opsawg
Internet Draft
Intended status: Standards track
Expires: March 17, 2018

Lucas
Cisco International Limited

September 13, 2017

aSSURE Data Security
draft-lucas-assure-data-security-00.txt

Abstract

aSSURE uses industry standards and best practice to provide a secure communications platform for device configuration and life cycle management across the entire range of smart devices, from the largest servers through to more constrained devices, with minimal human involvement. Based on extensions to current standard methods, aSSURE also provides secure end to end communication across any network type.

A new approach allows key distribution and encrypted channels to be established between devices that support RSA, EC and/or simple shared secrets. For devices that only support shared secrets, key derivation algorithms ensure that forward and backward compatible secrecy is supported so that secure change of ownership can be obtained. Owners prove ownership via a "case ID" known by the manufacturer and the "Trusted Authority" ID Server but not known by the device.

aSSURE defines end-to-end encryption links, called "channels", so that pairs of devices communicate with a unique set of encryption keys. These unique keys, coupled with the end-to-end encryption, mean communication is both secure and private.

DTLS supports both certificates and pre-shared keys, but does not cover key distribution or management. DTLS does not support client-specific pre-shared keys because the client cannot identify itself during the handshake. Herein are all the APIs required to support key distribution and management as well as an extension to the DTLS handshake that allows the client identity to be provided.

aSSURE cleanly integrates with the Open Interconnect Consortium (OIC) architecture. Both use CBOR encoded data with CoAP over UDP and DTLS. aSSURE URIs do not collide with OIC URIs and aSSURE channels can be used as a secure transport for OIC requests.

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-

Drafts is at http://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on March 17, 2018.

Copyright Notice

Table of Contents

Table of Figures

Glossary of Terms

API	Application Programming Interface
CA	Certificate Authority
CBOR	Concise Binary Object Representation, RFC-7049
CoAP	Constrained Application Protocol, RFC-7252
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DTLS	Datagram Transport Layer Security (v1.2), RFC-6347
EC	Elliptic Curve
ECDSA	E C Digital Signature Algorithm, NIST FIPS 186-4
ECIES	Elliptic Curve Integrated Encryption Scheme, ANSI X9.63
FQDN	Fully Qualified Domain Name
PKCS	Public Key Cryptography Service
PKI	Public Key Infrastructure
TA	Trusted Authority
TLS	Transport Layer Security (v1.2), RFC-5246

1. Introduction

This document provides the reference technical specification for aSSURE.

aSSURE uses industry standards and best practice to provide a secure communications platform for device configuration and life cycle management. Where possible, a minimal approach is taken to standards implementation so that the complexity and code footprint for implementation is kept to a minimum.

The underlying standards are:

- Transport Layer Security, TLS v1.2, RFC-5246
- Datagram Transport Layer Security, DTLS v1.2, RFC-6347
- Constrained Application Framework, CoAP, RFC-7252
- Concise Binary Object Representation, CBOR, RFC-7049
- CoAP Block-wise Transfers, https://www.ietf.org/id/draft-ietf-core-block-21.txt

The additional functionality provided by aSSURE is intended to work within existing communications frameworks. This allows aSSURE to provide an upgrade path to add a common security approach that provides both secure communications and lifecycle management including change of ownership. aSSURE uses a "Trusted Authority" (TA), similar to the role that a Certificate Authority (CA) plays in a Public Key Infrastructure (PKI) today, to track the manufacture and ownership of devices. Any number of Trusted Authorities may exist but each device will be assigned to a specific TA during its manufacture and will remain assigned to this TA for its entire life.

Device owners communicate with the various Trusted Authorities to assert ownership of individual devices and upload the initial

configuration for the device. When the device powers up, it will contact the Trusted Authority to obtain its initial configuration - this process is called "bootstrap". The initial configuration will provide sufficient information for the device to establish a secure communications channel to the system that will be managing it. Once this channel is established, additional configuration will be provided from the management system directly to the device and the device can enter "normal service".

The detailed device lifecycle flow is described elsewhere.

Note
aSSURE is designed to cleanly integrate with the Open Interconnect Consortium (OIC) architecture. Both OIC and aSSURE use CBOR encoded data with CoAP over UDP and DTLS. aSSURE URIs have been deliberately chosen not to collide with OIC URIs and aSSURE channels can be used as a secure transport for OIC requests.

2. The role of aSSURE in an IoT environment

2.1. Background

In any secure environment, there are five basic questions that any device must ask:

1. Who am I allowed to talk to?
2. How can I authenticate them?
3. What am I allowed to tell them?
4. What are they allowed to tell me?
5. How can I ensure that our communication is private?

If these basic questions can all be answered with confidence, there is the foundation for a secure system. If any of the above are uncertain then the system has weaknesses that may be exploited by an attacker.

The aSSURE standard provides an answer to all these questions in a way that allows devices to communicate across different network architectures and device capabilities yet still providing end-to end security at a level that is appropriate to the abilities of the devices that are communicating.

Furthermore, aSSURE provides this with a solution that involves minimal human involvement.

The following sections will address each of these questions in turn.

2.2. Who am I allowed to talk to?

In many ways, this is one of the biggest hurdles to overcome. If we want to be able to manufacture and sell "generic" product that

has no pre-configuration, how does that device know that we own it? There are a lot of different approaches to this with "Trusted On First Use" (TOFU) being an obvious one, but with all of them they either have weaknesses in the initial security or rely on public key cryptography.

Public key cryptography is fine in more powerful devices, but not an option in the smallest ones, so for a universally secure solution, a different approach is required.

The aSSURE standard uses a Trusted Authority (TA) as the reference for the device. The device is programmed with the identity and credentials of the TA during manufacture and, on first power up, will *only* talk to the TA. The user will register ownership of the device with the TA and securely upload the initial configuration data for the device to the TA. The TA will then forward that configuration to the device. That configuration includes the location and security parameters for the device to connect to the owner's systems, so now the device knows that it can trust its owner.

Once the device has connected to the owner's management system, this system can deliver additional configuration parameters, encryption keys, etc. to the device. This allows the management system to tell devices to set up secure peer-to-peer connections, connect to additional management systems and perform other actions.

2.3. How can I authenticate them?

The same sequence as for 2.2. above is used to provide the authentication details to the device. This information allows the device to authenticate the owner's systems and allows the owner's systems to authenticate the device.

2.4. What am I allowed to tell them?

The same sequence as for 2.2. above is used to provide the access control rules for access to the device data. This allows the device to know what information it can disclose.

2.5. What are they allowed to tell me?

The same sequence as for 2.2. above is used to provide the access control rules for commands and configuration sent to the device. This allows the device to know what parameters and commands it will accept from the owner's systems.

2.6. How can I ensure that our communication is private?

The aSSURE standard defines end-to-end encryption links, called "channels", that ensure each pair of devices communicate with a

unique set of encryption keys. These unique keys, coupled with the end-to-end encryption, means that their communication is both secure and private.

3. Terminology

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

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding The portions of this RFC covered by these keywords.

4. The role of the Management System in aSSURE

4.1. Overview

The Management System is a key part in the trust relationship that the device creates. The root of trust is the Trusted Authority. The Trusted Authority tells the device which management systems(s) it can trust. The Management Systems tell the device which other management systems and devices it can trust (if any) and what their permissions are on the device.

4.2. Creation of Communication Topologies

The Management System can instruct the aSSURE devices to form any topology that is within their capabilities. The limits on the topology types and complexity are only:

- Limitations set by the underlying network architecture
- Limitations set by the device memory and/or processing power
- and/or software
- Limitations set by the management software

In aSSURE terminology, each connection between devices is called a *channel*. The rules about how channel keys are determined and assigned is described in detail in section 13. below.

4.3. Examples of communication topologies

4.3.1. A "star" topology

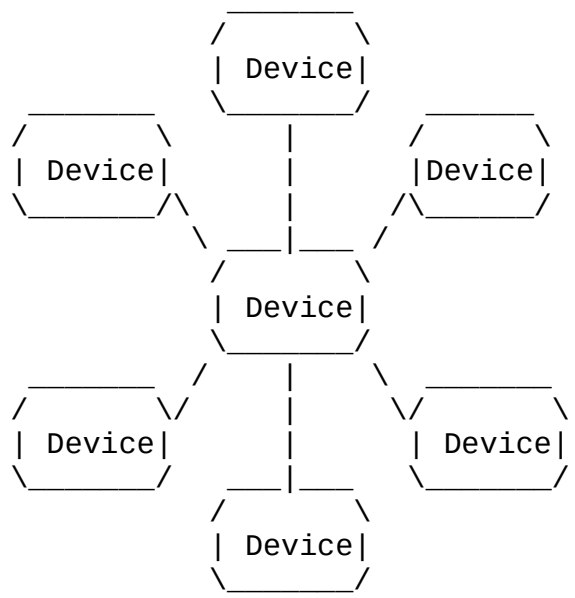


Figure 1 Star Topology

4.3.2. A "ring" topology

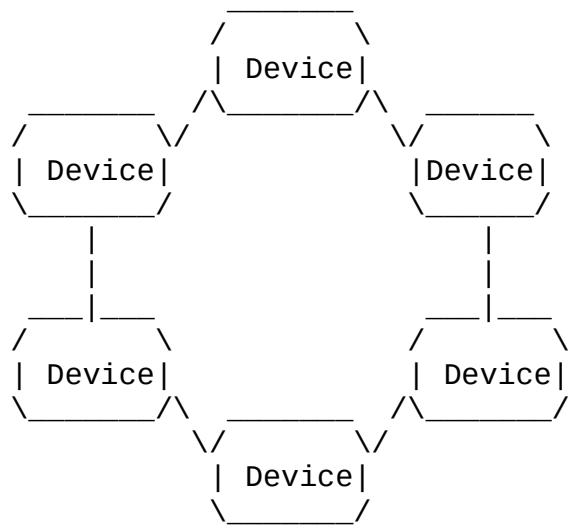


Figure 2 Ring Topology

4.3.3. A "tree" topology

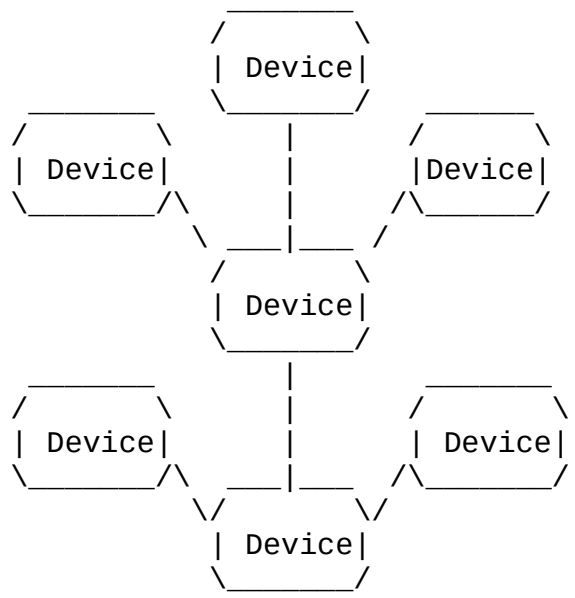


Figure 3 Tree Topology

4.3.4. A "fully connected" topology

The fully connected topology shows four devices where each device has a connector to all of the other three devices. If there are "n" devices they each have "n-1" connectors.

