SUIT
Internet-Draft
Intended status: Informational
Expires: December 4, 2020

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

June 02, 2020

An Information Model for Firmware Updates in IoT Devices
draft-ietf-suit-information-model-07

Abstract

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

One component of such a firmware update is a concise and machine-processable meta-data 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 Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on December 4, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 5
2. Conventions and Terminology 5
  2.1. Requirements Notation 6
3. Manifest Information Elements 6
  3.1. Manifest Element: Version ID of the manifest structure . 6
  3.2. Manifest Element: Monotonic Sequence Number 6
  3.3. Manifest Element: Vendor ID 6
    3.3.1. Example: Domain Name-based UUIDs 7
  3.4. Manifest Element: Class ID 7
    3.4.1. Example 1: Different Classes 8
    3.4.2. Example 2: Upgrading Class ID 8
    3.4.3. Example 3: Shared Functionality 9
    3.4.4. Example 4: White-labelling 9
  3.5. Manifest Element: Precursor Image Digest Condition . . . 10
  3.6. Manifest Element: Required Image Version List 10
  3.7. Manifest Element: Expiration Time 10
  3.8. Manifest Element: Payload Format 10
  3.9. Manifest Element: Processing Steps 11
  3.10. Manifest Element: Storage Location 11
    3.10.1. Example 1: Two Storage Locations 11
    3.10.2. Example 2: File System 11
    3.10.3. Example 3: Flash Memory 12
  3.11. Manifest Element: Component Identifier 12
  3.12. Manifest Element: Resource Indicator 12
  3.13. Manifest Element: Payload Digests 12
  3.14. Manifest Element: Size 13
  3.15. Manifest Element: Signature 13
  3.16. Manifest Element: Additional installation instructions . 13
  3.17. Manifest Element: Aliases 14
  3.18. Manifest Element: Dependencies 14
  3.19. Manifest Element: Encryption Wrapper 14
  3.20. Manifest Element: XIP Address 14
  3.21. Manifest Element: Load-time metadata 15
  3.22. Manifest Element: Run-time metadata 15
  3.23. Manifest Element: Payload 15
  3.24. Manifest Element: Key Claims 15

4. Security Considerations 15
4.1. Threat Model 16
4.2. Threat Descriptions 16
4.2.1. THREAT.IMG.EXPIRED: Old Firmware 16
4.2.2. THREAT.IMG.EXPIRED.ROLLBACK : Offline device + Old Firmware 16
4.2.3. THREAT.IMG.INCOMPATIBLE: Mismatched Firmware 17
4.2.4. THREAT.IMG.FORMAT: The target device misinterprets the type of payload 17
4.2.5. THREAT.IMG.LOCATION: The target device installs the payload to the wrong location 18
4.2.6. THREAT.NET.REDIRECT: Redirection to inauthentic payload hosting 18
4.2.7. THREAT.NET.MITM: Traffic interception 18
4.2.8. THREAT.IMG.REPLACE: Payload Replacement 19
4.2.9. THREAT.IMG.NON_AUTH: Unauthenticated Images 19
4.2.10. THREAT.UPD.WRONG_PRECURSOR: Unexpected Precursor images 19
4.2.11. THREAT.UPD.UNAPPROVED: Unapproved Firmware 20
4.2.12. THREAT.IMG.DISCLOSURE: Reverse Engineering Of Firmware Image for Vulnerability Analysis 21
4.2.13. THREAT.MFST.OVERRIDE: Overriding Critical Manifest Elements 22
4.2.14. THREAT.MFST.EXPOSURE: Confidential Manifest Element Exposure 22
4.2.15. THREAT.IMG.EXTRA: Extra data after image 22
4.2.16. THREAT.KEY.EXPOSURE: Exposure of signing keys 22
4.2.17. THREAT.MFST.MODIFICATION: Modification of manifest or payload prior to signing 23
4.2.18. THREAT.MFST.TOCTOU: Modification of manifest between authentication and use 23
4.3. Security Requirements 24
4.3.1. REQ.SEC.SEQUENCE: Monotonic Sequence Numbers 24
4.3.2. REQ.SEC.COMPATIBLE: Vendor, Device-type Identifiers . 24
4.3.3. REQ.SEC.EXP: Expiration Time 24
4.3.4. REQ.SEC.AUTHENTIC: Cryptographic Authenticity 25
4.3.5. REQ.SEC.AUTH.IMG_TYPE: Authenticated Payload Type . . 25
4.3.6. Security Requirement REQ.SEC.AUTH.IMG_LOC: Authenticated Storage Location 25
4.3.7. REQ.SEC.AUTH.REMOTE_LOC: Authenticated Remote Resource Location 25
4.3.8. REQ.SEC.AUTH.EXEC: Secure Execution 26
4.3.9. REQ.SEC.AUTH.PRECURSOR: Authenticated precursor images 26
4.3.10. REQ.SEC.AUTH.COMPATIBILITY: Authenticated Vendor and Class IDs 26
4.3.11. REQ.SEC.RIGHTS: Rights Require Authenticity 26
4.3.12. REQ.SEC.IMG.CONFIDENTIALITY: Payload Encryption . . . 27

