Internet Engineering Task Force (IETF)
Request for Comments: 9124
Category: Informational
ISSN: 2070-1721

B. Moran
H. Tschofenig
Arm Limited
H. Birkholz
Fraunhofer SIT

January 2022

A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices

Abstract

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism that is also suitable for constrained devices. Ensuring that devices function and remain secure over their service lifetime requires such an update mechanism to fix vulnerabilities, update configuration settings, and add new functionality.

One component of such a firmware update is a concise and machine-processable metadata document, or manifest, that describes the firmware image(s) and offers appropriate protection. This document describes the information that must be present in the manifest.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9124.

Table of Contents

1. Introduction

Vulnerabilities with Internet of Things (IoT) devices have raised the need for a reliable and secure firmware update mechanism that is also suitable for constrained devices. Ensuring that devices function and remain secure over their service lifetime requires such an update mechanism to fix vulnerabilities, update configuration settings, and add new functionality.

One component of such a firmware update is a concise and machine-processable metadata document, or manifest, that describes the firmware image(s) and offers appropriate protection. This document describes the information that must be present in the manifest.

This document describes all the information elements required in a manifest to secure firmware updates of IoT devices. Each information element is motivated by user stories and threats it aims to mitigate. These threats and user stories are not intended to be an exhaustive list of the threats against IoT devices and possible user stories that describe how to conduct a firmware update. Instead, they are intended to describe the threats against firmware updates in isolation and provide sufficient motivation to specify the information elements that cover a wide range of user stories.

To distinguish information elements from their encoding and serialization over the wire, this document presents an information model. RFC 3444 [RFC3444] describes the differences between information models and data models.

Because this document covers a wide range of user stories and a wide range of threats, not all information elements apply to all scenarios. As a result, various information elements are optional to implement and optional to use, depending on which threats exist in a particular domain of application and which user stories are important for deployments.

2. Requirements and Terminology

2.1. Requirements Notation

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Unless otherwise stated, these words apply to the design of the manifest format, not its implementation or application. Hence, whenever an information element is declared as "REQUIRED", this implies that the manifest format document has to include support for it.

2.2. Terminology

This document uses terms defined in [RFC9019]. The term "Operator" refers to either a device operator or a network operator.

"Secure time" and "secure clock" refer to a set of requirements on time sources. For local time sources, this primarily means that the clock must be monotonically increasing, including across power cycles, firmware updates, etc. For remote time sources, the provided time must be both authenticated and guaranteed to be correct to within some predetermined bounds, whenever the time source is accessible.

The term "Envelope" (or "Manifest Envelope") is used to describe an encoding that allows the bundling of a manifest with related information elements that are not directly contained within the manifest.

The term "payload" is used to describe the data that is delivered to a device during an update. This is distinct from a "firmware image", as described in [RFC9019], because the payload is often in an intermediate state, such as being encrypted, compressed, and/or encoded as a differential update. The payload, taken in isolation, is often not the final firmware image.

3. Manifest Information Elements

Each manifest information element is anchored in a security requirement or a usability requirement. The manifest elements are described below, justified by their requirements.

3.1. Version ID of the Manifest Structure

This is an identifier that describes which iteration of the manifest format is contained in the structure. This allows devices to identify the version of the manifest data model that is in use.

This element is REQUIRED.

3.2. Monotonic Sequence Number

This element provides a monotonically increasing (unsigned) sequence number to prevent malicious actors from reverting a firmware update against the policies of the relevant authority. This number must not wrap around.

For convenience, the monotonic sequence number may be a UTC timestamp. This allows global synchronization of sequence numbers without any additional management.

This element is REQUIRED.

Implements: REQ.SEC.SEQUENCE (Section 4.3.1)

3.3. Vendor ID

The Vendor ID element helps to distinguish between identically named products from different vendors. The Vendor ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only.

Recommended practice is to use version 5 Universally Unique Identifiers (UUIDs) [RFC4122] with the vendor's domain name and the DNS name space ID. Other options include type 1 and type 4 UUIDs.

Fixed-size binary identifiers are preferred because they are simple to match, unambiguous in length, explicitly non-parsable, and require no issuing authority. Guaranteed unique integers are preferred because they are small and simple to match; however, they may not be fixed length, and they may require an issuing authority to ensure uniqueness. Free-form text is avoided because it is variable length, prone to error, and often requires parsing outside the scope of the manifest serialization.

If human-readable content is required, it SHOULD be contained in a separate manifest information element: Manifest Text Information (Section 3.17).

This element is RECOMMENDED.

Implements: REQ.SEC.COMPATIBLE (Section 4.3.2),
REQ.SEC.AUTH.COMPATIBILITY (Section 4.3.10)

Here is an example for a domain-name-based UUID. Vendor A creates a UUID based on a domain name it controls, such as vendorId = UUID5(DNS, "vendor-a.example").

Because the DNS infrastructure prevents multiple registrations of the same domain name, this UUID is (with very high probability) guaranteed to be unique. Because the domain name is known, this UUID is reproducible. Type 1 and type 4 UUIDs produce similar guarantees of uniqueness, but not reproducibility.

This approach creates a contention when a vendor changes its name or relinquishes control of a domain name. In this scenario, it is possible that another vendor would start using that same domain name. However, this UUID is not proof of identity; a device's trust in a vendor must be anchored in a cryptographic key, not a UUID.

3.4. Class ID

A device "Class" is a set of different device types that can accept the same firmware update without modification. It thereby allows devices to determine the applicability of the firmware in an unambiguous way. Class IDs must be unique within the scope of a Vendor ID. This is to prevent similarly or identically named devices from colliding in their customer's infrastructure.

Recommended practice is to use version 5 UUIDs [RFC4122] with as much information as necessary to define firmware compatibility. Possible information used to derive the Class ID UUID includes:

* Model name or number

* Hardware revision

* Runtime library version

* Bootloader version

* ROM revision

* Silicon batch number

The Class ID UUID should use the Vendor ID as the name space identifier. Classes may be more fine-grained than is required to identify firmware compatibility. Classes must not be less granular than is required to identify firmware compatibility. Devices may have multiple Class IDs.

The Class ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only. A manifest serialization SHOULD NOT permit free-form text content to be used for the Class ID. A fixed-size binary identifier SHOULD be used.

Some organizations desire to keep the same product naming across multiple, incompatible hardware revisions for ease of user experience. If this naming is propagated into the firmware, then matching a specific hardware version becomes a challenge. An opaque,

non-readable binary identifier has no naming implications and so is more likely to be usable for distinguishing among incompatible device groupings, regardless of naming.

Fixed-size binary identifiers are preferred because they are simple to match, unambiguous in length, opaque and free from naming implications, and explicitly non-parsable. Free-form text is avoided because it is variable length, prone to error, often requires parsing outside the scope of the manifest serialization, and may be homogenized across incompatible device groupings.

If the Class ID is not implemented, then each logical device class must use a unique trust anchor for authorization.

This element is RECOMMENDED.

Implements: REQ.SEC.COMPATIBLE (Section 4.3.2), REQ.SEC.AUTH.COMPATIBILITY (Section 4.3.10)

3.4.1. Example 1: Different Classes

Vendor A creates Product Z and Product Y. The firmware images of Products Z and Y are not interchangeable. Vendor A creates UUIDs as follows:

* vendorId = UUID5(DNS, "vendor-a.example")

* ZclassId = UUID5(vendorId, "Product Z")

* YclassId = UUID5(vendorId, "Product Y")

This ensures that Vendor A's Product Z cannot install firmware for Product Y and Product Y cannot install firmware for Product Z.