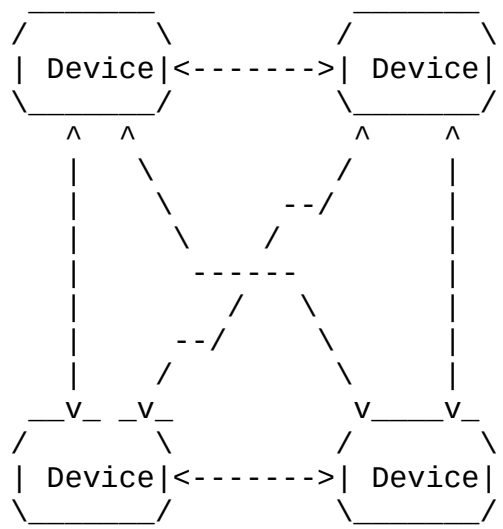


Figure 4 Fully Connected Topology

5. aSSURE Architecture

5.1. Internet Accessible Deployments

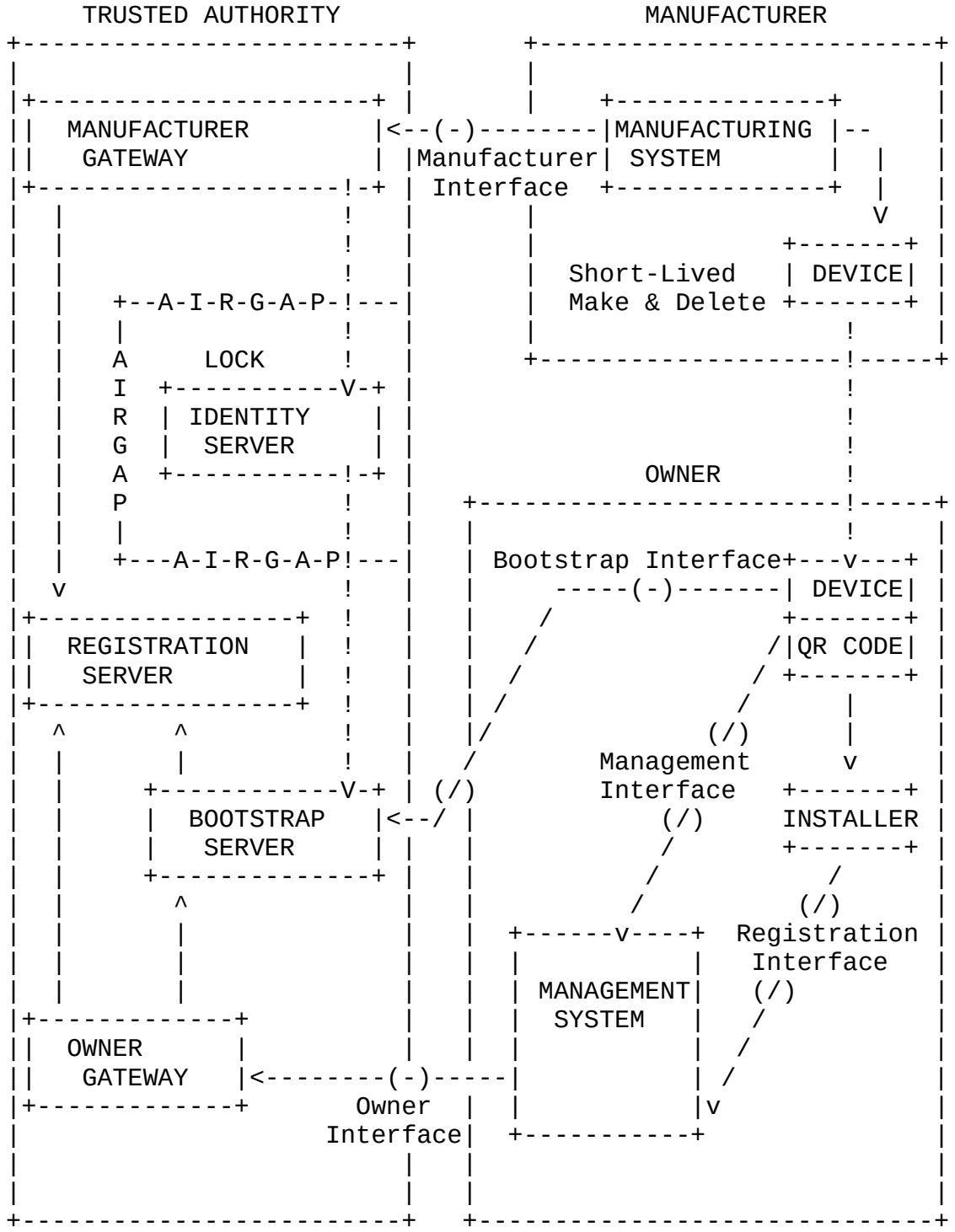


Figure 5 Internet-accessible architecture

5.2. Walled Garden Deployments

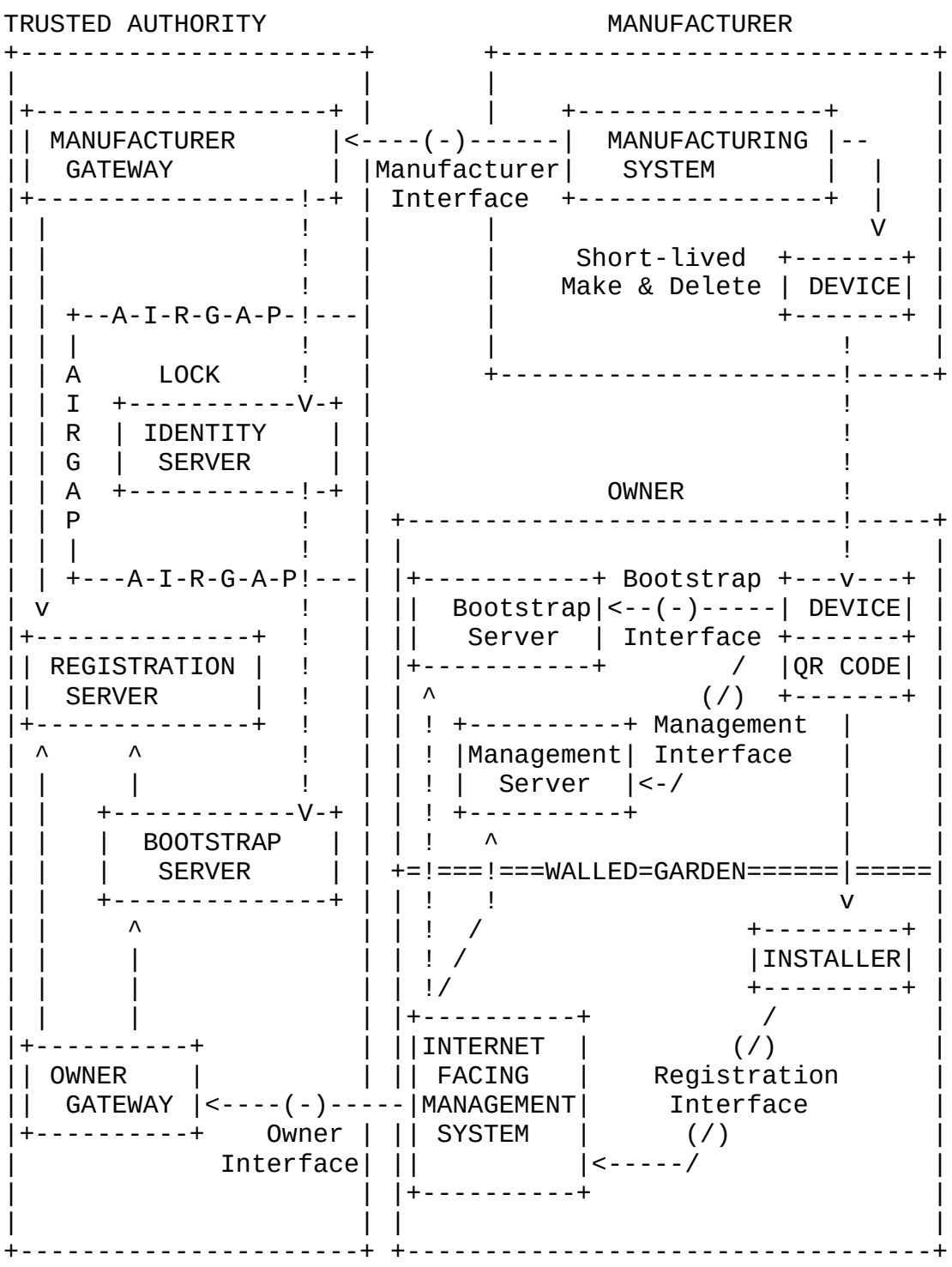


Figure 6 Walled-garden architecture

6. Security Considerations

6.1. Overview

The aSSURE framework is intended to be usable across the entire range of smart devices - from the largest servers through to more constrained devices (as defined in RFC-7228). This is a very challenging goal and means that some security approaches in common use today are not universally suitable.

Examples of approaches that are not universally suitable include:

- Public Key Cryptography, e.g. RSA, DSA, EC
  - This is computationally intensive and may take too long to be acceptable on devices with minimal processing ability.
  - This is computationally intensive and the necessary additional processing load may have an unacceptable impact on battery life.
  - This requires reasonably large code size to implement in software and this may not be available on the more constrained devices.
- X.509 certificates
  - These use time stamps and small devices may not have a real time clock.
  - These assume public key cryptography (RSA, DSA or EC) and constrained devices may not be able to support this as explained above.
  - These have a fixed lifetime, thus requiring them to be reissued before they expire.
  - These require a certificate authority to issue (and reissue) them.
  - Due to their complexity, these have been the target of various attacks in the past, so removing them reduces the attack surface.
- Complex text-based data representations such as ASN.1, HTML, XML, YAML or JSON
  - These are difficult to parse, requiring larger code libraries and more processing power than simpler formats such as CBOR.
  - Due to their complexity, these have been the target of various attacks in the past, so removing them reduces the attack surface.
- Complex protocols such as SOAP, HTML, etc.
  - Again, these are more difficult to parse, requiring larger code libraries and more processing power than simpler protocols such as CoAP.
- Full standards implementation
  - Lots of industry standards are large and have a lot of different options, most of which are unnecessary for a new implementation with no legacy support requirements.
  - For example, TLS supports over 200 different cipher suites and this list continues to grow.

- Due to their complexity, these have been the target of various attacks in the past, so reducing the scope of the implementation also reduces the attack surface.

Support for shared secrets

Very constrained devices that cannot support public key cryptography have to fall back on "shared secrets" (also known as "pre-shared keys") to identify themselves. Typical approaches using shared secrets require the secret to be disclosed to both parties to set up the secure connection. Once a secret has been disclosed, it can never be proved to have been forgotten. This makes transfer of device ownership problematic, as the previous owner of the device may still know the shared secret even after the device has been transferred to a new owner. In this scenario, the previous owner would be able to decrypt all traffic between the device and its new owner, making the device untrustworthy to the new owner.

aSSURE offers a new approach using shared secrets that allows the original secret to be concealed from the peer involved in the connection.

In the aSSURE scenario, a shared secret can be distributed to a device (or multiple devices) with one or two derivation functions. These derivation functions allow a device with the correct reference key to derive the shared secret. Different derivation functions exist to derive the shared secret from an Elliptic Curve key, an RSA key or another shared secret.

This ability to derive shared secrets is used to secure the ownership lifecycle for devices that do not support public key cryptography. With this approach, the device owner is provided with the parameters for the hashing function and the derived shared secret but not the original shared secret programmed into the device during manufacture. When the owner wishes to communicate with the device, the owner provides the device with the derived secret parameters, thus allowing the device to derive the new secret from the original secret. The owner is provided with a second derivation function that uses the owner key, so they can also derive the same secret and hence establish a secure communication link to the device. When the device changes owner, the new owner is given a different derived secret, not the original secret. The new owner can communicate with the device using the new parameters and the knowledge of the new derived secret. The previous owner, however, does not have the new owner's key nor the device key, so cannot derive the new shared secret and so is unable to decrypt the communications with the new owner.

The only entities that know the original device secret are the device itself and the Identity server within the Trusted Authority (this is described in more detail later).

Shared secrets are described more in 7.2. and 8.4. below.

6.2. Guidelines for manufacturers

6.2.1. Device UUID

All device UUIDs should be truly randomly generated. This means that they are completely unpredictable and knowledge of the UUID does not disclose any information about what the device is, who manufactured it or when.

6.2.2. Device Asymmetric Key

If a device uses an RSA or EC asymmetric key, this should be securely generated within the device and the private key must never be disclosed outside of the device. The device should have access to a suitable entropy source to ensure that the key is truly randomly generated.

6.2.3. Device Shared Secret

If a device uses a shared secret, this should be truly randomly generated and at least 128 bits in size. It may be generated inside the device or on the local manufacturing station but, if generated outside the device, must never be stored in an unencrypted form and must be wiped from RAM as soon as it is no longer needed. The device secret can only ever be stored in non volatile storage AFTER it has been formed into the device identity structure and encrypted as described in 9.2.1. below.

6.2.4. Case ID

The aSSURE solution needs a method for an owner to prove that a device is in their possession. This is done through knowledge of a "case ID", an identification string that is printed on the outside of the case. This number is NOT known to the device - it is only known to the manufacturer and the Trusted Authority's Identity Server. The case ID should be a randomly generated number that is at least 128 bits.

6.2.5. QR Code

The device ID and case ID should be printed in a QR Code on the side of the device. The data should be encoded in CBOR as follows:

```
ARRAY {
  INTEGER version    // Set to 1 for aSSURE v1
  BYTE STRING device_uuid
  BYTE STRING case_id
}
```

If the device UUID and case UUID are both 128 bits then this will encode in 36 bytes. This can be encoded in a type 3 QR code (27 x 27 pixels) with "M" level of error correction, a type 4 QR code (33 x 33 pixels) with a "Q" level of error or a type 5 QR code (37 x 37 pixels) with "H" level of error correction.

The manufacturer may choose any of these QR formats, but the type 5 is recommended as it has a higher level of error correction and is therefore less susceptible to damage.

7. Data Structures

7.1. Overview

Where possible, common data structures are used across the system. This simplifies the development and allows better code reuse.

7.2. Key Definition

A key is defined using one of the following CBOR formats depending on its type. All keys are identified by their ID, which is a 128 bit randomly assigned UUID.

Keys have a format which may be Elliptic Curve, RSA or derived secret. Elliptic Curve and RSA keys must include the public key part and may optionally also include the associated private key.

If the key is an Elliptic Curve key, the definition is:

```
ARRAY {
  INTEGER       content  // "Key Content Type", see 7.5. below
  INTEGER       format  // "EC key", see 7.6.  below
  BYTE STRING   key_id  // UUID
  BYTE STRING   public_key  // ASN.1 DER encoded string
  BYTE STRING   private_key  // ASN.1 DER encoded string
}
```

The `private_key` BYTE STRING must be zero length if only a public key is provided.

Note that the private key definition is not encrypted so if the private key is provided, the definition must be transferred over an encrypted channel and stored in a protected store.

If the key is an RSA key, the definition is:

```
ARRAY {
  INTEGER      content  // "Key Content Type", see 7.5. below
  INTEGER      format   // "RSA key", see 7.6.  below
  BYTE STRING  key_id   // UUID
  BYTE STRING  public_key  // ASN.1 DER encoded string
  BYTE STRING  private_key  // ASN.1 DER encoded string
```

```
}
```

The `private_key` BYTE STRING must be zero length if only a public key is provided.

Note that the private key definition is not encrypted so if the private key is provided, the definition must be transferred over an encrypted channel and stored in a protected store.

If the key is a derived secret, the definition is:

```
ARRAY {
  INTEGER         content  // "Key Content Type", see 7.5. below
  INTEGER         format   // "Derived shared secret", see 7.6.
below
                           // below
  BYTE STRING     key_id   // UUID
  ARRAY {
    // Derivation definition
    BYTE STRING     reference_A_key // UUID
    BYTE STRING     salt
    INTEGER         iterations_or_cipher
    BYTE_STRING     encrypted_secret
  }
  // Additional derivation definitions may be present in
  // the same format as above
}
```

The derived secret has one or more derivation definitions. All derivations can be disclosed publicly and all derivations must produce the same secret. A device may use any of the derivations for which it already knows the reference key.

If the reference key is a shared secret then the PBKDF2 algorithm is used with SHA2 as the digest function, the reference key shared secret used as the passphrase, the "iterations_or_cipher" used as the iteration count, salt and a length value determined by the length of the "encrypted_secret". After the PBKDF2 is completed, the result is XOR'd against the "encrypted_secret". This allows secure generation of any byte sequence.

If the reference key is an elliptic curve key, then the "salt" and "iterations_or_cipher" fields are ignored (and should be zero length and value zero respectively). The "encrypted_secret" contains the shared secret protected by the Cryptographic Message Syntax with "enveloped-data" as the ContentInfo (see RFC 5652 and RFC 5753) using the "Standard" variation of Ephemeral Static ECDH (see RFC 5753 section 3.1). The default choices for encryption cipher and hash function should be AES-128 and SHA-256 respectively.

If the reference key is an RSA key, then the "salt" and "iterations_or_cipher" fields are ignored (and should be zero

length and value zero respectively). The "encrypted_secret" contains the shared secret protected by the Cryptographic Message Syntax with "enveloped-data" as the ContentInfo (see RFC 5652) using RSAES-OAEP (see RFC 8017 section 7.1). The default choices for encryption cipher and hash function should be AES-128 and SHA 256 respectively. The SHA-1 hash should **not** be used.

