Workgroup: RATS Working Group
Internet-Draft: draft-ietf-rats-eat-11
Published: 23 October 2021
Intended Status: Standards Track
Expires: 26 April 2022
Authors: L. Lundblade
Security Theory LLC
G. Mandyam
Qualcomm Technologies Inc.
J. O'Donoghue
Qualcomm Technologies Inc.

The Entity Attestation Token (EAT)

Abstract

An Entity Attestation Token (EAT) provides a signed (attested) set of claims that describe state and characteristics of an entity, typically a device like a phone or an IoT device. These claims are used by a Relying Party to determine how much it wishes to trust the entity.

An EAT is either a CWT or JWT with some attestation-oriented claims. To a large degree, all this document does is extend CWT and JWT.

Status of This Memo

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

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

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

This Internet-Draft will expire on 26 April 2022.

Table of Contents

1. Introduction

Remote device attestation is a fundamental service that allows a remote device such as a mobile phone, an Internet-of-Things (IoT) device, or other endpoint to prove itself to a Relying Party, a server or a service. This allows the Relying Party to know some characteristics about the device and decide whether it trusts the device.

The notion of attestation here is large and may include, but is not limited to the following:

- Proof of the make and model of the device hardware (HW)
- Proof of the make and model of the device processor, particularly for security-oriented chips
- Measurement of the software (SW) running on the device
- Configuration and state of the device
- Environmental characteristics of the device such as its GPS location

This document uses the terminology and main operational model defined in [RATS.Architecture]. In particular it is a format that can be used for Attestation Evidence or Attestation Results as defined in the RATS architecture.

1.1. CWT, JWT, UCCS, UJCS and DEB

An EAT is a set of claims about an entity/device based on one of the following:

- CBOR Web Token (CWT), [RFC8392]
- Unprotected CWT Claims Sets (UCCS), [UCCS.Draft]
- JSON Web Token (JWT), [RFC7519]

All definitions, requirements, creation and validation procedures, security considerations, IANA registrations and so on from these carry over to EAT.

This specification extends those specifications by defining additional claims for attestation. This specification also describes the notion of a "profile" that can narrow the definition of an EAT, ensure interoperability and fill in details for specific usage scenarios. This specification also adds some considerations for registration of future EAT-related claims.

The identification of a protocol element as an EAT, whether CBOR or JSON encoded, follows the general conventions used by CWT, JWT and UCCS. Largely this depends on the protocol carrying the EAT. In some cases it may be by content type (e.g., MIME type). In other cases it may be through use of CBOR tags. There is no fixed mechanism across all use cases.

This specification adds two more top-level messages:

- *Unprotected JWT Claims Set (UJCS), Section 4

- *Detached EAT Bundle (DEB), Section 5

A DEB is simple structure to hold a collection of detached claims-sets and the EAT that separately provides integrity and authenticity protection for them. It can be either CBOR or JSON encoded.

1.2. CDDL, CBOR and JSON

An EAT can be encoded in either CBOR or JSON. The definition of each claim is such that it can be encoded either. Each token is either entirely CBOR or JSON, with only an exception for nested tokens.

To implement composite attestation as described in the RATS architecture document, one token has to be nested inside another. It is also possible to construct composite Attestation Results (see below) which may be expressed as one token nested inside another. So as to not force each end-end attestation system to be all JSON or all CBOR, nesting of JSON-encoded tokens in CBOR-encoded tokens and vice versa is accommodated by this specification. This is the only place that CBOR and JSON can be mixed.

This specification formally uses CDDL, [RFC8610], to define each claim. The implementor interprets the CDDL to come to either the CBOR [RFC8949] or JSON [ECMAScript] representation. In the case of JSON, Appendix E of [RFC8610] is followed. Additional rules are given in Section 8.2.2 where Appendix E is insufficient.

The CWT and JWT specifications were authored before CDDL was available and did not use CDDL. This specification includes a CDDL definition of most of what is defined in [RFC8392]. Similarly, this specification includes CDDL for most of what is defined in [RFC7519].

The UCCS specification does not include CDDL. This specification provides CDDL for it.

(TODO: The authors are open to modifications to this specification and the UCCS specification to include CDDL for UCCS and UJCS there instead of here.)

1.3. Operating Model and RATS Architecture

While it is not required that EAT be used with the RATS operational model described in Figure 1 in [RATS.Architecture], or even that it be used for attestation, this document is authored with an orientation around that model.

To summarize, an Attester on an entity/device generates Attestation Evidence. Attestation Evidence is a Claims Set describing various characteristics of the entity/device. Attestation Evidence also is usually signed by a key that proves the entity/device and the evidence it produces are authentic. The Claims Set includes a nonce or some other means to provide freshness. EAT is designed to carry Attestation Evidence. The Attestation Evidence goes to a Verifier where the signature is validated. Some of the Claims may also be validated against Reference Values. The Verifier then produces Attestation Results which is also usually a Claims Set. EAT is also designed to carry Attestation Results. The Attestation Results go to the Relying Party which is the ultimate consumer of the "Remote Attestaton Procedures", RATS. The Relying Party uses the Attestation Results as needed for the use case, perhaps allowing a device on the network, allowing a financial transaction or such.

Note that sometimes the Verifier and Relying Party are not separate and thus there is no need for a protocol to carry Attestation Results.

1.3.1. Use as Attestation Evidence

Any claim defined in this document or in the IANA CWT or JWT registry may be used in Attestation Evidence.

Attestation Evidence nearly always has to be signed or otherwise have authenticity and integrity protection because the Attester is remote relative to the Verifier. Usually, this is by using COSE/JOSE signing where the signing key is an attestation key provisioned into the entity/device by its manufacturer. The details of how this is

achieved are beyond this specification, but see Section 6. If there is already a suitable secure channel between the Attester and Verifier, UCCS may be used.

1.3.2. Use as Attestation Results

Any claim defined in this document or in the IANA CWT or JWT registry may be used in Attestation Results.

It is useful to characterize the relationship of claims in Evidence to those in Attestation Results.

Many claims in Attestation Evidence simply will pass through the Verifier to the Relying Party without modification. They will be verified as authentic from the device by the Verifier just through normal verification of the Attester's signature. The UEID, Section 3.4, and Location, Section 3.14, are examples of claims that may be passed through.

Some claims in Attestation Evidence will be verified by the Verifier by comparison to Reference Values. These claims will not likely be conveyed to the Relying Party. Instead, some claim indicating they were checked may be added to the Attestation Results or it may be tacitly known that the Verifier always does this check. For example, the Verifier receives the Software Evidence claim, Section 3.21, compares it to Reference Values and conveys the results to the Relying Party in a Software Measurement Results Claim, Section 3.22.

In some cases the Verifier may provide privacy-preserving functionality by stripping or modifying claims that do not posses sufficient privacy-preserving characteristics. For example, the data in the Location claim, Section 3.14, may be modified to have a precision of a few kilometers rather than a few meters.

When the Verifier is remote from the Relying Party, the Attestation Results must be protected for integrity, authenticity and possibly confidentiality. Often this will simply be HTTPS as per a normal web service, but COSE or JOSE may also be used. The details of this protection are beyond the scope of this document.

1.4. Entity Overview

An "entity" can be any device or device subassembly ("submodule") that can generate its own attestation in the form of an EAT. The attestation should be cryptographically verifiable by the EAT consumer. An EAT at the device-level can be composed of several submodule EAT's.

Modern devices such as a mobile phone have many different execution environments operating with different security levels. For example,

it is common for a mobile phone to have an "apps" environment that runs an operating system (OS) that hosts a plethora of downloadable apps. It may also have a TEE (Trusted Execution Environment) that is distinct, isolated, and hosts security-oriented functionality like biometric authentication. Additionally, it may have an eSE (embedded Secure Element) - a high security chip with defenses against HW attacks that is used to produce attestations. This device attestation format allows the attested data to be tagged at a security level from which it originates. In general, any discrete execution environment that has an identifiable security level can be considered an entity.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document reuses terminology from JWT [RFC7519] and CWT [RFC8392].

Claim: A piece of information asserted about a subject. A claim is represented as pair with a value and either a name or key to identify it.

Claim Name: A unique text string that identifies the claim. It is used as the claim name for JSON encoding.

Claim Key: The CBOR map key used to identify a claim.

Claim Value: The value portion of the claim. A claim value can be any CBOR data item or JSON value.

CWT/JWT Claims Set: The CBOR map or JSON object that contains the claims conveyed by the CWT or JWT.

This document reuses terminology from RATS Architecure [RATS.Architecture]

Attester: A role performed by an entity (typically a device) whose Evidence must be appraised in order to infer the extent to which the Attester is considered trustworthy, such as when deciding whether it is authorized to perform some operation.

Verifier: A role that appraises the validity of Attestation Evidence about an Attester and produces Attestation Results to be used by a Relying Party.

Relying Party:
A role that depends on the validity of information about an Attester, for purposes of reliably applying application specific actions. Compare /relying party/ in [RFC4949].

Attestation Evidence: A Claims Set generated by an Attester to be appraised by a Verifier. Attestation Evidence may include configuration data, measurements, telemetry, or inferences.

Attestation Results: The output generated by a Verifier, typically including information about an Attester, where the Verifier vouches for the validity of the results

Reference Values: A set of values against which values of Claims can be compared as part of applying an Appraisal Policy for Attestation Evidence. Reference Values are sometimes referred to in other documents as known-good values, golden measurements, or nominal values, although those terms typically assume comparison for equality, whereas here Reference Values might be more general and be used in any sort of comparison.

3. The Claims

This section describes new claims defined for attestation that are to be added to the CWT [IANA.CWT.Claims] and JWT [IANA.JWT.Claims] IANA registries.

This section also describes how several extant CWT and JWT claims apply in EAT.

CDDL, along with a text description, is used to define each claim independent of encoding. Each claim is defined as a CDDL group. In Section 8 on encoding, the CDDL groups turn into CBOR map entries and JSON name/value pairs.

Each claim described has a unique text string and integer that identifies it. CBOR encoded tokens MUST use only the integer for Claim Keys. JSON encoded tokens MUST use only the text string for Claim Names.

3.1. Token ID Claim (cti and jti)

CWT defines the "cti" claim. JWT defines the "jti" claim. These are equivalent to each other in EAT and carry a unique token identifier as they do in JWT and CWT. They may be used to defend against re use of the token but are distinct from the nonce that is used by the Relying Party to guarantee freshness and defend against replay.

3.2. Timestamp claim (iat)

The "iat" claim defined in CWT and JWT is used to indicate the date-of-creation of the token, the time at which the claims are collected and the token is composed and signed.

The data for some claims may be held or cached for some period of time before the token is created. This period may be long, even days. Examples are measurements taken at boot or a geographic position fix taken the last time a satellite signal was received. There are individual timestamps associated with these claims to indicate their age is older than the "iat" timestamp.

CWT allows the use floating-point for this claim. EAT disallows the use of floating-point. No token may contain an iat claim in float-point format. Any recipient of a token with a floating-point format iat claim may consider it an error. A 64-bit integer representation of epoch time can represent a range of +/- 500 billion years, so the only point of a floating-point timestamp is to have precession greater than one second. This is not needed for EAT.

3.3. Nonce Claim (nonce)

All EATs should have a nonce to prevent replay attacks. The nonce is generated by the Relying Party, the end consumer of the token. It is conveyed to the entity over whatever transport is in use before the token is generated and then included in the token as the nonce claim.

This documents the nonce claim for registration in the IANA CWT claims registry. This is equivalent to the JWT nonce claim that is already registered.

The nonce must be at least 8 bytes (64 bits) as fewer are unlikely to be secure. A maximum of 64 bytes is set to limit the memory a constrained implementation uses. This size range is not set for the already-registered JWT nonce, but it should follow this size recommendation when used in an EAT.

Multiple nonces are allowed to accommodate multistage verification and consumption.

```
$$claims-set-claims //=
    (nonce-label => nonce-type / [ 2* nonce-type ])

nonce-type = bstr .size (8..64)
```

3.4. Universal Entity ID Claim (ueid)

UEID's identify individual manufactured entities / devices such as a mobile phone, a water meter, a Bluetooth speaker or a networked security camera. It may identify the entire device or a submodule or subsystem. It does not identify types, models or classes of devices. It is akin to a serial number, though it does not have to be sequential.

UEID's must be universally and globally unique across manufacturers and countries. UEIDs must also be unique across protocols and systems, as tokens are intended to be embedded in many different protocols and systems. No two products anywhere, even in completely different industries made by two different manufacturers in two different countries should have the same UEID (if they are not global and universal in this way, then Relying Parties receiving them will have to track other characteristics of the device to keep devices distinct between manufacturers).

There are privacy considerations for UEID's. See Section 10.1.

The UEID is permanent. It never change for a given device / entity.

UEIDs are variable length. All implementations MUST be able to receive UEIDs that are 33 bytes long (1 type byte and 256 bits). The recommended maximum sent is also 33 bytes.

When the entity constructs the UEID, the first byte is a type and the following bytes the ID for that type. Several types are allowed to accommodate different industries and different manufacturing processes and to give options to avoid paying fees for certain types of manufacturer registrations.

Creation of new types requires a Standards Action [RFC8126].

Type Byte	Type Name	Specification
0x01	RAND	This is a 128, 192 or 256 bit random number generated once and stored in the device. This may be constructed by concatenating enough identifiers to make up an equivalent number of random bits and then feeding the concatenation through a cryptographic hash function. It may also be a cryptographic quality random number generated once at the beginning of the life of the device and stored. It may not be smaller than 128 bits.
0x02	IEEE EUI	This makes use of the IEEE company identification registry. An EUI is either an EUI-48, EUI-60 or EUI-64 and made up of an OUI, OUI-36 or a CID, different

Type Byte	Type Name	Specification
		registered company identifiers, and some unique per-device identifier. EUIs are often the same as or similar to MAC addresses. This type includes MAC-48, an obsolete name for EUI-48. (Note that while devices with multiple network interfaces may have multiple MAC addresses, there is only one UEID for a device) [IEEE.802-2001], [OUI.Guide]
0x03	IMEI	This is a 14-digit identifier consisting of an 8-digit Type Allocation Code and a 6-digit serial number allocated by the manufacturer, which SHALL be encoded as byte string of length 14 with each byte as the digit's value (not the ASCII encoding of the digit; the digit 3 encodes as 0x03, not 0x33). The IMEI value encoded SHALL NOT include Luhn checksum or SVN information. [ThreeGPP.IMEI]

Table 1: UEID Composition Types

UEID's are not designed for direct use by humans (e.g., printing on the case of a device), so no textual representation is defined.

The consumer (the Relying Party) of a UEID MUST treat a UEID as a completely opaque string of bytes and not make any use of its internal structure. For example, they should not use the OUI part of a type 0x02 UEID to identify the manufacturer of the device. Instead they should use the oemid claim that is defined elsewhere. The reasons for this are:

- *UEIDs types may vary freely from one manufacturer to the next.
- *New types of UEIDs may be created. For example, a type 0x07 UEID may be created based on some other manufacturer registration scheme.
- *Device manufacturers are allowed to change from one type of UEID to another anytime they want. For example, they may find they can optimize their manufacturing by switching from type 0x01 to type 0x02 or vice versa. The main requirement on the manufacturer is that UEIDs be universally unique.

