Authors: G. Mandyam   L. Lundblade
         Qualcomm Technologies Inc.   Security Theory LLC
         M. Ballesteros
         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.

Contributing

   TBD

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 4 June 2021.

Copyright Notice

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.

   Remote attestation is a fundamental service that can underlie other
   protocols and services that need to know about the trustworthiness
   of the device before proceeding. One good example is biometric
   authentication where the biometric matching is done on the device.
   The relying party needs to know that the device is one that is known
   to do biometric matching correctly. Another example is content
   protection where the relying party wants to know the device will
   protect the data. This generalizes on to corporate enterprises that
   might want to know that a device is trustworthy before allowing
   corporate data to be accessed by it.

   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

   TODO: mention use for Attestation Evidence and Results.

1.1. CWT, JWT and UCCS

For flexibility and ease of imlpementation in a wide variety of
environments, EATs can be either CBOR [RFC7049] or JSON [ECMAScript]
format. This specification simultaneously describes both formats.

An EAT is either a CWT as defined in [RFC8392], a UCCS as defined
in [UCCS.Draft], or a JWT as defined in [RFC7519]. This
specification extends those specifications with additional claims
for attestation.

The identification of a protocol element as an EAT, whether CBOR or
JSON format, 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.

1.2. CDDL

This specification uses CDDL, [RFC8610], as the primary formalism to
define each claim. The implementor then interprets the CDDL to come
to either the CBOR [RFC7049] or JSON [ECMAScript] representation. In
the case of JSON, Appendix E of [RFC8610] is followed. Additional
rules are given in Section 5.3.2 of this document where Appendix E
is insufficient. (Note that this is not to define a general means to
translate between CBOR and JSON, but only to define enough such that
the claims defined in this document can be rendered unambiguously in
JSON).

The CWT specification was authored before CDDL was available and did
not use it. This specification includes a CDDL definition of most of
what is described in [RFC8392].

1.3. 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. It is assumed that any entity that can create an
EAT does so by means of a dedicated root-of-trust (RoT).

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 can serve as a RoT. 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.

1.4. EAT Operating Models

TODO: Rewrite (or eliminate) this section in light of the RATS
architecture draft.

At least the following three participants exist in all EAT operating
models. Some operating models have additional participants.

The Entity. This is the phone, the IoT device, the sensor, the sub-
   assembly or such that the attestation provides information about.

The Manufacturer. The company that made the entity. This may be a
   chip vendor, a circuit board module vendor or a vendor of
   finished consumer products.

The Relying Party. The server, service or company that makes use of
   the information in the EAT about the entity.

In all operating models, the manufacturer provisions some secret
attestation key material (AKM) into the entity during manufacturing.
This might be during the manufacturer of a chip at a fabrication
facility (fab) or during final assembly of a consumer product or any
time in between. This attestation key material is used for signing
EATs.

In all operating models, hardware and/or software on the entity
create an EAT of the format described in this document. The EAT is
always signed by the attestation key material provisioned by the
manufacturer.

In all operating models, the relying party must end up knowing that
the signature on the EAT is valid and consistent with data from
claims in the EAT. This can happen in many different ways. Here are
some examples.

   *The EAT is transmitted to the relying party. The relying party
    gets corresponding key material (e.g. a root certificate) from
    the manufacturer. The relying party performs the verification.

   *The EAT is transmitted to the relying party. The relying party
    transmits the EAT to a verification service offered by the
    manufacturer. The server returns the validated claims.

*The EAT is transmitted directly to a verification service,
    perhaps operated by the manufacturer or perhaps by another party.
    It verifies the EAT and makes the validated claims available to
    the relying party. It may even modify the claims in some way and
    re-sign the EAT (with a different signing key).

   All these operating models are supported and there is no preference
   of one over the other. It is important to support this variety of
   operating models to generally facilitate deployment and to allow for
   some special scenarios. One special scenario has a validation
   service that is monetized, most likely by the manufacturer. In
   another, a privacy proxy service processes the EAT before it is
   transmitted to the relying party. In yet another, symmetric key
   material is used for signing. In this case the manufacturer should
   perform the verification, because any release of the key material
   would enable a participant other than the entity to create valid
   signed EATs.

1.5. What is Not Standardized

   The following is not standardized for EAT, just the same they are
   not standardized for CWT or JWT.

1.5.1. Transmission Protocol

   EATs may be transmitted by any protocol the same as CWTs and JWTs.
   For example, they might be added in extension fields of other
   protocols, bundled into an HTTP header, or just transmitted as
   files. This flexibility is intentional to allow broader adoption.
   This flexibility is possible because EAT's are self-secured with
   signing (and possibly additionally with encryption and anti-replay).
   The transmission protocol is not required to fulfill any additional
   security requirements.

