SUIT B. Moran
Internet-Draft M. Meriac
Intended status: Standards Track H. Tschofenig
Expires: August 2, 2018 Arm Limited
January 29, 2018

Firmware Manifest Format
draft-moran-suit-manifest-01

Abstract

This specification describes the format of a manifest. A manifest is a bundle of metadata about the firmware for an IoT device, where to find the firmware, the devices to which it applies, and cryptographic information protecting the manifest.

Status of This Memo

This Internet-Draft will expire on August 2, 2018.

Copyright Notice

Table of Contents

1. Introduction

A firmware update mechanism is an essential security feature for IoT devices to deal with vulnerabilities. While the transport of firmware images to the devices themselves is important there are already various techniques available, such as the Lightweight Machine-to-Machine (LwM2M) protocol offering device management of IoT devices. Equally important is the inclusion of meta-data about the conveyed firmware image (in the form of a manifest) and the use of end-to-end security protection to detect modifications and (optionally) to make reverse engineering more difficult. End-to-end security allows the author, who builds the firmware image, to be sure that no other party (including potential adversaries) installs firmware updates on IoT devices without adequate privileges. This authorization process is ensured by the use of dedicated symmetric or asymmetric keys installed on the IoT device: for use cases where only integrity protection is required it is sufficient to install a trust anchor on the IoT device. For confidentiality protected firmware images it is additionally required to install either one or multiple symmetric or asymmetric keys on the IoT device. Starting security protection at the author is a risk mitigation technique so that firmware images and manifests can be stored on untrusted respositories.

It is assumed that the reader is familiar with the high-level firmware update architecture [Architecture].

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

To describe the components of the manifest we use the terms structures and attributes. The manifest has a hierarchical structure and top level components are called structures and the attributes are the components within them.

3. Components

The key components of a manifest are shown in Figure 1 and are explained in the sub-sections below.

```
    +-------------------+    +-----------+
    | Manifest          |    | Condition |
    +-------------------+    +-----------+
    |  manifestVersion  |    |  type     |
    |  text             |    |  value    |     +-----------+
    |  nonce            |    +-----+-----+     | Directive |
    |  timestamp        |          |           +-----------+
    |  conditions ------+----------+           |  type     |
    |  directives ------+----------------------+  value    |
    |  aliases ---------+-----------------+    +-----------+
    |  dependencies ----+--------------+  |
    |  payloadInfo-+    |              |  |
    +--------------|----+        +-----+--+-----------+
                   |             | ResourceReference  |
                   |             +--------------------+
                   |             |  hash              |
                   |             |  uri               |
                   |             +--------------------+
                   |
    +--------------+-----+
    | PayloadInfo        |
    +--------------------+
    |  format            |
    |  digests           |
    |  storageIdentifier |
    |  uris              |
    |  size              |
    |  COSE_Encrypt      |
    +--------------------+
```

Figure 1: Components of a Manifest.

3.1. Manifest

The Manifest structure is the top-level construct that ties all other structures together. In addition to the structures explained in subsections below it contains

- a version number (in the 'manifestVersion' attribute)

- a textual description about the update, including the version / vendor / model of the device (in the 'text' attribute). This information is optional.

- a timestamp indicating when the manifest was created (in the 'timestamp' attribute).

The following CDDL fragment defines the manifest.

```
Manifest = [
    manifestVersion : uint,
    text : {* int => tstr } / nil,
    nonce : bstr,
    timestamp : uint,
    conditions: [ * condition ],
    directives: [ * directive ] / nil,
    aliases: [ * ResourceReference ] / nil,
    dependencies: [ * ResourceReference ] / nil,
    extensions: { * int => bstr } / nil,
    payloadInfo: ? PayloadInfo
]
```

In the text section, positive integers define standard text fields, described in this draft. Negative integers define application-specific text fields.

Field ID	Description
1	A text description of this manifest
2	A text description of the payload
3	A text representation of the target vendor name / manufacturer name
4	A text representation of the target model number / model name

Text fields are never used by the target. They are for informational purposes only.