A Device Indentifier URN is registered for UEIDs. See Section 9.3.4.

```
$$claims-set-claims //= (ueid-label => ueid-type)

ueid-type = bstr .size (7..33)
```

3.5. Semi-permanent UEIDs (SUEIDs)

An SEUID is of the same format as a UEID, but it may change to a different value on device life-cycle events. Examples of these events are change of ownership, factory reset and on-boarding into an IoT device management system. A device may have both a UEID and SUEIDs, neither, one or the other.

There may be multiple SUEIDs. Each one has a text string label the purpose of which is to distinguish it from others in the token. The label may name the purpose, application or type of the SUEID. Typically, there will be few SUEDs so there is no need for a formal labeling mechanism like a registry. The EAT profile may describe how SUEIDs should be labeled. If there is only one SUEID, the claim remains a map and there still must be a label. For example, the label for the SUEID used by FIDO Onboarding Protocol could simply be "FDO".

There are privacy considerations for SUEID's. See Section 10.1.

A Device Indentifier URN is registered for SUEIDs. See Section 9.3.4.

```
$$claims-set-claims //= (sueids-label => sueids-type)

sueids-type = {
    + tstr => ueid-type
}
```

3.6. Hardware OEM Identification (oemid)

This claim identifies the OEM of the hardware. Any of the three forms may be used at the convenience of the attester implementation. The receiver of this claim MUST be able to handle all three forms.

3.6.1. Random Number Based

This format is always 16 bytes in size (128 bits).

The OEM may create their own ID by using a cryptographic-quality random number generator. They would perform this only once in the life of the company to generate the single ID for said company. They would use that same ID in every device they make. This uniquely identifies the OEM on a statistical basis and is large enough should there be ten billion companies.

The OEM may also use a hash like SHA-256 and truncate the output to 128 bits. The input to the hash should be somethings that have at least 96 bits of entropy, but preferably 128 bits of entropy. The

input to the hash may be something whose uniqueness is managed by a central registry like a domain name.

This is to be base64url encoded in JSON.

3.6.2. IEEE Based

The IEEE operates a global registry for MAC addresses and company IDs. This claim uses that database to identify OEMs. The contents of the claim may be either an IEEE MA-L, MA-M, MA-S or an IEEE CID [IEEE.RA]. An MA-L, formerly known as an OUI, is a 24-bit value used as the first half of a MAC address. MA-M similarly is a 28-bit value uses as the first part of a MAC address, and MA-S, formerly known as OUI-36, a 36-bit value. Many companies already have purchased one of these. A CID is also a 24-bit value from the same space as an MA-L, but not for use as a MAC address. IEEE has published Guidelines for Use of EUI, OUI, and CID [OUI.Guide] and provides a lookup services [OUI.Lookup].

Companies that have more than one of these IDs or MAC address blocks should pick one and prefer that for all their devices.

Commonly, these are expressed in Hexadecimal Representation [IEEE.802-2001] also called the Canonical format. When this claim is encoded the order of bytes in the bstr are the same as the order in the Hexadecimal Representation. For example, an MA-L like "AC-DE-48" would be encoded in 3 bytes with values 0xAC, 0xDE, 0x48. For JSON encoded tokens, this is further base64url encoded.

This format is always 3 bytes in size in CBOR.

3.6.3. IANA Private Enterprise Number

IANA maintains a simple integer-based company registry called the Private Enterprise Number (PEN) [PEN].

PENs are often used to create an OID. That is not the case here. They are used only as a simple integer.

In CBOR this is encoded as a major type 0 integer in CBOR and is typically 3 bytes. It is encoded as a number in JSON.

```
oemid-pen = int

oemid-ieee = bstr .size 3

oemid-random = bstr .size 16


$$claims-set-claims //= (
    oemid-label =>
        oemid-random / oemid-ieee / oemid-pen
)
```

3.7. Hardware Version Claims (hardware-version-claims)

The hardware version can be claimed at three different levels, the chip, the circuit board and the final device assembly. An EAT can include any combination these claims.

The hardware version is a simple text string the format of which is set by each manufacturer. The structure and sorting order of this text string can be specified using the version-scheme item from CoSWID [CoSWID].

The hardware version can also be given by a 13-digit [EAN-13]. A new CoSWID version scheme is registered with IANA by this document in Section 9.3.3. An EAN-13 is also known as an International Article Number or most commonly as a bar code.

```
$$claims-set-claims //=  (
    chip-version-label => hw-version-type
)

$$claims-set-claims //=  (
    board-version-label => hw-version-type
)

$$claims-set-claims //=  (
    device-version-label => hw-version-type
)

hw-version-type = [
    version:  tstr,
    scheme:  $version-scheme
]
```

3.8. Software Name Claim

This is a simple free-form text claim for the name of the software. A CoSWID manifest or other type of manifest can be used instead if this is too simple.

```
$$claims-set-claims //= ( sw-name-label => tstr )
```

3.9. Software Version Claim

This makes use of the CoSWID version scheme data type to give a simple version for the software. A full CoSWID manifest or other type of manifest can be instead if this is too simple.

```
$$claims-set-claims //= (sw-version-label => sw-version-type)

sw-version-type = [
    version:  tstr,
    scheme:  $version-scheme / As defined by CoSWID /
]
```

3.10. The Security Level Claim (security-level)

This claim characterizes the device/entity ability to defend against attacks aimed at capturing the signing key, forging claims and at forging EATs. This is by defining four security levels as described below.

These claims describe security environment and countermeasures available on the end-entity/client device where the attestation key resides and the claims originate.

1 - Unrestricted: There is some expectation that implementor will protect the attestation signing keys at this level. Otherwise, the EAT provides no meaningful security assurances.

2 - Restricted: Entities at this level are not general-purpose operating environments that host features such as app download systems, web browsers and complex productivity applications. It is akin to the secure-restricted level (see below) without the security orientation. Examples include a Wi-Fi subsystem, an IoT camera, or sensor device. Often these can be considered more secure than unrestricted just because they are much simpler and a smaller attack surface, but this won't always be the case. Some unrestricted devices may be implemented in a way that provides poor protection of signing keys.

3 - Secure-Restricted: Entities at this level must meet the criteria defined in section 4 of FIDO Allowed Restricted Operating Environments [FIDO.AROE]. Examples include TEE's and schemes using virtualization-based security. Like the FIDO security goal, security at this level is aimed at defending well against large-scale network/remote attacks against the device.

4 - Hardware:

Entities at this level must include substantial defense against physical or electrical attacks against the device itself. It is assumed any potential attacker has captured the device and can disassemble it. Examples include TPMs and Secure Elements.

The entity should claim the highest security level it achieves and no higher. This set is not extensible so as to provide a common interoperable description of security level to the Relying Party. If a particular implementation considers this claim to be inadequate, it can define its own proprietary claim. It may consider including both this claim as a coarse indication of security and its own proprietary claim as a refined indication.

This claim is not intended as a replacement for a proper end-device security certification scheme such as those based on FIPS 140 [FIPS-140] or those based on Common Criteria [Common.Criteria]. The claim made here is solely a self-claim made by the Attester.

```
$$claims-set-claims //= (
    security-level-label =>
        security-level-cbor-type /
        security-level-json-type
)

security-level-cbor-type = &(
    unrestricted: 1,
    restricted: 2,
    secure-restricted: 3,
    hardware: 4
)

security-level-json-type =
    "unrestricted" /
    "restricted" /
    "secure-restricted" /
    "hardware"
```

3.11. Secure Boot Claim (secure-boot)

The value of true indicates secure boot is enabled. Secure boot is considered enabled when base software, the firmware and operating system, are under control of the entity manufacturer identified in the OEMID claim described in Section 3.6. This may because the software is in ROM or because it is cryptographically authenticated or some combination of the two or other.

```
$$claims-set-claims //= (secure-boot-label => bool)
```

3.12. Debug Status Claim (debug-status)

This applies to system-wide or submodule-wide debug facilities of the target device / submodule like JTAG and diagnostic hardware built into chips. It applies to any software debug facilities related to root, operating system or privileged software that allow system-wide memory inspection, tracing or modification of non-system software like user mode applications.

This characterization assumes that debug facilities can be enabled and disabled in a dynamic way or be disabled in some permanent way such that no enabling is possible. An example of dynamic enabling is one where some authentication is required to enable debugging. An example of permanent disabling is blowing a hardware fuse in a chip. The specific type of the mechanism is not taken into account. For example, it does not matter if authentication is by a global password or by per-device public keys.

As with all claims, the absence of the debug level claim means it is not reported. A conservative interpretation might assume the Not Disabled state. It could however be that it is reported in a proprietary claim.

This claim is not extensible so as to provide a common interoperable description of debug status to the Relying Party. If a particular implementation considers this claim to be inadequate, it can define its own proprietary claim. It may consider including both this claim as a coarse indication of debug status and its own proprietary claim as a refined indication.

The higher levels of debug disabling requires that all debug disabling of the levels below it be in effect. Since the lowest level requires that all of the target's debug be currently disabled, all other levels require that too.

There is no inheritance of claims from a submodule to a superior module or vice versa. There is no assumption, requirement or guarantee that the target of a superior module encompasses the targets of submodules. Thus, every submodule must explicitly describe its own debug state. The Verifier or Relying Party receiving an EAT cannot assume that debug is turned off in a submodule because there is a claim indicating it is turned off in a superior module.

An individual target device / submodule may have multiple debug facilities. The use of plural in the description of the states refers to that, not to any aggregation or inheritance.

The architecture of some chips or devices may be such that a debug facility operates for the whole chip or device. If the EAT for such

a chip includes submodules, then each submodule should independently report the status of the whole-chip or whole-device debug facility. This is the only way the Relying Party can know the debug status of the submodules since there is no inheritance.

3.12.1. Enabled

If any debug facility, even manufacturer hardware diagnostics, is currently enabled, then this level must be indicated.

3.12.2. Disabled

This level indicates all debug facilities are currently disabled. It may be possible to enable them in the future, and it may also be possible that they were enabled in the past after the target device/sub-system booted/started, but they are currently disabled.

3.12.3. Disabled Since Boot

This level indicates all debug facilities are currently disabled and have been so since the target device/sub-system booted/started.

3.12.4. Disabled Permanently

This level indicates all non-manufacturer facilities are permanently disabled such that no end user or developer cannot enable them. Only the manufacturer indicated in the OEMID claim can enable them. This also indicates that all debug facilities are currently disabled and have been so since boot/start.

3.12.5. Disabled Fully and Permanently

This level indicates that all debug capabilities for the target device/sub-module are permanently disabled.

```
$$claims-set-claims //=  (
    debug-status-label =>
        debug-status-cbor-type / debug-status-json-type
)

debug-status-cbor-type = &(
    enabled: 0,
    disabled: 1,
    disabled-since-boot: 2,
    disabled-permanently: 3,
    disabled-fully-and-permanently: 4
)

debug-status-json-type =
    "enabled" /
    "disabled" /
    "disabled-since-boot" /
    "disabled-permanently" /
    "disabled-fully-and-permanently"
```

3.13. Including Keys

An EAT may include a cryptographic key such as a public key. The signing of the EAT binds the key to all the other claims in the token.

The purpose for inclusion of the key may vary by use case. For example, the key may be included as part of an IoT device onboarding protocol. When the FIDO protocol includes a pubic key in its attestation message, the key represents the binding of a user, device and Relying Party. This document describes how claims containing keys should be defined for the various use cases. It does not define specific claims for specific use cases.

Keys in CBOR format tokens SHOULD be the COSE_Key format [RFC8152] and keys in JSON format tokens SHOULD be the JSON Web Key format [RFC7517]. These two formats support many common key types. Their use avoids the need to decode other serialization formats. These two formats can be extended to support further key types through their IANA registries.

The general confirmation claim format [RFC8747], [RFC7800] may also be used. It provides key encryption. It also allows for inclusion by reference through a key ID. The confirmation claim format may employed in the definition of some new claim for a a particular use case.

When the actual confirmation claim is included in an EAT, this document associates no use case semantics other than proof of posession. Different EAT use cases may choose to associate further

semantics. The key in the confirmation claim MUST be protected the same as the key used to sign the EAT. That is, the same, equivalent or better hardware defenses, access controls, key generation and such must be used.

3.14. The Location Claim (location)

The location claim gives the location of the device entity from which the attestation originates. It is derived from the W3C Geolocation API [W3C.GeoLoc]. The latitude, longitude, altitude and accuracy must conform to [WGS84]. The altitude is in meters above the [WGS84] ellipsoid. The two accuracy values are positive numbers in meters. The heading is in degrees relative to true north. If the device is stationary, the heading is NaN (floating-point not-a-number). The speed is the horizontal component of the device velocity in meters per second.

When encoding floating-point numbers half-precision should not be used. It usually does not provide enough precision for a geographic location. It is not a requirement that the receiver of an EAT implement half-precision, so the receiver may not be able to decode the location.

The location may have been cached for a period of time before token creation. For example, it might have been minutes or hours or more since the last contact with a GPS satellite. Either the timestamp or age data item can be used to quantify the cached period. The timestamp data item is preferred as it a non-relative time.

The age data item can be used when the entity doesn't know what time it is either because it doesn't have a clock or it isn't set. The entity must still have a "ticker" that can measure a time interval. The age is the interval between acquisition of the location data and token creation.

See location-related privacy considerations in Section 10.2 below.

```
$$claims-set-claims //= (location-label => location-type)

location-type = {
    latitude => number,
    longitude => number,
    ? altitude => number,
    ? accuracy => number,
    ? altitude-accuracy => number,
    ? heading => number,
    ? speed => number,
    ? timestamp => ~time-int,
    ? age => uint
}

latitude = 1 / "latitude"
longitude = 2 / "longitude"
altitude = 3 / "altitude"
accuracy = 4 / "accuracy"
altitude-accuracy = 5 / "altitude-accuracy"
heading = 6 / "heading"
speed = 7 / "speed"
timestamp = 8 / "timestamp"
age = 9 / "age"
```

3.15. The Uptime Claim (uptime)

The "uptime" claim contains a value that represents the number of seconds that have elapsed since the entity or submod was last booted.

```
$$claims-set-claims //= (uptime-label => uint)
```

3.16. The Boot Seed Claim (boot-seed)

The Boot Seed claim is a random value created at system boot time that will allow differentiation of reports from different boot sessions. This value is usually public and not protected. It is not the same as a seed for a random number generator which must be kept secret.

```
$$claims-set-claims //=  (boot-seed-label => bytes)
```

3.17. The Intended Use Claim (intended-use)

EAT's may be used in the context of several different applications. The intended-use claim provides an indication to an EAT consumer about the intended usage of the token. This claim can be used as a way for an application using EAT to internally distinguish between different ways it uses EAT.

1 - Generic

Generic attestation describes an application where the EAT consumer requres the most up-to-date security assessment of the attesting entity. It is expected that this is the most commonly-used application of EAT.

2- Registration Entities that are registering for a new service may be expected to provide an attestation as part of the registration process. This intended-use setting indicates that the attestation is not intended for any use but registration.

3 - Provisioning Entities may be provisioned with different values or settings by an EAT consumer. Examples include key material or device management trees. The consumer may require an EAT to assess device security state of the entity prior to provisioning.

4 - Certificate Issuance (Certificate Signing Request) Certifying authorities (CA's) may require attestations prior to the issuance of certificates related to keypairs hosted at the entity. An EAT may be used as part of the certificate signing request (CSR).