4.3.13. REQ.SEC.ACCESS_CONTROL: Access Control 27
4.3.14. REQ.SEC.MFST.CONFIDENTIALITY: Encrypted Manifests . . 27
4.3.15. REQ.SEC.IMG.COMPLETE_DIGEST: Whole Image Digest . . . 28
4.3.16. REQ.SEC.REPORTING: Secure Reporting 28
4.3.17. REQ.SEC.KEY.PROTECTION: Protected storage of signing keys 28
4.3.18. REQ.SEC.MFST.CHECK: Validate manifests prior to deployment 28
4.3.19. REQ.SEC.MFST.TRUSTED: Construct manifests in a trusted environment 29
4.3.20. REQ.SEC.MFST.CONST: Manifest kept immutable between check and use 29
4.4. User Stories 29
4.4.1. USER_STORY.INSTALL.INSTRUCTIONS: Installation Instructions 29
4.4.2. USER_STORY.MFST.FAIL_EARLY: Fail Early 30
4.4.3. USER_STORY.OVERRIDE: Override Non-Critical Manifest Elements 30
4.4.4. USER_STORY.COMPONENT: Component Update 31
4.4.5. USER_STORY.MULTI_AUTH: Multiple Authorisations . . . 31
4.4.6. USER_STORY.IMG.FORMAT: Multiple Payload Formats . . . 31
4.4.7. USER_STORY.IMG.CONFIDENTIALITY: Prevent Confidential Information Disclosures 31
4.4.8. USER_STORY.IMG.UNKNOWN_FORMAT: Prevent Devices from Unpacking Unknown Formats 31
4.4.9. USER_STORY.IMG.CURRENT_VERSION: Specify Version Numbers of Target Firmware 32
4.4.10. USER_STORY.IMG.SELECT: Enable Devices to Choose Between Images 32
4.4.11. USER_STORY.EXEC.MFST: Secure Execution Using Manifests 32
4.4.12. USER_STORY.EXEC.DECOMPRESS: Decompress on Load . . . 32
4.4.13. USER_STORY.MFST.IMG: Payload in Manifest 32
4.4.14. USER_STORY.MFST.PARSE: Simple Parsing 33
4.4.15. USER_STORY.MFST.DELEGATION: Delegated Authority in Manifest 33
4.4.16. USER_STORY.MFST.PRE_CHECK: Update Evaluation 33
4.5. Usability Requirements 33
4.5.1. REQ.USE.MFST.PRE_CHECK: Pre-Installation Checks . . . 33
4.5.2. REQ.USE.MFST.OVERRIDE_REMOTE: Override Remote Resource Location 34
4.5.3. REQ.USE.MFST.COMPONENT: Component Updates 34
4.5.4. REQ.USE.MFST.MULTI_AUTH: Multiple authentications . . 35
4.5.5. REQ.USE.IMG.FORMAT: Format Usability 35
4.5.6. REQ.USE.IMG.NESTED: Nested Formats 36
4.5.7. REQ.USE.IMG.VERSIONS: Target Version Matching 36
4.5.8. REQ.USE.IMG.SELECT: Select Image by Destination . . . 36
4.5.9. REQ.USE.EXEC: Executable Manifest 36

4.5.10. REQ.USE.LOAD: Load-Time Information 37
4.5.11. REQ.USE.PAYLOAD: Payload in Manifest Superstructure . 37
4.5.12. REQ.USE.PARSE: Simple Parsing 38
4.5.13. REQ.USE.DELEGATION: Delegation of Authority in Manifest 38
5. IANA Considerations 38
6. Acknowledgements 38
7. References 39
7.1. Normative References 39
7.2. Informative References 39
Authors' Addresses 39

1. Introduction

The information model describes all the information elements required to secure firmware updates of IoT devices from the threats described in Section 4.1 and enables the user stories captured in Section 4.4. These threats and user stories are not intended to be an exhaustive list of the threats against IoT devices, nor of the 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. The information model does not define the serialization, encoding, ordering, or structure of information elements, only their semantics.