3.2. PayloadInfo

The PayloadInfo structure contains information about the firmware image:

- format: contains the firmware image type (such as rawBinary, hexLocationLengthData, ELF). Format is an array of: an integer (positive for standard types, negative for application-specific types) and a bstr that encapsulates any information needed by the format processor, that is not included in the firmware image itself.

- size: offers information about the size of the firmware image in bytes. If the size of the obtained firmware image differs from the size stated in the manifest then the obtained image MUST be consider corrupted.

- nonce: contains a (short) random value to ensure that a given manifest is unique. This separates the function of the timestamp, which is provided for rollback protection, from the function of the nonce, which is for uniqueness. Keeping these functions separate ensures that a number of edge cases are catered for, for example: the creation of manifests quickly enough that they have the same timestamp.

- storageIdentifier: indicates where the image should be placed on the device. This value is useful, for example, when an IoT device contains multiple microcontrollers (MCUs) and the decision needs to be made to which MCU to send which firmware image. Another example is when an IoT device contains both firmware and configuration and the configuration must be updated while the firmware remains the same.

- uris: a set of ranked references for where to find the payload. By using a ranking, the device can select which the preferred URIs are. If several URIs have the same preference, then devices SHOULD select randomly from the available URIs of the same rank. URIs need not be a URL, a URN is acceptable if the target understands it. The uri can allow a device to use a ranked search pattern to choose the best location to look for the payload in complex distribution scenarios, such as attempting to find the payload on a gateway device prior to looking on a fileserver.

- digestAlgorithm: This defines the type of digest used for all entries in the digests list. The type must be a standard COSE MAC algorithm or a message digest algorithm (these are not yet defined in COSE). An optional 'parameters' bstr is provided in case one of these algorithms requires additional configuration that would normally be present in the 'protected' or 'unprotected' fields of the COSE_Mac object.

- digests: This is a map of possible digests. It is indexed by integer: positive for standardized digests and negative for application-specific digests.

- payload: a COSE_Encrypt object, a bstr, or nil. Note that a COSE_Mac could be used instead of a bstr / nil, but this would be redundant since the whole structure is already authenticated.

NOTE: digests needs some form of key derivation to prevent the need for multiple keys. It is expected that the same key be used, with a KDF of some kind, to derive a key from the key used to sign the manifest in the case of COSE_Mac manifests. Where manifests use COSE_Sign at the top level, it is expected that digests will use standard message digest algorithms instead of MAC algorithms.

The following CDDL fragment defines the payload info:

```
PayloadInfo = [
    format = [
        type: int,
        ? parameters : bstr
    ],
    size: uint,
    storageIdentifier: bstr,
    uris: [*[
        rank: int,
        uri: tstr
    ]] / nil,
    digestAlgorithm = [
        type : int,
        ? parameters: bstr
    ] / nil,
    digests = {* int => bstr} / nil,
    payload = COSE_Encrypt / bstr / nil
]
```

Digests can contain several kinds of digest:

Digest ID	Description
1	raw payload digest: the digest of the payload with no modification. This is the digest of the plaintext. This data is redundant when an AEAD algo is used.
2	installed payload digest: the digest of the payload, post-installation. This is most useful in differential updates.
3	ciphertext digest: The digest of the ciphertext of the payload. This is useful when compressed or differential updates are used, since it can be used to verify the downloaded package prior to decryption.
4	pre-image digest: The digest of the image that must already be present in the device in order to install the payload.

There are several ways that this format can reference a payload:

1. The payload can be contained by the COSE_Encrypt object. In this case, no URIs are expected, since the payload is contained in COSE_Encrypt.

2. The COSE_Encrypt object is present, but its 'ciphertext' is nil. This means that the ciphertext payload is delivered separately. In this case, at least one URI is expected in uris.

3. The payload is a bstr. This encapsulates a plaintext payload. A raw payload digest is redundant. No URIs are expected.

4. The payload is a nil. This means that the plaintext payload is delivered separately. In this case, at least one URI is expected in uris. At least one digest is expected in digests.

Most importantly, however, the PayloadInfo structure contains a reference to the firmware image (in the 'reference' attribute) or the image is embedded inside the PayloadInfo structure (within the 'integrated' attribute). A referenced image first needs to be fetched by the device before the update can be applied. The 'reference' attribute contains a 'hash' and a 'uri' attribute: the value in the 'hash' attribute allows the device to determine whether it has already obtained this firmware image and, since it is included in the digitally signed manifest, it protects the firmware image against modifications. The 'uri' attribute references the image.