3.4.2. Example 2: Upgrading Class ID

Vendor A creates Product X. Later, Vendor A adds a new feature to Product X, creating Product X v2. Product X requires a firmware update to work with firmware intended for Product X v2.

Vendor A creates UUIDs as follows:

* vendorId = UUID5(DNS, "vendor-a.example")

* XclassId = UUID5(vendorId, "Product X")

* Xv2classId = UUID5(vendorId, "Product X v2")

When Product X receives the firmware update necessary to be compatible with Product X v2, part of the firmware update changes the Class ID to Xv2classId.

3.4.3. Example 3: Shared Functionality

Vendor A produces two products: Product X and Product Y. These components share a common core (such as an operating system (OS)) but

have different applications. The common core and the applications can be updated independently. To enable X and Y to receive the same common core update, they require the same Class ID. To ensure that only Product X receives Application X and only Product Y receives Application Y, Product X and Product Y must have different Class IDs. The vendor creates Class IDs as follows:

* vendorId = UUID5(DNS, "vendor-a.example")

* XclassId = UUID5(vendorId, "Product X")

* YclassId = UUID5(vendorId, "Product Y")

* CommonClassId = UUID5(vendorId, "common core")

Product X matches against both XclassId and CommonClassId. Product Y matches against both YclassId and CommonClassId.

3.4.4. Example 4: Rebranding

Vendor A creates a Product A and its firmware. Vendor B sells the product under its own name as Product B with some customized configuration. The vendors create the Class IDs as follows:

* vendorIdA = UUID5(DNS, "vendor-a.example")

* classIdA = UUID5(vendorIdA, "Product A-Unlabeled")

* vendorIdB = UUID5(DNS, "vendor-b.example")

* classIdB = UUID5(vendorIdB, "Product B")

The product will match against each of these Class IDs. If Vendor A and Vendor B provide different components for the device, the implementor may choose to make ID matching scoped to each component. Then, the vendorIdA, classIdA match the component ID supplied by Vendor A, and the vendorIdB, classIdB match the component ID supplied by Vendor B.

3.5. Precursor Image Digest Condition

This element provides information about the payload that needs to be present on the device for an update to apply. This may, for example, be the case with differential updates.

This element is OPTIONAL.

Implements: REQ.SEC.AUTH.PRECURSOR (Section 4.3.9)

3.6. Required Image Version List

Payloads may only be applied to a specific firmware version or multiple firmware versions. For example, a payload containing a differential update may be applied only to a specific firmware version.

When a payload applies to multiple versions of firmware, the required image version list specifies which firmware versions must be present for the update to be applied. This allows the update author to target specific versions of firmware for an update, while excluding those to which it should not or cannot be applied.

This element is OPTIONAL.

Implements: REQ.USE.IMG.VERSIONS (Section 4.5.8)

3.7. Expiration Time

This element tells a device the time at which the manifest expires and should no longer be used. This element should be used where a secure source of time is provided and firmware is intended to expire predictably. This element may also be displayed (e.g., via an app) for user confirmation, since users typically have a reliable knowledge of the date.

Special consideration is required for end-of-life if firmware will not be updated again -- for example, if a business stops issuing updates to a device. In this case, the last valid firmware should not have an expiration time.

This element is OPTIONAL.

Implements: REQ.SEC.EXP (Section 4.3.3)

3.8. Payload Format

This element describes the payload format within the signed metadata. It is used to enable devices to decode payloads correctly.

This element is REQUIRED.

Implements: REQ.SEC.AUTH.IMG_TYPE (Section 4.3.5),
REQ.USE.IMG.FORMAT (Section 4.5.6)

3.9. Processing Steps

This element provides a representation of the processing steps required to decode a payload -- in particular, those that are compressed, packed, or encrypted. The representation must describe which algorithms are used and must convey any additional parameters required by those algorithms.

A processing step may indicate the expected digest of the payload after the processing is complete.

This element is RECOMMENDED.

Implements: REQ.USE.IMG.NESTED (Section 4.5.7)

3.10. Storage Location

This element tells the device where to store a payload within a given

component. The device can use this to establish which permissions are necessary and the physical storage location to use.

This element is REQUIRED.

Implements: REQ.SEC.AUTH.IMG_LOC (Section 4.3.6)

3.10.1. Example 1: Two Storage Locations

A device supports two components: an OS and an application. These components can be updated independently, expressing dependencies to ensure compatibility between the components. The author chooses two storage identifiers:

* "OS"

* "APP"

3.10.2. Example 2: Filesystem

A device supports a full-featured filesystem. The author chooses to use the storage identifier as the path at which to install the payload. The payload may be a tarball, in which case it unpacks the tarball into the specified path.

3.10.3. Example 3: Flash Memory

A device supports flash memory. The author chooses to make the storage identifier the offset where the image should be written.

3.11. Component Identifier

In a device with more than one storage subsystem, a storage identifier is insufficient to identify where and how to store a payload. To resolve this, a component identifier indicates to which part of the storage subsystem the payload shall be placed.

A serialization may choose to combine the use of a component identifier and storage location (Section 3.10).

This element is OPTIONAL.

Implements: REQ.USE.MFST.COMPONENT (Section 4.5.4)

3.12. Payload Indicator

This element provides the information required for the device to acquire the payload. This functionality is only needed when the target device does not intrinsically know where to find the payload.

This can be encoded in several ways:

* One URI

* A list of URIs

* A prioritized list of URIs

* A list of signed URIs

This element is OPTIONAL.

Implements: REQ.SEC.AUTH.REMOTE_LOC (Section 4.3.7)

3.13. Payload Digests

This element contains one or more digests of one or more payloads. This allows the target device to ensure authenticity of the payload(s) when combined with the Signature (Section 3.15) element. A manifest format must provide a mechanism to select one payload from a list based on system parameters, such as an execute-in-place (XIP) installation address.

This element is REQUIRED. Support for more than one digest is OPTIONAL.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.USE.IMG.SELECT (Section 4.5.9)

3.14. Size

This element provides the size of the payload in bytes, which informs the target device how big of a payload to expect. Without it, devices are exposed to some classes of denial-of-service attacks.

This element is REQUIRED.

Implements: REQ.SEC.AUTH.EXEC (Section 4.3.8)

3.15. Manifest Envelope Element: Signature

The signature element contains all the information necessary to protect the contents of the manifest against modification and to offer authentication of the signer. Because the signature element authenticates the manifest, it cannot be contained within the manifest. Instead, either the manifest is contained within the signature element or the signature element is a member of the Manifest Envelope and bundled with the manifest.

The signature element represents the foundation of all security properties of the manifest. Manifests, which are included as dependencies by other manifests, should include a signature so that the recipient can distinguish between different actors with different permissions.

The signature element must support multiple signers and multiple signing algorithms. A manifest format may allow multiple manifests to be covered by a single signature element.

This element is REQUIRED in non-dependency manifests.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.SEC.RIGHTS

(Section 4.3.11), REQ.USE.MFST.MULTI_AUTH (Section 4.5.5)

3.16. Additional Installation Instructions

Additional installation instructions are machine-readable commands the device should execute when processing the manifest. This information is distinct from the information necessary to process a payload. Additional installation instructions include information such as update timing (for example, install only on Sunday, at 0200), procedural considerations (for example, shut down the equipment under control before executing the update), and pre- and post-installation steps (for example, run a script). Other installation instructions could include requesting user confirmation before installing.

This element is OPTIONAL.

Implements: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

3.17. Manifest Text Information

This is textual information pertaining to the update described by the manifest. This information is for human consumption only. It MUST NOT be the basis of any decision made by the recipient.

This element is OPTIONAL.

