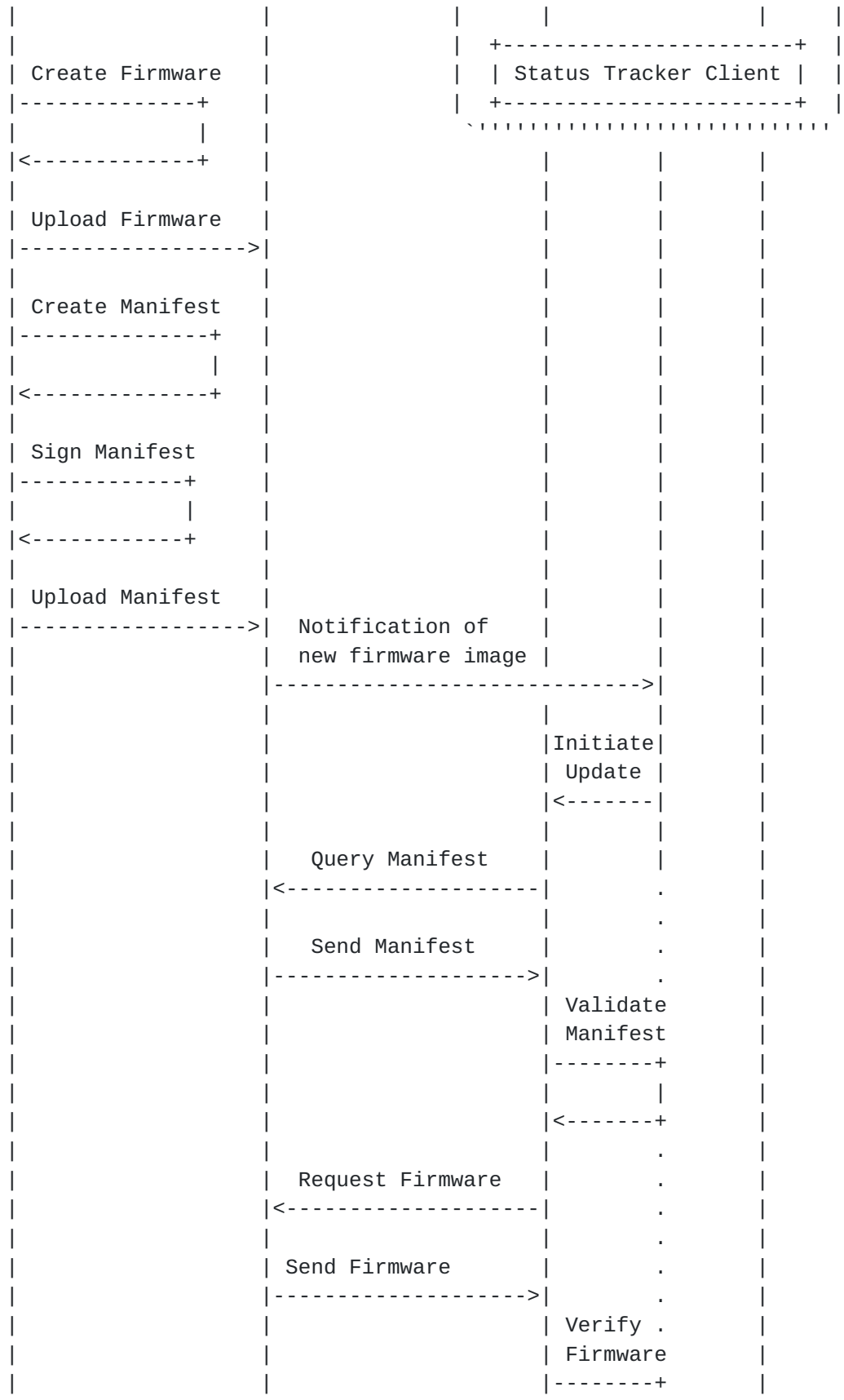
In this architecture the authorization policy is separated from the underlying communication architecture. This is accomplished by separating the entities from their permissions. For example, an author may not have the authority to install a firmware image on a device in critical infrastructure without 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.

8. Example

Figure 2 illustrates an example message flow for distributing a firmware image to a device. The firmware and manifest are stored on the same firmware server and distributed in a detached manner.