7.3. Signature Definition

Rather than use X.509 signatures, which require public key cryptography and ASN.1 encoding, aSSURE uses a simpler approach using CBOR that can also be used with derived secrets as well as public keys.

If the signature is generated by an Elliptic Curve private key, the signature structure is:

```
ARRAY {
    INTEGER     content // "Signature Content Type", see
                        // 7.5. below
    INTEGER     format // Signature format, see 7.9. below
    INTEGER     created_at
    INTEGER     valid_until
    BYTE STRING key_id
    BYTE STRING r
    BYTE STRING s
}
```

Here, "r" and "s" are the signature values as defined in the Elliptic Curve Digital Signature Algorithm (ECDSA).

If the signature is generated by an RSA private key, the signature structure is:

```
ARRAY {
    INTEGER     content // "Signature Content Type", see
                        // 7.5.  below
    INTEGER     format  // Signature format, see 7.9. below
    INTEGER     created_at
    INTEGER     valid_until
    BYTE STRING key_id
    BYTE STRING s
}
```

Here, "s" are the signature values as defined in the RSA Digital Signature Algorithm (PKCS #1 v1.5).

If the signature is generated by a derived secret, the signature structure is:

```
ARRAY {
    INTEGER     content // "Signature Content Type", see
                        // 7.5. below
```

```
        INTEGER     format  // Signature format, see 7.9. below
        INTEGER     created_at
        INTEGER     valid_until
        BYTE STRING key_id
        BYTE STRING hmac
    }
```

In all cases, the signature covers the original data structure and the signature structure with the "r", "s" and/or "hmac" fields being treated as zero length when performing the sign or validation actions (i.e. the BYTE STRING definition is considered to be the value 0b010_00000).

Two timestamps are part of each signature:

- The timestamp of when the signature was created ("created_at")
- The timestamp after which the signature invalid ("valid_until")
-

Each signing authority is free to choose its own validity duration for the signatures that it issues. The authority can therefore balance the rate of re-issue of signatures against the time that a signature or a compromised key would remain valid. These fields can be set to zero to indicate a signature that is always valid and never expires. These fields may also be zero if the signature authority knows that the signature is being issued to a device with no real-time clock capability. If a device has no knowledge of the true time, these fields should be ignored when performing signature validation.

7.4. Authenticated Key Definition

Rather than use X.509 certificates which require public key cryptography, ASN.1 encoding and knowledge of real time, aSSURE uses a simpler yet more flexible structure to authenticate a public key or a key derived from a shared secret.

An authenticated key definition combines a header, key definition as in 7.2. above, optional metadata and signature as in 7.3. above.

```
ARRAY {
    INTEGER content  // "Identity Content Type", see 7.5. below
    INTEGER class    // Identity class, see 7.7.  below
    ARRAY {
        // Key definition, see 7.2.  above
        INTEGER content // "Key Content Type", see 7.5. below
    }
    MAP {
        // Metadata (key + value pairs), see 7.10.  below
    }
    ARRAY {
        // Signature definition, see 7.3.  above
```

```
          INTEGER content  // "Signature Content Type", see
                           // 7.5.  below
      }
   }
```

The device should validate an "authenticated key" as follows:

1. Check that the key used to generate its signature is already present in the device
2. Check that current time is within the timestamp range for the signature
3. For each key in the hierarchy of keys required to validate this signature, check that current time is within the timestamp range for that key's signature
4. Check that the class of the key used to generate the signature is allowed to sign the class of the new authenticated key. The rules for this are in 7.7. below.
5. Check that the signature matches the authenticated key data

If all the above checks pass, the authenticated key can be accepted.

7.5. Content Type IDs

Value	Meaning
0	Identity Content Type
1	Key Content Type
2	Configuration Content Type
3	Encrypted Key Content Type
4	Signature Content Type
5	Owner Content Type

Value	Meaning
0	Identity Content Type
1	Key Content Type
2	Configuration Content Type
3	Encrypted Key Content Type
4	Signature Content Type
5	Owner Content Type

7.6. Key Format IDs

Value	Meaning
0	Elliptic Curve Key (ASN1.DER encoded following industry standards)
1	RSA Key (ASN1.DER encoded following industry standards)
2	Configuration Content Type

7.7. Identity Class IDs

Valid	Name	Identity Classes: Manufacturer	Trusted Authority	Management System	Bootstrap Service	Device	Channel	Signing Permissions: Manufacturer Device	Trusted Authority	Bootstrap
0	Manufacturer	Y	Y	N	N	Y	N	N	N	N
1	Trusted Authority	N	Y	Y	Y	Y	N	N	N	N
2	Management System	N	N	Y	Y	Y	Y	Y	Y	Y
3	Bootstrap Service	N	N	N	N	N	N	N	N	N
4	Device	N	N	N	N	N	Y	N	N	N
5	Channel	N	N	N	N	N	N	N	N	N

The *Manufacturer* is the manufacturer of the device. A manufacturer

may install *Device* identification keys at the factory to allow the device to authenticate itself. If the device is public key
>> capable, the manufacturer MUST install the identity for the *Trusted Authority*.

The *Trusted Authority* refers to all systems running within the *Trusted Authority*. A *Trusted Authority* is only allowed to authenticate other systems running within the *Trusted Authority*, the *Bootstrap* Service, the owner's *Management Systems* or the *Device*. It is not allowed to authenticate *Bootstrap* data or device communications *Channels*.

The *Management System* is not allowed to authenticate *Manufacturers* or *Trusted Authorities* but it can authenticate all other identities used by the device.

The *Bootstrap Service* is not allowed to authenticate anything else - it is only able to deliver bootstrap data to the device.

The *Device* is only allowed to authenticate channels.

The *Channel* is not allowed to authenticate anything else - it is only allowed to transport data.

7.8. Cipher Suite IDs

Value	Meaning
0	"AES-128 CBC" (OID 2.16.840.1.101.3.4.2)"
1	"AES-192 CBC" (OID 2.16.840.1.101.3.4.22)"
2	"AES-256 CBC" (OID 2.16.840.1.101.3.4.42)"
3	"AES-128 CCM" (OID 2.16.840.1.101.3.4.7)"
4	"AES-192 CCM" (OID 2.16.840.1.101.3.4.27)"
5	"AES-256 CCM" (OID 2.16.840.1.101.3.4.47)"
6	"AES-128 GCM" (OID 2.16.840.1.101.3.4.6)"
7	"AES-192 GCM" (OID 2.16.840.1.101.3.4.26)"
8	"AES-256 GCM" (OID 2.16.840.1.101.3.4.46)"

7.9. Signature Format IDs

Value	Meaning
0	ecdsa-with-SHA256 (OID 1.2.840.10045.4.3.2)
1	ecdsa-with-SHA384 (OID 1.2.840.10045.4.3.3)
2	ecdsa-with-SHA512 (OID 1.2.840.10045.4.3.4)
3-7	Reserved..future expansion of ECDSA signatures

8	sha256-with-rsa-signature (OID 1.2.840.113549.1.1.11)	
9	sha384-with-rsa-signature (OID 1.2.840.113549.1.1.12)	
10	sha512-with-rsa-signature (OID 1.2.840.113549.1.1.13)	
11-15	Reserved..future expansion of RSA signatures	
16	hmacWithSHA256 (OID 1.2.840.113549.2.9)	
17	hmacWithSHA384 (OID 1.2.840.113549.2.10)	
18	hmacWithSHA512 (OID 1.2.840.113549.2.11)	

7.10. Authenticated Key Metadata

Key	Value	Usage
INTEGER(0)	BYTE STRING (<device uuid>)	Indicates the device owning the key

7.11. aSSURE timestamps

7.11.1. Simple timestamps

aSSURE uses a variant of the Unix time format for all its timestamps. An aSSURE simple timestamp is an integer that tracks the number of seconds since midnight on Friday January 1st 2010 GMT rather than midnight on January 1st 1970 GMT. This allows a signed 32-bit number to provide a valid timestamp until 2078 rather than 2038.

```
aSSURE_timestamp_secs = unix_timestamp_secs - 1262304000
```

A simple timestamp is defined in CBOR as:

```
INTEGER timestamp_secs
```

7.11.2. Precision timestamps

If more precision is required, a fractional part may also be provided. This holds the fractional part of the second as a 32-bit value (so the precision is ~233ps).

A precision timestamp is defined in CBOR as:

```
ARRAY {
  INTEGER timestamp_secs
  INTEGER timestamp_frac
}
```

8. DTLS with aSSURE key identities

8.1. Overview

aSSURE uses DTLS for all secure connections due to its low overhead both in code and operation. DTLS supports both certificates and pre-shared keys, but does not cover how the certificate authorities or pre-shared keys are to be securely distributed.

This section shows how we extend the basic DTLS standard with additional RFC to provide the functionality required for aSSURE. It also shows how to setup DTLS connections in an aSSURE environment.

8.2. Extension to (D)TLS

aSSURE needs to be able to provide the client identity to the server during the DTLS handshake. This would normally be done using X.509 certificates, but aSSURE avoids the complexity and overhead of X.509 certificates so an alternative approach is required.
aSSURE provides the client identity to the server using a new TLS extension, "Peer Name Indication" that is lightweight and similar to the "Server Name Indication" extension.

8.2.1. Peer Name Indication

TLS does not provide a mechanism for a client to tell a server the name of the client that is connecting before the ServerHello is returned. It may be desirable for clients to provide this information to facilitate secure connections to servers where the ServerHello should vary according to the client identity.

In order to provide any of the names, clients MAY include an extension of type "peer_name" in the (extended) client hello. The "extension_data" field of this extension SHALL contain "PeerNameList" where:

```
struct {
    NameType name_type;
    select (name_type) {
        case uuid: UUID;
    } name;
} PeerName;
```

```
enum {
    uuid(0), (255)
} NameType;

opaque UUID[16];

struct {
    PeerName peer_name_list<1..2^16-1>
} PeerNameList;
```

This allows the client to provide its UUID in a compact format to the server. It also allows other client formats to be used in the future.

8.3. Proof of identity by public key clients

A typical DTLS handshake uses X.509 certificates to allow both mutual authentication of identity and key exchange to set up the secure connection. aSSURE does not use X.509 certificates for the reasons explained in section 6.1. above so an alternative approach is required to allow mutual authentication and key exchange.

aSSURE uses the device API calls to allow the management system to securely provide identity credentials for other peers to the device. An aSSURE device WILL NOT communicate with any peer that has not had the peer identity credentials provided to it by an authorised management system over a secured connection.

aSSURE uses the Peer Name Indication extension (see 8.2.1. above) to allow a device to indicate its ID to the peer during the initial connection request. This allows the peer to check that the device is in its list of trusted devices. If the device is not in the list, the peer refuses the connection.

Similarly, the server uses the Peer Name Indication extension to provide its ID to the client in the initial connection response. This allows the client to quickly check its database to check that the server is in its list of trusted peers. If the server is not in the list, the client aborts the connection attempt.

aSSURE then uses the Raw Public Key Extension defined in RFC-7250 to allow just the public keys to be exchanged between device and peer. Both the device and the peer cross-check the public keys provided in the DTLS handshake against the expected public keys their list of trusted devices. If either device or peer detect a public key mismatch, they abort the handshake.

If all checks complete without error, the handshake is allowed to continue normally and the secure connection is established.

8.4. Proof of identity by shared secret clients

When establishing a secure connection, if either device cannot support public keys, they must use the derived shared secret method of authentication. Derived shared secrets are a weaker form of security than public keys, but if used carefully, can still provide a reasonable level of security.

DTLS connections protected by derived shared secrets differ from public key cryptography in that both parties must know the same secret to allow the DTLS handshake to complete. Hence, a derived shared secret can be used to prove a *link relationship* but not an endpoint identity.

The management system will have been provided with a unique derived shared secret for each device - this is provided to the device in the bootstrap configuration so that the device can trust the *link relationship* from itself to the management system.

The management system will then generate a unique derived shared secret for each link that the device needs to establish using its knowledge of the available keys on the devices at each end of the link. The management system will then push that derived shared secret to both devices indicating the peer device ID to which the derived shared secret relates and instruct one of them to act as the client to establish the channel.

The client will then attempt to connect to the peer using DTLS.

aSSURE uses the Peer Name Indication extension (see 8.2.1. above) to allow a device to indicate its ID to the peer during the initial connection request. This allows the peer to check that the device is in its list of trusted devices. If the device is not in the list, the peer refuses the connection.

Similarly, the server uses the Peer Name Indication extension to provide its ID to the client in the initial connection response. This allows the client to quickly check its database to check that the server is in its list of trusted peers. If the server is not in the list, the client aborts the connection attempt.

aSSURE then uses the Pre-shared Key Identity Hint Extension defined in RFC-4279 to allow the server to provide the necessary key information to the client. The key information is encoded in Base64 because RFC-4279 recommends that the key information only contains printable characters. The key information will be one of:

- The UUID of a key known to both client and server. The UUID is provided as a CBOR BYTE STRING of 16 bytes.
- A key definition as in 7.2. above.

The client can determine which is provided by inspecting the data.

A simple UUID is a CBOR BYTE STRING whilst a key definition is a CBOR ARRAY.

The client checks that the indicated (or reference) key is available and appropriate for this connection, derives the secret as appropriate and uses this as the pre-shared key for the DTLS session. The server also derives the secret from the key and uses this as the pre-shared key for the DTLS session.

The DTLS handshake then continues as normal and the session is established.

9. Trusted Authority APIs

9.1. Overview

The Trusted Authority provides three distinct APIs as follows:

- Manufacturer API
- Owner API
- Bootstrap API

The Manufacturer API is used by the manufacturer to upload manufacturing details about each device when it is created. Only the minimum amount of information about the device is uploaded and this information is stored in One-Time Programmable (OTP) memory on the device, so can never be changed. For this reason, the Manufacturing API has no requirement to allow device information to be updated after manufacture, such as during Return Merchandise Authorisation (RMA), because this information can never be changed.

The Owner API is used by the device owner to assert ownership of the device, update the device bootstrap configuration and change device ownership.

The Bootstrap API is used by the devices themselves to download their bootstrap configuration so that they can connect to their management systems and enter service.