Implements: REQ.USE.MFST.TEXT (Section 4.5.2)

3.18. Aliases

Aliases provide a mechanism for a manifest to augment or replace URIs or URI lists defined by one or more of its dependencies.

This element is OPTIONAL.

Implements: REQ.USE.MFST.OVERRIDE_REMOTE (Section 4.5.3)

3.19. Dependencies

This is a list of other manifests that are required by the current manifest. Manifests are identified in an unambiguous way, such as a cryptographic digest.

This element is REQUIRED to support deployments that include both multiple authorities and multiple payloads.

Implements: REQ.USE.MFST.COMPONENT (Section 4.5.4)

3.20. Encryption Wrapper

Encrypting firmware images requires symmetric content encryption keys. The encryption wrapper provides the information needed for a device to obtain or locate a key that it uses to decrypt the firmware.

This element is REQUIRED for encrypted payloads.

Implements: REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12)

3.21. XIP Address

In order to support XIP systems with multiple possible base addresses, it is necessary to specify which address the payload is linked for.

For example, a microcontroller may have a simple bootloader that chooses one of two images to boot. That microcontroller then needs to choose one of two firmware images to install, based on which of its two images is older.

This element is OPTIONAL.

Implements: REQ.USE.IMG.SELECT (Section 4.5.9)

3.22. Load-Time Metadata

Load-time metadata provides the device with information that it needs in order to load one or more images. This metadata may include any of the following:

* The source (e.g., non-volatile storage)

* The destination (e.g., an address in RAM)

* Cryptographic information

* Decompression information

* Unpacking information

Typically, loading is done by copying an image from its permanent storage location into its active use location. The metadata allows operations such as decryption, decompression, and unpacking to be performed during that copy.

This element is OPTIONAL.

Implements: REQ.USE.LOAD (Section 4.5.11)

3.23. Runtime Metadata

Runtime metadata provides the device with any extra information needed to boot the device. This may include the entry point of an XIP image or the kernel command line to boot a Linux image.

This element is OPTIONAL.

Implements: REQ.USE.EXEC (Section 4.5.10)

3.24. Payload

The Payload element is contained within the manifest or Manifest

Envelope and enables the manifest and payload to be delivered simultaneously. This is used for delivering small payloads, such as cryptographic keys or configuration data.

This element is OPTIONAL.

Implements: REQ.USE.PAYLOAD (Section 4.5.12)

3.25. Manifest Envelope Element: Delegation Chain

The delegation chain offers enhanced authorization functionality via authorization tokens, such as Concise Binary Object Representation (CBOR) Web Tokens [RFC8392] with Proof-of-Possession Key Semantics [RFC8747]. Each token itself is protected and does not require another layer of protection. Each authorization token typically includes a public key or a public key fingerprint; however, this is dependent on the tokens used. Each token MAY include additional metadata, such as key usage information. Because the delegation chain is needed to verify the signature, it must be placed in the Manifest Envelope, rather than the manifest.

The first token in any delegation chain MUST be authenticated by the recipient's trust anchor. Each subsequent token MUST be authenticated using the previous token. This allows a recipient to discard each antecedent token after it has authenticated the subsequent token. The final token MUST enable authentication of the manifest. More than one delegation chain MAY be used if more than one signature is used. Note that no restriction is placed on the encoding order of these tokens; the order of elements is logical only.

This element is OPTIONAL.

Implements: REQ.USE.DELEGATION (Section 4.5.14),
REQ.SEC.KEY.ROTATION (Section 4.3.18)

4. Security Considerations

The following subsections describe the threat model, user stories, security requirements, and usability requirements. This section also provides the motivations for each of the manifest information elements.

Note that it is worthwhile to recall that a firmware update is, by definition, remote code execution. Hence, if a device is configured to trust an entity to provide firmware, it trusts this entity to behave correctly. Many classes of attacks can be mitigated by verifying that a firmware update came from a trusted party and that no rollback is taking place. However, if the trusted entity has been compromised and distributes attacker-provided firmware to devices, then the possibilities for defense are limited.

4.1. Threat Model

The following subsections aim to provide information about the threats that were considered, the security requirements that are

derived from those threats, and the fields that permit implementation of the security requirements. This model uses the Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege (STRIDE) approach [STRIDE]. Each threat is classified according to the following:

* Spoofing identity

* Tampering with data

* Repudiation

* Information disclosure

* Denial of service

* Elevation of privilege

This threat model only covers elements related to the transport of firmware updates. It explicitly does not cover threats outside of the transport of firmware updates. For example, threats to an IoT device due to physical access are out of scope.

4.2. Threat Descriptions

Many of the threats detailed in this section contain a "threat escalation" description. This explains how the described threat might fit together with other threats and produce a high-severity threat. This is important because some of the described threats may seem low severity but could be used with others to construct a high-severity compromise.

4.2.1. THREAT.IMG.EXPIRED: Old Firmware

Classification: Elevation of Privilege

An attacker sends an old, but valid, manifest with an old, but valid, firmware image to a device. If there is a known vulnerability in the provided firmware image, this may allow an attacker to exploit the vulnerability and gain control of the device.

Threat Escalation: If the attacker is able to exploit the known vulnerability, then this threat can be escalated to all types.

Mitigated by: REQ.SEC.SEQUENCE (Section 4.3.1)

4.2.2. THREAT.IMG.EXPIRED.OFFLINE: Offline Device + Old Firmware

Classification: Elevation of Privilege

An attacker targets a device that has been offline for a long time and runs an old firmware version. The attacker sends an old, but valid, manifest to a device with an old, but valid, firmware image. The attacker-provided firmware is newer than the installed firmware but older than the most recently available firmware. If there is a known vulnerability in the provided firmware image, then this may

allow an attacker to gain control of a device. Because the device has been offline for a long time, it is unaware of any new updates. As such, it will treat the old manifest as the most current.

The exact mitigation for this threat depends on where the threat comes from. This requires careful consideration by the implementor. If the threat is from a network actor, including an on-path attacker, or an intruder into a management system, then a user confirmation can mitigate this attack, simply by displaying an expiration date and requesting confirmation. On the other hand, if the user is the attacker, then an online confirmation system (for example, a trusted timestamp server) can be used as a mitigation system.

Threat Escalation: If the attacker is able to exploit the known vulnerability, then this threat can be escalated to all types.

Mitigated by: REQ.SEC.EXP (Section 4.3.3), REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.2.3. THREAT.IMG.INCOMPATIBLE: Mismatched Firmware

Classification: Denial of Service

An attacker sends a valid firmware image, for the wrong type of device, signed by an actor with firmware installation permission on both device types. The firmware is verified by the device positively because it is signed by an actor with the appropriate permission. This could have wide-ranging consequences. For devices that are similar, it could cause minor breakage or expose security vulnerabilities. For devices that are very different, it is likely to render devices inoperable.

Mitigated by: REQ.SEC.COMPATIBLE (Section 4.3.2)

For example, suppose that two vendors -- Vendor A and Vendor B -- adopt the same trade name in different geographic regions, and they both make products with the same names, or product name matching is not used. This causes firmware from Vendor A to match devices from Vendor B.

If the vendors are the firmware authorities, then devices from Vendor A will reject images signed by Vendor B, since they use different credentials. However, if both devices trust the same author, then devices from Vendor A could install firmware intended for devices from Vendor B.

4.2.4. THREAT.IMG.FORMAT: The Target Device Misinterprets the Type of Payload

Classification: Denial of Service

If a device misinterprets the format of the firmware image, it may cause a device to install a firmware image incorrectly. An incorrectly installed firmware image would likely cause the device to stop functioning.