Create Firmware
Upload Firmware
Create Manifest
Sign Manifest
Upload Manifest
Status Tracker Client
Notification of new firmware image
Initiate Update
Query Manifest
Send Manifest
Validate Manifest
Request Firmware
Send Firmware
Verify Firmware

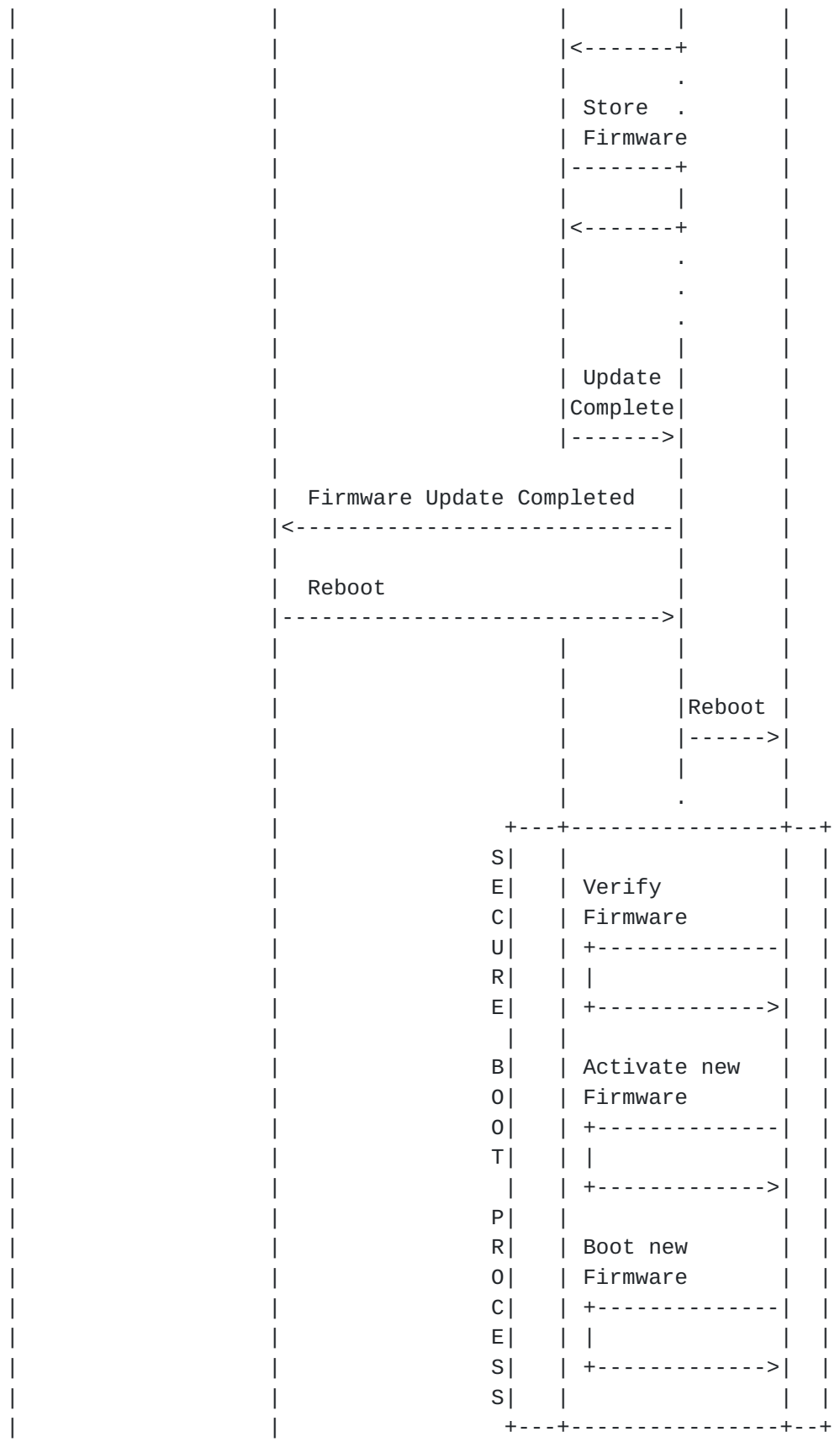
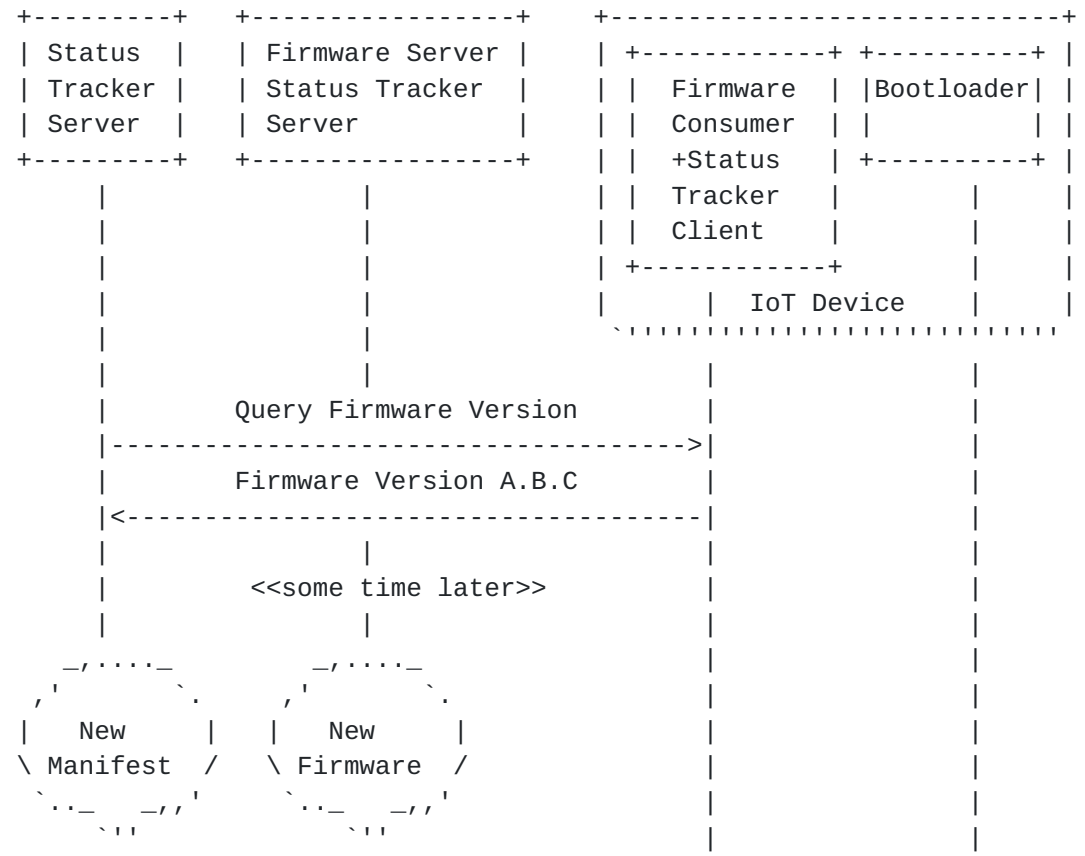