9.2. Manufacturer API

The Manufacturer API is used by the manufacturer to upload data about the device when it is manufactured. The manufacturer API is a RESTful interface using JSON over HTTPS with client authentication. Generation of the client key and issuing of the client certificate is out of scope for this document (this information could easily be exchanged via email). Similarly, delivery of the Trusted Authority "Identity Service" certificate to the manufacturer and disclosure of the Manufacturer API URL is also out of scope (again, this information could easily be provided via email).

9.2.1. PUT /v1/devices/<uuid>

After a device is manufactured, the manufacturer builds the device data into a JSON data structure as follows:

```
{
  "id": "<device UUID>",
  "bootstrap_server": <bootstrap server ID used by device>,
  "case_string": "<string printed on case>",
  "shared_secret": "<Device shared secret as HEX>"
  "public_key": "<X.509 PEM encoded device public key>"
  "parameter_set": "<parameter set UUID>"
  "capabilities": {  // Device bootstrap capabilities
    "ec_capable": <boolean>,
    "rsa_capable": <boolean>,
    "sha384_capable": <boolean>,
    "sha512_capable": <boolean>,
    "aes256_capable": <boolean>,
    // Additional capabilities may be added in the future
  }
}
```

If the device uses an RSA or EC key as its device key, the "shared_secret" will not present. If the device is only shared secret capable then the "public_key" will not be present. The JSON data will NEVER have both "shared_secret" and "public_key" fields.

All devices must support SHA-256, AES-128 and shared secrets. If the device can support other keys, hashing algorithms or ciphers during bootstrap, these should be indicated here.

The manufacturer's production line will have encrypted the device data immediately after the device has been tested. The device data is encrypted using ECIES and the Identity Service public key (which must be a strong Elliptic Curve key). The ECIES configuration uses SHA512 and AES-256. The encrypted data is then provided as a binary payload in the request.

No payload is returned. The response code will be one of:

Response	Description
201 Created	The new device has been created
403 Forbidden	The manufacturer cannot update the device because it already exists
503 Service Unavailable	The device cannot be created at this time

Only a single device is uploaded in each PUT request, but the client may re-use the HTTPS session to send additional requests.

9.2.2. POST /v1/parametersets

This allows a manufacturer to upload a parameter set definition. The format of the parameter set definition is TBD. The parameter set is defined in CBOR and provided as the request payload. The parameter set is provided in the configuration data to the Management System to guide the bootstrap data definition as described in 15.2. below.

The payload will contain the assigned UUID as a simple ASCII string. The response code will be one of:

Response	Description
201 Created	The new parameter set has been created
503 Service Unavailable	The parameter set cannot be created at this time

9.2.3. PUT /v1/parametersets/<uuid>

This allows a manufacturer to replace a parameter set definition. The format of the parameter set definition is TBD. The parameter set is defined in CBOR and provided as the request payload. The parameter set is provided in the configuration data to the Management System to guide the bootstrap data definition as described in 15.2. below.

No payload is returned. The response code will be one of:

Response	Description
201 Created	The new parameter set has been created
403 Forbidden	The manufacturer cannot update the parameter set because it does not exist or belongs to a different manufacturer
503 Service Unavailable	The parameter set cannot be created at this time

Only a single parameter set is uploaded in each POST request, but the client may re-use the HTTPS session to send additional requests.

9.2.4. GET /v1/parametersets/<uuid>

This allows a manufacturer to download a parameter set definition.

The format of the parameter set definition is TBD and is exactly the data as provided in 9.2.2. above.

The returned payload is the parameter set data. The response code will be one of:

Response	Description
200 OK	The parameter set has been returned
403 Forbidden	The parameter set does not exist or was provided by a different manufacturer

9.3. Owner API

The Owner API is used to take ownership of a device, set the bootstrap data for the device and transfer ownership of the device.

As with the Manufacturer API, the Owner API is a RESTful interface using JSON over HTTPS with client authentication.

9.3.1. POST /v1/managementsystems

This is used by management systems to register with the Owner API. The connection does not require client authentication.

Before making this request, the management system should generate a new Elliptic Curve key and associated X.509 certificate signing request (CSR). The management system provides the CSR in DER format as the request payload. The Owner API will immediately issue a certificate for this CSR and return it as the response payload in DER format.

All management systems must support both Elliptic Curve and RSA public keys, SHA-256, SHA-512, AES-128 and AES-256 so that they can correctly interoperate with all devices irrespective of the device abilities (or lack of abilities).

The response code will be one of:

Response	Description
201 Created	The new management system ID has been created
503 Service Unavailable	The management system ID cannot be created at this time

Note: Some Trusted Authorities may require a username and password to be provided to authenticate this request to prevent attackers trying to overload the system with requests. Alternative validation approaches are also possible. Such extensions are out of scope of this document.

Note: The serial number of the issued certificate is used as the "manufacturer ID" in owner management API requests (9.3.4. below and 9.3.5. below). For security reasons, the manufacturer ID should not be predictable so ought to be a large random number (>= 64 bits).

9.3.2. PUT /v1/devices/<uuid>/owner?case_string=<string>

This is used by management systems to take ownership of a device. The connection must be authenticated with the issued client certificate.

The case identification string from the QR code must be provided as the "case_string" parameter. No payload is provided.
No payload is returned and the response code will be one of:

Response	Description
204 Changed	Ownership has been assigned to this management system
403 Forbidden	The device is owned by a different management system, the wrong case string was provided or the device does not exist

9.3.3. PUT /v1/devices/<uuid>/owner?mgmtid=<string>

This is a variant of 9.3.2. above and used by management systems to transfer ownership of a device to a different management system. The connection must be authenticated with the issued client certificate.
The serial number of the certificate issued to the new management system must be provided as the "mgmtid" parameter. No payload is provided.

No payload is returned and the response code will be one of:

Response	Description
204 Changed	Ownership has been assigned to the new owner
403 Forbidden	The device is owned by a different management system, or the device does not exist

9.3.4. PUT /v1/devices/<uuid>/owner?case_string=<string>&mgmtid=<string>

This is a combination of 9.3.2. and 9.3.3. above. It is used by management systems to take ownership of a device and immediately assign it to a different management system. The connection must be authenticated with the issued client certificate.

The case identification string must be provided as the "case_string" parameter and the serial number of the certificate issued to the new management system must be provided as the "mgmtid" parameter. No payload is provided.

No payload is returned and the response code will be one of:

Response	Description
204 Changed	Ownership has been assigned to the new owner
403 Forbidden	The device is owned by a different management system, or the device does not exist

9.3.5. PUT /v1/devices/<uuid>/owner?mgmtid=NULL

This is a variant of 9.3.2. above and used by management systems to release ownership of a device. Once a device has been released from ownership, *any* management system may take ownership of the device if knows the device UUID and case string. The connection must be authenticated with the issued client certificate.

No payload is provided or returned and the response code will be one of:

Response	Description
204 Changed	Ownership has been released
403 Forbidden	The device is owned by a different management system, or the device does not exist

9.3.6. GET /v1/devices/<uuid>/parameterset

This is used by management systems to obtain the parameter set required for the device. The connection must be authenticated with the issued client certificate.

No payload is provided. If the request is successful, the parameter set will be returned in the payload. The format of the parameter set data is TBD.

Note that this returns the same data as 9.2.4. above but references the device not the parameter set UUID itself.

The response code will be one of:

Response	Description
200 OK	The device parameter set has been returned
403 Forbidden	The device is owned by a different management system, or the device does not exist

9.3.7. PUT /v1/devices/<uuid>/bootstrap

This is used by management systems to set the bootstrap data for a device. The connection must be authenticated with the issued client certificate.

The bootstrap data is provided as a binary payload. The format of the device bootstrap data is device dependent and detailed in the device parameter set returned in 9.3.6. above. No payload is returned. The response code will be one of:

Response	Description
204 Changed	The device bootstrap data has been accepted
403 Forbidden	The device is owned by a different management system, or the device does not exist

9.3.8. GET /v1/devices/<uuid>/bootstrap

This is used by management systems to return the bootstrap data for a device. The connection must be authenticated with the issued client certificate.

No payload is provided. If successful, the bootstrap data is returned as a binary payload. The response code will be one of:

Response	Description

200 OK	The device bootstrap data has been returned
403 Forbidden	The device is owned by a different management system, or the device does not exist

9.4. Bootstrap API

The Bootstrap API is used by the device to download its bootstrap data. The Bootstrap API is a RESTful interface using CBOR and CoAP over DTLS with client authentication.

9.4.1. GET /v1/devices/<uuid>/bootstrap

This is used by the device to obtain its bootstrap data. The DTLS connection must be authenticated with the device key as described in section 8. above. The <uuid> in the URI must belong to the device authenticated during the DTLS handshake.

No payload is provided. If successful, the bootstrap data is returned as a binary payload. The response code will be one of:

Response	Description
2.05	The device bootstrap data has been returned
4.03	The device is not allowed to access the requested bootstrap data, or the device does not exist

10. Device Management API

The device API is used by the management system to control the aSSURE functionality in the device. Like the Bootstrap API, the Device API is RESTful using CBOR and CoAP over DTLS. The transport of the DTLS messages will vary depending on the device type and installation - examples of different physical and/or network layers are provided in section 11. below.

All requests on the Device Management API must be made over authenticated DTLS connections, known as "channels". The definition of channels is in 4.2. above.

The Management API is used to define what privileges are assigned to each channel. These privileges are:

Privilege	Description

Key Management	The connection is allowed to create, reconfigure and delete keys on the device.
Channel Management	The connection is allowed to create, reconfigure and delete channel definitions on the device.
System Management	The connection is allowed to instruct the device to perform system level actions such as bootstrap or reboot.

10.1.1. PUT /v1/keys/<uuid>

This request requires "Key Management" privileges on the requesting channel.

This instructs the device to add the indicated key to its key store. The key is provided in the payload as a CBOR object as defined in 7.4. above.

The response code will be one of:

Response	Description
2.01	The key has been created
2.04	The key already exists on the device
4.01	The channel does not have privileges to manage keys
4.13	The device has no space to add more keys

10.1.2. POST /v1/keys/generate?type=<key_type>&persistent=<boolean>

This request requires "Key Management" privileges on the requesting channel.

This instructs the device to create a new key of the indicated type in its key store.

Type	Description
0	RSA 2048 bits
1	Elliptic Curve (NIST P-256)

2	Elliptic Curve (NIST P-384)
3	Elliptic Curve (NIST P-521)

If the persistent flag is not set, the key will only exist in RAM and will be lost when the device next reboots or loses power.

If the creation is successful, the device will create an authenticate key as defined in 7.4. above. The device will add its own ID in the MAP field and create the signature using its device key. The authenticate key is then returned in the response payload.

The response code will be one of:

Response	Description
2.01	The key has been created
4.01	The channel does not have privileges to manage keys
4.13	The device has no space to add more keys
5.01	Unsupported key type

10.1.3. GET /v1/keys/<uuid>

This request requires "Key Management" privileges on the requesting channel.

This instructs the device to return the indicated key in its key store. The key is returned in the payload as a CBOR object as defined in 7.4. above.

The response code will be one of:

Response	Description
2.05	The key information has been returned
4.01	The channel does not have privileges to return keys
4.04	The key does not exist

10.1.4. DELETE /v1/keys/<uuid>

This request requires "Key Management" privileges on the requesting channel.

This instructs the device to delete the indicated key from its key store. The key must not be in use by any channel nor must it be used as a reference key by any other keys. The device will not permit any key defined in the bootstrap data to be deleted.

The response code will be one of:

Response	Description
2.02	The key has been deleted or did not exist
4.01	The channel does not have privileges to manage keys
4.03	The key is in use and cannot be deleted at this time or is not allowed to be deleted

10.1.5. GET /v1/keys

This request requires "Key Management" privileges on the requesting channel.

This instructs the device to list the keys in its key store. The key list is returned as a CBOR array of authenticated keys as defined in 7.4. above. For security reasons, no private keys will be disclosed. Instead, if the device has the private data for the key, the string "PRESENT" will be returned as the "private_key" byte string. If the device does not have the private data for the key, the "private_key" byte string will be zero length.

The response code will be one of:

Response	Description
2.05	The key list has been returned
4.01	The channel does not have privileges to manage keys

10.1.6. PUT /v1/channels

This request requires "Channel Management" privileges on the requesting channel.

This instructs the device to create a new channel. The channel configuration will be provided in the request payload in CBOR format as below:

ARRAY {

```
        BYTE STRING local_key_id
        BYTE STRING peer_key_id   // Zero length if the same as
                                  // local_key_id
        ARRAY {
            BOOLEAN   persistent_across_reboots
            BOOLEAN   open_immediately
            BOOLEAN   channel_management_privilege
            BOOLEAN   system_management_privilege
            BOOLEAN   key_management_privilege
            // Additional configuration flags may follow
        }
        ARRAY {
            INTEGER address_type
            // Address content, structure varies depending on
            // address_type
        }
    }
```

The format of the address content depends on the target device network and physical layer. As support for additional network and physical layers are added, additional address types and associated address content format will be defined. The list of assigned address types is in 12.4. below.

The assigned channel ID will be returned in the response payload as a CBOR integer.

The response code will be one of:

Response	Description
2.01	The channel has been created
4.01	The channel does not have privileges to manage channels
4.13	The device has no space to add more channels

10.1.7. PUT /v1/channels/<id>

As per 10.1.6. above, "PUT /v1/channels" but where the channel ID is explicitly provided. The request and response payload formats are unchanged.

Additional response codes may be:

Response	Description
2.04	The channel already exists

10.1.8. PUT /v1/channels/<channel_id>/open

This request requires "Channel Management" privileges on the requesting channel.

This instructs the device to open the indicated channel. No request payload is provided.

The response code will be one of:

Response	Description
2.04	The device will try to open the channel
4.01	The channel does not have privileges to manage channels
4.04	The channel does not exist

Note that a 2.04 response DOES NOT mean that the channel has been successfully opened. Instead, it means that the device WILL TRY to open the channel. This may take some time and the status can be monitored with the channel status request in 10.1.11. below.

10.1.9. PUT /v1/channels/<channel_id>/close

This request requires "Channel Management" privileges on the requesting channel.

This instructs the device to close the indicated channel. No request payload is provided.

The response code will be one of:

Response	Description
2.04	The device will try to close the channel
4.01	The channel does not have privileges to manage channels
4.04	The channel does not exist

Note that a 2.04 response DOES NOT mean that the channel has been successfully closed. Instead, it means that the device WILL TRY to close the channel. This may take some time and the status can be monitored with the channel status request in 10.1.11. below.