Encryption is handled by the COSE_Encrypt structure. Most encryption modes are already supported via the COSE_Encrypt structure, only per-device pre-shared keys (or per-device ECDH derivation of pre-shared keys) needs to be described. When using an encrypted image key, shared between many devices, the COSE_Encrypt recipients structure should be filled out as follows:

```
/ recipients / [
    [
        / protected / h'a1011820' / {
            \ alg \ 1:32 \ AES-CCM-64-128-128 \
        } / ,
        / unprotected / {
            / kid / 4:'<uri for key lookup>' /,
            / iv / 5:h'<7 bytes>'
        },
        / ciphertext / nil
    ]
]
```

Figure 2: AES-CCM-64-128-128 COSE Example.

This allows a manifest to direct a device to fetch keys from a particular location, identify them by name, or perform another fetch/lookup operation. The exact method for key distribution is out of scope. (However, an array of COSE_Encrypt objects, each containing a single key object, with a simple recipient object seems appropriate.)

This mode is tailored to use cases where a single encrypted firmware image is transmitted to many IoT devices.

3.3. Condition and Directive

The Condition and the Directive structures together allow "If <...> Then <...>" rules to be expressed.

It offers the following functionality:

- Apply an update immediately (Directive.applyImmediately)

- Apply an update only to devices that match the vendorId, classId, deviceId attributes

- Apply an update only if the device system time is before the time indicated in the Condition.lastApplicationTime.

- Wait to apply an update until the device system time is after an indicated time.

The following CDDL fragment defines the structure of a condition:

```
condition = [
    type : int,
    parameters : bstr
]
directive = condition
```

Some condition types are predefined:

Condition ID	Description
1	Vendor ID. parameters contains the 128-bit vendor ID to match.
2	Class ID. parameters contains the 128-bit device class ID to match.
3	Device ID. parameters contains the 128-bit device ID to match.
4	Best Before. Do not apply the update after time. parameters is serialized as an uint timestamp encoded in the bstr.

Some directive types are predefined:

Directive ID	Description
1	Apply Immediately. Apply right away. Do not wait. parameters MUST contain a True or False, serialized in the bstr. Setting this value to False will cause the target to wait until a new manifest arrives with Apply Immediately set to true and a dependency on this manifest.
2	Apply After. Wait until time to apply update. parameters is serialized as a uint timestamp encoded in the bstr.

Application-specific conditions and directives MUST use negative identifiers.

3.4. Dependencies and Aliases

In some situations an IoT device may require more than a single firmware image. To express the requirement that more than one image has to be installed on a device the dependencies structure is used, which is of type ResourceReference.

The following CDDL fragment defines the ResourceReference:

```
ResourceReference = [
    uri : tstr,
    digest : bstr
]
```

Aliases are used to refer to alternative locations of firmware images. This is useful in environments where organizations cache firmware images (and their corresponding manifests) on premise to avoid the need to fetch images from repositories maintained by the developer’s organizations (such a device manufacturer or an OEM).

3.5. Device Identification

A device is identified by a combination of three identifiers:

- A vendor identifier

- A device class identifier

- A device identifier

3.5.1. Vendor ID

The vendor ID is a 128-bit number that conforms to RFC 4122, type 5. This number is used by the device to verify manifests.

The Vendor ID should be derived from the manufacturer's domain name using the algorithm defined in Section 4.3 of RFC 4122.

A vendor ID is typically compiled into a firmware image since it is static for the lifetime of the firmware.

3.5.2. Device Class ID

The device class is a 128-bit number that conforms to RFC 4122, type 5. This number is used by the client to verify manifests. The Device Class ID SHOULD use the Vendor ID as the namespace, but the ID within the namespace can be arbitrary.

A class ID is also typically compiled into a firmware image since it is static for the lifetime of the firmware.