5 - Proof-of-Possession An EAT consumer may require an attestation as part of an accompanying proof-of-possession (PoP) appication. More precisely, a PoP transaction is intended to provide to the recipient cryptographically-verifiable proof that the sender has posession of a key. This kind of attestation may be neceesary to verify the security state of the entity storing the private key used in a PoP application.

```
$$claims-set-claims //= (
    intended-use-label =>
        intended-use-cbor-type / intended-use-json-type
)

intended-use-cbor-type = &(
    generic: 1,
    registration: 2,
    provisioning: 3,
    csr: 4,
    pop: 5
)

intended-use-json-type =
    "generic" /
    "registration" /
    "provisioning" /
    "csr" /
    "pop"
```

3.18. The Profile Claim (profile)

See Section 7 for the detailed description of a profile.

A profile is identified by either a URL or an OID. Typically, the URI will reference a document describing the profile. An OID is just a unique identifier for the profile. It may exist anywhere in the OID tree. There is no requirement that the named document be publicly accessible. The primary purpose of the profile claim is to uniquely identify the profile even if it is a private profile.

The OID is encoded in CBOR according to [CBOR.OID] and the URI according to [RFC8949]. Both are unwrapped and thus not tags. The OID is always absolute and never relative. If the claims CBOR type is a text string it is a URI and if a byte string it is an OID.

Note that this named "eat_profile" for JWT and is distinct from the already registered "profile" claim in the JWT claims registry.

```
$$claims-set-claims //= (profile-label => ~uri / ~oid)

oid = #6.4000(bstr) ; TODO: Replace with CDDL from OID RFC
```

3.19. The DLOA (Digital Letter or Approval) Claim (dloas)

A DLOA (Digital Letter of Approval) [DLOA] is an XML document that describes a certification that a device or entity has received. Examples of certifications represented by a DLOA include those issued by Global Platform and those based on Common Criteria. The DLOA is unspecific to any particular certification type or those issued by any particular organization.

This claim is typically issued by a Verifier, not an Attester. When this claim is issued by a Verifier, it MUST be because the entity, device or submodule has received the certification in the DLOA.

This claim can contain more than one DLOA. If multiple DLOAs are present, it MUST be because the entity, device or submodule received all of the certifications.

DLOA XML documents are always fetched from a registrar that stores them. This claim contains several data items used to construct a URL for fetching the DLOA from the particular registrar.

The first data item is a URI for the registrar. The second data item is a platform label to indicate the particular platform that was certified. For platform certifications only these two are needed.

A DLOA may equally apply to an application. In that case it has the URI for the registrar, a platform label and additionally an application label.

The method of combining the registrar URI, platform label and possibly application label is specified in [DLOA].

```
$$claims-set-claims //= (
    dloas-label => [ + dloa-type ]
)

dloa-type = [
    dloa_registrar: ~uri
    dloa_platform_label: text
    ? dloa_application_label: text
]
```

3.20. The Software Manifests Claim (manifests)

This claim contains descriptions of software that is present on the device. These manifests are installed on the device when the software is installed or are created as part of the installation process. Installation is anything that adds software to the device, possibly factory installation, the user installing elective applications and so on. The defining characteristic is that they are created by the software manufacturer. The purpose of these claims in an EAT is to relay them without modification to the Verifier and/or the Relying Party.

In some cases these will be signed by the software manufacturer independent of any signing for the purpose of EAT attestation. Manifest claims should include the manufacturer's signature (which will be signed over by the attestation signature). In other cases the attestation signature will be the only one.

This claim allows multiple formats for the manifest. For example the manifest may be a CBOR-format CoSWID, an XML-format SWID or other. Identification of the type of manifest is always by a CBOR tag. In many cases, for examples CoSWID, a tag will already be registered with IANA. If not, a tag MUST be registered. It can be in the first-come-first-served space which has minimal requirements for registration.

The claim is an array of one or more manifests. To facilitate hand off of the manifest to a decoding library, each manifest is contained in a byte string. This occurs for CBOR-format manifests as well as non-CBOR format manifests.

If a particular manifest type uses CBOR encoding, then the item in the array for it MUST be a byte string that contains a CBOR tag. The EAT decoder must decode the byte string and then the CBOR within it to find the tag number to identify the type of manifest. The contents of the byte string is then handed to the particular manifest processor for that type of manifest. CoSWID and SUIT manifest are examples of this.

If a particular manifest type does not use CBOR encoding, then the item in the array for it must be a CBOR tag that contains a byte string. The EAT decoder uses the tag to identify the processor for that type of manifest. The contents of the tag, the byte string, are handed to the manifest processor. Note that a byte string is used to contain the manifest whether it is a text based format or not. An example of this is an XML format ISO/IEC 19770 SWID.

It is not possible to describe the above requirements in CDDL so the type for an individual manifest is any in the CDDL below. The above text sets the encoding requirement.

This claim allows for multiple manifests in one token since multiple software packages are likely to be present. The multiple manifests may be of multiple formats. In some cases EAT submodules may be used instead of the array structure in this claim for multiple manifests.

When the [CoSWID] format is used, it MUST be a payload CoSWID, not an evidence CoSWID.

```
$$claims-set-claims //= (
    manifests-label => manifests-type
)

manifests-type = [+ $$manifest-formats]

; Must be a CoSWID payload type
; TODO: signed CoSWIDs
coswid-that-is-a-cbor-tag-xx = tagged-coswid<concise-swid-tag>

$$manifest-formats /= bytes .cbor coswid-that-is-a-cbor-tag-xx

; TODO: make this work too
;$$manifest-formats /= bytes .cbor SUIT_Envelope_Tagged
```

3.21. The Software Evidence Claim (swevidence)

This claim contains descriptions, lists, evidence or measurements of the software that exists on the device. The defining characteristic of this claim is that its contents are created by processes on the device that inventory, measure or otherwise characterize the

software on the device. The contents of this claim do not originate from the software manufacturer.

In most cases the contents of this claim are signed as part of attestation signing, but independent signing in addition to the attestation signing is not ruled out when a particular evidence format supports it.

This claim uses the same mechanism for identification of the type of the swevidence as is used for the type of the manifest in the manifests claim. It also uses the same byte string based mechanism for containing the claim and easing the hand off to a processing library. See the discussion above in the manifests claim.

When the [CoSWID] format is used, it MUST be evidence CoSWIDs, not payload CoSWIDS.

```
$$claims-set-claims //= (
    swevidence-label => swevidence-type
)

swevidence-type = [+ $$swevidence-formats]

; Must be a CoSWID evidence type that is a CBOR tag
; TODO: fix the CDDL so a signed CoSWID is allowed too
coswid-that-is-a-cbor-tag = tagged-coswid<concise-swid-tag>
$$swevidence-formats /= bytes .cbor coswid-that-is-a-cbor-tag
```

3.22. The SW Measurement Results Claim (swresults)

This claims reports the outcome of the comparison of a measurement on some software to the expected Reference Values. It may report a successful comparison, failed comparison or other.

This claim may be generated by the Verifier and sent to the Relying Party. For example, it could be the results of the Verifier comparing the contents of the swevidence claim to Reference Values.

This claim can also be generated on the device if the device has the ability for one subsystem to measure another subsystem. For example, a TEE might have the ability to measure the software of the rich OS and may have the Reference Values for the rich OS.

Within an attestation target or submodule, multiple results can be reported. For example, it may be desirable to report the results for the kernel and each individual application separately.

For each software objective, the following can be reported.

3.22.1. Scheme

This is the free-form text name of the verification system or scheme that performed the verification. There is no official registry of schemes or systems. It may be the name of a commercial product or such.

3.22.2. Objective

This roughly characterizes the coverage of the software measurement software. This corresponds to the attestation target or the submodule. If all of the indicated target is not covered, the measurement must indicate partial.

1 - all Indicates all the software has been verified, for example, all the software in the attestation target or the submodule

2 - firmware Indicates all of and only the firmware

3 - kernel Refers to all of the most-privileged software, for example the Linux kernel

4 - privileged Refers to all of the software used by the root, system or administrative account

5 - system-libs Refers to all of the system libraries that are broadly shared and used by applications and such

6 - partial Some other partial set of the software

3.22.3. Results

This describes the result of the measurement and also the comparison to Reference Values.

1 - verificaton-not-run
Indicates no attempt was made to run the verification

2 - verification-indeterminite The verification was attempted, but it did not produce a result; perhaps it ran out of memory, the battery died or such

3 - verification-failed The verification ran to completion, the comparison was completed and did not compare correctly to the Reference Values

4 - fully-verified The verification ran to completion and all measurements compared correctly to Reference Values

5 - partially-verified The verification ran to completion and some, but not all measurements compared correctly to Reference Values

3.22.4. Objective Name

This is a free-form text string that describes the objective. For example, "Linux kernel" or "Facebook App"

```
$$claims-set-claims //= (swresults-label => [ + swresult-type ])

verification-result-cbor-type = &(
    verification-not-run: 1,
    verification-indeterminate: 2,
    verification-failed: 3,
    fully-verified: 4,
    partially-verified: 5,
)

verification-result-json-type =
    "verification-not-run" /
    "verification-indeterminate" /
    "verification-failed" /
    "fully-verified" /
    "partially-verified"


verification-objective-cbor-type = &(
    all: 1,
    firmware: 2,
    kernel: 3,
    privileged: 4,
    system-libs: 5,
    partial: 6,
)

verification-objective-json-type =
    "all" /
    "firmware" /
    "kernel" /
    "privileged" /
    "system-libs" /
    "partial"


swresult-type = [
    verification-system: tstr,
    objective: verification-objective-cbor-type /
        verification-objective-json-type,
    result: verification-result-cbor-type /
        verification-result-json-type,
    ? objective-name: tstr
]
```

3.23. Submodules (submods)

Some devices are complex, having many subsystems. A mobile phone is a good example. It may have several connectivity subsystems for communications (e.g., Wi-Fi and cellular). It may have subsystems

for low-power audio and video playback. It may have one or more security-oriented subsystems like a TEE or a Secure Element.

The claims for a subsystem can be grouped together in a submodule or submod.

The submods are in a single map/object, one entry per submodule. There is only one submods map/object in a token. It is identified by its specific label. It is a peer to other claims, but it is not called a claim because it is a container for a claims set rather than an individual claim. This submods part of a token allows what might be called recursion. It allows claims sets inside of claims sets inside of claims sets...

3.23.1. Submodule Types

The following sections define the three major types of submodules:

- *A submodule Claims-Set
- *A nested token, which can be any valid EAT token, CBOR or JSON
- *The digest of a detached Claims-Set

These are distinguished primarily by their data type which may be a map/object, string or array.

3.23.1.1. Submodule Claims-Set

This is simply a subordinate Claims-Set containing claims about the submodule.

The submodule claims-set is produced by the same Attester as the surrounding token. It is secured using the same mechanism as the enclosing token (e.g., it is signed by the same attestation key). It roughly corresponds to an Attester Target Environment as described in the RATS architecture.

It may contain claims that are the same as its surrounding token or superior submodules. For example, the top-level of the token may have a UEID, a submod may have a different UEID and a further subordinate submodule may also have a UEID.

The encoding of a submodule Claims-Set is always the same as the encoding as the token it is part of.

This data type for this type of submodule is a map/object as that is the type of a Claims-Set.

3.23.1.2. Nested Token

This type of submodule is a fully formed complete token. It is typically produced by a separate Attester. It is typically used by a Composite Device as described in RATS Architecture [RATS.Architecture]

In being a submodule of the surrounding token, it is cryptographically bound to the surrounding token. If it was conveyed in parallel with the surrounding token, there would be no such binding and attackers could substitute a good attestation from another device for the attestation of an errant subsystem.

A nested token does NOT need to use the same encoding as the enclosing token. This is to allow Composite Devices to be built without regards to the encoding supported by their Attesters.

Thus a CBOR-encoded token like a CWT or UCCS can have a JWT as a nested token submodule and a JSON-encoded token can have a CWT or UCCS as a nested token submodule.

The data type for this type of submodule is either a text or byte string.

Mechanisms are defined for identifying the encoding and type of the nested token. These mechanisms are different for CBOR and JSON encoding. The type of a CBOR-encoded nested token is identified using the CBOR tagging mechanism and thus is in common with identification used when any CBOR-encoded token is part of a CBOR-based protocol. A new simple type mechanism is defined for indication of the type of a JSON-encoded token since there is no JSON equivalent of tagging.

3.23.1.2.1. Surrounding EAT is CBOR-Encoded

If the submodule is a byte string, then the nested token is CBOR-encoded. The byte string always wraps a token that is a tag. The tag identifies whether the nested token is a CWT, a UCCS or a CBOR-encoded DEB.

If the submodule is a text string, then the nested token is JSON-encoded. The text string contains JSON. That JSON is the exactly the JSON described in the next section with one exception. The token can't be CBOR-encoded.

```
; This specifies how one fully-formed token is nested inside a
; CBOR-format token.  The fully-formed nested token is any valid
; token, CBOR or JSON (JWT, CWT, UCCS, DEB...)  The mechanism for
; identifying the type of the nested token is specific to the format
; of the surrounding token, CBOR in this case.
;
; A primary reason this is encoding-specific is that JSON does not
; have an equivalent to CBOR tags.
;
; If the data type here is text, then the nested token is JSON
; format, one of a JWT, UJCS or JSON-encoded DEB. The means for
; distinguishing which is in the definition of JSON-encoded
; Nested-Token.  If the data type is bstr, then the nested token
; is CBOR format. It is byte-string wrapped and identified by a
;CBOR tag.

Nested-Token =
    tstr / ; A JSON-encoded Nested-Token (see json-nested-token.cddl)
    bstr .cbor Tagged-CBOR-Token
```

3.23.1.2.2. Surrounding EAT is JSON-Encoded

A nested token in a JSON-encoded token is an array of two items. The first is a string that indicates the type of the second item as follows:

"JWT" A JWT formatted according to [RFC7519]

"CBOR" Some base64url-encoded CBOR that is a tag that is either a CWT, UCCS or CBOR-encoded DEB

"UJCS" A UJCS-Message. (A UJCS-Message is identical to a JSON-encoded Claims-Set)

"DEB" A JSON-encoded Detached EAT Bundle.

```
; This describes a nested token that occurs inside a JSON-encoded
; token. It uses an array that is made up of a type indicator and the
; actual token.  This is a substitute for the CBOR tag mechanism that
; JSON does not have.

Nested-Token = [
   type : "JWT" / "CBOR" / "UJCS" / "DEB",
   nested-token : JWT-Message /
                  B64URL-Tagged-CBOR-Token /
                  DEB-JSON-Message /
                  UJCS-Message
]


; This text is a Tagged-CBOR-Token (see cbor-token.cddl) that is
; base64url encoded.  For example, it is a CWT that is a COSE_Sign1
; that is a CBOR tag that has been base64url encoded.

B64URL-Tagged-CBOR-Token = tstr .regexp "[A-Za-z0-9_=-]+"
```

3.23.1.3. Detached Submodule Digest

This is type of submodule equivalent to a Claims-Set submodule, except the Claims-Set is conveyed separately outside of the token.

This type of submodule consists of a digest made using a cryptographic hash of a Claims-Set. The Claims-Set is not included in the token. It is conveyed to the Verifier outside of the token. The submodule containing the digest is called a detached digest. The separately conveyed Claims-Set is called a detached claims set.

The input to the digest is exactly the byte-string wrapped encoded form of the Claims-Set for the submodule. That Claims-Set can include other submodules including nested tokens and detached digests.