Threat Escalation: An attacker that can cause a device to misinterpret the received firmware image may gain elevation of privilege and potentially expand this to all types of threats.

Mitigated by: REQ.SEC.AUTH.IMG_TYPE (Section 4.3.5)

4.2.5. THREAT.IMG.LOCATION: The Target Device Installs the Payload to the Wrong Location

Classification: Denial of Service

If a device installs a firmware image to the wrong location on the device, then it is likely to break. For example, a firmware image installed as an application could cause a device and/or application to stop functioning.

Threat Escalation: An attacker that can cause a device to misinterpret the received code may gain elevation of privilege and potentially expand this to all types of threats.

Mitigated by: REQ.SEC.AUTH.IMG_LOC (Section 4.3.6)

4.2.6. THREAT.NET.REDIRECT: Redirection to Inauthentic Payload Hosting

Classification: Denial of Service

If a device is tricked into fetching a payload for an attacker-controlled site, the attacker may send corrupted payloads to devices.

Mitigated by: REQ.SEC.AUTH.REMOTE_LOC (Section 4.3.7)

4.2.7. THREAT.NET.ONPATH: Traffic Interception

Classification: Spoofing Identity, Tampering with Data

An attacker intercepts all traffic to and from a device. The attacker can monitor or modify any data sent to or received from the device. This can take the form of manifests, payloads, status reports, and capability reports being modified or not delivered to the intended recipient. It can also take the form of analysis of data sent to or from the device, in content, size, or frequency.

Mitigated by: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12), REQ.SEC.AUTH.REMOTE_LOC (Section 4.3.7), REQ.SEC.MFST.CONFIDENTIALITY (Section 4.3.14), REQ.SEC.REPORTING (Section 4.3.16)

4.2.8. THREAT.IMG.REPLACE: Payload Replacement

Classification: Elevation of Privilege

An attacker replaces newly downloaded firmware after a device finishes verifying a manifest. This could cause the device to execute the attacker's code. This attack likely requires physical access to the device. However, it is possible that this attack is

carried out in combination with another threat that allows remote execution. This is a typical Time Of Check / Time Of Use (TOCTOU) attack.

Threat Escalation: If the attacker is able to exploit a known vulnerability or if the attacker can supply their own firmware, then this threat can be escalated to all types.

Mitigated by: REQ.SEC.AUTH.EXEC (Section 4.3.8)

4.2.9. THREAT.IMG.NON_AUTH: Unauthenticated Images

Classification: Elevation of Privilege / all types

If an attacker can install their firmware on a device -- for example, by manipulating either payload or metadata -- then they have complete control of the device.

Mitigated by: REQ.SEC.AUTHENTIC (Section 4.3.4)

4.2.10. THREAT.UPD.WRONG_PRECURSOR: Unexpected Precursor Images

Classification: Denial of Service / all types

Modifications of payloads and metadata allow an attacker to introduce a number of denial-of-service attacks. Below are some examples.

An attacker sends a valid, current manifest to a device that has an unexpected precursor image. If a payload format requires a precursor image (for example, delta updates) and that precursor image is not available on the target device, it could cause the update to break.

An attacker that can cause a device to install a payload against the wrong precursor image could gain elevation of privilege and potentially expand this to all types of threats. However, it is unlikely that a valid differential update applied to an incorrect precursor would result in functional, but vulnerable, firmware.

Mitigated by: REQ.SEC.AUTH.PRECURSOR (Section 4.3.9)

4.2.11. THREAT.UPD.UNAPPROVED: Unapproved Firmware

Classification: Denial of Service, Elevation of Privilege

This threat can appear in several ways; however, it is ultimately about ensuring that devices retain the behavior required by their owner or Operator. The owner or Operator of a device typically requires that the device maintain certain features, functions, capabilities, behaviors, or interoperability constraints (more generally, behavior). If these requirements are broken, then a device will not fulfill its purpose. Therefore, if any party other than the device's owner or the owner's contracted device operator has the ability to modify device behavior without approval, then this constitutes an elevation of privilege.

Similarly, a network operator may require that devices behave in a

particular way in order to maintain the integrity of the network. If device behavior on a network can be modified without the approval of the network operator, then this constitutes an elevation of privilege with respect to the network.

For example, if the owner of a device has purchased that device because of Features A, B, and C, and a firmware update that removes Feature A is issued by the manufacturer, then the device may not fulfill the owner's requirements any more. In certain circumstances, this can cause significantly greater threats. Suppose that Feature A is used to implement a safety-critical system, whether the manufacturer intended this behavior or not. When unapproved firmware is installed, the system may become unsafe.

In a second example, the owner or Operator of a system of two or more interoperating devices needs to approve firmware for their system in order to ensure interoperability with other devices in the system. If the firmware is not qualified, the system as a whole may not work. Therefore, if a device installs firmware without the approval of the device owner or Operator, this is a threat to devices or the system as a whole.

Similarly, the Operator of a network may need to approve firmware for devices attached to the network in order to ensure favorable operating conditions within the network. If the firmware is not qualified, it may degrade the performance of the network. Therefore, if a device installs firmware without the approval of the network operator, this is a threat to the network itself.

Threat Escalation: If the network operator expects configuration that is present in devices deployed in Network A, but not in devices deployed in Network B, then the device may experience degraded security, leading to threats of all types.

Mitigated by: REQ.SEC.RIGHTS (Section 4.3.11), REQ.SEC.ACCESS_CONTROL (Section 4.3.13)

4.2.11.1. Example 1: Multiple Network Operators with a Single Device Operator

In this example, assume that device operators expect the rights to create firmware but that network operators expect the rights to qualify firmware as "fit for purpose" on their networks. Additionally, assume that device operators manage devices that can be deployed on any network, including Network A and Network B in our example.

An attacker may obtain a manifest for a device on Network A. Then, this attacker sends that manifest to a device on Network B. Because Network A and Network B are under the control of different Operators, and the firmware for a device on Network A has not been qualified to be deployed on Network B, the target device on Network B is now in violation of Operator B's policy and may be disabled by this unqualified, but signed, firmware.

This is a denial of service because it can render devices inoperable.

This is an elevation of privilege because it allows the attacker to make installation decisions that should be made by the Operator.

4.2.11.2. Example 2: Single Network Operator with Multiple Device Operators

Multiple devices that interoperate are used on the same network and communicate with each other. Some devices are manufactured and managed by Device Operator A and other devices by Device Operator B. New firmware is released by Device Operator A that breaks compatibility with devices from Device Operator B. An attacker sends the new firmware to the devices managed by Device Operator A without the approval of the network operator. This breaks the behavior of the larger system, causing denial of service and, possibly, other threats. Where the network is a distributed Supervisory Control and Data Acquisition (SCADA) system, this could cause misbehavior of the process that is under control.

4.2.12. THREAT.IMG.DISCLOSURE: Reverse Engineering of Firmware Image for Vulnerability Analysis

Classification: all types

An attacker wants to mount an attack on an IoT device. To prepare the attack, the provided firmware image is reverse engineered and analyzed for vulnerabilities.

Mitigated by: REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12)

4.2.13. THREAT.MFST.OVERRIDE: Overriding Critical Manifest Elements

Classification: Elevation of Privilege

An authorized actor, but not the author, uses an override mechanism (USER_STORY.OVERRIDE (Section 4.4.3)) to change an information element in a manifest signed by the author. For example, if the authorized actor overrides the digest and URI of the payload, the actor can replace the entire payload with a payload of their choice.

Threat Escalation: By overriding elements such as payload installation instructions or a firmware digest, this threat can be escalated to all types.