Because the information model 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 could be considered optional to implement and optional to use, depending on which threats exist in a particular domain of application and which user stories are required. Elements marked as REQUIRED provide baseline security and usability properties that are expected to be required for most applications. Those elements are required to be implemented and used. Elements marked as RECOMMENDED provide important security or usability properties that are needed on most devices. Elements marked as OPTIONAL enable security or usability properties that are useful in some applications.

The definition of some of the information elements include examples that illustrate their semantics and how they are intended to be used.

2. Conventions and Terminology

This document uses terms defined in [I-D.ietf-suit-architecture]. The term 'Operator' refers to both Device and Network Operator.

This document treats devices with a homogeneous storage architecture as devices with a heterogeneous storage architecture, but with a single storage subsystem.

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.

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. Manifest Element: Version ID of the manifest structure

An identifier that describes which iteration of the manifest format is contained in the structure.

This element is REQUIRED in order to allow devices to identify the version of the manifest data model that is in use.

3.2. Manifest Element: Monotonic Sequence Number

A monotonically increasing sequence number. For convenience, the monotonic sequence number MAY be a UTC timestamp. This allows global synchronisation of sequence numbers without any additional management. This number MUST be easily accessible so that code choosing one out of several manifests can choose which is the latest.

This element is REQUIRED and is necessary to prevent malicious actors from reverting a firmware update against the policies of the relevant authority.

Implements: REQ.SEC.SEQUENCE (Section 4.3.1)

3.3. Manifest Element: Vendor ID

Vendor IDs must be unique. This is to prevent similarly, or identically named entities from different geographic regions from colliding in their customer's infrastructure. Recommended practice is to use [RFC4122] version 5 UUIDs with the vendor's domain name and the DNS name space ID. Other options include type 1 and type 4 UUIDs.

Vendor ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only.

The use of a Vendor ID is RECOMMENDED. It helps to distinguish between identically named products from different vendors.

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

3.3.1. Example: Domain Name-based UUIDs

Vendor A creates a UUID based on their domain name:

```
vendorId = UUID5(DNS, "vendor-a.com")
```

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. Manifest Element: Class ID

A device "Class" is a set of different device types that can accept the same firmware update without modification. Class IDs MUST be unique within the scope of a Vendor ID. This is to prevent similarly, or identically named devices colliding in their customer's infrastructure.

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

- model name or number
- hardware revision
- runtime library version
- bootloader version
- ROM revision

- silicon batch number

The Class Identifier UUID SHOULD use the Vendor ID as the name space ID. Other options include version 1 and 4 UUIDs. Classes MAY be more granular 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.

Class ID is not intended to be a human-readable element. It is intended for binary match/mismatch comparison only.

The use of Class ID is RECOMMENDED. It allows devices to determine applicability of a firmware in an unambiguous way.

If Class ID is not implemented, then each logical device class MUST use a unique trust anchor for authorisation.

Implements: Security Requirement 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.com")

- 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.com")

- 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), 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.com")
- 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: White-labelling

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

- vendorIdA = UUID5(DNS, "vendor-a.com")
- classIdA = UUID5(vendorIdA, "Product A-Unlabelled")
- vendorIdB = UUID5(DNS, "vendor-b.com")
- 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. Manifest Element: Precursor Image Digest Condition

When a precursor image is required by the payload format, a precursor image digest condition MUST be present in the conditions list. The precursor image may be installed or stored as a candidate.

This element is OPTIONAL to implement.

Enables feature: differential updates.

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

3.6. Manifest Element: Required Image Version List

When a payload applies to multiple versions of a firmware, the required image version list specifies which 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 be applied.

Where an update can only be applied over specific predecessor versions, that version MUST be specified by the Required Image Version List.

This element is OPTIONAL.

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

3.7. Manifest Element: Expiration Time

This element tells a device the time at which the manifest expires and should no longer be used. This is only usable in conjunction with a secure source of time.

This element is OPTIONAL and may enable user stories where a secure source of time is provided and firmware is intended to expire predictably.

Implements: REQ.SEC.EXP (Section 4.3.3)

3.8. Manifest Element: Payload Format

The format of the payload MUST be indicated to devices in an unambiguous way. This element provides a mechanism to describe the payload format, within the signed metadata.