The primary use for this is to facilitate the implementation of a small and secure attester, perhaps purely in hardware. This small, secure attester implements COSE signing and only a few claims, perhaps just UEID and hardware identification. It has inputs for digests of submodules, perhaps 32-byte hardware registers. Software running on the device constructs larger claim sets, perhaps very large, encodes them and digests them. The digests are written into the small secure attesters registers. The EAT produced by the small secure attester only contains the UEID, hardware identification and digests and is thus simple enough to be implemented in hardware. Probably, every data item in it is of fixed length.

The integrity protection for the larger Claims Sets will not be as secure as those originating in hardware block, but the key material

and hardware-based claims will be. It is possible for the hardware to enforce hardware access control (memory protection) on the digest registers so that some of the larger claims can be more secure. For example, one register may be writable only by the TEE, so the detached claims from the TEE will have TEE-level security.

The data type for this type of submodule is an array It contains two data items, an algorithm identifier and a byte string containing the digest.

A DEB, described in Section 5, may be used to convey detached claims sets and the token with their detached digests. EAT, however, doesn't require use of a DEB. Any other protocols may be used to convey detached claims sets and the token with their detached digests. Note that since detached Claims-Sets are usually signed, protocols conveying them must make sure they are not modified in transit.

3.23.2. No Inheritance

The subordinate modules do not inherit anything from the containing token. The subordinate modules must explicitly include all of their claims. This is the case even for claims like the nonce.

This rule is in place for simplicity. It avoids complex inheritance rules that might vary from one type of claim to another.

3.23.3. Security Levels

The security level of the non-token subordinate modules should always be less than or equal to that of the containing modules in the case of non-token submodules. It makes no sense for a module of lesser security to be signing claims of a module of higher security. An example of this is a TEE signing claims made by the non-TEE parts (e.g. the high-level OS) of the device.

The opposite may be true for the nested tokens. They usually have their own more secure key material. An example of this is an embedded secure element.

3.23.4. Submodule Names

The label or name for each submodule in the submods map is a text string naming the submodule. No submodules may have the same name.

3.23.5. CDDL for submods

```
; This is the part of a token that contains all the submodules.  It
; is a peer with the claims in the token, but not a claim, only a
; map/object to hold all the submodules.

$$claims-set-claims //= (submods-label => { + text => Submodule })


; A submodule can be:
; - A simple Claims-Set (encoded in the same format as the token)
; - A digest of a detached Claims-Set (encoded in the same format as
;    the token)
; - A nested token which may be either CBOR or JSON format. Further,
;   the mechanism for identifying and containing the nested token
;   depends on the format of the surrounding token, particularly
;   because JSON doesn't have any equivalent of a CBOR tag so a
;   JSON-specific mechanism is invented. Also, there is the issue
;   that binary data must be B64 encoded when carried in
;   JSON. Nested-Token is defined in the format specific CDDL, not
;   here.

; Note that at nested token can either be a signed token like a CWT
; or JWT, an unsigned token like a UCCS or UJCS, or a DEB (detached
; EAT bundle).  The specific encoding of these is format-specific
; so it doesn't appear here.

Submodule = Claims-Set / Nested-Token / Detached-Submodule-Digest


; This is for both JSON and CBOR.  JSON uses text label for
; algorithm from JOSE registry. CBOR uses integer label for
; algorithm from COSE registry. In JSON the digest is base64
; encoded.

Detached-Submodule-Digest = [
   algorithm : int / text,
   digest : bstr
]
```

4. Unprotected JWT Claims-Sets

This is simply the JSON equivalent of an Unprotected CWT Claims-Set [UCCS.Draft].

It has no protection of its own so protections must be provided by the protocol carrying it. These are extensively discussed in

[UCCS.Draft]. All the security discussion and security considerations in [UCCS.Draft] apply to UJCS.

(Note: The EAT author is open to this definition being moved into the UCCS draft, perhaps along with the related CDDL. It is place here for now so that the current UCCS draft plus this document are complete. UJCS is needed for the same use cases that a UCCS is needed. Further, JSON will commonly be used to convey Attestation Results since JSON is common for server to server communications. Server to server communications will often have established security (e.g., TLS) therefore the signing and encryption from JWS and JWE are unnecssary and burdensome).

5. Detached EAT Bundles

A detached EAT bundle is a structure to convey a fully-formed and signed token plus detached claims set that relate to that token. It is a top-level EAT message like a CWT, JWT, UCCS and UJCS. It can be used any place that CWT, JWT, UCCS or UJCS messages are used. It may also be sent as a submodule.

A DEB has two main parts.

The first part is a full top-level token. This top-level token must have at least one submodule that is a detached digest. This top-level token may be either CBOR or JSON-encoded. It may be a CWT, JWT, UCCS or UJCS, but not a DEB. The same mechanism for distinguishing the type for nested token submodules is used here.

The second part is a map/object containing the detached Claims-Sets corresponding to the detached digests in the full token. When the DEB is CBOR-encoded, each Claims-Set is wrapped in a byte string. When the DEB is JSON-encoded, each Claims-Set is base64url encoded. All the detached Claims-Sets MUST be encoded in the same format as the DEB. No mixing of encoding formats is allowed for the Claims-Sets in a DEB.

For CBOR-encoded DEBs, tag TBD602 can be used to identify it. The normal rules apply for use or non-use of a tag. When it is sent as a submodule, it is always sent as a tag to distinguish it from the other types of nested tokens.

The digests of the detached claims sets are associated with detached claims-sets by label/name. It is up to the constructor of the detached EAT bundle to ensure the names uniquely identify the detached claims sets. Since the names are used only in the detached EAT bundle, they can be very short, perhaps one byte.

```
; Top-level definition of a DEB for CBOR and JSON

Detached-EAT-Bundle = [
    main-token : Nested-Token,
    detached-claims-sets: {
        + tstr => cbor-wrapped-claims-set / json-wrapped-claims-set
    }
]


; text content is a base64url encoded JSON-format Claims-Set

json-wrapped-claims-set = tstr .regexp "[A-Za-z0-9_=-]+"


cbor-wrapped-claims-set = bstr .cbor Claims-Set
```

6. Endorsements and Verification Keys

The Verifier must possess the correct key when it performs the cryptographic part of an EAT verification (e.g., verifying the COSE/JOSE signature). This section describes several ways to identify the verification key. There is not one standard method.

The verification key itself may be a public key, a symmetric key or something complicated in the case of a scheme like Direct Anonymous Attestation (DAA).

RATS Architecture [RATS.Architecture] describes what is called an Endorsement. This is an input to the Verifier that is usually the basis of the trust placed in an EAT and the Attester that generated it. It may contain the public key for verification of the signature on the EAT. It may contain Reference Values to which EAT claims are compared as part of the verification process. It may contain implied claims, those that are passed on to the Relying Party in Attestation Results.

There is not yet any standard format(s) for an Endorsement. One format that may be used for an Endorsement is an X.509 certificate. Endorsement data like Reference Values and implied claims can be carried in X.509 v3 extensions. In this use, the public key in the X.509 certificate becomes the verification key, so identification of the Endorsement is also identification of the verification key.

The verification key identification and establishment of trust in the EAT and the attester may also be by some other means than an Endorsement.

For the components (Attester, Verifier, Relying Party,...) of a particular end-end attestation system to reliably interoperate, its definition should specify how the verification key is identified. Usually, this will be in the profile document for a particular attestation system.

6.1. Identification Methods

Following is a list of possible methods of key identification. A specific attestation system may employ any one of these or one not listed here.

The following assumes Endorsements are X.509 certificates or equivalent and thus does not mention or define any identifier for Endorsements in other formats. If such an Endorsement format is created, new identifiers for them will also need to be created.

6.1.1. COSE/JWS Key ID

The COSE standard header parameter for Key ID (kid) may be used. See [RFC8152] and [RFC7515]

COSE leaves the semantics of the key ID open-ended. It could be a record locator in a database, a hash of a public key, an input to a KDF, an authority key identifier (AKI) for an X.509 certificate or other. The profile document should specify what the key ID's semantics are.

6.1.2. JWS and COSE X.509 Header Parameters

COSE X.509 [COSE.X509.Draft] and JSON Web Siganture [RFC7515] define several header parameters (x5t, x5u,...) for referencing or carrying X.509 certificates any of which may be used.

The X.509 certificate may be an Endorsement and thus carrying additional input to the Verifier. It may be just an X.509 certificate, not an Endorsement. The same header parameters are used in both cases. It is up to the attestation system design and the Verifier to determine which.

6.1.3. CBOR Certificate COSE Header Parameters

Compressed X.509 and CBOR Native certificates are defined by CBOR Certificates [CBOR.Cert.Draft]. These are semantically compatible with X.509 and therefore can be used as an equivalent to X.509 as described above.

These are identified by their own header parameters (c5t, c5u,...).

6.1.4. Claim-Based Key Identification

For some attestation systems, a claim may be re-used as a key identifier. For example, the UEID uniquely identifies the device and therefore can work well as a key identifier or Endorsement identifier.

This has the advantage that key identification requires no additional bytes in the EAT and makes the EAT smaller.

This has the disadvantage that the unverified EAT must be substantially decoded to obtain the identifier since the identifier is in the COSE/JOSE payload, not in the headers.

6.2. Other Considerations

In all cases there must be some way that the verification key is itself verified or determined to be trustworthy. The key identification itself is never enough. This will always be by some out-of-band mechanism that is not described here. For example, the Verifier may be configured with a root certificate or a master key by the Verifier system administrator.

Often an X.509 certificate or an Endorsement carries more than just the verification key. For example, an X.509 certificate might have key usage constraints and an Endorsement might have Reference Values. When this is the case, the key identifier must be either a protected header or in the payload such that it is cryptographically bound to the EAT. This is in line with the requirements in section 6 on Key Identification in JSON Web Signature [RFC7515].

7. Profiles

This EAT specification does not gaurantee that implementations of it will interoperate. The variability in this specification is necessary to accommodate the widely varying use cases. An EAT profile narrows the specification for a specific use case. An ideal EAT profile will guarantee interoperability.

The profile can be named in the token using the profile claim described in Section 3.18.

A profile can apply to Attestation Evidence or to Attestation Results or both.

7.1. Format of a Profile Document

A profile document doesn't have to be in any particular format. It may be simple text, something more formal or a combination.

In some cases CDDL may be created that replaces CDDL in this or other document to express some profile requirements. For example, to require the altitude data item in the location claim, CDDL can be written that replicates the location claim with the altitude no longer optional.

7.2. List of Profile Issues

The following is a list of EAT, CWT, UCCS, JWS, UJCS, COSE, JOSE and CBOR options that a profile should address.

7.2.1. Use of JSON, CBOR or both

The profile should indicate whether the token format should be CBOR, JSON, both or even some other encoding. If some other encoding, a specification for how the CDDL described here is serialized in that encoding is necessary.

This should be addressed for the top-level token and for any nested tokens. For example, a profile might require all nested tokens to be of the same encoding of the top level token.

7.2.2. CBOR Map and Array Encoding

The profile should indicate whether definite-length arrays/maps, indefinite-length arrays/maps or both are allowed. A good default is to allow only definite-length arrays/maps.

An alternate is to allow both definite and indefinite-length arrays/maps. The decoder should accept either. Encoders that need to fit on very small hardware or be actually implement in hardware can use indefinite-length encoding.

This applies to individual EAT claims, CWT and COSE parts of the implementation.

7.2.3. CBOR String Encoding

The profile should indicate whether definite-length strings, indefinite-length strings or both are allowed. A good default is to allow only definite-length strings. As with map and array encoding, allowing indefinite-length strings can be beneficial for some smaller implementations.

7.2.4. CBOR Preferred Serialization

The profile should indicate whether encoders must use preferred serialization. The profile should indicate whether decoders must accept non-preferred serialization.

7.2.5. COSE/JOSE Protection

COSE and JOSE have several options for signed, MACed and encrypted messages. EAT/CWT has the option to have no protection using UCCS and JOSE has a NULL protection option. It is possible to implement no protection, sign only, MAC only, sign then encrypt and so on. All combinations allowed by COSE, JOSE, JWT, CWT, UCCS and UJCS are allowed by EAT.

The profile should list the protections that must be supported by all decoders implementing the profile. The encoders them must implement a subset of what is listed for the decoders, perhaps only one.

Implementations may choose to sign or MAC before encryption so that the implementation layer doing the signing or MACing can be the smallest. It is often easier to make smaller implementations more secure, perhaps even implementing in solely in hardware. The key material for a signature or MAC is a private key, while for encryption it is likely to be a public key. The key for encryption requires less protection.

7.2.6. COSE/JOSE Algorithms

The profile document should list the COSE algorithms that a Verifier must implement. The Attester will select one of them. Since there is no negotiation, the Verifier should implement all algorithms listed in the profile. If detached submodules are used, the COSE algorithms allowed for their digests should also be in the profile.

7.2.7. DEB Support

A Detatched EAT Bundle Section 5 is a special case message that will not often be used. A profile may prohibit its use.

7.2.8. Verification Key Identification

Section Section 6 describes a number of methods for identifying a verification key. The profile document should specify one of these or one that is not described. The ones described in this document are only roughly described. The profile document should go into the full detail.

7.2.9. Endorsement Identification

Similar to, or perhaps the same as Verification Key Identification, the profile may wish to specify how Endorsements are to be identified. However note that Endorsement Identification is optional, where as key identification is not.

7.2.10. Freshness

Just about every use case will require some means of knowing the EAT is recent enough and not a replay of an old token. The profile should describe how freshness is achieved. The section on Freshness in [RATS.Architecture] describes some of the possible solutions to achieve this.

7.2.11. Required Claims

The profile can list claims whose absence results in Verification failure.

7.2.12. Prohibited Claims

The profile can list claims whose presence results in Verification failure.

7.2.13. Additional Claims

The profile may describe entirely new claims. These claims can be required or optional.

7.2.14. Refined Claim Definition

The profile may lock down optional aspects of individual claims. For example, it may require altitude in the location claim, or it may require that HW Versions always be described using EAN-13.

7.2.15. CBOR Tags

The profile should specify whether the token should be a CWT Tag or not. Similarly, the profile should specify whether the token should be a UCCS tag or not.

When COSE protection is used, the profile should specify whether COSE tags are used or not. Note that RFC 8392 requires COSE tags be used in a CWT tag.

Often a tag is unncessary because the surrounding or carrying protocol identifies the object as an EAT.

7.2.16. Manifests and Software Evidence Claims

The profile should specify which formats are allowed for the manifests and software evidence claims. The profile may also go on to say which parts and options of these formats are used, allowed and prohibited.

8. Encoding and Collected CDDL

An EAT is fundamentally defined using CDDL. This document specifies how to encode the CDDL in CBOR or JSON. Since CBOR can express some things that JSON can't (e.g., tags) or that are expressed differently (e.g., labels) there is some CDDL that is specific to the encoding format.

8.1. Claims-Set and CDDL for CWT and JWT

CDDL was not used to define CWT or JWT. It was not available at the time.

This document defines CDDL for both CWT and JWT as well as UCCS. This document does not change the encoding or semantics of anything in a CWT or JWT.

A Claims-Set is the central data structure for EAT, CWT, JWT and UCCS. It holds all the claims and is the structure that is secured by signing or other means. It is not possible to define EAT, CWT, JWT or UCCS in CDDL without it. The CDDL definition of Claims-Set here is applicable to EAT, CWT, JWT and UCCS.