Mitigated by: REQ.SEC.ACCESS_CONTROL (Section 4.3.13)

4.2.14. THREAT.MFST.EXPOSURE: Confidential Manifest Element Exposure

Classification: Information Disclosure

A third party may be able to extract sensitive information from the manifest.

Mitigated by: REQ.SEC.MFST.CONFIDENTIALITY (Section 4.3.14)

4.2.15. THREAT.IMG.EXTRA: Extra Data after Image

Classification: all types

If a third party modifies the image so that it contains extra code after a valid, authentic image, that third party can then use their own code in order to make better use of an existing vulnerability.

Mitigated by: REQ.SEC.IMG.COMPLETE_DIGEST (Section 4.3.15)

4.2.16. THREAT.KEY.EXPOSURE: Exposure of Signing Keys

Classification: all types

If a third party obtains a key or even indirect access to a key -- for example, in a hardware security module (HSM) -- then they can perform the same actions as the legitimate owner of the key. If the key is trusted for firmware updates, then the third party can perform firmware updates as though they were the legitimate owner of the key.

For example, if manifest signing is performed on a server connected to the internet, an attacker may compromise the server and then be able to sign manifests, even if the keys for manifest signing are held in an HSM that is accessed by the server.

Mitigated by: REQ.SEC.KEY.PROTECTION (Section 4.3.17), REQ.SEC.KEY.ROTATION (Section 4.3.18)

4.2.17. THREAT.MFST.MODIFICATION: Modification of Manifest or Payload prior to Signing

Classification: all types

If an attacker can alter a manifest or payload before it is signed, they can perform all the same actions as the manifest author. This allows the attacker to deploy firmware updates to any devices that trust the manifest author. If an attacker can modify the code of a payload before the corresponding manifest is created, they can insert their own code. If an attacker can modify the manifest before it is signed, they can redirect the manifest to their own payload.

For example, the attacker deploys malware to the developer's computer or signing service that watches manifest creation activities and inserts code into any binary that is referenced by a manifest.

For example, the attacker deploys malware to the developer's computer or signing service that replaces the referenced binary (digest) and URI with the attacker's binary (digest) and URI.

Mitigated by: REQ.SEC.MFST.CHECK (Section 4.3.19), REQ.SEC.MFST.TRUSTED (Section 4.3.20)

4.2.18. THREAT.MFST.TOCTOU: Modification of Manifest between Authentication and Use

Classification: all types

If an attacker can modify a manifest after it is authenticated (time

of check) but before it is used (time of use), then the attacker can place any content whatsoever in the manifest.

Mitigated by: REQ.SEC.MFST.CONST (Section 4.3.21)

4.3. Security Requirements

The security requirements here are a set of policies that mitigate the threats described in Section 4.1.

4.3.1. REQ.SEC.SEQUENCE: Monotonic Sequence Numbers

Only an actor with firmware installation authority is permitted to decide when device firmware can be installed. To enforce this rule, manifests MUST contain monotonically increasing sequence numbers. Manifests may use UTC epoch timestamps to coordinate monotonically increasing sequence numbers across many actors in many locations. If UTC epoch timestamps are used, they must not be treated as times; they must be treated only as sequence numbers. Devices must reject manifests with sequence numbers smaller than any onboard sequence number, i.e., there is no sequence number rollover.

> Note: This is not a firmware version field. It is a manifest sequence number. A firmware version may be rolled back by creating a new manifest for the old firmware version with a later sequence number.

Mitigates: THREAT.IMG.EXPIRED (Section 4.2.1)

Implemented by: Monotonic Sequence Number (Section 3.2)

4.3.2. REQ.SEC.COMPATIBLE: Vendor, Device-Type Identifiers

Devices MUST only apply firmware that is intended for them. Devices must know that a given update applies to their vendor, model, hardware revision, and software revision. Human-readable identifiers are often prone to error in this regard, so unique identifiers should be used instead.

Mitigates: THREAT.IMG.INCOMPATIBLE (Section 4.2.3)

Implemented by: Vendor ID Condition (Section 3.3), Class ID Condition (Section 3.4)

4.3.3. REQ.SEC.EXP: Expiration Time

A firmware manifest MAY expire after a given time, and devices may have a secure clock (local or remote). If a secure clock is provided and the firmware manifest has an expiration timestamp, the device must reject the manifest if the current time is later than the expiration time.

Special consideration is required for end-of-life in cases where a device will not be updated again -- for example, if a business stops issuing updates for a device. The last valid firmware should not have an expiration time.

If a device has a flawed time source (either local or remote), an old update can be deployed as new.

Mitigates: THREAT.IMG.EXPIRED.OFFLINE (Section 4.2.2)

Implemented by: Expiration Time (Section 3.7)

4.3.4. REQ.SEC.AUTHENTIC: Cryptographic Authenticity

The authenticity of an update MUST be demonstrable. Typically, this means that updates must be digitally signed. Because the manifest contains information about how to install the update, the manifest's authenticity must also be demonstrable. To reduce the overhead required for validation, the manifest contains the cryptographic digest of the firmware image, rather than a second digital signature. The authenticity of the manifest can be verified with a digital signature or Message Authentication Code. The authenticity of the firmware image is tied to the manifest by the use of a cryptographic digest of the firmware image.

Mitigates: THREAT.IMG.NON_AUTH (Section 4.2.9), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Signature (Section 3.15), Payload Digests (Section 3.13)

4.3.5. REQ.SEC.AUTH.IMG_TYPE: Authenticated Payload Type

The type of payload MUST be authenticated. For example, the target must know whether the payload is XIP firmware, a loadable module, or configuration data.

Mitigates: THREAT.IMG.FORMAT (Section 4.2.4)

Implemented by: Payload Format (Section 3.8), Signature (Section 3.15)

4.3.6. REQ.SEC.AUTH.IMG_LOC: Authenticated Storage Location

The location on the target where the payload is to be stored MUST be authenticated.

Mitigates: THREAT.IMG.LOCATION (Section 4.2.5)

Implemented by: Storage Location (Section 3.10)

4.3.7. REQ.SEC.AUTH.REMOTE_LOC: Authenticated Remote Payload

The location where a target should find a payload MUST be authenticated. Remote resources need to receive an equal amount of cryptographic protection as the manifest itself, when dereferencing URIs. The security considerations of Uniform Resource Identifiers (URIs) are applicable [RFC3986].

Mitigates: THREAT.NET.REDIRECT (Section 4.2.6), THREAT.NET.ONPATH

(Section 4.2.7)

Implemented by: Payload Indicator (Section 3.12)

4.3.8. REQ.SEC.AUTH.EXEC: Secure Execution

The target SHOULD verify firmware at the time of boot. This requires authenticated payload size and firmware digest.

Mitigates: THREAT.IMG.REPLACE (Section 4.2.8)

Implemented by: Payload Digests (Section 3.13), Size (Section 3.14)

4.3.9. REQ.SEC.AUTH.PRECURSOR: Authenticated Precursor Images

If an update uses a differential compression method, it MUST specify the digest of the precursor image, and that digest MUST be authenticated.

Mitigates: THREAT.UPD.WRONG_PRECURSOR (Section 4.2.10)

Implemented by: Precursor Image Digest (Section 3.5)

4.3.10. REQ.SEC.AUTH.COMPATIBILITY: Authenticated Vendor and Class IDs

The identifiers that specify firmware compatibility MUST be authenticated to ensure that only compatible firmware is installed on a target device.