This element is REQUIRED and MUST be present to enable devices to decode payloads correctly.

Implements: REQ.SEC.AUTH.IMG_TYPE (Section 4.3.5), REQ.USE.IMG.FORMAT (Section 4.5.5)

3.9. Manifest Element: Processing Steps

A representation of the Processing Steps required to decode a payload. The representation MUST describe which algorithm(s) is used and any additional parameters required by the algorithm(s). The representation MAY group Processing Steps together in predefined combinations.

A Processing Step MAY indicate the expected digest of the payload after the processing is complete.

Processing steps are RECOMMENDED to implement.

Enables feature: Encrypted, compressed, packed formats

Implements: REQ.USE.IMG.NESTED (Section 4.5.6)

3.10. Manifest Element: 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 and MUST be present to enable devices to store payloads to the correct location.

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: File System

A device supports a full 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. Manifest Element: Component Identifier

In a heterogeneous storage architecture, a storage identifier is insufficient to identify where and how to store a payload. To resolve this, a component identifier indicates which part of the storage architecture is targeted by the payload. In a homogeneous storage architecture, this element is unnecessary.

This element is OPTIONAL and only necessary in heterogeneous storage architecture devices.

N.B. A manifest format MAY choose to combine Component Identifier and Storage Location (Section 3.10)

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

3.12. Manifest Element: Resource Indicator

This element provides the information required for the device to acquire the resource. This can be encoded in several ways:

- One URI

- A list of URIs

- A prioritised list of URIs

- A list of signed URIs

This element is OPTIONAL and only needed when the target device does not intrinsically know where to find the payload.

N.B. Devices will typically require URIs.

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

3.13. Manifest Element: 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). A manifest format MUST provide a mechanism to select one payload from a list based on system parameters, such as Execute-In-Place Installation Address.

This element is REQUIRED to implement and fundamentally necessary to ensure the authenticity and integrity of the payload. Support for more than one digest is OPTIONAL to implement in a recipient device.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.USE.IMG.SELECT (Section 4.5.8)

3.14. Manifest Element: Size

The size of the payload in bytes.

Variable-size storage locations MUST be set to exactly the size listed in this element.

This element is REQUIRED and informs the target device how big of a payload to expect. Without it, devices are exposed to some classes of denial of service attack.

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

3.15. Manifest Element: Signature

This is not strictly a manifest element. Instead, the manifest is wrapped by a standardised authentication container. The authentication container MUST support multiple signers and multiple signature algorithms.

This element is REQUIRED in non-dependency manifests and represents the foundation of all security properties of the manifest. Manifests which are included as dependencies by another manifest SHOULD include a signature so that the recipient can distinguish between different actors with different permissions.

A manifest MUST NOT be considered authenticated by channel security even if it contains only channel information (such as URIs). If the authenticated remote or channel were compromised, the threat actor could induce recipients to query traffic over any accessible network. Lightweight authentication with pre-existing relationships SHOULD be done with MAC.

Implements: REQ.SEC.AUTHENTIC (Section 4.3.4), REQ.SEC.RIGHTS (Section 4.3.11), REQ.USE.MFST.MULTI_AUTH (Section 4.5.4)

3.16. Manifest Element: Additional installation instructions

Instructions that 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), pre- and post-installation steps (for example, run a script).

This element is OPTIONAL.

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

3.17. Manifest Element: Aliases

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 and enables some user stories.

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

3.18. Manifest Element: Dependencies

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

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

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

3.19. Manifest Element: 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 MAY be included in a decryption step contained in Processing Steps (Section 3.9).

This element is REQUIRED to use for encrypted payloads,

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

3.20. Manifest Element: 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.

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

3.21. Manifest Element: Load-time metadata

Load-time metadata provides the device with information that it needs in order to load one or more images. This is effectively a copy operation from the permanent storage location of an image into the active use location of that image. The metadata contains the source and destination of the image as well as any operations that are performed on the image.

Implements: REQ.USE.LOAD (Section 4.5.10)

3.22. Manifest Element: Run-time metadata

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

Implements: REQ.USE.EXEC (Section 4.5.9)

3.23. Manifest Element: Payload

The Payload element provides a recipient device with the whole payload, contained within the manifest superstructure. This enables the manifest and payload to be delivered simultaneously.