This document specifies how to encode a Claims-Set in CBOR or JSON.

With the exception of nested tokens and some other externally defined structures (e.g., SWIDs) an entire Claims-Set must be in encoded in either CBOR or JSON, never a mixture.

CDDL for the seven claims defined by [RFC8392] and [RFC7519] is included here.

8.2. Encoding Data Types

This makes use of the types defined in [RFC8610] Appendix D, Standard Prelude.

8.2.1. Common Data Types

time-int is identical to the epoch-based time, but disallows floating-point representation.

Unless expliclity indicated, URIs are not the URI tag defined in [RFC8949]. They are just text strings that contain a URI.

```
string-or-uri = tstr

time-int = #6.1(int)
```

8.2.2. JSON Interoperability

JSON should be encoded per [RFC8610] Appendix E. In addition, the following CDDL types are encoded in JSON as follows:

- *bstr - must be base64url encoded
- *time - must be encoded as NumericDate as described section 2 of [RFC7519].
- *string-or-uri - must be encoded as StringOrURI as described section 2 of [RFC7519].
- *uri - must be a URI [RFC3986].
- *oid - encoded as a string using the well established dotted-decimal notation (e.g., the text "1.2.250.1").

8.2.3. Labels

Map labels, including Claims-Keys and Claim-Names, and enumerated-type values are always integers when encoding in CBOR and strings when encoding in JSON. There is an exception to this for naming submodules and detached claims sets in a DEB. These are strings in CBOR.

The CDDL in most cases gives both the integer label and the string label as it is not convenient to have conditional CDDL for such.

8.3. CBOR Interoperability

CBOR allows data items to be serialized in more than one form. If the sender uses a form that the receiver can't decode, there will not be interoperability.

This specification gives no blanket requirements to narrow CBOR serialization for all uses of EAT. This allows individual uses to tailor serialization to the environment. It also may result in EAT implementations that don't interoperate.

One way to guarantee interoperability is to clearly specify CBOR serialization in a profile document. See Section 7 for a list of serialization issues that should be addressed.

EAT will be commonly used where the device generating the attestation is constrained and the receiver/Verifier of the attestation is a capacious server. Following is a set of serialization requirements that work well for that use case and are guaranteed to interoperate. Use of this serialization is recommended

where possible, but not required. An EAT profile may just reference the following section rather than spell out serialization details.

8.3.1. EAT Constrained Device Serialization

- Preferred serialization described in section 4.1 of [RFC8949] is not required. The EAT decoder must accept all forms of number serialization. The EAT encoder may use any form it wishes.

- The EAT decoder must accept indefinite length arrays and maps as described in section 3.2.2 of [RFC8949]. The EAT encoder may use indefinite length arrays and maps if it wishes.

- The EAT decoder must accept indefinite length strings as described in section 3.2.3 of [RFC8949]. The EAT encoder may use indefinite length strings if it wishes.

- Sorting of maps by key is not required. The EAT decoder must not rely on sorting.

- Deterministic encoding described in Section 4.2 of [RFC8949] is not required.

- Basic validity described in section 5.3.1 of [RFC8949] must be followed. The EAT encoder must not send duplicate map keys/labels or invalid UTF-8 strings.

8.4. Collected Common CDDL

```
; This is the fundamental definition of a Claims-Set for both CBOR
; and JSON. It is a set of label-value pairs each of which is a
; claim.
;
; In CBOR the labels can be integers or strings with a strong
; preference for integers.  For JSON, the labels are always strings.
;
; The values can be anything, with some consideration for types that
; can work in both CBOR and JSON.

Claims-Set = {
    * $$claims-set-claims,
    * Claim-Label .feature "extended-label" => any
}

Claim-Label = int / text
string-or-uri = tstr

time-int = #6.1(int)
; This is CDDL for the 7 individual claims that are defined in CWT
; and JWT.  This CDDL works for either CBOR format CWT or JSON format
; JWT The integer format CWT Claim Keys (the labels) are defined in
; cwt-labels.cddl.  The string format JWT Claim Names (the labels)
; are defined in jwt-labels.cddl.

; $$claims-set-claims is defined in claims-set.cddl

$$claims-set-claims //= (iss-label => text)
$$claims-set-claims //= (sub-label => text)
$$claims-set-claims //= (aud-label => text)
$$claims-set-claims //= (exp-label => ~time)
$$claims-set-claims //= (nbf-label => ~time)
$$claims-set-claims //= (iat-label => ~time)

; TODO: how does the bstr get handled in JSON validation with the
; cddl tool?  TODO: should this be a text for JSON?
; $$claims-set-claims //= (cti-label : bytes)
$$claims-set-claims //=
    (nonce-label => nonce-type / [ 2* nonce-type ])

nonce-type = bstr .size (8..64)

$$claims-set-claims //= (ueid-label => ueid-type)

ueid-type = bstr .size (7..33)
$$claims-set-claims //= (sueids-label => sueids-type)

sueids-type = {
    + tstr => ueid-type
}
```

```
oemid-pen = int

oemid-ieee = bstr .size 3

oemid-random = bstr .size 16


$$claims-set-claims //= (
    oemid-label =>
        oemid-random / oemid-ieee / oemid-pen
)
$$claims-set-claims //=  (
    chip-version-label => hw-version-type
)

$$claims-set-claims //=  (
    board-version-label => hw-version-type
)

$$claims-set-claims //=  (
    device-version-label => hw-version-type
)

hw-version-type = [
    version:  tstr,
    scheme:  $version-scheme
]
$$claims-set-claims //= ( sw-name-label => tstr )

$$claims-set-claims //= (
    security-level-label =>
        security-level-cbor-type /
        security-level-json-type
)

security-level-cbor-type = &(
    unrestricted: 1,
    restricted: 2,
    secure-restricted: 3,
    hardware: 4
)

security-level-json-type =
    "unrestricted" /
    "restricted" /
    "secure-restricted" /
    "hardware"
$$claims-set-claims //= (secure-boot-label => bool)
$$claims-set-claims //=  (
```

```
    debug-status-label =
        debug-status-cbor-type / debug-status-json-type
)

debug-status-cbor-type = &(
    enabled: 0,
    disabled: 1,
    disabled-since-boot: 2,
    disabled-permanently: 3,
    disabled-fully-and-permanently: 4
)

debug-status-json-type =
    "enabled" /
    "disabled" /
    "disabled-since-boot" /
    "disabled-permanently" /
    "disabled-fully-and-permanently"
$$claims-set-claims //= (location-label => location-type)

location-type = {
    latitude => number,
    longitude => number,
    ? altitude => number,
    ? accuracy => number,
    ? altitude-accuracy => number,
    ? heading => number,
    ? speed => number,
    ? timestamp => ~time-int,
    ? age => uint
}

latitude = 1 / "latitude"
longitude = 2 / "longitude"
altitude = 3 / "altitude"
accuracy = 4 / "accuracy"
altitude-accuracy = 5 / "altitude-accuracy"
heading = 6 / "heading"
speed = 7 / "speed"
timestamp = 8 / "timestamp"
age = 9 / "age"

$$claims-set-claims //= (uptime-label => uint)
$$claims-set-claims //=  (boot-seed-label => bytes)
$$claims-set-claims //= (
    intended-use-label =
        intended-use-cbor-type / intended-use-json-type
)
```

```
intended-use-cbor-type = &(
    generic: 1,
    registration: 2,
    provisioning: 3,
    csr: 4,
    pop: 5
)

intended-use-json-type =
    "generic" /
    "registration" /
    "provisioning" /
    "csr" /
    "pop"

$$claims-set-claims //= (
    dloas-label => [ + dloa-type ]
)

dloa-type = [
    dloa_registrar: ~uri
    dloa_platform_label: text
    ? dloa_application_label: text
]


$$claims-set-claims //= (profile-label => ~uri / ~oid)

oid = #6.4000(bstr) ; TODO: Replace with CDDL from OID RFC

$$claims-set-claims //= (
    manifests-label => manifests-type
)

manifests-type = [+ $$manifest-formats]

; Must be a CoSWID payload type
; TODO: signed CoSWIDs
coswid-that-is-a-cbor-tag-xx = tagged-coswid<concise-swid-tag>

$$manifest-formats /= bytes .cbor coswid-that-is-a-cbor-tag-xx

; TODO: make this work too
;$$manifest-formats /= bytes .cbor SUIT_Envelope_Tagged

$$claims-set-claims //= (
    swevidence-label => swevidence-type
)

swevidence-type = [+ $$swevidence-formats]
```

```
; Must be a CoSWID evidence type that is a CBOR tag
; TODO: fix the CDDL so a signed CoSWID is allowed too
coswid-that-is-a-cbor-tag = tagged-coswid<concise-swid-tag>
$$swevidence-formats /= bytes .cbor coswid-that-is-a-cbor-tag


$$claims-set-claims //= (swresults-label => [ + swresult-type ])

verification-result-cbor-type = &(
    verification-not-run: 1,
    verification-indeterminate: 2,
    verification-failed: 3,
    fully-verified: 4,
    partially-verified: 5,
)

verification-result-json-type =
    "verification-not-run" /
    "verification-indeterminate" /
    "verification-failed" /
    "fully-verified" /
    "partially-verified"


verification-objective-cbor-type = &(
    all: 1,
    firmware: 2,
    kernel: 3,
    privileged: 4,
    system-libs: 5,
    partial: 6,
)

verification-objective-json-type =
    "all" /
    "firmware" /
    "kernel" /
    "privileged" /
    "system-libs" /
    "partial"


swresult-type = [
    verification-system: tstr,
    objective: verification-objective-cbor-type /
        verification-objective-json-type,
    result: verification-result-cbor-type /
        verification-result-json-type,
    ? objective-name: tstr
```

```
]
; This is the part of a token that contains all the submodules.  It
; is a peer with the claims in the token, but not a claim, only a
; map/object to hold all the submodules.

$$claims-set-claims //= (submods-label => { + text => Submodule })


; A submodule can be:
; - A simple Claims-Set (encoded in the same format as the token)
; - A digest of a detached Claims-Set (encoded in the same format as
;    the token)
; - A nested token which may be either CBOR or JSON format. Further,
;   the mechanism for identifying and containing the nested token
;   depends on the format of the surrounding token, particularly
;   because JSON doesn't have any equivalent of a CBOR tag so a
;   JSON-specific mechanism is invented. Also, there is the issue
;   that binary data must be B64 encoded when carried in
;   JSON. Nested-Token is defined in the format specific CDDL, not
;   here.

; Note that at nested token can either be a signed token like a CWT
; or JWT, an unsigned token like a UCCS or UJCS, or a DEB (detached
; EAT bundle).  The specific encoding of these is format-specific
; so it doesn't appear here.

Submodule = Claims-Set / Nested-Token / Detached-Submodule-Digest


; This is for both JSON and CBOR.  JSON uses text label for
; algorithm from JOSE registry. CBOR uses integer label for
; algorithm from COSE registry. In JSON the digest is base64
; encoded.

Detached-Submodule-Digest = [
   algorithm : int / text,
   digest : bstr
]


; Top-level definition of a DEB for CBOR and JSON

Detached-EAT-Bundle = [
    main-token : Nested-Token,
    detached-claims-sets: {
        + tstr => cbor-wrapped-claims-set / json-wrapped-claims-set
    }
]
```

```
; text content is a base64url encoded JSON-format Claims-Set

json-wrapped-claims-set = tstr .regexp "[A-Za-z0-9_=-]+"


cbor-wrapped-claims-set = bstr .cbor Claims-Set
```

8.5. Collected CDDL for CBOR

```
; The top-level definition of a CBOR-encoded token.

CBOR-Token = Tagged-CBOR-Token / Untagged-CBOR-Token


; All forms of a CBOR-encoded token that are a CBOR tag.

Tagged-CBOR-Token  = CWT-Tagged-Message
Tagged-CBOR-Token /= UCCS-Tagged-Message
Tagged-CBOR-Token /= DEB-Tagged-Message


; All forms of a CBOR-encoded token that are not a CBOR tag.

Untagged-CBOR-Token  = CWT-Untagged-Message
Untagged-CBOR-Token /= UCCS-Untagged-Message
Untagged-CBOR-Token /= DEB-Untagged-Message


; The payload of the COSE message is always a Claims-Set

CWT-Tagged-Message = COSE_Tagged_Message
CWT-Untagged-Message = COSE_Untagged_Message


UCCS-Message = UCCS-Tagged-Message / UCCS-Untagged-Message

UCCS-Tagged-Message = #6.601(UCCS-Untagged-Message)

UCCS-Untagged-Message = Claims-Set


DEB-Tagged-Message = #6.602(DEB-Untagged-Message)

DEB-Untagged-Message = Detached-EAT-Bundle


; This specifies how one fully-formed token is nested inside a
; CBOR-format token.  The fully-formed nested token is any valid
; token, CBOR or JSON (JWT, CWT, UCCS, DEB...)  The mechanism for
; identifying the type of the nested token is specific to the format
; of the surrounding token, CBOR in this case.
;
; A primary reason this is encoding-specific is that JSON does not
; have an equivalent to CBOR tags.
;
; If the data type here is text, then the nested token is JSON
; format, one of a JWT, UJCS or JSON-encoded DEB. The means for
; distinguishing which is in the definition of JSON-encoded
; Nested-Token.  If the data type is bstr, then the nested token
```

```
; is CBOR format. It is byte-string wrapped and identified by a
;CBOR tag.

Nested-Token =
    tstr / ; A JSON-encoded Nested-Token (see json-nested-token.cddl)
    bstr .cbor Tagged-CBOR-Token


; This is the CDDL definition of the labels for a CBOR format web
; token, a CWT.  The CDDL for the claims is in web-token-claims.cddl

iss-label = 1
sub-label = 2
aud-label = 3
exp-label = 4
nbf-label = 5
iat-label = 6
cti-label = 7; The following Claim Keys (labels) are pre-assigned by IAN
; They are for CBOR-based tokens (CWT and UCCS).
; They are not expected to change in the final publication as an RFC.

nonce-label = 10
ueid-label = 11
oemid-label = 13
security-level-label = 14
secure-boot-label = 15
debug-status-label = 16
location-label = 17
profile-label = 18
submods-label = 20

; These are not yet assigned in any way and may change.
; These are intentionally above 24 so as to not use up
; single-byte labels.

sueids-label = <TBD25>
chip-version-label = <TBD26>
board-version-label = <TBD27>
device-version-label = <TBD28>
sw-name-label = <TBD29>
sw-version-label = <TBD30>
uptime-label = <TBD31>
boot-seed-label = <TBD32>
intended-use-label = <TBD33>
dloas-label = <TBD34>
manifests-label = <TBD35>
swevidence-label = <TBD36>
swresults-label = <TBD37>
```

8.6. Collected CDDL for JSON

```
; A JWT message is either a JWS or JWE in compact serialization form
; with the payload a Claims-Set. Compact serialization is the
; protected headers, payload and signature, each b64url encoded and
; separated by a ".". This CDDL simply matches top-level syntax of of
; a JWS or JWE since it is not possible to do more in CDDL.

JWT-Message = text .regexp [A-Za-z0-9_=-]+\.[A-Za-z0-9_=-]+\.[A-Za-z0-9_


; This defines the JSON equivalent of a UCCS message, a token with
; no integrity or authenticity protection.

UJCS-Message = Claims-Set

; This describes a nested token that occurs inside a JSON-encoded
; token. It uses an array that is made up of a type indicator and the
; actual token.  This is a substitute for the CBOR tag mechanism that
; JSON does not have.

Nested-Token = [
   type : "JWT" / "CBOR" / "UJCS" / "DEB",
   nested-token : JWT-Message /
                  B64URL-Tagged-CBOR-Token /
                  DEB-JSON-Message /
                  UJCS-Message
]


; This text is a Tagged-CBOR-Token (see cbor-token.cddl) that is
; base64url encoded.  For example, it is a CWT that is a COSE_Sign1
; that is a CBOR tag that has been base64url encoded.

B64URL-Tagged-CBOR-Token = tstr .regexp "[A-Za-z0-9_=-]+"
; This is the CDDL definition of the labels for a JSON format web
; token, a JWT.  The CDDL for the claims is in web-token-claims.cddl

iss-label = "iss"
sub-label = "sub"
aud-label = "aud"
exp-label = "exp"
nbf-label = "nbf"
iat-label = "iat"
cti-label = "cti"; The following are claim names for JSON encoded tokens

ueid-label /= "ueid"
sueids-label /= "sueids"
nonce-label /= "nonce"
oemid-label /= "oemid"
security-level-label /= "seclevel"
secure-boot-label /= "secboot"
```

```
debug-status-label /= "dbgstat"
location-label /= "location"
uptime-label /= "uptime"
profile-label /= "eat-profile"
intended-use-label /= "intuse"
boot-seed-label /= "bootseed"
submods-label /= "submods"
timestamp /= "timestamp"
manifests-label /= "manifests"
swevidence-label /= "swevidence"
dloas-label /= "dloas"
swresults-label /= "swresults"
sw-name-label /= "swname"
sw-version-label /= "swversion"

latitude /= "lat"
longitude /= "long"
altitude /= "alt"
accuracy /= "accry"
altitude-accuracy /= "alt-accry"
heading /= "heading"
speed /= "speed"
```

9. IANA Considerations

9.1. Reuse of CBOR and JSON Web Token (CWT and JWT) Claims Registries

Claims defined for EAT are compatible with those of CWT and JWT so the CWT and JWT Claims Registries, [IANA.CWT.Claims] and [IANA.JWT.Claims], are re used. No new IANA registry is created.