Mitigates: THREAT.IMG.INCOMPATIBLE (Section 4.2.3)

Implemented by: Vendor ID Condition (Section 3.3), Class ID Condition (Section 3.4)

4.3.11. REQ.SEC.RIGHTS: Rights Require Authenticity

If a device grants different rights to different actors, exercising those rights MUST be accompanied by proof of those rights, in the form of proof of authenticity. Authenticity mechanisms, such as those required in REQ.SEC.AUTHENTIC (Section 4.3.4), can be used to prove authenticity.

For example, if a device has a policy that requires that firmware have both an Authorship right and a Qualification right and if that device grants Authorship and Qualification rights to different parties, such as a device operator and a network operator, respectively, then the firmware cannot be installed without proof of rights from both the device operator and the network operator.

Mitigates: THREAT.UPD.UNAPPROVED (Section 4.2.11)

Implemented by: Signature (Section 3.15)

4.3.12. REQ.SEC.IMG.CONFIDENTIALITY: Payload Encryption

The manifest information model MUST enable encrypted payloads.

Encryption helps to prevent third parties, including attackers, from reading the content of the firmware image. This can protect against confidential information disclosures and discovery of vulnerabilities through reverse engineering. Therefore, the manifest must convey the information required to allow an intended recipient to decrypt an encrypted payload.

Mitigates: THREAT.IMG.DISCLOSURE (Section 4.2.12), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Encryption Wrapper (Section 3.20)

4.3.13. REQ.SEC.ACCESS_CONTROL: Access Control

If a device grants different rights to different actors, then an exercise of those rights MUST be validated against a list of rights for the actor. This typically takes the form of an Access Control List (ACL). ACLs are applied to two scenarios:

1. An ACL decides which elements of the manifest may be overridden and by which actors.

2. An ACL decides which component identifier / storage identifier pairs can be written by which actors.

Mitigates: THREAT.MFST.OVERRIDE (Section 4.2.13), THREAT.UPD.UNAPPROVED (Section 4.2.11)

Implemented by: Client-side code, not specified in manifest

4.3.14. REQ.SEC.MFST.CONFIDENTIALITY: Encrypted Manifests

A manifest format MUST allow encryption of selected parts of the manifest or encryption of the entire manifest to prevent sensitive content of the firmware metadata from being leaked.

Mitigates: THREAT.MFST.EXPOSURE (Section 4.2.14), THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Manifest Encryption Wrapper / Transport Security

4.3.15. REQ.SEC.IMG.COMPLETE_DIGEST: Whole Image Digest

The digest SHOULD cover all available space in a fixed-size storage location. Variable-size storage locations MUST be restricted to exactly the size of deployed payload. This prevents any data from being distributed without being covered by the digest. For example, XIP microcontrollers typically have fixed-size storage. These devices should deploy a digest that covers the deployed firmware image, concatenated with the default erased value of any remaining space.

Mitigates: THREAT.IMG.EXTRA (Section 4.2.15)

Implemented by: Payload Digests (Section 3.13)

4.3.16. REQ.SEC.REPORTING: Secure Reporting

Status reports from the device to any remote system MUST be performed over an authenticated, confidential channel in order to prevent modification or spoofing of the reports.

Mitigates: THREAT.NET.ONPATH (Section 4.2.7)

Implemented by: Transport Security / Manifest format triggering generation of reports

4.3.17. REQ.SEC.KEY.PROTECTION: Protected Storage of Signing Keys

Cryptographic keys for signing/authenticating manifests SHOULD be stored in a manner that is inaccessible to networked devices -- for example, in an HSM or an air-gapped computer. This protects against an attacker obtaining the keys.

Keys SHOULD be stored in a way that limits the risk of a legitimate, but compromised, entity (such as a server or developer computer) issuing signing requests.

Mitigates: THREAT.KEY.EXPOSURE (Section 4.2.16)

Implemented by: Hardware-assisted isolation technologies, which are outside the scope of the manifest format

4.3.18. REQ.SEC.KEY.ROTATION: Protected Storage of Signing Keys

Cryptographic keys for signing/authenticating manifests SHOULD be replaced from time to time. Because it is difficult and risky to replace a trust anchor, keys used for signing updates SHOULD be delegates of that trust anchor.

If key expiration is performed based on time, then a secure clock is needed. If the time source used by a recipient to check for expiration is flawed, an old signing key can be used as current, which compounds THREAT.KEY.EXPOSURE (Section 4.2.16).

Mitigates: THREAT.KEY.EXPOSURE (Section 4.2.16)

Implemented by: Secure storage technology, which is a system design/ implementation aspect outside the scope of the manifest format

4.3.19. REQ.SEC.MFST.CHECK: Validate Manifests prior to Deployment

Manifests SHOULD be verified prior to deployment. This reduces problems that may arise with devices installing firmware images that damage devices unintentionally.

Mitigates: THREAT.MFST.MODIFICATION (Section 4.2.17)

Implemented by: Testing infrastructure. While outside the scope of the manifest format, proper testing of low-level software is essential for avoiding unnecessary downtime or worse situations.

4.3.20. REQ.SEC.MFST.TRUSTED: Construct Manifests in a Trusted Environment

For high-risk deployments, such as large numbers of devices or devices that provide critical functions, manifests SHOULD be constructed in an environment that is protected from interference, such as an air-gapped computer. Note that a networked computer connected to an HSM does not fulfill this requirement (see THREAT.MFST.MODIFICATION (Section 4.2.17)).

Mitigates: THREAT.MFST.MODIFICATION (Section 4.2.17)

Implemented by: Physical and network security for protecting the environment where firmware updates are prepared to avoid unauthorized access to this infrastructure

4.3.21. REQ.SEC.MFST.CONST: Manifest Kept Immutable between Check and Use

Both the manifest and any data extracted from it MUST be held immutable between its authenticity verification (time of check) and its use (time of use). To make this guarantee, the manifest MUST fit within internal memory or secure memory, such as encrypted memory. The recipient SHOULD defend the manifest from tampering by code or hardware resident in the recipient -- for example, other processes or debuggers.

If an application requires that the manifest be verified before storing it, then this means the manifest MUST fit in RAM.

Mitigates: THREAT.MFST.TOCTOU (Section 4.2.18)

Implemented by: Proper system design with sufficient resources and implementation avoiding TOCTOU attacks

4.4. User Stories

User stories provide expected use cases. These are used to feed into usability requirements.

4.4.1. USER_STORY.INSTALL.INSTRUCTIONS: Installation Instructions

As a device operator, I want to provide my devices with additional installation instructions so that I can keep process details out of my payload data.

Some installation instructions might be as follows:

* Use a table of hashes to ensure that each block of the payload is validated before writing.

* Do not report progress.

* Pre-cache the update, but do not install.

* Install the pre-cached update matching this manifest.

* Install this update immediately, overriding any long-running tasks.

Satisfied by: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.4.2. USER_STORY.MFST.FAIL_EARLY: Fail Early

As a designer of a resource-constrained IoT device, I want bad updates to fail as early as possible to preserve battery life and limit consumed bandwidth.

Satisfied by: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.4.3. USER_STORY.OVERRIDE: Override Non-critical Manifest Elements

As a device operator, I would like to be able to override the non-critical information in the manifest so that I can control my devices more precisely. The authority to override this information is provided via the installation of a limited trust anchor by another authority.

Some examples of potentially overridable information:

URIs (Section 3.12): This allows the device operator to direct devices to their own infrastructure in order to reduce network load.

Conditions: This allows the device operator to impose additional constraints on the installation of the manifest.

Directives (Section 3.16): This allows the device operator to add more instructions, such as time of installation.

Processing Steps (Section 3.9): If an intermediary performs an action on behalf of a device, it may need to override the processing steps. It is still possible for a device to verify the final content and the result of any processing step that specifies a digest. Some processing steps should be non-overridable.