Implements: REQ.USE.PAYLOAD (Section 4.5.11)

3.24. Manifest Element: Key Claims

The Key Claims element is not authenticated by the Signature (Section 3.15), instead, it provides a chain of key delegations (or references to them) for the device to follow in order to verify the key that authenticated the manifest using a trusted key.

Implements: REQ.USE.DELEGATION (Section 4.5.13)

4. Security Considerations

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

4.1. Threat Model

The following sub-sections 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 S.T.R.I.D.E. [STRIDE] approach. Each threat is classified according to:

- 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

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.ROLLBACK : 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 one 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.

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)

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 types of device. 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)

4.2.3.1. 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 threat.

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 an 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 threat.

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

4.2.6. THREAT.NET.REDIRECT: Redirection to inauthentic payload hosting

Classification: Denial of Service

If a device does not know where to obtain the payload for an update, it may be redirected to an attacker's server. This would allow an attacker to provide broken payloads to devices.

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

4.2.7. THREAT.NET.MITM: 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, either 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 a 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 threat.

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, 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

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 threat. However, it is unlikely that a valid differential update applied to an incorrect precursor would result in a 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 behaviour required by their Owner, Device Operator, or Network Operator. The owner or operator of a device typically requires that the device maintain certain features, functions, capabilities, behaviours, or interoperability constraints (more generally, behaviour). 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 behaviour 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 devices behaviour 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 is issued by the manufacturer, which removes Feature A, 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 behaviour 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 favourable 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 firmware 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 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 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 the 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. A 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 approval of the Network Operator. This breaks the behaviour of the larger system causing denial of service and possibly other threats. Where the network is a distributed SCADA system, this could cause misbehaviour 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 he or she retrieves the provided firmware image and

performs reverse engineering of the firmware image to analyze it for specific 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 authorised 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 authorised 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 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 an HSM, then they can perform the same actions as the legitimate owner of the key. If the key is trusted for firmware

update, 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)

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 4.3.18), REQ.SEC.MFST.TRUSTED (Section 4.3.19)

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 4.3.20)

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.

Note: This is not a firmware version. 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 with fine granularity that a given update applies to their vendor, model, hardware revision, software revision. Human-readable identifiers are often error-prone in this regard, so unique identifiers SHOULD be used.

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. Devices MAY provide 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 current time is later than the expiration time.

Mitigates: THREAT.IMG.EXPIRED.ROLLBACK (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 authenticated. 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 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 digest of the firmware image.

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

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

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

The type of payload (which may be independent of format) 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), Storage Location (Section 3.10)

4.3.6. Security Requirement 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 Resource Location

The location where a target should find a payload MUST be authenticated.

Mitigates: THREAT.NET.REDIRECT (Section 4.2.6), THREAT.NET.MITM (Section 4.2.7)

Implemented by: Resource Indicator (Section 3.12)

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

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

Mitigates: THREAT.IMG.REPLACE (Section 4.2.8)

Implemented by: Payload Digest (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.MITM (Section 4.2.7)

Implemented by: Encryption Wrapper (Section 3.19)

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

It MUST be possible to encrypt part or all of the manifest. This may be accomplished with either transport encryption or with at-rest encryption.

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

Implemented by: External 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 SHOULD be performed over an authenticated, confidential channel in order to prevent modification or spoofing of the reports.

Mitigates: THREAT.NET.MITM (Section 4.2.7)

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)

4.3.18. REQ.SEC.MFST.CHECK: Validate manifests prior to deployment

Manifests SHOULD be parsed and examined prior to deployment to validate that their contents have not been modified during creation and signing.

Mitigates: THREAT.MFST.MODIFICATION (Section 4.2.17)

4.3.19. REQ.SEC.MFST.TRUSTED: Construct manifests in a trusted environment

For high risk deployments, such as large numbers of devices or critical function devices, 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)

4.3.20. 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 an internal memory or a 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 is verified before storing it, then this means the manifest MUST fit in RAM.

Mitigates: THREAT.MFST.TOCTOU (Section 4.2.18)

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:

- Use a table of hashes to ensure that each block of the payload is validate 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 pose 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: USER_STORY.OVERRIDE (Section 4.4.3), REQ.USE.MFST.COMPONENT (Section 4.5.3)

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 4.5.3)

4.4.5. USER_STORY.MULTI_AUTH: Multiple Authorisations

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 4.5.4), 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 optimise the bandwidth used by my devices.

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.5)

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

As a firmware author, I want to prevent confidential information from being disclosed during firmware updates. It is assumed that channel security or at-rest encryption is adequate to protect the manifest itself against information disclosure.