All EAT claims defined in this document are placed in both registries. All new EAT claims defined subsequently should be placed in both registries.

9.2. Claim Characteristics

The following is design guidance for creating new EAT claims, particularly those to be registered with IANA.

Much of this guidance is generic and could also be considered when designing new CWT or JWT claims.

9.2.1. Interoperability and Relying Party Orientation

It is a broad goal that EATs can be processed by Relying Parties in a general way regardless of the type, manufacturer or technology of the device from which they originate. It is a goal that there be general-purpose verification implementations that can verify tokens for large numbers of use cases with special cases and configurations

for different device types. This is a goal of interoperability of the semantics of claims themselves, not just of the signing, encoding and serialization formats.

This is a lofty goal and difficult to achieve broadly requiring careful definition of claims in a technology neutral way. Sometimes it will be difficult to design a claim that can represent the semantics of data from very different device types. However, the goal remains even when difficult.

9.2.2. Operating System and Technology Neutral

Claims should be defined such that they are not specific to an operating system. They should be applicable to multiple large high-level operating systems from different vendors. They should also be applicable to multiple small embedded operating systems from multiple vendors and everything in between.

Claims should not be defined such that they are specific to a SW environment or programming language.

Claims should not be defined such that they are specific to a chip or particular hardware. For example, they should not just be the contents of some HW status register as it is unlikely that the same HW status register with the same bits exists on a chip of a different manufacturer.

The boot and debug state claims in this document are an example of a claim that has been defined in this neutral way.

9.2.3. Security Level Neutral

Many use cases will have EATs generated by some of the most secure hardware and software that exists. Secure Elements and smart cards are examples of this. However, EAT is intended for use in low-security use cases the same as high-security use case. For example, an app on a mobile device may generate EATs on its own.

Claims should be defined and registered on the basis of whether they are useful and interoperable, not based on security level. In particular, there should be no exclusion of claims because they are just used only in low-security environments.

9.2.4. Reuse of Extant Data Formats

Where possible, claims should use already standardized data items, identifiers and formats. This takes advantage of the expertise put into creating those formats and improves interoperability.

Often extant claims will not be defined in an encoding or serialization format used by EAT. It is preferred to define a CBOR and JSON format for them so that EAT implementations do not require a plethora of encoders and decoders for serialization formats.

In some cases, it may be better to use the encoding and serialization as is. For example, signed X.509 certificates and CRLs can be carried as-is in a byte string. This retains interoperability with the extensive infrastructure for creating and processing X.509 certificates and CRLs.

9.2.5. Proprietary Claims

EAT allows the definition and use of proprietary claims.

For example, a device manufacturer may generate a token with proprietary claims intended only for verification by a service offered by that device manufacturer. This is a supported use case.

In many cases proprietary claims will be the easiest and most obvious way to proceed, however for better interoperability, use of general standardized claims is preferred.

9.3. Claims Registered by This Document

This specification adds the following values to the "JSON Web Token Claims" registry established by [RFC7519] and the "CBOR Web Token Claims Registry" established by [RFC8392]. Each entry below is an addition to both registries (except for the nonce claim which is already registered for JWT, but not registered for CWT).

The "Claim Description", "Change Controller" and "Specification Documents" are common and equivalent for the JWT and CWT registries. The "Claim Key" and "Claim Value Types(s)" are for the CWT registry only. The "Claim Name" is as defined for the CWT registry, not the JWT registry. The "JWT Claim Name" is equivalent to the "Claim Name" in the JWT registry.

9.3.1. Claims for Early Assignment

RFC Editor: in the final publication this section should be combined with the following section as it will no longer be necessary to distinguish claims with early assignment. Also, the following paragraph should be removed.

The claims in this section have been (requested for / given) early assignment according to [RFC7120]. They have been assigned values and registered before final publication of this document. While their semantics is not expected to change in final publication, it

is possible that they will. The JWT Claim Names and CWT Claim Keys are not expected to change.

*Claim Name: Nonce

*Claim Description: Nonce

*JWT Claim Name: "nonce" (already registered for JWT)

*Claim Key: 10

*Claim Value Type(s): byte string

*Change Controller: IESG

*Specification Document(s): [OpenIDConnectCore], **this document**

*Claim Name: UEID

*Claim Description: The Universal Entity ID

*JWT Claim Name: "ueid"

*CWT Claim Key: 11

*Claim Value Type(s): byte string

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: OEMID

*Claim Description: IEEE-based OEM ID

*JWT Claim Name: "oemid"

*Claim Key: 13

*Claim Value Type(s): byte string

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: Security Level

*Claim Description: Characterization of the security of an Attester or submodule

*JWT Claim Name: "seclevel"

*Claim Key: 14

*Claim Value Type(s): integer

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: Secure Boot

*Claim Description: Indicate whether the boot was secure

*JWT Claim Name: "secboot"

*Claim Key: 15

*Claim Value Type(s): Boolean

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: Debug Status

*Claim Description: Indicate status of debug facilities

*JWT Claim Name: "dbgstat"

*Claim Key: 16

*Claim Value Type(s): integer

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: Location

*Claim Description: The geographic location

*JWT Claim Name: "location"

*Claim Key: 17

*Claim Value Type(s): map

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: Profile

*Claim Description: Indicates the EAT profile followed

*JWT Claim Name: "eat_profile"

*Claim Key: 18

*Claim Value Type(s): map

*Change Controller: IESG

*Specification Document(s): **this document**

*Claim Name: Submodules Section

*Claim Description: The section containing submodules (not actually a claim)

*JWT Claim Name: "submods"

*Claim Key: 20

*Claim Value Type(s): map

*Change Controller: IESG

*Specification Document(s): **this document**

9.3.2. To be Assigned Claims

TODO: add the rest of the claims in here

9.3.3. Version Schemes Registered by this Document

IANA is requested to register a new value in the "Software Tag Version Scheme Values" established by [CoSWID].

The new value is a version scheme a 13-digit European Article Number [EAN-13]. An EAN-13 is also known as an International Article Number or most commonly as a bar code. This version scheme is the ASCII text representation of EAN-13 digits, the same ones often printed with a bar code. This version scheme must comply with the EAN allocation and assignment rules. For example, this requires the manufacturer to obtain a manufacture code from GS1.

Index	Version Scheme Name	Specification
5	ean-13	This document

Table 2

9.3.4. UEID URN Registered by this Document

IANA is requested to register the following new subtypes in the "DEV URN Subtypes" registry under "Device Identification". See [RFC9039].

Subtype	Description	Reference
ueid	Universal Entity Identifier	This document
sueid	Semi-permanent Universal Entity Identifier	This document

Table 3

9.3.5. Tag for Detached EAT Bundle

In the registry [IANA.cbor-tags], IANA is requested to allocate the following tag from the FCFS space, with the present document as the specification reference.

Tag	Data Items	Semantics
TBD602	array	Detached EAT Bundle Section 5

Table 4

10. Privacy Considerations

Certain EAT claims can be used to track the owner of an entity and therefore, implementations should consider providing privacy-preserving options dependent on the intended usage of the EAT. Examples would include suppression of location claims for EAT's provided to unauthenticated consumers.

10.1. UEID and SUEID Privacy Considerations

A UEID is usually not privacy-preserving. Any set of Relying Parties that receives tokens that happen to be from a single device will be able to know the tokens are all from the same device and be able to track the device. Thus, in many usage situations UEID violates governmental privacy regulation. In other usage situations a UEID will not be allowed for certain products like browsers that give privacy for the end user. It will often be the case that tokens will not have a UEID for these reasons.

An SUEID is also usually not privacy-preserving. In some cases it may have fewer privacy issues than a UEID depending on when and how and when it is generated.

There are several strategies that can be used to still be able to put UEIDs and SUEIDs in tokens:

* The device obtains explicit permission from the user of the device to use the UEID/SUEID. This may be through a prompt. It may also be through a license agreement. For example, agreements

for some online banking and brokerage services might already cover use of a UEID/SUEID.

* The UEID/SUEID is used only in a particular context or particular use case. It is used only by one Relying Party.

* The device authenticates the Relying Party and generates a derived UEID/SUEID just for that particular Relying Party. For example, the Relying Party could prove their identity cryptographically to the device, then the device generates a UEID just for that Relying Party by hashing a proofed Relying Party ID with the main device UEID/SUEID.

Note that some of these privacy preservation strategies result in multiple UEIDs and SUEIDs per device. Each UEID/SUEID is used in a different context, use case or system on the device. However, from the view of the Relying Party, there is just one UEID and it is still globally universal across manufacturers.

10.2. Location Privacy Considerations

Geographic location is most always considered personally identifiable information. Implementers should consider laws and regulations governing the transmission of location data from end user devices to servers and services. Implementers should consider using location management facilities offered by the operating system on the device generating the attestation. For example, many mobile phones prompt the user for permission when before sending location data.

11. Security Considerations

The security considerations provided in Section 8 of [RFC8392] and Section 11 of [RFC7519] apply to EAT in its CWT and JWT form, respectively. In addition, implementors should consider the following.

11.1. Key Provisioning

Private key material can be used to sign and/or encrypt the EAT, or can be used to derive the keys used for signing and/or encryption. In some instances, the manufacturer of the entity may create the key material separately and provision the key material in the entity itself. The manfuacturer of any entity that is capable of producing an EAT should take care to ensure that any private key material be suitably protected prior to provisioning the key material in the entity itself. This can require creation of key material in an enclave (see [RFC4949] for definition of "enclave"), secure transmission of the key material from the enclave to the entity using an appropriate protocol, and persistence of the private key

material in some form of secure storage to which (preferably) only the entity has access.

11.1.1. Transmission of Key Material

Regarding transmission of key material from the enclave to the entity, the key material may pass through one or more intermediaries. Therefore some form of protection ("key wrapping") may be necessary. The transmission itself may be performed electronically, but can also be done by human courier. In the latter case, there should be minimal to no exposure of the key material to the human (e.g. encrypted portable memory). Moreover, the human should transport the key material directly from the secure enclave where it was created to a destination secure enclave where it can be provisioned.

11.2. Transport Security

As stated in Section 8 of [RFC8392], "The security of the CWT relies upon on the protections offered by COSE". Similar considerations apply to EAT when sent as a CWT. However, EAT introduces the concept of a nonce to protect against replay. Since an EAT may be created by an entity that may not support the same type of transport security as the consumer of the EAT, intermediaries may be required to bridge communications between the entity and consumer. As a result, it is RECOMMENDED that both the consumer create a nonce, and the entity leverage the nonce along with COSE mechanisms for encryption and/or signing to create the EAT.

Similar considerations apply to the use of EAT as a JWT. Although the security of a JWT leverages the JSON Web Encryption (JWE) and JSON Web Signature (JWS) specifications, it is still recommended to make use of the EAT nonce.

11.3. Multiple EAT Consumers

In many cases, more than one EAT consumer may be required to fully verify the entity attestation. Examples include individual consumers for nested EATs, or consumers for individual claims with an EAT. When multiple consumers are required for verification of an EAT, it is important to minimize information exposure to each consumer. In addition, the communication between multiple consumers should be secure.

For instance, consider the example of an encrypted and signed EAT with multiple claims. A consumer may receive the EAT (denoted as the "receiving consumer"), decrypt its payload, verify its signature, but then pass specific subsets of claims to other consumers for evaluation ("downstream consumers"). Since any COSE encryption will be removed by the receiving consumer, the communication of claim

subsets to any downstream consumer should leverage a secure protocol (e.g.one that uses transport-layer security, i.e. TLS),

However, assume the EAT of the previous example is hierarchical and each claim subset for a downstream consumer is created in the form of a nested EAT. Then transport security between the receiving and downstream consumers is not strictly required. Nevertheless, downstream consumers of a nested EAT should provide a nonce unique to the EAT they are consuming.

12. References

12.1. Normative References

[CBOR.OID] Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", Work in Progress, Internet-Draft, draft-ietf-cbor-tags-oid-08, 21 May 2021, <https://www.ietf.org/archive/id/draft-ietf-cbor-tags-oid-08.txt>.

[CoSWID] Birkholz, H., Fitzgerald-McKay, J., Schmidt, C., and D. Waltermire, "Concise Software Identification Tags", Work in Progress, Internet-Draft, draft-ietf-sacm-coswid-19, 20 October 2021, <https://www.ietf.org/archive/id/draft-ietf-sacm-coswid-19.txt>.

[DLOA] "Digital Letter of Approval", November 2015, <https://globalplatform.org/wp-content/uploads/2015/12/GPC_DigitalLetterOfApproval_v1.0.pdf>.

[EAN-13] GS1, "International Article Number - EAN/UPC barcodes", 2019, <https://www.gs1.org/standards/barcodes/ean-upc>.

[FIDO.AROE] The FIDO Alliance, "FIDO Authenticator Allowed Restricted Operating Environments List", November 2020, <https://fidoalliance.org/specs/fido-security-requirements/fido-authenticator-allowed-restricted-operating-environments-list-v1.2-fd-20201102.html>.

[IANA.cbor-tags] "IANA CBOR Tags Registry", n.d., <https://www.iana.org/assignments/cbor-tags/cbor-tags.xhtml>.