10.1.10. DELETE /v1/channels/<channel_id>

This request requires "Channel Management" privileges on the requesting channel.

This instructs the device to delete the indicated channel, closing it automatically if it is currently open. No request payload is provided. The device will not permit any channel defined in the bootstrap data to be deleted.

The response code will be one of:

Response	Description
2.02	The channel has been deleted or does not exist
2.04	The channel was open so will be cleanly closed then deleted
4.01	The channel does not have privileges to manage channels
4.03	The channel is not allowed to be deleted

Note that after a 2.04 response the client can poll the channel status using the "GET" request as in 10.1.11. below. If the channel is still being closed, the response will be a 2.05 with status = 4. As soon as the close completes, the channel will be deleted and the "GET" request in 10.1.11. below will return 4.04 because the channel no longer exists.

10.1.11. GET /v1/channels/<id>

This request requires "Channel Management" privileges on the requesting channel.

This requests the device to return the configuration and status of the channel. No request payload is provided. The channel configuration and status will be provided in the request payload in CBOR format as below:

```
ARRAY {
    ARRAY {
        // Configuration structure as in 10.1.6.  above
    }
    ARRAY {
        INTEGER  channel_id
        INTEGER  state
        // Additional status flags may follow
    }
}
```

The "status" integer will be one of:

Response	Description
0	The channel is closed
1	The channel is requested to be opened
2	The channel handshake is in progress
3	The channel is open
4	The channel is being closed
10	The channel handshake failed because the peer did not answer
11	The channel handshake failed because the peer provided the wrong key ID
12	The channel handshake failed because the peer failed authentication
13	The channel has experienced some other error

The response code will be one of:

Response	Description
2.05	The channel information has been returned
4.01	The channel does not have privileges to manage channels
4.04	The channel does not exist

10.1.12. GET /v1/channels

This request requires "Channel Management" privileges on the requesting channel.
This requests the device to return the configuration and status of all channels. No request payload is provided. The information is returned as a CBOR array of channel responses as defined in 10.1.11. above.

The response code will be one of:

Response	Description
2.05	The configuration and status for all

		channels has been returned
4.01	The channel does not have privileges to manage channels	

10.1.13. PUT /v1/reboot

This request requires "System Management" privileges on the requesting channel.

This instructs the device to reboot. The device will reboot after returning the response. No request payload is provided and no response payload is returned.

The response code will be one of:

Response	Description
2.04	The device is about to reboot
4.01	The channel does not have privileges to manage the system

10.1.14. PUT /v1/shutdown

This request requires "System Management" privileges on the requesting channel.

This instructs the device to shut down. The device will shut down after returning the response. No request payload is provided and no response payload is returned.

The response code will be one of:

Response	Description
2.04	The device is about to shutdown
4.01	The channel does not have privileges to manage the system

10.1.15. PUT /v1/bootstrap

This request requires "System Management" privileges on the requesting channel.

This instructs the device to perform an aSSURE bootstrap. The device will bootstrap after returning the response. No request payload is provided and no response payload is returned.

The response code will be one of:

Response	Description
2.04	The device is about to bootstrap
4.01	The channel does not have privileges to manage the system

10.1.16. GET /v1/ping

This request requires no privileges on the requesting channel. This is used to check that the device is online and able to respond to requests.

A payload may be provided. The device will respond with the same payload (or as much of the payload as the device can return if it is a constrained device). The response code will be:

Response	Description
2.05	Ping response

10.1.17. GET /v1/info

This request requires no privileges on the requesting channel.

This is used to return basic information about the device. The device will return the CBOR structure below:

```
ARRAY {
    BYTE STRING  device_id
    BYTE STRING  device_mac_address
    TEXT STRING  device_manufacturer
    TEXT STRING  device_product_code
    TEXT STRING  device_serial_number
    TEXT STRING  device_build_date
    TEXT STRING  software_manufacturer
    TEXT STRING  software_product_code
    TEXT STRING  software_version
}
```

The only field that must be present is the device_id. All other fields may be zero length if the device is unable to provide them for any reason.

The response code will be:

Response	Description
2.05	Information returned

11. Management Server API

11.1. Overview

The management server must support the registration and presence APIs. The registration API is used to allow devices to be registered with the management system. The presence API is used by devices after bootstrap to inform the management system of their presence on the network.

11.2. Registration API

The Registration API is used to register the device with the management system when it is installed. The Registration API is a RESTful interface using JSON over HTTPS. The authentication behaviour is determined by the Management Server implementation but either username + passphrase or client certificates are recommended.

11.2.1. POST /v1/devices/<uuid>?case_string=<case_string>

This is used to register the indicated device UUID with the management system. The case string is provided to prove the device is physically present. The UUID and case string would normally come from a QR code or similar attached to the device (use of an RFID is not recommended as the source for obvious security reasons).

No payload is returned. The response code will be one of:

Response	Description
201 Created	The new device has been registered
401 Unauthorised	The device cannot be registered because it already exists
503 Service Unavailable	The device cannot be registered at this time

Only a single device is uploaded in each POST request, but the client may re-use the HTTPS session to send additional requests.

11.2.2. POST /v1/devices/<old_uuid>/replace?uuid=<new_uuid>

This is used to tell the management system that the indicated device with UUID <old_uuid> has been replaced by the device with UUID <new_uuid>. For example, this would occur when a device has failed and a maintenance engineer has replaced it. The management system can then update its database, etc. to allow the new device to "seamlessly replace" the old device (e.g. by applying the old device configuration to the new device). For security reasons, the management system should immediately disable the old device as this call indicates that the old device is no longer in use.

The new device must be registered with the call in 11.2.1. above before performing this call to replace the old device.

No payload is returned. The response code will be one of:

Response	Description
201 Created	The new device has replaced the old one
401	The old device does not exist, the client is not allowed to update the state of the old device, the new device does not exist or the client is not allowed to update the state of the new device.
503 Service Unavailable	The device cannot be replaced at this time

11.2.3. GET /v1/devices/<uuid>/status

This is used to get the status of the device. The registration, configuration, bootstrap and presence of a device may take some time. A registration client may obtain the status of the device here to check if the device has entered service.

If successful, the payload is a JSON structure:

```
{
    status: <value>,
    description: "<helpful_text>"
}
```

11.2.4. GET /v1/devices/<uuid>/info

This is used to get the information for the device. This is only available after the device has sent is presence message to the management system.

If successful, the payload is a JSON structure with the same

information as in 10.1.17. above:

```
{
    "device_id": "<device uuid>",
    "device_mac_address": "<device MAC address>",
    "device_manufacturer": "...",
    "device_product_code": "...",
    "device_serial_number": "...",
    "device_build_date": "...",
    "software_manufacturer": "...",
    "software_product_code": "...",
    "software_version": "..."
}
```

The response code will be one of:

Response	Description
200 OK	The device information has been returned
401 Unauthorised	The client is not authorised to access this device
503 Service Unavailable	Device information cannot be returned at this time

Only a single device response is permitted per GET request, but the client may re-use the HTTPS session to send additional requests.

11.3. Presence API

The Presence API is used by the device to indicate to the management system that its bootstrap process has completed and it is now online in the network. Like the Bootstrap API and Device Management APIs, the Presence API is RESTful using CBOR and CoAP over DTLS.

11.3.1. PUT /v1/devices/<uuid>/info

This is used by the device to confirm it is active and has completed its bootstrap.

The device provides the same CBOR structure as in 10.1.17. above as the request payload.

No payload is returned. The response code will be one of:

Response	Description
2.01 Created	The device presence has been acknowledged

4.01 Unauthorized	The device has attempted to provide information for a different device

11.3.2. PUT /v1/devices/<uuid>/goodbye

This is used by the device to indicate it is deliberately going offline. It will send this to all connected management systems.

The device will provide a single INTEGER in CBOR format to indicate the reason.

Value	Reason
0	Device is shutting down
1	Device has been instructed to reboot by Management System
2	Device has been instructed to bootstrap by Management System
3	Device has been instructed to reboot by local controls (e.g. button)
4	Device has been instructed to bootstrap by local controls (e.g. button)

No payload is returned. The response code will be one of:

Response	Description
2.04 Changed	The device goodbye has been acknowledged
4.01 Unauthorized	The device has attempted to provide information for a different device

11.4. Miscellaneous

11.4.1. GET /v1/timestamp

This is used to get the current time from a trusted source. A device may use this to set its clock without having to include support for other protocols such as NTP. This request may be made by any client with or without authentication.

No payload is provided. The response is a timestamp in CBOR as defined in 7.11. above (both integer and fractional parts).

The response code will be one of:

Response	Description
200 OK	The timestamp has been returned
503 Service Unavailable	Device information cannot be returned at this time

12. Physical / Network Layer Implementations

12.1. BACnet

aSSURE traffic is fully compatible with existing BACnet traffic and is identified as Network Control messages using Message Types 0x80 - 0x82 and Vendor ID 0x_**TBD**_.

Message Types 0x80 through 0x82 are used to identify the aSSURE traffic as belonging to one of three logical groups.

Message Type	Description
0x80	aSSURE Bootstrap
0x81	aSSURE Secure Management Channels
0x82	aSSURE Secure Data Channels

When an address is indicated in a CBOR message, the address format is:

```
ARRAY {
    INTEGER     0        // Address type 1 = BACnet
    INTEGER     net
    BYTE STRING addr      // "len" is the BYTE STRING length
}
```

If the "net" field is a NULL (CBOR Major: 7, Value: 22 => 0xF6) rather than an INTEGER (CBOR Major: 0), this indicates a local network address. When creating the BACnet NPDU, the NPDU destination specifier for that address would not be present.

12.1.1. aSSURE Bootstrap

The aSSURE Bootstrap messages are used to identify the bootstrap gateways on the BACnet network and assign a secure data channel for communication with the bootstrap server. This traffic cannot be secured because it happens BEFORE the bootstrap data has been

received so the device has no keys for communicating with peers on the local network.

All Insecure Control Messages are RESTful using CoAP in the NSDU part of an NPDU frame. They are therefore independent of any specific BACnet LLC such as BACnet IP or BACnet MS/TP.

12.1.1.1. GET /v1/gateway/<bootstrap_server_id>

This message is broadcast on BACnet to discover the best gateway capable of routing traffic to the indicated bootstrap server. No payload is provided.

All gateways capable of routing traffic to the broadcast server should respond with their assigned priority for handling traffic (or zero if it has not been explicitly set). This means that a device may receive multiple replies to its GET message and it should be able to handle this. The device should wait a reasonable time (e.g. 5 seconds) for all replies to be received and should pick the gateway with the lowest priority value. If multiple gateways respond with the same lowest priority value, a gateway should be chosen at random.

If no gateways respond, the device should backoff and retry.

The gateway response should be a 2.00 status code with a CBOR payload containing the following content:

INTEGER priority

12.1.1.2. POST /v1/gateway/channel?server=<bootstrap_server_id>

This message is sent to the chosen gateway that responded to the discovery message described in 12.1.1.1. above. The gateway should assign a channel that routes messages to the indicated bootstrap server. No payload is provided. If successful, the gateway response should be a CBOR payload with the following content:

INTEGER channel_id
BYTE STRING token

The cookie should be a random string of at least 4 bytes. The device must provide this token when closing the channel.

The status code will be one of:

Response	Description
2.01	The channel has been created
4.04	The gateway cannot find a route to the

	indicated bootstrap server	
5.03	The gateway cannot create the channel at this time	

12.1.1.3. DELETE/v1/gateway/channel/<channel_id>?token=<token_in_hex>

This message is sent to the chosen gateway that responded to the discovery message described in 12.1.1.1. above. The gateway should assign a channel that routes messages to the indicated bootstrap server. No payload is provided and no payload is returned. The device must offer the token returned by the gateway in the channel assign request in 12.1.1.2. above and the DELETE request must come from the same network address as the POST request.

The status code will be one of:

Response	Description
2.02	The channel has been deleted
4.01	The device is not allowed to delete this channel

12.1.2. aSSURE Secure Management Channels

These messages are used for encrypted aSSURE-protected DTLS sessions accessing the Device Management API defined in 10. above. All messages on the aSSURE Secure Management Channel have the following CBOR structure:

INTEGER channel_id
BYTE STRING dtls_data

12.1.3. aSSURE Secure Data Channels

These messages are used for all other aSSURE-protected DTLS session such as secure bootstrap or peer-to-peer channels. All messages on the aSSURE Secure Data Channel have the following CBOR structure:

INTEGER channel_id
BYTE STRING dtls_data

12.2. IP

When deployed on IP networks, aSSURE traffic uses UDP port **TBD** for Secure Management Channels and UDP port **TBD** for Secure Data

Channels. The UDP payload is the DTLS data. The channel ID is inferred from the local port and remote address and port.

When compared to BACnet, the "aSSURE Bootstrap" messages are not required because an IP network is already able to route traffic directly to the bootstrap servers without the explicit establishment of a channel. The device is assumed to be capable of DHCP and DNS to obtain a local IP address, determine the subnet gateway and resolve the bootstrap server FQDN.

When an address is indicated in a CBOR message, the address format is:

```
ARRAY {
    INTEGER     1           // Address type 1 = IP
    BYTE STRING remote_addr // 4 bytes for IPv4,network byte order
                            //16 bytes for IPv6,network byte order
    INTEGER     local_port  // Local UDP port, 0=any
    INTEGER     remote_port // Remote UDP port
}
```

12.2.1. Bootstrap Server FQDN

The bootstrap server ID can be converted to a Fully Qualified Domain Name (FQDN) by suffixing the decimal value of the ID with the string ".***TBD***.net". Similarly, an aSSURE bootstrap server FQDN can be converted to the server ID by reversing the process.

12.3. Bluetooth

Bluetooth implementations should use the "Internet Protocol Support Profile" to allow the device to send and receive IPv6 traffic.

When an address is indicated in a CBOR message, the address format should be type 1 as in 12.2. above.
When a Bluetooth MAC address is specifically indicated in a CBOR message, the address format is:

```
ARRAY {
    INTEGER     2       // Address type 2 = Bluetooth
    BYTE STRING mac     // MAC address, 6 bytes
}
```

In all other respects, the Bluetooth implementation follows the "IP" implementation as in 12.2. above.