Store
Firmware
Update
Complete
Firmware Update Completed
Reboot
Reboot
SECURE BOOT PROCESS
Verify
Firmware
Activate new
Firmware
Boot new
Firmware

Figure 2: First Example Flow for a Firmware Update.

Figure 3 shows an exchange that starts with the status tracker querying the device for its current firmware version. Later, a new firmware version becomes available and since this device is running an older version the status tracker server interacts with the device to initiate an update.

The manifest and the firmware are stored on different servers in this example. When the device processes the manifest it learns where to download the new firmware version. 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. Finally, the device reports the successful boot of the new firmware version.



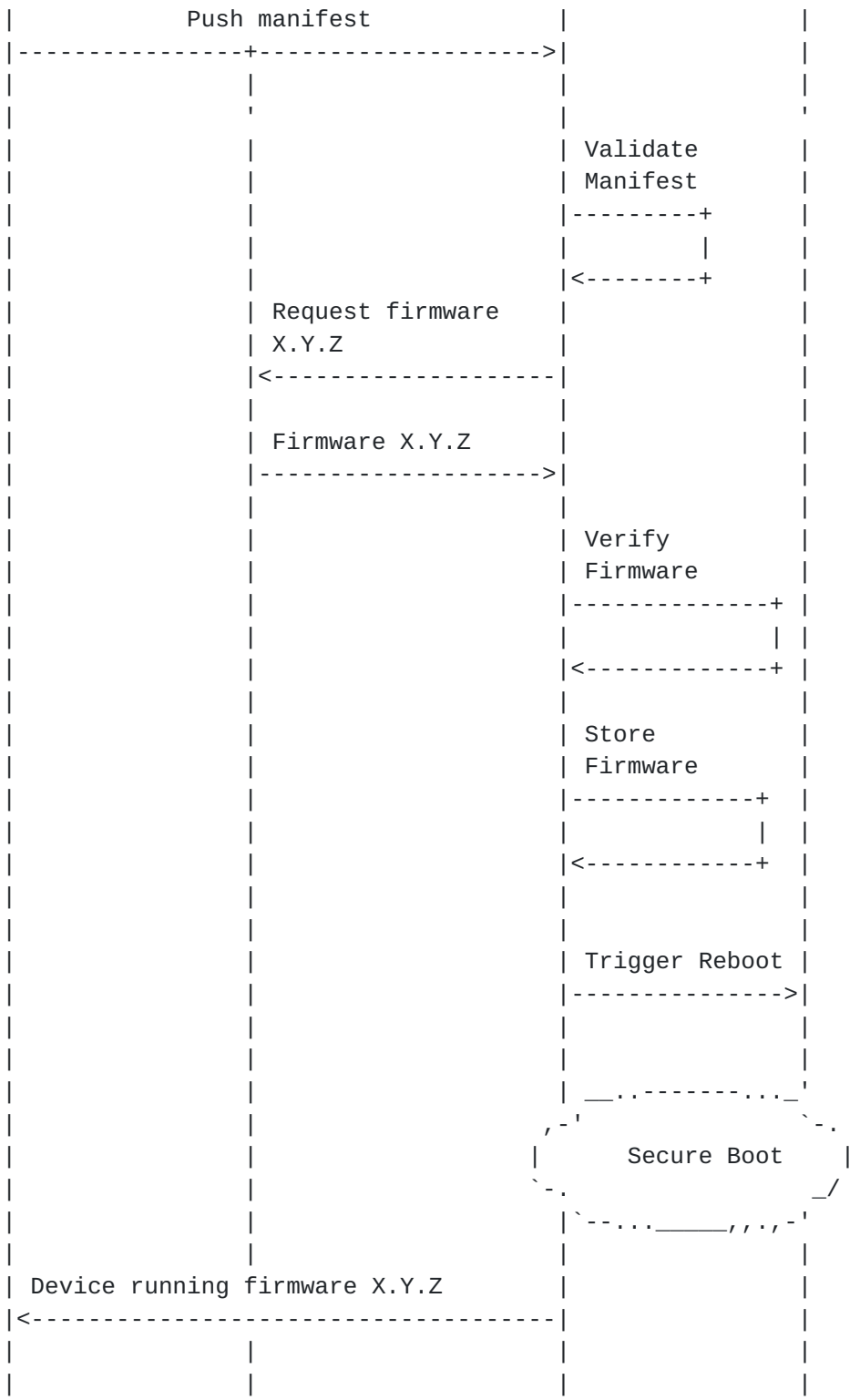


Figure 3: Second Example Flow for a Firmware Update.

9. IANA Considerations

This document does not require any actions by IANA.

10. Security Considerations

This document describes terminology, requirements and an architecture for firmware updates of IoT devices. The content of the document is thereby focused on improving security of IoT devices via firmware update mechanisms and informs the standardization of a manifest format.

An in-depth examination of the security considerations of the architecture is presented in [I-D.ietf-suit-information-model].

11. 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
- Bob Briscoe
- Roman Danyliw
- Brian Carpenter
- Theresa Enghardt
- Rich Salz
- Mohit Sethi
- Eric Vyncke
- Alvaro Retana
- Barry Leiba
- Benjamin Kaduk
- Martin Duke
- Robert Wilton

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

12. Informative References

[I-D.ietf-rats-architecture]
Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote Attestation Procedures Architecture", draft-ietf-rats-architecture-08 (work in progress), December 2020.

[I-D.ietf-suit-information-model]
Moran, B., Tschofenig, H., and H. Birkholz, "An Information Model for Firmware Updates in IoT Devices", draft-ietf-suit-information-model-08 (work in progress), October 2020.

[I-D.ietf-suit-manifest]
Moran, B., Tschofenig, H., Birkholz, H., and K. Zandberg, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest", draft-ietf-suit-manifest-11 (work in progress), December 2020.

[I-D.ietf-teep-architecture]
Pei, M., Tschofenig, H., Thaler, D., and D. Wheeler, "Trusted Execution Environment Provisioning (TEEP) Architecture", draft-ietf-teep-architecture-13 (work in progress), November 2020.

[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>.

[quantum-factorization]
Jiang, S., Britt, K., McCaskey, A., Humble, T., and S. Kais, "Quantum Annealing for Prime Factorization", December 2018, <https://www.nature.com/articles/s41598-018-36058-z>.

[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>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <https://www.rfc-editor.org/info/rfc6763>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

[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>.

[RFC8778] Housley, R., "Use of the HSS/LMS Hash-Based Signature Algorithm with CBOR Object Signing and Encryption (COSE)", RFC 8778, DOI 10.17487/RFC8778, April 2020, <https://www.rfc-editor.org/info/rfc8778>.

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@arm.com

David Brown
Linaro

EMail: david.brown@linaro.org

Milosch Meriac
Consultant

EMail: milosch@meriac.com