[IANA.CWT.Claims] IANA, "CBOR Web Token (CWT) Claims", <http://www.iana.org/assignments/cwt>.

[IANA.JWT.Claims] IANA, "JSON Web Token (JWT) Claims", <https://www.iana.org/assignments/jwt>.

[OpenIDConnectCore] Sakimura, N., Bradley, J., Jones, M., Medeiros, B. D., and C. Mortimore, "OpenID Connect Core 1.0

incorporating errata set 1", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[PEN] "Private Enterprise Number (PEN) Request", n.d., <https://pen.iana.org/pen/PenApplication.page>.

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

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

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

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

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

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[ThreeGPP.IMEI] 3GPP, "3rd Generation Partnership Project; Technical Specification Group Core Network and Terminals; Numbering, addressing and identification", 2019, <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=729>.

[UCCS.Draft] Birkholz, H., O'Donoghue, J., Cam-Winget, N., and C. Bormann, "A CBOR Tag for Unprotected CWT Claims Sets", Work in Progress, Internet-Draft, draft-ietf-rats-uccs-01, 12 July 2021, <https://www.ietf.org/archive/id/draft-ietf-rats-uccs-01.txt>.

[WGS84] National Geospatial-Intelligence Agency (NGA), "WORLD GEODETIC SYSTEM 1984, NGA.STND.0036_1.0.0_WGS84", 8 July 2014, <https://earth-info.nga.mil/php/download.php?file=coord-wgs84>.

12.2. Informative References

[BirthdayAttack] "Birthday attack", <https://en.wikipedia.org/wiki/Birthday_attack.>.

[CBOR.Cert.Draft] Mattsson, J. P., Selander, G., Raza, S., Höglund, J., and M. Furuhed, "CBOR Encoded X.509 Certificates (C509 Certificates)", Work in Progress, Internet-Draft, draft-ietf-cose-cbor-encoded-cert-02, 12 July 2021, <https://www.ietf.org/archive/id/draft-ietf-cose-cbor-encoded-cert-02.txt>.

[Common.Criteria] "Common Criteria for Information Technology Security Evaluation", April 2017, <https://www.commoncriteriaportal.org/cc/>.

[COSE.X509.Draft]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Header parameters for carrying and referencing X.509 certificates", Work in Progress, Internet-Draft, draft-ietf-cose-x509-08, 14 December 2020, <https://www.ietf.org/archive/id/draft-ietf-cose-x509-08.txt>.

[ECMAScript] "Ecma International, "ECMAScript Language Specification, 5.1 Edition", ECMA Standard 262", June 2011, <http://www.ecma-international.org/ecma-262/5.1/ECMA-262.pdf>.

[FIPS-140] National Institue of Standards, "Security Requirements for Cryptographic Modules", May 2001, <https://csrc.nist.gov/publications/detail/fips/140/2/final>.

[IEEE.802-2001] "IEEE Standard For Local And Metropolitan Area Networks Overview And Architecture", 2007, <https://webstore.ansi.org/standards/ieee/ieee8022001r2007>.

[IEEE.802.1AR] "IEEE Standard, "IEEE 802.1AR Secure Device Identifier"", December 2009, <http://standards.ieee.org/findstds/standard/802.1AR-2009.html>.

[IEEE.RA] "IEEE Registration Authority", <https://standards.ieee.org/products-services/regauth/index.html>.

[OUI.Guide] "Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)", August 2017, <https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/eui.pdf>.

[OUI.Lookup] "IEEE Registration Authority Assignments", <https://regauth.standards.ieee.org/standards-ra-web/pub/view.html#registries>.

[RATS.Architecture] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote Attestation Procedures Architecture", Work in Progress, Internet-Draft, draft-ietf-rats-architecture-12, 23 April 2021, <https://www.ietf.org/archive/id/draft-ietf-rats-architecture-12.txt>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI

10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, DOI 10.17487/RFC7120, January 2014, <https://www.rfc-editor.org/info/rfc7120>.

[RFC9039] Arkko, J., Jennings, C., and Z. Shelby, "Uniform Resource Names for Device Identifiers", RFC 9039, DOI 10.17487/RFC9039, June 2021, <https://www.rfc-editor.org/info/rfc9039>.

[W3C.GeoLoc] Worldwide Web Consortium, "Geolocation API Specification 2nd Edition", January 2018, <https://www.w3.org/TR/geolocation-API/#coordinates_interface>.

Appendix A. Examples

These examples are either UCCS, shown as CBOR diagnostic, or UJCS messages. Full CWT and JWT examples with signing and encryption are not given.

All UCCS examples can be the payload of a CWT. To do so, they must be converted from the UCCS message to a Claims-Set, which is achieve by "removing" the tag.

UJCS messages can be directly used as the payload of a JWT.

WARNING: These examples use tag and label numbers not yet assigned by IANA.

A.1. Simple TEE Attestation

This is a simple attestation of a TEE that includes a manifest that is a payload CoSWID to describe the TEE's software.

```
/ This is a UCCS EAT that describes a simple TEE. /

601({
    / nonce /           10: h'948f8860d13a463e',
    / security-level /  14: 3, / secure-restricted /
    / secure-boot /     15: true,
    / debug-status /    16: 2, / disabled-since-boot /
    / manfests /        35: [
                               / This is byte-string wrapped      /
                               / payload CoSWID. It gives the TEE /
                               / software name, the version and   /
                               / the  name of the file it is in.  /
                               h' da53574944a60064336132340c01016b
                                  41636d6520544545204f530d65332e31
                                  2e340282a2181f6b41636d6520544545
                                  204f53182101a2181f6b41636d652054
                                  4545204f5318210206a111a118186e61
                                  636d655f7465655f332e657865'
                           ]
})


/ A payload CoSWID created by the SW vendor. All this really does /
/ is name the TEE SW, its version and lists the one file that     /
/ makes up the TEE. /

1398229316({
    / Unique CoSWID ID /    0: "3a24",
    / tag-version /        12: 1,
    / software-name /       1: "Acme TEE OS",
    / software-version /   13: "3.1.4",
    / entity /              2: [
                                   {
        / entity-name /                31: "Acme TEE OS",
        / role        /                33: 1 / tag-creator /
                                   },
                                   {
        / entity-name /                31: "Acme TEE OS",
        / role        /                33: 2 / software-creator /
                                   }
                               ],
    / payload /                6: {
        / ...file /                17: {
            / ...fs-name /             24: "acme_tee_3.exe"
                                   }
                               }
})
```

A.2. EAT Produced by Attestation Hardware Block

```
/ This is an example of a token produced by a HW block           /
/ purpose-built for attestation.  Only the nonce claim changes   /
/ from one attestation to the next as the rest  either come      /
/ directly from the hardware or from one-time-programmable memory /
/ (e.g. a fuse). 47 bytes encoded in CBOR (8 byte nonce, 16 byte  /
/ UEID). /

601({
    / nonce /           10: h'948f8860d13a463e',
    / UEID /            11: h'0198f50a4ff6c05861c8860d13a638ea',
    / OEMID /           13: 64242, / Private Enterprise Number /
    / security-level /  14: 4, / hardware level security /
    / secure-boot /     15: true,
    / debug-status /    16: 3, / disabled-permanently /
    / chip-version /    26: [ "3.1", 1 ] / Type is multipartnumeric /
})
```

A.3. Detached EAT Bundle

In this DEB main token is produced by a HW attestation block. The detached Claims-Set is produced by a TEE and is largely identical to the Simple TEE examples above. The TEE digests its Claims-Set and feeds that digest to the HW block.

In a better example the attestation produced by the HW block would be a CWT and thus signed and secured by the HW block. Since the signature covers the digest from the TEE that Claims-Set is also secured.

The DEB itself can be assembled by untrusted SW.

```
/ This is a detached EAT bundle (DEB) tag.  /

602([

    / First part is a full EAT token with claims like nonce and /
    / UEID. Most importantly, it includes a submodule that is a /
    / detached digest which is the hash of the "TEE" claims set /
    / in the next section.                                      /
    /                                                          /
    / This token here is in UCCS format (unsigned). In a more   /
    / realistic example, it would be a signed CWT.              /
    h'd90259a80a48948f8860d13a463e0b500198f50a4ff6c058
      61c8860d13a638ea0d19faf20e040ff51003181a8263332e
      310114a163544545822f5820e5cf95fd24fab71446742dd5
      8d43dae178e55fe2b94291a9291082ffc2635a0b',

    {
       / A CBOR-encoded byte-string wrapped EAT claims-set. It /
       / contains claims suitable for a TEE                   /
       "TEE" : h'a50a48948f8860d13a463e0e030ff51002182381
                 585dda53574944a60064336132340c01016b4163
                 6d6520544545204f530d65332e312e340282a218
                 1f6b41636d6520544545204f53182101a2181f6b
                 41636d6520544545204f5318210206a111a11818
                 6e61636d655f7465655f332e657865'
    }
 ])
```

```
/ This example contains submodule that is a detached digest,   /
/ which is the hash of a Claims-Set convey outside this token. /
/ Other than that is is the other example of a token from an   /
/ attestation HW block                                         /

601({
    / nonce /           10: h'948f8860d13a463e',
    / UEID /            11: h'0198f50a4ff6c05861c8860d13a638ea',
    / OEMID /           13: 64242, / Private Enterprise Number /
    / security-level /  14: 4, / hardware level security /
    / secure-boot /     15: true,
    / debug-status /    16: 3, / disabled-permanently /
    / chip-version /    26: [ "3.1", 1 ], / multipartnumeric /
    / submods/          20: {
                                "TEE": [ / detached digest submod /
                                    -16, / SHA-256 /
                                    h'e5cf95fd24fab7144674
                                      2dd58d43dae178e55fe2
                                      b94291a9291082ffc2635
                                      a0b'
                                ]
                            }
})
```

A.4. Key / Key Store Attestation

```
/ This is an attestation of a public key and the key store     /
/ implementation that protects and manages it. The key store   /
/ implementation is in a security-oriented execution           /
/ environment separate from the high-level OS, for example a   /
/ TEE. The key store is the Attester.                          /
/                                                              /
/ There is some attestation of the high-level OS, just version /
/ and boot & debug status. It is a Claims-Set submodule because/
/ it has lower security level than the key store. The key      /
/ store's implementation has access to info about the HLOS, so /
/ it is able to include it.                                    /
/                                                              /
/ A key and an indication of the user authentication given to  /
/ allow access to the key is given. The labels for these are   /
/ in the private space since this is just a hypothetical       /
/ example, not part of a standard protocol.                    /
/                                                              /
/ This is similar to Android Key Attestation.                  /


601({
    / nonce /           10: h'948f8860d13a463e',
    / security-level /  14: 3, / secure-restricted /
    / debug-status /    16: 2, / disabled-since-boot /
    / secure-boot /     15: true,
    / manifests /       35: [
                                h'da53574944a600683762623334383766
                                  0c000169436172626f6e6974650d6331
                                  2e320e0102a2181f75496e6475737472
                                  69616c204175746f6d6174696f6e1821
                                  02'
                                / Above is an encoded CoSWID     /
                                / with the following data        /
                                /   SW Name: "Carbonite"         /
                                /   SW Vers: "1.2"               /
                                /   SW Creator:                  /
                                /      "Industrial Automation"   /
                            ],
    / expiration /       4: 1634324274, / 2021-10-15T18:57:54Z /
    / creation time /    6: 1634317080, / 2021-10-15T16:58:00Z /
                   -80000 : "fingerprint",
                   -80001 : { / The key -- A COSE_Key  /
                / kty /       1: 2, / EC2, eliptic curve with x & y /
                / kid /       2: h'36675c206f96236c3f51f54637b94ced',
                / curve /    -1: 2, / curve is P-256 /
                / x-coord /  -2: h'65eda5a12577c2bae829437fe338701a
                                   10aaa375e1bb5b5de108de439c08551d',
                / y-coord /  -3: h'1e52ed75701163f7f9e40ddf9f341b3d
                                   c9ba860af7e0ca7ca7e9eecd0084d19c'
```

```
            },

    / submods /        20 : {
                            "HLOS" : { / submod for high-level OS /
         / nonce /              10: h'948f8860d13a463e',
           / security-level /   14: 1, / unrestricted /
           / secure-boot /      15: true,
           / manifests /        35: [
                                     h'da53574944a600687337
                                       6537346b78380c000168
                                       44726f6964204f530d65
                                       52322e44320e0302a218
                                       1F75496E647573747269
                                       616c204175746f6d6174
                                       696f6e182102'
                                     / Above is an encoded CoSWID /
                                     / with the following data:   /
                                     /   SW Name: "Droid OS"      /
                                     /   SW Vers: "R2.D2"         /
                                     /   SW Creator:              /
                                     /     "Industrial Automation"/
                                ]
                            }
                        }
})
```

A.5. SW Measurements of an IoT Device

This is a simple token that might be for and IoT device. It includes CoSWID format measurments of the SW. The CoSWID is in byte-string wrapped in the token and also shown in diagnostic form.

```
/ This EAT UCCS is for an IoT device with a TEE. The attestation   /
/ is produced by the TEE. There is a submodule for the IoT OS (the /
/ main OS of the IoT device that is not as secure as the TEE). The /
/ submodule contains claims for the IoT OS. The TEE also measures  /
/ the IoT OS and puts the measurements in the submodule.           /

601({
    / nonce /           10: h'948f8860d13a463e',
    / security-level /  14: 3, / secure-restricted /
    / secure-boot /     15: true,
    / debug-status /    16: 2, / disabled-since-boot /
    / OEMID /           13: h'8945ad', / IEEE CID based /
    / UEID /            11: h'0198f50a4ff6c05861c8860d13a638ea',
    / sumods /          20: {
                            "OS" : {
        / security-level /      14: 2, / restricted /
        / secure-boot /         15: true,
        / debug-status /        16: 2, / disabled-since-boot /
        / swevidence /          36: [
                                    / This is a byte-string wrapped /
                                    / evidence CoSWID. It has       /
                                    / hashes of the main files of   /
                                    / the IoT OS.  /
                                    h'da53574944a600663463613234350c
                                      17016d41636d6520522d496f542d4f
                                      530d65332e312e3402a2181f724163
                                      6d652042617365204174746573746 5
                                      7218210103a11183a318187161636d
                                      655f725f696f745f6f732e65786514
                                      1a0044b349078201582005f6b327c1
                                      73b4192bd2c3ec248a292215eab456
                                      611bf7a783e25c1782479905a31818
                                      6d7265736f75726365732e72736314
                                      1a000c38b10782015820c142b9aba4
                                      280c4bb8c75f716a43c99526694caa
                                      be529571f5569bb7dc542f98a31818
                                      6a636f6d6d6f6e2e6c6962141a0023
                                      3d3b0782015820a6a9dcdfb3884da5
                                      f884e4e1e8e8629958c2dbc7027414
                                      43a913e34de9333be6'
                                ]
                            }
                        }
})
```

```
/ An evidence CoSWID created for the "Acme R-IoT-OS" created by /
/ the "Acme Base Attester" (both fictious names).  It provides  /
/ measurements of the SW (other than the attester SW) on the    /
/ device. /

1398229316({
    / Unique CoSWID ID /    0: "4ca245",
    / tag-version /        12: 23, / Attester-maintained counter /
    / software-name /       1: "Acme R-IoT-OS",
    / software-version /   13: "3.1.4",
    / entity /              2: {
        / entity-name /        31: "Acme Base Attester",
        / role        /        33: 1 / tag-creator /
                            },
    / evidence /            3: {
        / ...file /             17: [
                                    {
            / ...fs-name /              24: "acme_r_iot_os.exe",
            / ...size    /              20: 4502345,
            / ...hash    /               7: [
                                             1, / SHA-256 /
                                             h'05f6b327c173b419
                                               2bd2c3ec248a2922
                                               15eab456611bf7a7
                                               83e25c1782479905'
                                            ]
                                    },
                                    {
            / ...fs-name /              24: "resources.rsc",
            / ...size    /              20: 800945,
            / ...hash    /               7: [
                                              1, / SHA-256 /
                                             h'c142b9aba4280c4b
                                               b8c75f716a43c995
                                               26694caabe529571
                                               f5569bb7dc542f98'
                                            ]
                                    },
                                    {
            / ...fs-name /              24: "common.lib",
            / ...size    /              20: 2309435,
            / ...hash    /               7: [
                                             1, / SHA-256 /
                                             h'a6a9dcdfb3884da5
                                               f884e4e1e8e86299
                                               58c2dbc702741443
                                               a913e34de9333be6'
                                            ]
                                    }
```

```
                                ]
                            }
})
```

A.6. Attestation Results in JSON format

This is a UJCS format token that might be the output of a Verifier that evaluated the IoT Attestation example immediately above.