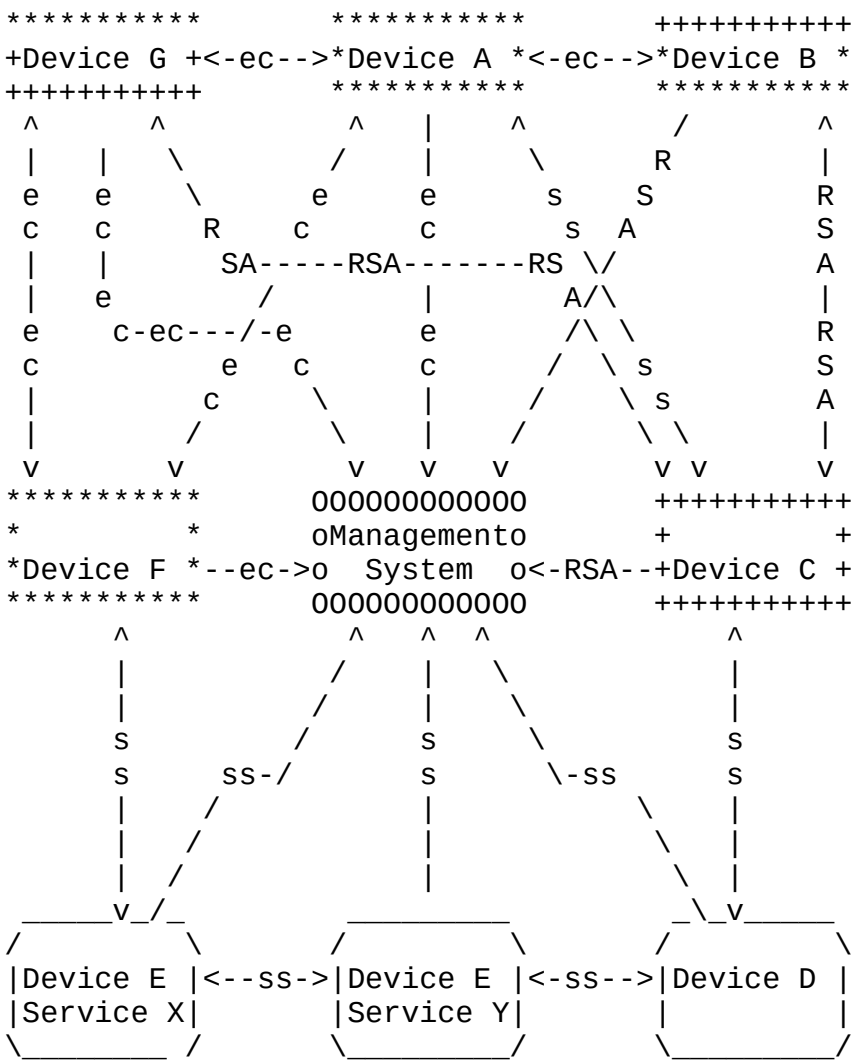
12.4. Assigned address types

Address Type	Description
0	BACnet NPDU address

1	Destination IP address and UDP ports
2	Bluetooth MAC address

13. DTLS Connection Configuration Examples

13.1. Example Topology



*** - (Device A & F) Elliptic Curve and Shared Secret Capable
+++ - (Device C) RSA and Shared Secret Capable

*** _ (Device G) Elliptic Curve, RSA and Shared Secret Capable
+++ (EC Device Key)

+++ _ (Device B) Elliptic Curve, RSA and Shared Secret Capable
*** (RSA Device Key)

___ - (device D and E) Only Shared Secret Capable
(Shared Secret Device Key)

"ec", "ss", "RSA" are the channels

Figure 7 DTLS Connection Example Topology

13.2. Elliptic Curve device ↔ Elliptic Curve device

e.g. Device A ↔ Device F
Both endpoints have Elliptic Curve keys so no additional keys need to be created. Each endpoint is sent a channel definition indicating the local device key, the peer device key, the peer address and the privileges for the channel.

13.3. Elliptic Curve device ↔ RSA device

We cannot use the two device keys to directly secure the DTLS connection. There are three possible solutions to allow a secure link. These are presented in the order of most to least preferred.

13.3.1. Option 1 - Issue EC key to RSA device

e.g. Device A ↔ Device B
If the RSA device can support Elliptic Curve keys, then a new Elliptic Curve key should be created on (or issued to) the RSA device by the Management System. The Management System should sign the new key as assigned to the device.

Now both devices have an EC key, so 13.2. above can be followed.

13.3.2. Option 2 - Issue RSA key to EC device

e.g. Device C ↔ Device G
If the EC device can support RSA keys, then a new RSA key should be created on (or issued to) the EC device by the Management System. The Management System should sign the new key as assigned to the device.

Now both devices have an RSA key, so 13.5. below can be followed.

13.3.3. Option 3 - Issue Shared Secret to both devices

e.g. Device A ↔ Device C
Shared secrets must be used, so 13.7. below is followed.

13.4. Elliptic Curve device ↔ Shared Secret device

e.g. Device F ↔ Device E (Service X)
Shared secrets must be used, so 13.7. below is followed.

13.5. RSA device ↔ RSA device

e.g. Device B ↔ Device C
Both endpoints have RSA key so no additional keys need to be

created. Each endpoint is sent a channel definition indicating the local device key, the peer device key, the peer address and the privileges for the channel.

13.6. RSA device ↔ Shared Secret device

e.g. Device C ↔ Device D
Shared secrets must be used, so 13.7. below is followed.

13.7. Shared Secret device ↔ Shared Secret device

e.g. Device D ↔ Device E (Service Y)
The management system must issue a new shared secret (called the "Channel Key") to both devices to identify the channel. Each endpoint is sent a channel definition indicating the Channel Key, the peer address and the privileges for the channel.

14. Message Sequence Diagrams

14.1. Manufacturing Flow

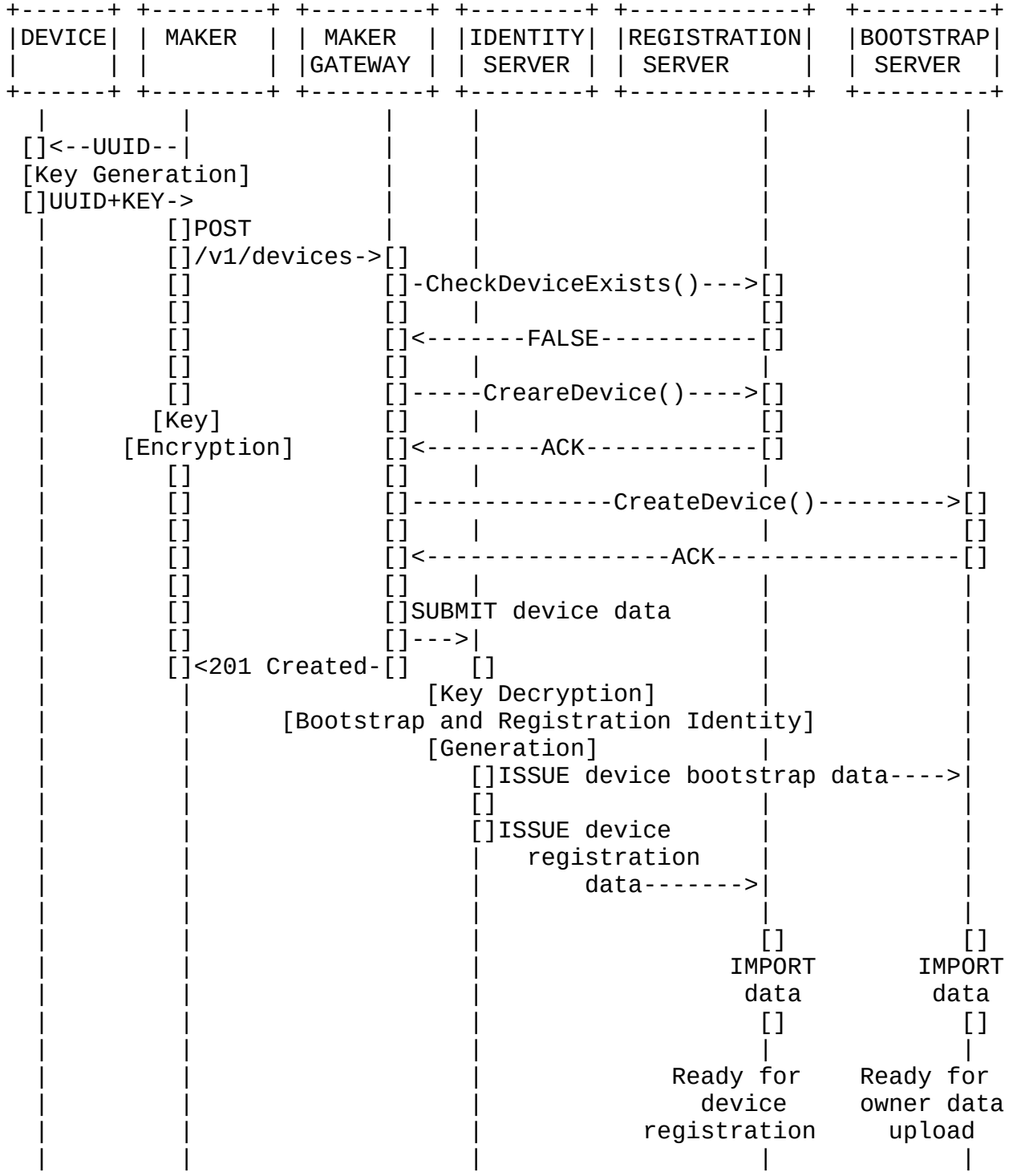
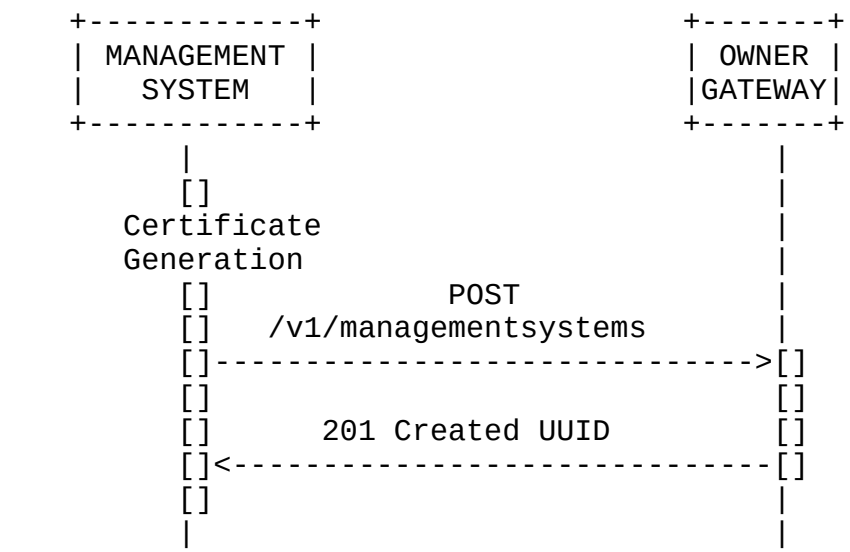