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 Resource Indicator (Section 3.12).

Satisfied by: REQ.USE.IMG.FORMAT (Section 4.5.5), REQ.USE.IMG.NESTED (Section 4.5.6), REQ.USE.MFST.OVERRIDE_REMOTE (Section 4.5.2)

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 4.5.7)

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 4.5.8)

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 4.5.9)

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 4.5.10)

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 encryption keys, configuration information, public keys, authorization tokens, or X.509 certificates.

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

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 4.5.12)

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 4.5.13)

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)

4.5. Usability Requirements

The following usability requirements satisfy the user stories listed above.

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

It MUST be possible for a manifest author to place ALL information required to process an update in the manifest.

For example: Information about which precursor image is required for a differential update MUST be placed in the manifest, not in the differential compression header.

Satisfies: [USER_STORY.MFST.PRE_CHECK(#user-story-mfst-pre-check), USER_STORY.INSTALL.INSTRUCTIONS (Section 4.4.1)

Implemented by: Additional installation instructions (Section 3.16)

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

It MUST be possible 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 3.17)

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

It MUST be possible 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 effectively means 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 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 Manifest Element: Dependencies, StorageIdentifier, ComponentIdentifier

4.5.3.1. Example 1: Multiple Microcontrollers

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

4.5.3.2. 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.

4.5.3.3. 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.

4.5.4. REQ.USE.MFST.MULTI_AUTH: Multiple authentications

It MUST be possible to authenticate a manifest multiple times so that authorisations from multiple parties with different permissions can be required in order to authorise installation of a manifest.

Satisfies: USER_STORY.MULTI_AUTH (Section 4.4.5)

Implemented by: Signature (Section 3.15)

4.5.5. 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:

- 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)

4.5.6. 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)

4.5.7. 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)

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

The manifest format MUST provide a mechanism to list multiple equivalent payloads by Execute-In-Place Installation Address, including the payload digest and, optionally, payload URIs.

Satisfies: USER_STORY.IMG.SELECT (Section 4.4.10)

Implemented by: XIP Address (Section 3.20)

4.5.9. REQ.USE.EXEC: Executable Manifest

It MUST be possible to describe an executable system with a manifest on both Execute-In-Place microcontrollers and on complex operating systems. This requires the manifest to specify the digest of each statically linked dependency. 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: Run-time metadata (Section 3.22)

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

It MUST be possible to specify additional metadata for load time processing of a payload, such as cryptographic information, load-address, and compression algorithm.

N.B. load comes before exec/boot.

Satisfies: USER_STORY.EXEC.DECOMPRESS (Section 4.4.12)

Implemented by: Load-time metadata (Section 3.21)

4.5.11. REQ.USE.PAYLOAD: Payload in Manifest Superstructure

It MUST be possible to place 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, wrapped encryption keys, configuration information, public keys, authorization tokens, or X.509 certificates.

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 4.3.20)), 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: 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 organised 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 4.3.20).

Satisfies: USER_STORY.MFST.IMG (Section 4.4.13)

Implemented by: Payload (Section 3.23)

4.5.12. REQ.USE.PARSE: Simple Parsing

The structure of the manifest MUST be simple to parse, without need for a general-purpose parser.

Satisfies: USER_STORY.MFST.PARSE (Section 4.4.14)

Implemented by: N/A

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

Any manifest format MUST enable the delivery of a key claim with, but not authenticated by, a manifest. This key claim delivers a new key with which the recipient can verify the manifest.

Satisfies: USER_STORY.MFST.DELEGATION (Section 4.4.15)

Implemented by: Key Claims (Section 3.24)

5. IANA Considerations

This document does not require any actions by IANA.

6. 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 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 Kvamtro, Oyvind Ronningstad, Michael Richardson, Jan-Frederik Rieckers, Francisco Acosta, Anton Gerasimov, Matthias Waehlisch, Max Groening, Daniel Petry, Gaetan Harter, Ralph Hamm, Steve Patrick, Fabio Utzig, Paul Lambert, Benjamin Kaduk, Said 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 Philips, and Gary Thomson.

7. References

7.1. Normative References

[I-D.ietf-suit-architecture]
Moran, B., Tschofenig, H., Brown, D., and M. Meriac, "A Firmware Update Architecture for Internet of Things", draft-ietf-suit-architecture-11 (work in progress), May 2020.

[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>.

7.2. Informative References

[STRIDE] Microsoft, "The STRIDE Threat Model", May 2018, <https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx>.

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