Satisfied by: REQ.USE.MFST.COMPONENT (Section 4.5.4)

4.4.4. USER_STORY.COMPONENT: Component Update

As a device operator, I want to divide my firmware into components, so that I can reduce the size of updates, make different parties responsible for different components, and divide my firmware into frequently updated and infrequently updated components.

Satisfied by: REQ.USE.MFST.COMPONENT (Section 4.5.4)

4.4.5. USER_STORY.MULTI_AUTH: Multiple Authorizations

As a device operator, I want to ensure the quality of a firmware update before installing it, so that I can ensure interoperability of all devices in my product family. I want to restrict the ability to

make changes to my devices to require my express approval.

Satisfied by: REQ.USE.MFST.MULTI_AUTH (Section 4.5.5), REQ.SEC.ACCESS_CONTROL (Section 4.3.13)

4.4.6. USER_STORY.IMG.FORMAT: Multiple Payload Formats

As a device operator, I want to be able to send multiple payload formats to suit the needs of my update, so that I can optimize the bandwidth used by my devices.

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.6)

4.4.7. USER_STORY.IMG.CONFIDENTIALITY: Prevent Confidential Information Disclosures

As a firmware author, I want to prevent confidential information in the manifest from being disclosed when distributing manifests and firmware images. Confidential information may include information about the device these updates are being applied to as well as information in the firmware image itself.

Satisfied by: REQ.SEC.IMG.CONFIDENTIALITY (Section 4.3.12)

4.4.8. USER_STORY.IMG.UNKNOWN_FORMAT: Prevent Devices from Unpacking Unknown Formats

As a device operator, I want devices to determine whether they can process a payload prior to downloading it.

In some cases, it may be desirable for a third party to perform some processing on behalf of a target. For this to occur, the third party MUST indicate what processing occurred and how to verify it against the Trust Provisioning Authority's intent.

This amounts to overriding Processing Steps (Section 3.9) and Payload Indicator (Section 3.12).

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.6), REQ.USE.IMG.NESTED (Section 4.5.7), REQ.USE.MFST.OVERRIDE_REMOTE (Section 4.5.3)

4.4.9. USER_STORY.IMG.CURRENT_VERSION: Specify Version Numbers of Target Firmware

As a device operator, I want to be able to target devices for updates based on their current firmware version, so that I can control which versions are replaced with a single manifest.

Satisfied by: REQ.USE.IMG.VERSIONS (Section 4.5.8)

4.4.10. USER_STORY.IMG.SELECT: Enable Devices to Choose between Images

As a developer, I want to be able to sign two or more versions of my firmware in a single manifest so that I can use a very simple bootloader that chooses between two or more images that are executed in place.

Satisfied by: REQ.USE.IMG.SELECT (Section 4.5.9)

4.4.11. USER_STORY.EXEC.MFST: Secure Execution Using Manifests

As a signer for both secure execution/boot and firmware deployment, I would like to use the same signed document for both tasks so that my data size is smaller, I can share common code, and I can reduce signature verifications.

Satisfied by: REQ.USE.EXEC (Section 4.5.10)

4.4.12. USER_STORY.EXEC.DECOMPRESS: Decompress on Load

As a developer of firmware for a run-from-RAM device, I would like to use compressed images and to indicate to the bootloader that I am using a compressed image in the manifest so that it can be used with secure execution/boot.

Satisfied by: REQ.USE.LOAD (Section 4.5.11)

4.4.13. USER_STORY.MFST.IMG: Payload in Manifest

As an Operator of devices on a constrained network, I would like the manifest to be able to include a small payload in the same packet so that I can reduce network traffic.

Small payloads may include, for example, wrapped content encryption keys, configuration information, public keys, authorization tokens, or X.509 certificates.

Satisfied by: REQ.USE.PAYLOAD (Section 4.5.12)

4.4.14. USER_STORY.MFST.PARSE: Simple Parsing

As a developer for constrained devices, I want a low-complexity library for processing updates so that I can fit more application code on my device.

Satisfied by: REQ.USE.PARSE (Section 4.5.13)

4.4.15. USER_STORY.MFST.DELEGATION: Delegated Authority in Manifest

As a device operator that rotates delegated authority more often than delivering firmware updates, I would like to delegate a new authority when I deliver a firmware update so that I can accomplish both tasks in a single transmission.

Satisfied by: REQ.USE.DELEGATION (Section 4.5.14)

4.4.16. USER_STORY.MFST.PRE_CHECK: Update Evaluation

As an Operator of a constrained network, I would like devices on my network to be able to evaluate the suitability of an update prior to initiating any large download so that I can prevent unnecessary consumption of bandwidth.

Satisfied by: REQ.USE.MFST.PRE_CHECK (Section 4.5.1)

4.4.17. USER_STORY.MFST.ADMINISTRATION: Administration of Manifests

As a device operator, I want to understand what an update will do and to which devices it applies so that I can make informed choices about which updates to apply, when to apply them, and which devices should be updated.

Satisfied by: REQ.USE.MFST.TEXT (Section 4.5.2)

4.5. Usability Requirements

The following usability requirements satisfy the user stories listed above.

4.5.1. REQ.USE.MFST.PRE_CHECK: Pre-installation Checks

A manifest format MUST be able to carry all information required to process an update.

For example, information about which precursor image is required for a differential update must be placed in the manifest.

Satisfies: USER_STORY.MFST.PRE_CHECK (Section 4.4.16), USER_STORY.INSTALL.INSTRUCTIONS (Section 4.4.1)

Implemented by: Additional Installation Instructions (Section 3.16)

4.5.2. REQ.USE.MFST.TEXT: Descriptive Manifest Information

It MUST be possible for a device operator to determine what a manifest will do and which devices will accept it prior to distribution.

Satisfies: USER_STORY.MFST.ADMINISTRATION (Section 4.4.17)

Implemented by: Manifest Text Information (Section 3.17)

4.5.3. REQ.USE.MFST.OVERRIDE_REMOTE: Override Remote Resource Location

A manifest format MUST be able to redirect payload fetches. This applies where two manifests are used in conjunction. For example, a device operator creates a manifest specifying a payload and signs it, and provides a URI for that payload. A network operator creates a second manifest, with a dependency on the first. They use this second manifest to override the URIs provided by the device operator, directing them into their own infrastructure instead. Some devices may provide this capability, while others may only look at canonical sources of firmware. For this to be possible, the device must fetch the payload, whereas a device that accepts payload pushes will ignore this feature.

Satisfies: USER_STORY.OVERRIDE (Section 4.4.3)

Implemented by: Aliases (Section 3.18)

4.5.4. REQ.USE.MFST.COMPONENT: Component Updates

A manifest format MUST be able to express the requirement to install one or more payloads from one or more authorities so that a multi-payload update can be described. This allows multiple parties with different permissions to collaborate in creating a single update for the IoT device, across multiple components.

This requirement implies that it must be possible to construct a tree of manifests on a multi-image target.

In order to enable devices with a heterogeneous storage architecture, the manifest must enable specification of both a storage system and the storage location within that storage system.

Satisfies: USER_STORY.OVERRIDE (Section 4.4.3), USER_STORY.COMPONENT (Section 4.4.4)

Implemented by: Dependencies, StorageIdentifier, ComponentIdentifier

4.5.4.1. Example 1: Multiple Microcontrollers

An IoT device with multiple microcontrollers in the same physical device will likely require multiple payloads with different component identifiers.