Figure 8 Manufacturing Flow Sequence Diagram

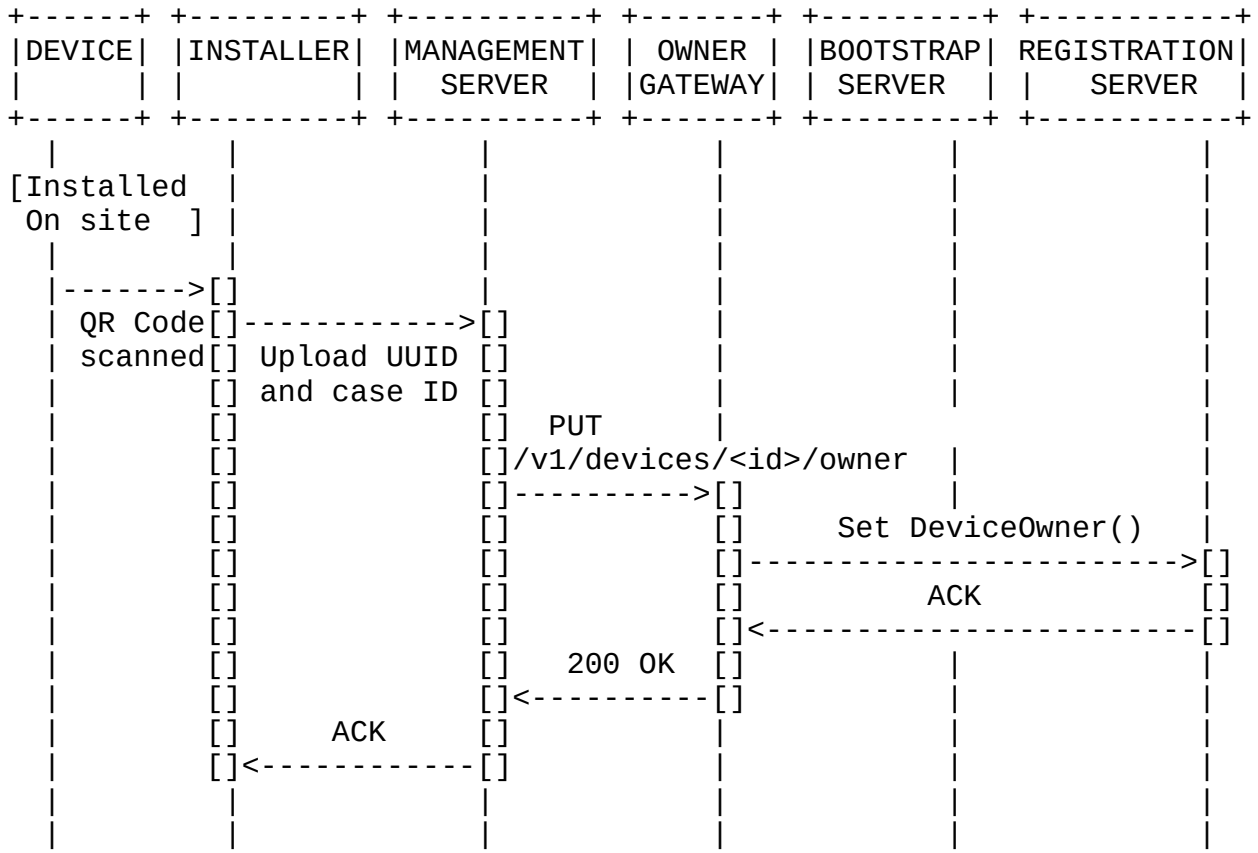
14.2. Management System Preparation



Registration of the Management System with the Trusted Authority

Figure 9 Management System Preparation Sequence Diagram

14.3. Device Registration



Registration of the device with the management systems

Figure 10 Device Registration Sequence Diagram

14.4. Device Ownership State Machine

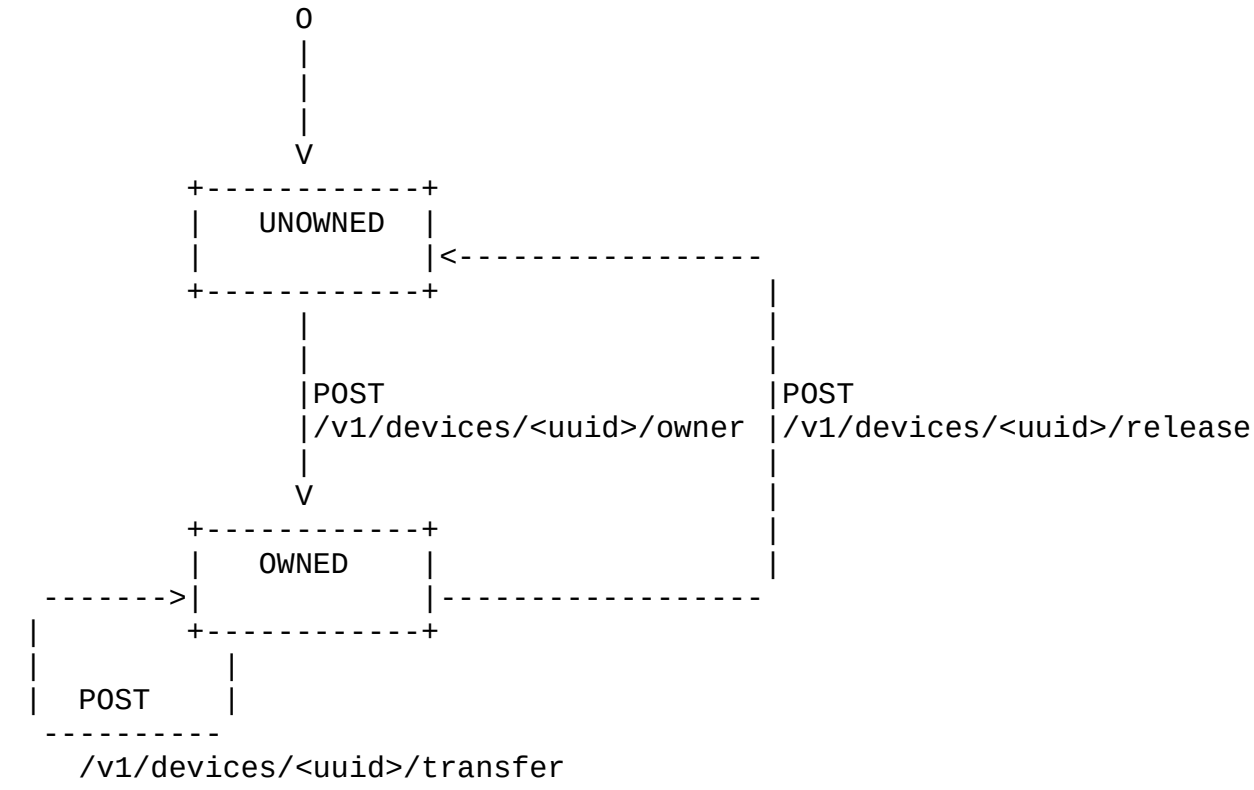


Figure 11 Device Ownership State Machine

14.5. Device Configuration and Bootstrap

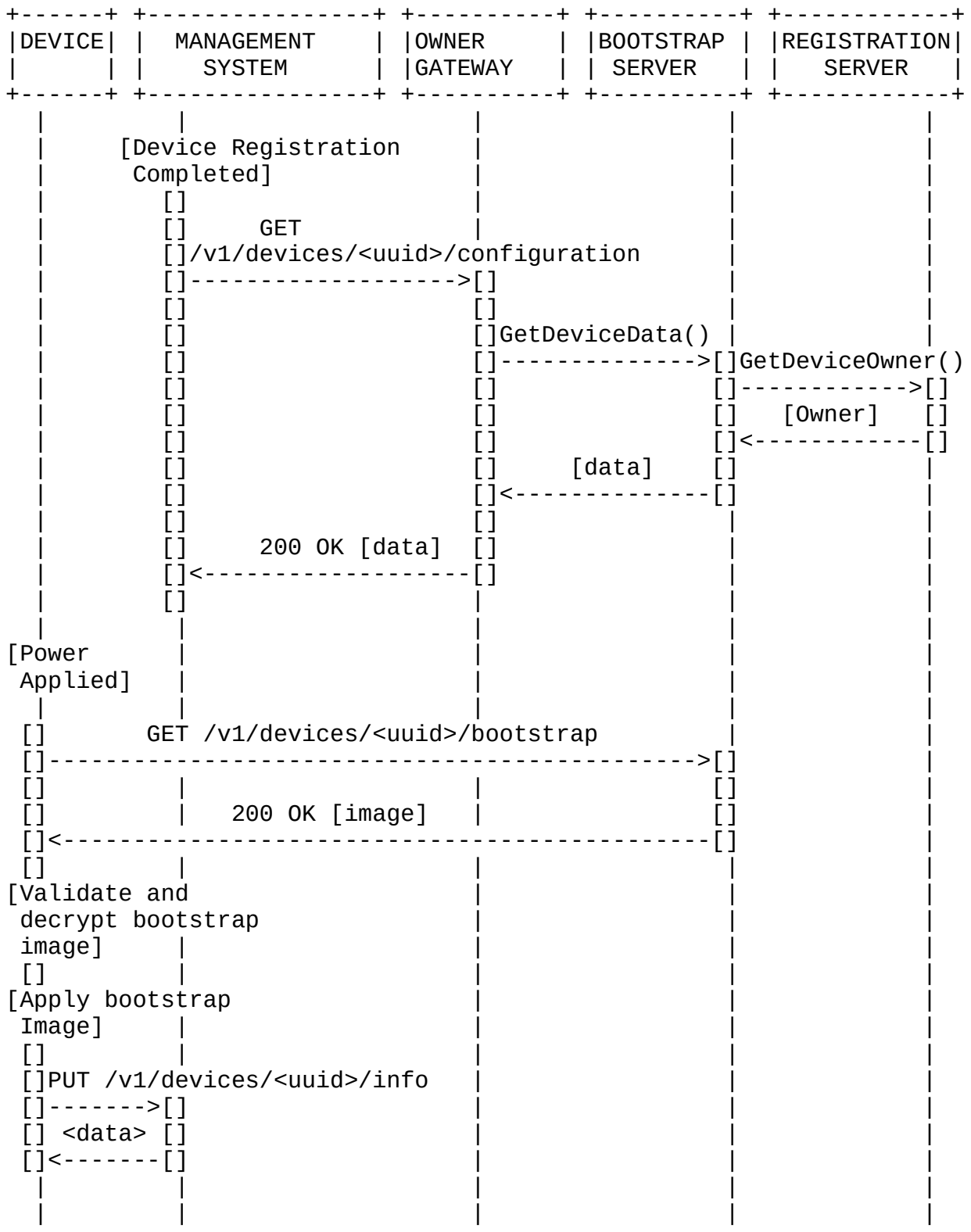
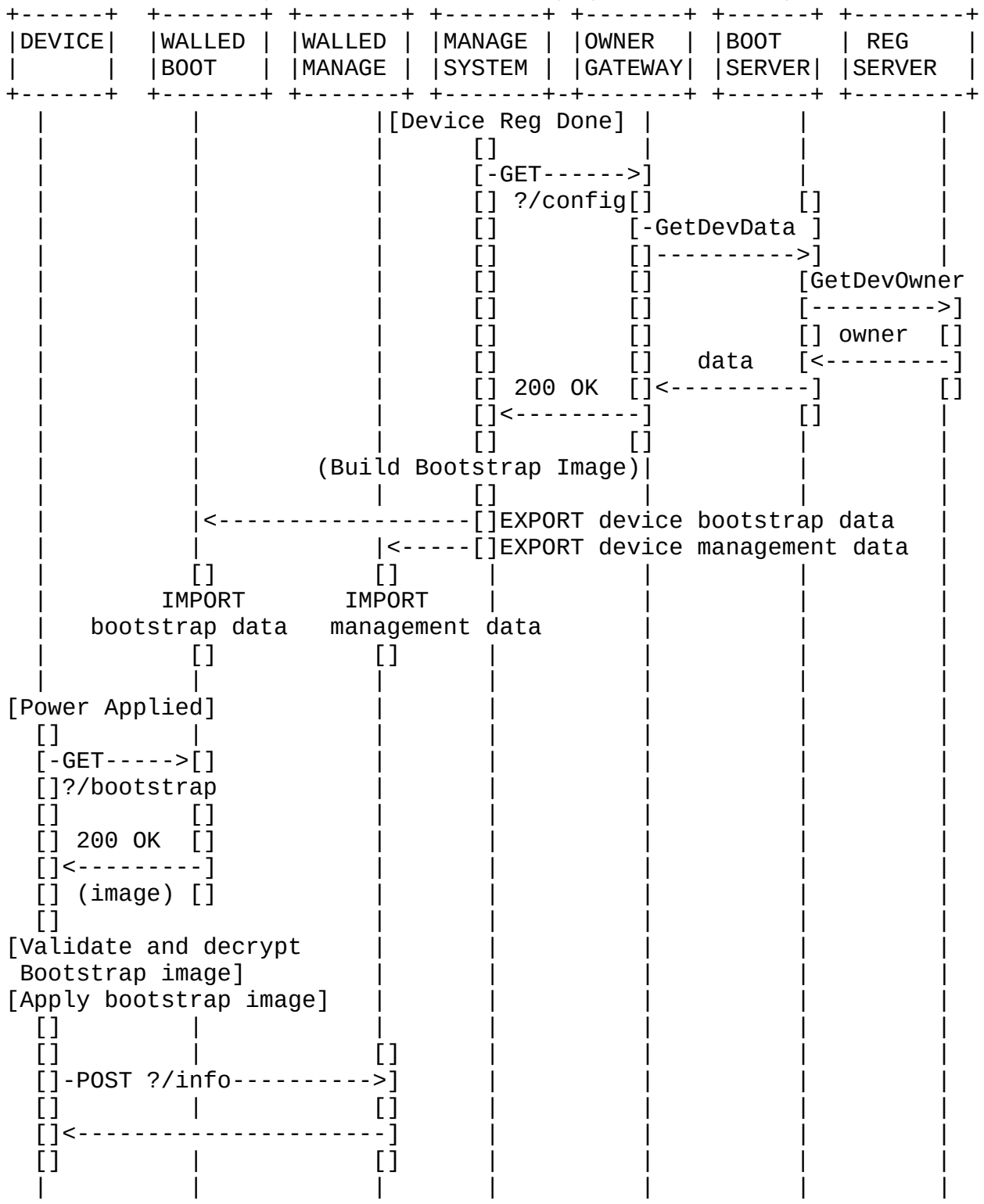


Figure 12 Device Configuration and Bootstrap Sequence Diagram

14.6. Device Configuration and Bootstrap (Walled Garden)



?/ is /v1/devices/<uuid>/
Configuration of the device to connect to the management system

Figure 13 Device Configuration and Bootstrap Sequence Diagram (Walled Garden)