This particular Verifier knows enough about the TEE Attester to be able to pass claims like security level directly through to the Relying Party. The Verifier also knows the Reference Values for the measured SW components and is able to check them. It informs the Relying Party that they were correct in the swresults claim. "Trustus Verifications" is the name of the services that verifies the SW component measurements.

This UJCS is identical to JSON-encoded Claims-Set that could be a JWT payload.

```
{
    "nonce" : "lI+IYNE6Rj4=",
    "seclevel" : "secure-restricted",
    "secboot" : true,
    "dbgstat" : "disabled-since-boot",
    "OEMID" : "iUWt",
    "UEID" : "AZj1Ck/2wFhhyIYNE6Y4",
    "submods" : {
        "seclevel" : "restricted",
        "secboot" : true,
        "dbgstat" : "disabled-since-boot",
        "swname" : "Acme R-IoT-OS",
        "sw-version" : [
            "3.1.4"
        ],
        "swresults" : [
            [
                "Trustus Verifications",
                "all",
                "fully-verified"
            ]
       ]
}
```

Appendix B. UEID Design Rationale

B.1. Collision Probability

This calculation is to determine the probability of a collision of UEIDs given the total possible entity population and the number of entities in a particular entity management database.

Three different sized databases are considered. The number of devices per person roughly models non-personal devices such as traffic lights, devices in stores they shop in, facilities they work in and so on, even considering individual light bulbs. A device may have individually attested subsystems, for example parts of a car or a mobile phone. It is assumed that the largest database will have at most 10% of the world's population of devices. Note that databases that handle more than a trillion records exist today.

The trillion-record database size models an easy-to-imagine reality over the next decades. The quadrillion-record database is roughly at the limit of what is imaginable and should probably be accommodated. The 100 quadrillion datadbase is highly speculative perhaps involving nanorobots for every person, livestock animal and domesticated bird. It is included to round out the analysis.

Note that the items counted here certainly do not have IP address and are not individually connected to the network. They may be connected to internal buses, via serial links, Bluetooth and so on. This is not the same problem as sizing IP addresses.

People	Devices / Person	Subsystems / Device	Database Portion	Database Size
10 billion	100	10	10%	trillion (10^12)
10 billion	100,000	10	10%	quadrillion (10^15)
100 billion	1,000,000	10	10%	100 quadrillion (10^17)

Table 5

This is conceptually similar to the Birthday Problem where m is the number of possible birthdays, always 365, and k is the number of people. It is also conceptually similar to the Birthday Attack where collisions of the output of hash functions are considered.

The proper formula for the collision calculation is

```
p = 1 - e^{-k^2/(2n)}

p   Collision Probability
n   Total possible population
k   Actual population
```

However, for the very large values involved here, this formula requires floating point precision higher than commonly available in calculators and SW so this simple approximation is used. See [BirthdayAttack].

```
 p = k^2 / 2n
```

For this calculation:

```
 p  Collision Probability
 n  Total population based on number of bits in UEID
 k  Population in a database
```

Database Size	128-bit UEID	192-bit UEID	256-bit UEID
trillion (10^12)	2 * 10^-15	8 * 10^-35	5 * 10^-55
quadrillion (10^15)	2 * 10^-09	8 * 10^-29	5 * 10^-49
100 quadrillion (10^17)	2 * 10^-05	8 * 10^-25	5 * 10^-45

Table 6

Next, to calculate the probability of a collision occurring in one year's operation of a database, it is assumed that the database size is in a steady state and that 10% of the database changes per year. For example, a trillion record database would have 100 billion states per year. Each of those states has the above calculated probability of a collision.

This assumption is a worst-case since it assumes that each state of the database is completely independent from the previous state. In reality this is unlikely as state changes will be the addition or deletion of a few records.

The following tables gives the time interval until there is a probability of a collision based on there being one tenth the number of states per year as the number of records in the database.

```
t = 1 / ((k / 10) * p)

t  Time until a collision
p  Collision probability for UEID size
k  Database size
```

Database Size	128-bit UEID	192-bit UEID	256-bit UEID
trillion (10^12)	60,000 years	10^24 years	10^44 years

Database Size	128-bit UEID	192-bit UEID	256-bit UEID
quadrillion (10^15)	8 seconds	10^14 years	10^34 years
100 quadrillion (10^17)	8 microseconds	10^11 years	10^31 years

Table 7

Clearly, 128 bits is enough for the near future thus the requirement that UEIDs be a minimum of 128 bits.

There is no requirement for 256 bits today as quadrillion-record databases are not expected in the near future and because this time-to-collision calculation is a very worst case. A future update of the standard may increase the requirement to 256 bits, so there is a requirement that implementations be able to receive 256-bit UEIDs.

B.2. No Use of UUID

A UEID is not a UUID [RFC4122] by conscious choice for the following reasons.

UUIDs are limited to 128 bits which may not be enough for some future use cases.

Today, cryptographic-quality random numbers are available from common CPUs and hardware. This hardware was introduced between 2010 and 2015. Operating systems and cryptographic libraries give access to this hardware. Consequently, there is little need for implementations to construct such random values from multiple sources on their own.

Version 4 UUIDs do allow for use of such cryptographic-quality random numbers, but do so by mapping into the overall UUID structure of time and clock values. This structure is of no value here yet adds complexity. It also slightly reduces the number of actual bits with entropy.

UUIDs seem to have been designed for scenarios where the implementor does not have full control over the environment and uniqueness has to be constructed from identifiers at hand. UEID takes the view that hardware, software and/or manufacturing process directly implement UEID in a simple and direct way. It takes the view that cryptographic quality random number generators are readily available as they are implemented in commonly used CPU hardware.

Appendix C. EAT Relation to IEEE.802.1AR Secure Device Identity (DevID)

This section describes several distinct ways in which an IEEE IDevID [IEEE.802.1AR] relates to EAT, particularly to UEID and SUEID.

[IEEE.802.1AR] orients around the definition of an implementation called a "DevID Module." It describes how IDevIDs and LDevIDs are stored, protected and accessed using a DevID Module. A particular level of defense against attack that should be achieved to be a DevID is defined. The intent is that IDevIDs and LDevIDs are used with an open set of network protocols for authentication and such. In these protocols the DevID secret is used to sign a nonce or similar to proof the association of the DevID certificates with the device.

By contrast, EAT defines network protocol for proving trustworthiness to a Relying Party, the very thing that is not defined in [IEEE.802.1AR]. Nor does not give details on how keys, data and such are stored protected and accessed. EAT is intended to work with a variety of different on-device implementations ranging from minimal protection of assets to the highest levels of asset protection. It does not define any particular level of defense against attack, instead providing a set of security considerations.

EAT and DevID can be viewed as complimentary when used together or as competing to provide a device identity service.

C.1. DevID Used With EAT

As just described, EAT defines a network protocol and [IEEE.802.1AR] doesn't. Vice versa, EAT doesn't define a an device implementation and DevID does.

Hence, EAT can be the network protocol that a DevID is used with. The DevID secret becomes the attestation key used to sign EATs. The DevID and its certificate chain become the Endorsement sent to the Verifier.

In this case the EAT and the DevID are likely to both provide a device identifier (e.g. a serial number). In the EAT it is the UEID (or SUEID). In the DevID (used as an endorsement), it is a device serial number included in the subject field of the DevID certificate. It is probably a good idea in this use for them to be the same serial number or for the UEID to be a hash of the DevID serial number.

C.2. How EAT Provides an Equivalent Secure Device Identity

The UEID, SUEID and other claims like OEM ID are equivalent to the secure device identity put into the subject field of a DevID certificate. These EAT claims can represent all the same fields and values that can be put in a DevID certificate subject. EAT explicitly and carefully defines a variety of useful claims.

EAT secures the conveyance of these claims by having them signed on the device by the attestation key when the EAT is generated. EAT also signs the nonce that gives freshness at this time. Since these claims are signed for every EAT generated, they can include things that vary over time like GPS location.

DevID secures the device identity fields by having them signed by the manufacturer of the device sign them into a certificate. That certificate is created once during the manufacturing of the device and never changes so the fields cannot change.

So in one case the signing of the identity happens on the device and the other in a manufacturing facility, but in both cases the signing of the nonce that proves the binding to the actual device happens on the device.

While EAT does not specify how the signing keys, signature process and storage of the identity values should be secured against attack, an EAT implementation may have equal defenses against attack. One reason EAT uses CBOR is because it is simple enough that a basic EAT implementation can be constructed entirely in hardware. This allows EAT to be implemented with the strongest defenses possible.

C.3. An X.509 Format EAT

It is possible to define a way to encode EAT claims in an X.509 certificate. For example, the EAT claims might be mapped to X.509 v3 extensions. It is even possible to stuff a whole CBOR-encoded unsigned EAT token into a X.509 certificate.

If that X.509 certificate is an IDevID or LDevID, this becomes another way to use EAT and DevID together.

Note that the DevID must still be used with an authentication protocol that has a nonce or equivalent. The EAT here is not being used as the protocol to interact with the rely party.

C.4. Device Identifier Permanence

In terms of permanence, an IDevID is similar to a UEID in that they do not change over the life of the device. They cease to exist only when the device is destroyed.

An SUEID is similar to an LDevID. They change on device life-cycle events.

[IEEE.802.1AR] describes much of this permanence as resistant to attacks that seek to change the ID. IDevID permanence can be described this way because [IEEE.802.1AR] is oriented around the

definition of an implementation with a particular level of defense against attack.

EAT is not defined around a particular implementation and must work on a range of devices that have a range of defenses against attack. EAT thus can't be defined permanence in terms of defense against attack. EAT's definition of permanence is in terms of operations and device lifecycle.

Appendix D. Changes from Previous Drafts

The following is a list of known changes from the previous drafts. This list is non-authoritative. It is meant to help reviewers see the significant differences.

D.1. From draft-rats-eat-01

*Added UEID design rationale appendix

D.2. From draft-mandyam-rats-eat-00

This is a fairly large change in the orientation of the document, but no new claims have been added.

*Separate information and data model using CDDL.

*Say an EAT is a CWT or JWT

*Use a map to structure the boot_state and location claims

D.3. From draft-ietf-rats-eat-01

*Clarifications and corrections for OEMID claim

*Minor spelling and other fixes

*Add the nonce claim, clarify jti claim

D.4. From draft-ietf-rats-eat-02

*Roll all EUIs back into one UEID type

*UEIDs can be one of three lengths, 128, 192 and 256.

*Added appendix justifying UEID design and size.

*Submods part now includes nested eat tokens so they can be named and there can be more tha one of them

*Lots of fixes to the CDDL

*Added security considerations

D.5. From draft-ietf-rats-eat-03

*Split boot_state into secure-boot and debug-disable claims

*Debug disable is an enumerated type rather than Booleans

D.6. From draft-ietf-rats-eat-04

*Change IMEI-based UEIDs to be encoded as a 14-byte string

*CDDL cleaned up some more

*CDDL allows for JWTs and UCCSs

*CWT format submodules are byte string wrapped

*Allows for JWT nested in CWT and vice versa

*Allows UCCS (unsigned CWTs) and JWT unsecured tokens

*Clarify tag usage when nesting tokens

*Add section on key inclusion

*Add hardware version claims

*Collected CDDL is now filled in. Other CDDL corrections.

*Rename debug-disable to debug-status; clarify that it is not extensible

*Security level claim is not extensible

*Improve specification of location claim and added a location privacy section

*Add intended use claim

D.7. From draft-ietf-rats-eat-05

*CDDL format issues resolved

*Corrected reference to Location Privacy section

D.8. From draft-ietf-rats-eat-06

*Added boot-seed claim

*Rework CBOR interoperability section

*Added profiles claim and section

D.9. From draft-ietf-rats-eat-07

*Filled in IANA and other sections for possible preassignment of Claim Keys for well understood claims

D.10. From draft-ietf-rats-eat-08

*Change profile claim to be either a URL or an OID rather than a test string

D.11. From draft-ietf-rats-eat-09

*Add SUEIDs

*Add appendix comparing IDevID to EAT

*Added section on use for Evidence and Attestation Results

*Fill in the key ID and endorsements identificaiton section

*Remove origination claim as it is replaced by key IDs and endorsements

*Added manifests and software evidence claims

*Add string labels non-claim labels for use with JSON (e.g. labels for members of location claim)

*EAN-13 HW versions are no longer a separate claim. Now they are folded in as a CoSWID version scheme.

D.12. From draft-ietf-rats-eat-10

*Hardware version is made into an array of two rather than two claims

*Corrections and wording improvements for security levels claim

*Add swresults claim

*Add dloas claim - Digitial Letter of Approvals, a list of certifications

*CDDL for each claim no longer in a separate sub section

*Consistent use of terminology from RATS architecture document

*Consistent use of terminology from CWT and JWT documents

*Remove operating model and procedures; refer to CWT, JWT and RATS architecture instead

*Some reorganization of Section 1

*Moved a few references, including RATS Architecture, to informative.

*Add detached submodule digests and detached eat bundles (DEBs)

*New simpler and more universal scheme for identifying the encoding of a nested token

*Made clear that CBOR and JSON are only mixed when nesting a token in another token

*Clearly separate CDDL for JSON and CBOR-specific data items

*Define UJCS (unsigned JWTs)

*Add CDDL for a general Claims-Set used by UCCS, UJCS, CWT, JWT and EAT

*Top level CDDL for CWT correctly refers to COSE

*OEM ID is specifically for HW, not for SW

*HW OEM ID can now be a PEN

*HW OEM ID can now be a 128-bit random number

*Expand the examples section

*Add software and version claims as easy / JSON alternative to CoSWID

Authors' Addresses

Laurence Lundblade
Security Theory LLC

Email: lgl@securitytheory.com

Giridhar Mandyam
Qualcomm Technologies Inc.
5775 Morehouse Drive
San Diego, California
United States of America

Phone: +1 858 651 7200

Email: mandyam@qti.qualcomm.com

Jeremy O'Donoghue
Qualcomm Technologies Inc.
279 Farnborough Road
Farnborough
GU14 7LS
United Kingdom

Phone: +44 1252 363189
Email: jodonogh@qti.qualcomm.com