3.5.3. Device ID

The device ID is also a 128-bit number that conforms to RFC 4122. The device ID can come from a variety of sources. For example, a device may obtain this identifier during the manufacturing phase (together with other configuration information and manufacturer-provided credentials). In this case, we recommend using RFC 4122, type 1, where the node ID is the factory tool ID, which provides traceability of a device back to the origin of manufacture. A device ID can also come from on-device resources, such as device unique-ID registers or device identifiers in CPUs. Our recommendation is to provide unique CPU resources to a generator function similar to the one used for the class_id. In this example, the device_info may be a combination of several components, such as:

- MAC address

- Device unique identifier

Where multiple sources of unique identity are available, they should all be provided to the UUID function, since it combines them to create a single, unique identifier.

3.6. Authentication of Manifests

At the top level, manifests are authenticated using either the COSE_Mac or COSE_Sign structures, depending on application. The considerations that apply to encryption keys in PayloadInfo apply equally to the use of Mac keys in COSE_Mac.

3.7. Minimum Feature Set

Not all devices will support the full feature set described in this specification. If features become complex enough, it may be necessary to report the features used by a manifest. Since this is redundant information, it is excluded from this draft.

At minimum, targets MUST support the following manifest fields:

1. manifestVersion, so that the target can tell which version of manifest is in use.

2. text (this is ignored by the target)

3. nonce (this is ignored by the target)

4. timestamp

5. conditions

6. payloadInfo

A target attempting to parse a manifest that contains non-nil fields that it does not support SHALL report an error in validation.

At minimum, targets MUST support the ONE of the following payload modes:

1. digestAlgorithm, digests, bstr

2. digestAlgorithm, digests, uris, nil payload

3. COSE_Encrypt, inline ciphertext

4. COSE_Encrypt, nil ciphertext

4. Manifest CDDL Specification

The following CDDL code describes the entire manifest format.

```
condition = [
    type : int,
    parameters : bstr
]
directive = condition

ResourceReference = [
    uri : tstr,
    digest : bstr
]

PayloadInfo = [
    format = [
        type: int,
        ? parameters : bstr
    ],
    size: uint,
    storageIdentifier: bstr,
    uris: [*[
        rank: int,
        uri: tstr
    ]] / nil,
    digestAlgorithm = [
        type : int,
        ? parameters: bstr
    ] / nil,
    digests = {* int => bstr} / nil,
    payload = COSE_Encrypt / bstr / nil
]

Manifest = [
    manifestVersion : uint,
    text : {* int => tstr } / nil,
    nonce : bstr,
    timestamp : uint,
    conditions: [ * condition ],
    directives: [ * directive ] / nil,
    aliases: [ * ResourceReference ] / nil,
    dependencies: [ * ResourceReference ] / nil,
    extensions: { * int => bstr } / nil,
    payloadInfo: ? payloadInfo
]
```

The manifest itself is encapsulated in either a COSE_Mac or a COSE_Sign block.

5. IANA Considerations

Editor's Note: A few registries would be good to allow easier allocation of new features.

6. Security Considerations

This document is about a manifest format describing and protecting firmware images and as such it is part of a larger solution for offering a standardized way of delivering firmware updates to IoT devices. A more detailed discussion about security can be found in the architecture document [Architecture].

7. Mailing List Information

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

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

8. Acknowledgements

We would like the following persons for their support in designing this mechanism

- Geraint Luff

- Amyas Phillips

- Dan Ros

- Thomas Eichinger

- Michael Richardson

- Emmanuel Baccelli

- Ned Smith

- David Brown

- Jim Schaad

- Carsten Bormann

- Cullen Jennings

- Olaf Bergmann

- Suhas Nandakumar

- Phillip Hallam-Baker

We would also like to thank the WG chairs, Russ Housley, David Waltermire, Dave Thaler and the responsible security area director, Kathleen Moriarty, for their support.

9. References

9.1. 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>.

9.2. Informative References

[Architecture] Tschofenig, H., "A Firmware Update Architecture for Internet of Things Devices", January 2018.

9.3. URIs

[1] mailto:suit@ietf.org

Authors' Addresses

Brendan Moran
Arm Limited

EMail: Brendan.Moran@arm.com

Milosch Meriac
Arm Limited

EMail: Milosch.Meriac@arm.com

Hannes Tschofenig
Arm Limited

EMail: hannes.tschofenig@gmx.net