14.7. Device Change Owner

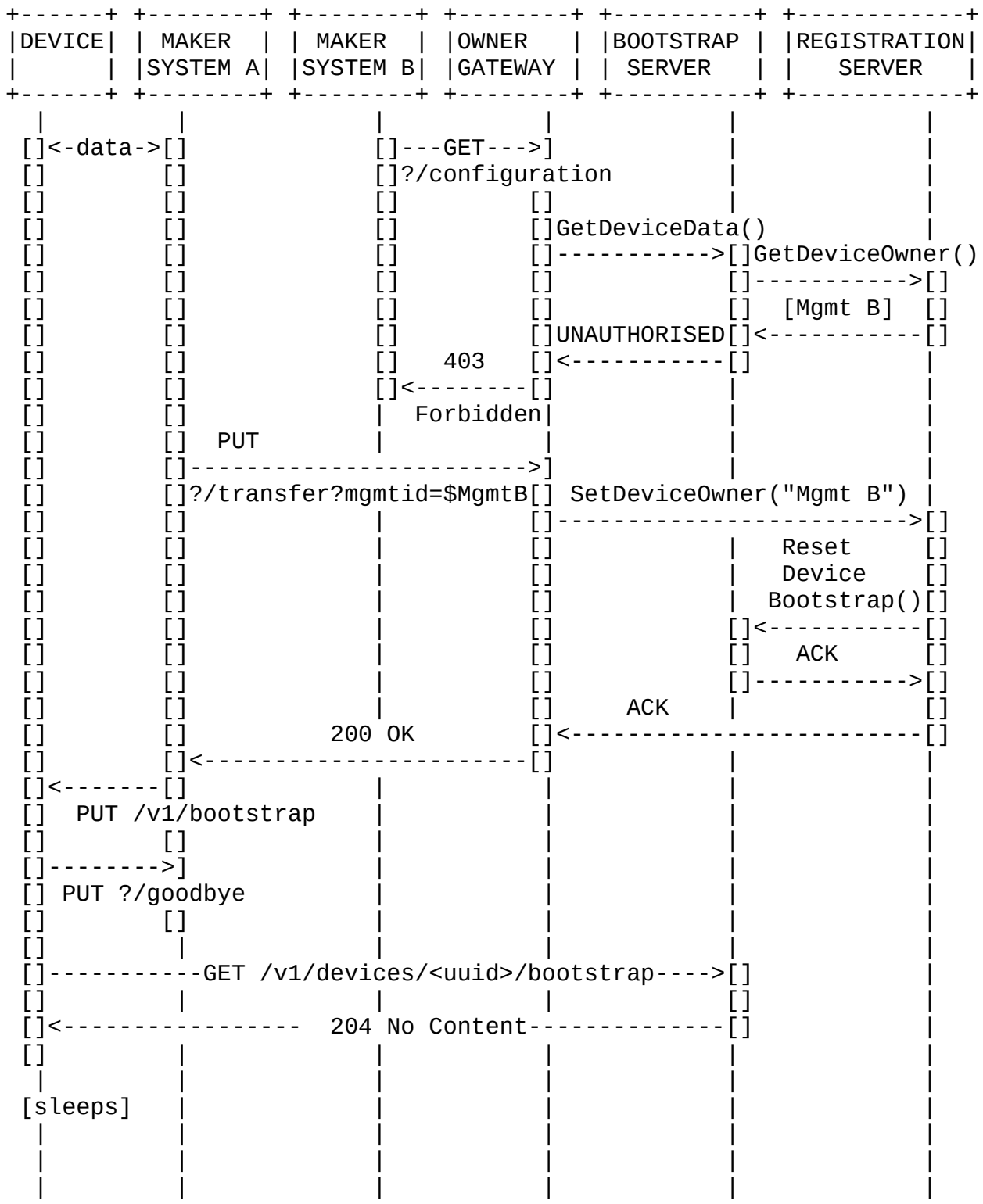
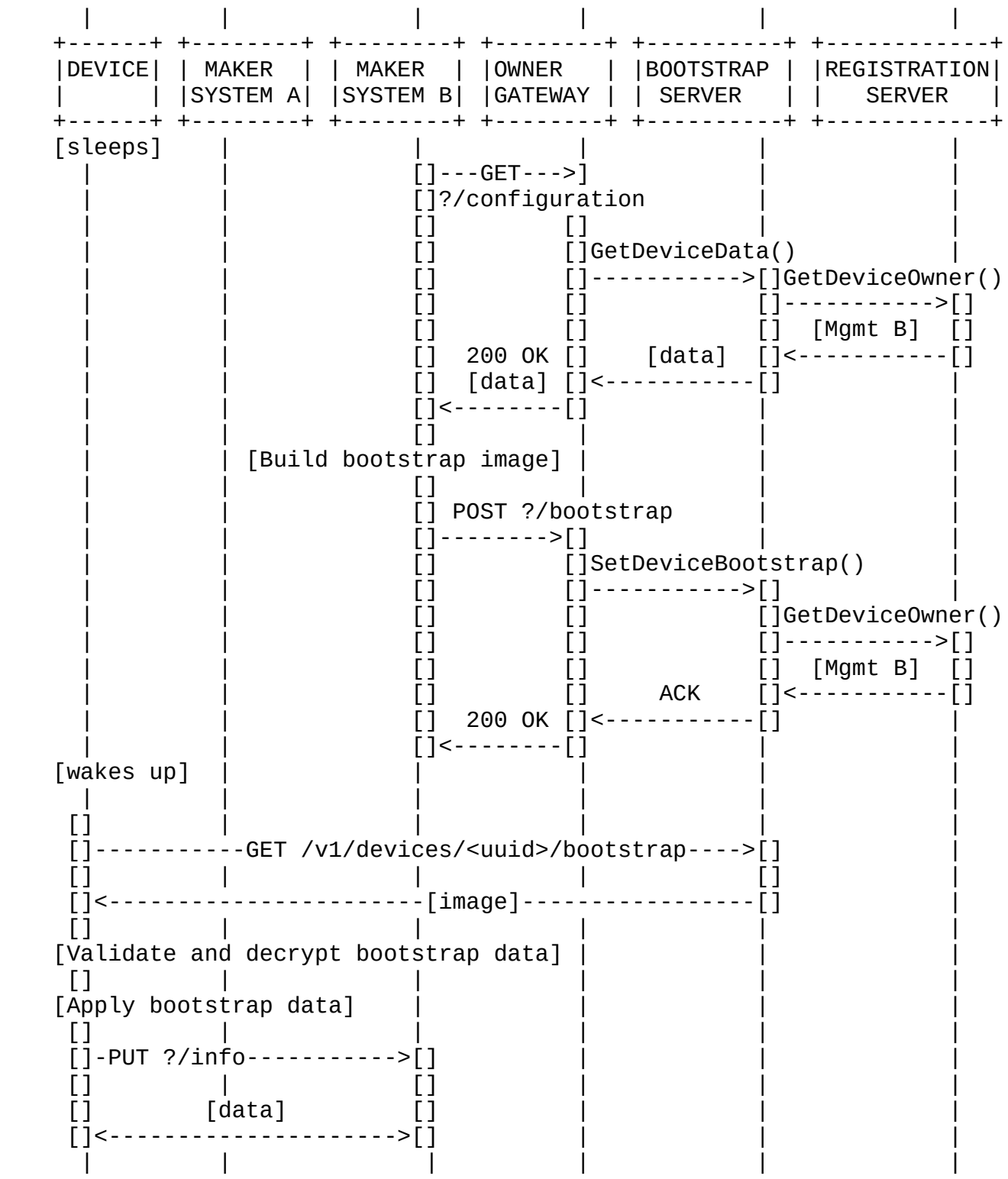


Figure 14 Device Change Owner Sequence Diagram (first part)



?/ is /v1/devices/<uuid>/
Device Change of Ownership
Change the management system authorized for the device

Figure 15 Device Change Owner Sequence Diagram (second part)

15. Configuration and Bootstrap Data Formats

15.1. Overview

The bootstrap data is critical to the device to determine ownership and allow authentication of the management system. Additional parameters may be provided to the device as part of the bootstrap data. The Management System uploads the bootstrap data for a device to the bootstrap server so that the bootstrap server can be provide it to the device during the bootstrap sequence. he Management System therefore encrypts the bootstrap data so that only the target device can decode it (in the case of shared secret devices, the Identity Service is also theoretically capable of this decryption). This protects the data from exposure should the bootstrap server be compromised.

The Management System needs to know what information the device needs to complete its bootstrap and it requests this in the request defined in 9.3.6. above.

15.2. Configuration data format

The configuration data for a device provides the manufacturing data for the device and information about the key to use to encrypt the bootstrap data. The data is in JSON format as below:

```
{
  "device": {
    "id": "<device UUID>",
    "bootstrap_server": <bootstrap server ID used by device>,
    "capabilities": {   // Device bootstrap capabilities
      "ec_capable": <boolean>,
      "rsa_capable": <boolean>,
      "sha384_capable": <boolean>,
      "sha512_capable": <boolean>,
      "aes256_capable": <boolean>,
      // Additional capabilities may be added in the future
    },
  },
  "authenticated_keys": [
    // Array of base-64 encoded key identity strings as in
    // 7.4.  above
  ],
  "owner_information": "<base64-encoded signed owner data
                                          in 15.3.  below>",
  "parameter_choices": [
      // List of sets of parameter choices. The Management
      // System should provide exactly one of the sets of
      // parameters, but it may choose to provide a different
      // parameter set if it has additional information about
      // what the device can support.
  ],
}
```

The format of the "parameter_choices" array depends on the types of messages that are required by the device. The exact format is TBD at this time.

15.3. Device connection to the bootstrap server using DTLS using pre-shared secrets

The array of "authenticated_keys" provided in the configuration data will include a bootstrap server key. This key and all keys that it relies on to derive it from the device key must be provided to the device by the bootstrap server during the DTLS handshake so that the device can establish the DTLS connection. The bootstrap service in the Trusted Authority will do this automatically. If a bootstrap service is used within the Walled Garden, it must be careful to include all these keys in the correct order (from device key to bootstrap server) so that the device can derive the key necessary for the DTLS session.

15.4. Device connection to the bootstrap server using DTLS using public keys

There is a special requirement for the device behaviour when establishing the DTLS connection to the bootstrap server. The DTLS handshake (with extensions as in RFC-7250 allowing raw public keys) uses public keys rather than certificates, so the device cannot authenticate the bootstrap server key during the DTLS handshake.

The device must therefore *temporarily* accept the public key from the bootstrap server during the DTLS handshake and download the bootstrap data. The device must then check that the public key from the bootstrap server is in the list of identities in the bootstrap and that it has the "Bootstrap Service" identity class.

Once the identity of the bootstrap server has been confirmed, validation of the bootstrap data can continue. If the identity of the bootstrap server cannot be confirmed, the bootstrap data should be discarded.

15.5. Bootstrap data format

The bootstrap data is constructed by the management system based on the configuration data and the additional information that the management system needs to provide.

The bootstrap data is in CBOR format comprises three sections - a header, the encrypted content and the signature. The header includes one or more authenticated keys and the owner information. All authenticated keys in the configuration data must be included in the authenticated key list in the order provided. The management system may then append additional keys if it wishes.

The order is important because the device will validate and import the authenticated keys in the order provided. If a key is invalid or depends on a key that is not yet imported, the bootstrap data will be rejected.

The owner information tells the device the number of owners the device has had. This number starts at zero and is incremented each time the owner changes. The device must store this number in non volatile memory and only accept bootstrap data if the owner sequence_number in the bootstrap data is the same or higher than the owner_sequence_number stored in non-volatile memory. This prevents replay attacks of older owner data in an attempt to reclaim ownership of a device. The owner data must be signed by a manufacturer or registration identity as defined in 7.7. above.

```
ARRAY {
  // One or more authenticated key definitions in 7.4.  above
  ARRAY {
    ...  // Authenticated key definition
  }

  // Owner information
  ARRAY {
    INTEGER content  //"Owner Content Type" as in 7.5.  above
    INTEGER owner_sequence_number
    ARRAY {
      // Signature for owner data, provided by bootstrap
      // server as in 7.3.  above
    }
  }
  // End of owner information

  // Start of encryption information
  BYTE STRING decryption_key_id  // Key UUID
  BYTE STRING encrypted_payload

  // Signature for the entire bootstrap data
  ARRAY {
    // Signature as in 7.3.  above, signed by Management
    // Systems key
    // Signature covers ENCRYPTED payload, so signature
    //.validation is done before decryption
  }
}
```

15.5.1. Payload protected by Elliptic Curve keys

If the decryption key refers to an Elliptic Curve key, the encrypted_payload is a Cryptographic Message Syntax object containing an enveloped-data block (see RFC 5652 and RFC 5753). The enveloped-data should be encrypted using the "Standard" variation of Ephemeral Static ECDH (see RFC 5753 section 3.1). The

default choices for encryption cipher and hash function should be AES-128 and SHA-256 respectively.

The "enveloped-data", after decryption, contains the payload CBOR structure as defined in 15.5.4. below.

15.5.2. Payload protected by RSA keys

If the decryption key refers to an RSA key, then the encrypted_payload is a Cryptographic Message Syntax object containing an enveloped-data block (see RFC 5652).

The enveloped-data should be encrypted using RSAES-OAEP (see RFC 8017 section 7.1). The default choices for encryption cipher and hash function should be AES-128 and SHA-256 respectively. The SHA-1 hash should NOT be used.

The "enveloped-data", after decryption, contains the payload CBOR structure as defined in 15.5.4. below.

15.5.3. Payload protected by shared secrets

If the decryption key refers to a shared secret then the encrypted_payload contains the CBOR structure below. The salt, iterations_or_cipher and encrypted_secret fields are used to derive a decryption key for the cipher in the same way as a derived secret is obtained in section 7.2. above. This key is then used with the indicated cipher_suite with the cipher_IV and optional tag to decrypt the encrypted_data.

```
ARRAY {
  BYTE STRING     salt
  INTEGER         iterations_or_cipher
  BYTE_STRING     encrypted_secret
  INTEGER         cipher_suite  // As in 7.8.  above
  BYTE STRING     cipher_IV
  BYTE STRING     tag        // Zero length for CBC ciphers
  BYTE STRING     encrypted_data
}
```

The "encrypted_data", after decryption, contains the payload CBOR structure as defined in 15.5.4. below.

15.5.4. Decrypted payload content

The payload has the following CBOR format:

```
ARRAY {
  BYTE STRING coap_message0
  BYTE STRING coap_message1
  ...
  BYTE STRING coap_messageN
}
```

Each message is replayed to the local API in the order in the payload.

If the device requires configuration messages to be replayed to a different API, a local API function should be created that understands how to replay the message content to the other API. E.g. replay of a BACnet APDU to the local device.

16. Security Considerations

This whole draft concerns security considerations. See Chapter 6.

17. IANA Considerations

None

18. Conclusions

End to end certificate handling and encrypted communication using "channels" within the DTLS framework can easily be achieved without inventing new standards, just by enhancing current ones. This covers devices from high end servers down to resource constrained devices across different types of network.

The underlying standards are:

Transport Layer Security, TLS v1.2, RFC-5246
Datagram Transport Layer Security, DTLS v1.2, RFC-6347
Constrained Application Framework, CoAP, RFC-7252
Concise Binary Object Representation, CBOR, RFC-7049
CoAP Block-wise Transfers, https://www.ietf.org/id/draft-ietf core-block-21.txt

The aSSURE specification lends itself to the industrial manufacture and distribution of IoT and other connected pieces of equipment and can serve many markets both in retrofit and new build. Indeed IoT is currently disgorging millions of devices in architectures that are not secure enough and could be repaired using the aSSURE framework and philosophy. This problem is better described by Bob Hinden in his paper "Internet of Insecure Things" published in the Internet Protocol Journal.

19. References

19.1. Normative References

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

[RFC4279] Eronen, P., Ed., and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC4279, DOI

10.17487/RFC4279, December 2005, <http://www.rfc editor.org/info/rfc4279>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc editor.org/info/rfc5246>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <http://www.rfc-editor.org/info/rfc7049>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <http://www.rfc-editor.org/info/rfc7250>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

19.2. Informative References

CoAP Block-wise Transfers, https://www.ietf.org/id/draft-ietf core-block-21.txt

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <http://www.rfc-editor.org/info/rfc2898>.

[RFC5753] Turner, S. and D. Brown, "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)", RFC 5753, DOI 10.17487/RFC5753, January 2010, <http://www.rfc-editor.org/info/rfc5753>.

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

"Internet of Insecure Things", Hinden, B., Internet Protocol Journal March 2017 Vol 20, Number 1, Page 12. <http://ipj.dreamhosters.com/wp-content/uploads/issues/2017/ipj20-1.pdf>.

20. Acknowledgments

This document is a byproduct of the "aSSURE" project, partially funded by Innovate UK. It is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranties of fitness for a particular purpose. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the aSSURE project or Innovate UK.

Author's Address

Roger Lucas
c/o Cisco International Limited
10, New Square Park
Bedfont Lakes
Feltham
TW14 8HA
United Kingdom
Email: iot@hiddenengine.co.uk