4.5.4.2. Example 2: Code and Configuration

A firmware image can be divided into two payloads: code and configuration. These payloads may require authorizations from different actors in order to install (see REQ.SEC.RIGHTS (Section 4.3.11) and REQ.SEC.ACCESS_CONTROL (Section 4.3.13)). This structure means that multiple manifests may be required, with a dependency structure between them.

4.5.4.3. Example 3: Multiple Software Modules

A firmware image can be divided into multiple functional blocks for separate testing and distribution. This means that code would need to be distributed in multiple payloads. For example, this might be desirable in order to ensure that common code between devices is identical in order to reduce distribution bandwidth.

4.5.5. REQ.USE.MFST.MULTI_AUTH: Multiple Authentications

A manifest format MUST be able to carry multiple signatures so that authorizations from multiple parties with different permissions can be required in order to authorize installation of a manifest.

Satisfies: USER_STORY.MULTI_AUTH (Section 4.4.5)

Implemented by: Signature (Section 3.15)

4.5.6. REQ.USE.IMG.FORMAT: Format Usability

The manifest format MUST accommodate any payload format that an Operator wishes to use. This enables the recipient to detect which format the Operator has chosen. Some examples of payload format are as follows:

* Binary

* Executable and Linkable Format (ELF)

* Differential

* Compressed

* Packed configuration

* Intel HEX

* Motorola S-Record

Satisfies: USER_STORY.IMG.FORMAT (Section 4.4.6)
USER_STORY.IMG.UNKNOWN_FORMAT (Section 4.4.8)

Implemented by: Payload Format (Section 3.8)

4.5.7. REQ.USE.IMG.NESTED: Nested Formats

The manifest format MUST accommodate nested formats, announcing to the target device all the nesting steps and any parameters used by those steps.

Satisfies: USER_STORY.IMG.CONFIDENTIALITY (Section 4.4.7)

Implemented by: Processing Steps (Section 3.9)

4.5.8. REQ.USE.IMG.VERSIONS: Target Version Matching

The manifest format MUST provide a method to specify multiple version numbers of firmware to which the manifest applies, either with a list or with range matching.

Satisfies: USER_STORY.IMG.CURRENT_VERSION (Section 4.4.9)

Implemented by: Required Image Version List (Section 3.6)

4.5.9. REQ.USE.IMG.SELECT: Select Image by Destination

The manifest format MUST provide a mechanism to list multiple equivalent payloads by execute-in-place (XIP) installation address, including the payload digest and, optionally, payload URIs.

Satisfies: USER_STORY.IMG.SELECT (Section 4.4.10)

Implemented by: XIP Address (Section 3.21)

4.5.10. REQ.USE.EXEC: Executable Manifest

The manifest format MUST allow the description of an executable system with a manifest on both XIP microcontrollers and complex operating systems. In addition, the manifest format MUST be able to express metadata, such as a kernel command line, used by any loader or bootloader.

Satisfies: USER_STORY.EXEC.MFST (Section 4.4.11)

Implemented by: Runtime Metadata (Section 3.23)

4.5.11. REQ.USE.LOAD: Load-Time Information

The manifest format MUST enable carrying additional metadata for load-time processing of a payload, such as cryptographic information, load address, and compression algorithm. Note that load comes before execution/boot.

Satisfies: USER_STORY.EXEC.DECOMPRESS (Section 4.4.12)

Implemented by: Load-Time Metadata (Section 3.22)

4.5.12. REQ.USE.PAYLOAD: Payload in Manifest Envelope

The manifest format MUST allow placing a payload in the same structure as the manifest. This may place the payload in the same packet as the manifest.

Integrated payloads may include, for example, binaries as well as configuration information, and keying material.

When an integrated payload is provided, this increases the size of the manifest. Manifest size can cause several processing and storage concerns that require careful consideration. The payload can prevent the whole manifest from being contained in a single network packet, which can cause fragmentation and the loss of portions of the manifest in lossy networks. This causes the need for reassembly and retransmission logic. The manifest MUST be held immutable between verification and processing (see REQ.SEC.MFST.CONST (Section 4.3.21)), so a larger manifest will consume more memory with immutability guarantees -- for example, internal RAM or NVRAM, or external secure memory. If the manifest exceeds the available immutable memory, then it MUST be processed modularly, evaluating each of the following: delegation chains; the security container; and the actual manifest, which includes verifying the integrated payload. If the security model calls for downloading the manifest and validating it before storing to NVRAM in order to prevent wear to NVRAM and energy expenditure in NVRAM, then either increasing memory allocated to manifest storage or modular processing of the received manifest may be required. While the manifest has been organized to enable this type of processing, it creates additional complexity in the parser. If the manifest is stored in NVRAM prior to processing, the integrated payload may cause the manifest to exceed the available storage. Because the manifest is received prior to validation of applicability, authority, or correctness, integrated payloads cause the recipient to expend network bandwidth and energy that may not be

required if the manifest is discarded, and these costs vary with the size of the integrated payload.

See also: REQ.SEC.MFST.CONST (Section 4.3.21)

Satisfies: USER_STORY.MFST.IMG (Section 4.4.13)

Implemented by: Payload (Section 3.24)

4.5.13. REQ.USE.PARSE: Simple Parsing

The structure of the manifest MUST be simple to parse to reduce the attack vectors against manifest parsers.

Satisfies: USER_STORY.MFST.PARSE (Section 4.4.14)

Implemented by: N/A

4.5.14. REQ.USE.DELEGATION: Delegation of Authority in Manifest

A manifest format MUST enable the delivery of delegation information. This information delivers a new key with which the recipient can verify the manifest.

Satisfies: USER_STORY.MFST.DELEGATION (Section 4.4.15)

Implemented by: Delegation Chain (Section 3.25)

5. IANA Considerations

This document has no IANA actions.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR

Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[RFC9019] Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", RFC 9019, DOI 10.17487/RFC9019, April 2021, <https://www.rfc-editor.org/info/rfc9019>.

6.2. Informative References

[RFC3444] Pras, A. and J. Schoenwaelder, "On the Difference between Information Models and Data Models", RFC 3444, DOI 10.17487/RFC3444, January 2003, <https://www.rfc-editor.org/info/rfc3444>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[STRIDE] Microsoft, "The STRIDE Threat Model", November 2009, <https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)>.

Acknowledgements

We would like to thank our working group chairs -- Dave Thaler, Russ Housley, and David Waltermire -- for their review comments and their support.

We would like to thank the participants of the 2018 Berlin Software Updates for Internet of Things (SUIT) Hackathon and the June 2018 virtual design team meetings for their discussion input.

In particular, we would like to thank Koen Zandberg, Emmanuel Baccelli, Carsten Bormann, David Brown, Markus Gueller, Frank Audun Kvamtrø, Øyvind Rønningstad, Michael Richardson, Jan-Frederik Rieckers, Francisco Acosta, Anton Gerasimov, Matthias Wählisch, Max Gröning, Daniel Petry, Gaëtan Harter, Ralph Hamm, Steve Patrick, Fabio Utzig, Paul Lambert, Saïd Gharout, and Milen Stoychev.

We would like to thank those who contributed to the development of this information model. In particular, we would like to thank Milosch Meriac, Jean-Luc Giraud, Dan Ros, Amyas Phillips, and Gary Thomson.

Finally, we would like to thank the following IESG members for their review feedback: Erik Kline, Murray Kucherawy, Barry Leiba, Alissa Cooper, Stephen Farrell, and Benjamin Kaduk.

Authors' Addresses

Brendan Moran
Arm Limited

Email: Brendan.Moran@arm.com

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@gmx.net

Henk Birkholz
Fraunhofer SIT

Email: henk.birkholz@sit.fraunhofer.de