   For certain devices, a direct connection may not exist between the
   EAT-producing device and the Relying Party. In such cases, the EAT
   should be protected against malicious access. The use of COSE and
   JOSE allows for signing and encryption of the EAT. Therefore, even
   if the EAT is conveyed through intermediaries between the device and
   Relying Party, such intermediaries cannot easily modify the EAT
   payload or alter the signature.

1.5.2. Signing Scheme

   The term "signing scheme" is used to refer to the system that
   includes end-end process of establishing signing attestation key
   material in the entity, signing the EAT, and verifying it. This
   might involve key IDs and X.509 certificate chains or something
   similar but different. The term "signing algorithm" refers just to

the algorithm ID in the COSE signing structure. No particular
signing algorithm or signing scheme is required by this standard.

There are three main implementation issues driving this. First,
secure non-volatile storage space in the entity for the attestation
key material may be highly limited, perhaps to only a few hundred
bits, on some small IoT chips. Second, the factory cost of
provisioning key material in each chip or device may be high, with
even millisecond delays adding to the cost of a chip. Third,
privacy-preserving signing schemes like ECDAA (Elliptic Curve Direct
Anonymous Attestation) are complex and not suitable for all use
cases.

Over time to faciliate interoperability, some signing schemes may be
defined in EAT profiles or other documents either in the IETF or
outside.

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], COSE [RFC8152],
and CWT [RFC8392].

Claim Name. The human-readable name used to identify a claim.

Claim Key. The CBOR map key or JSON name 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 Claims Set. The CBOR map or JSON object that contains the
   claims conveyed by the CWT or JWT.

Attestation Key Material (AKM). The key material used to sign the
   EAT token. If it is done symmetrically with HMAC, then this is a
   simple symmetric key. If it is done with ECC, such as an IEEE
   DevID [IDevID], then this is the private part of the EC key pair.
   If ECDAA is used, (e.g., as used by Enhanced Privacy ID, i.e.
   EPID) then it is the key material needed for ECDAA.

3. The Claims

This section describes new claims defined for attestation. It also
mentions several claims defined by CWT and JWT that are particularly
important for EAT.

Note also: * Any claim defined for CWT or JWT may be used in an EAT
including those in the CWT [IANA.CWT.Claims] and JWT IANA
[IANA.JWT.Claims] claims registries.

  *All claims are optional

  *No claims are mandatory

  *All claims that are not understood by implementations MUST be
   ignored

There are no default values or meanings assigned to absent claims
other than they are not reported. The reason for a claim's absence
may be the implementation not supporting the claim, an inability to
determine its value, or a preference to report in a different way
such as a proprietary claim.

CDDL along with text descriptions is used to define each claim
indepdent of encoding. Each claim is defined as a CDDL group (the
group is a general aggregation and type definition feature of CDDL).
In the encoding section Section 5, the CDDL groups turn into CBOR
map entries and JSON name/value pairs.

TODO: add paragraph here about use for Attestation Evidence and for
Results.

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.

3.3.1. nonce CDDL

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

nonce-claim = (
    nonce => nonce-type / [ 2* nonce-type ]
)
```

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 7.1.

The UEID should be permanent. It should never change for a given device / entity. In addition, it should not be reprogrammable. UEID's are variable length. All implementations MUST be able to receive UEID's 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 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.

3.4.1. ueid CDDL

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

ueid-claim = (
    ueid => ueid-type
)
```

3.5. Origination Claim (origination)

   TODO: this claim is likely to be dropped in favor of Endorsement
   identifier and locators.

   This claim describes the parts of the device or entity that are
   creating the EAT. Often it will be tied back to the device or chip
   manufacturer. The following table gives some examples:

Name	Description
Acme-TEE	The EATs are generated in the TEE authored and configured by "Acme"
Acme-TPM	The EATs are generated in a TPM manufactured by "Acme"
Acme-Linux-Kernel	The EATs are generated in a Linux kernel configured and shipped by "Acme"
Acme-TA	The EATs are generated in a Trusted Application (TA) authored by "Acme"

Table 2

TODO: consider a more structure approach where the name and the URI and other are in separate fields.

TODO: This needs refinement. It is somewhat parallel to issuer claim in CWT in that it describes the authority that created the token.

3.5.1. origination CDDL

```
origination-claim = (
    origination => string-or-uri
)
```

3.6. OEM Identification by IEEE (oemid)

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.

3.6.1. oemid CDDL

```
oemid-claim = (
    oemid => bstr
)
```

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

Both the simple version string and EAN-13 versions may be included for the same hardware.

```
chip-version-claim = (
    chip-version => tstr
)

chip-version-scheme-claim = (
    chip-version-scheme => $version-scheme
)

board-version-claim = (
    board-version => tstr
)

board-version-scheme-claim = (
    board-version-scheme => $version-scheme
)

device-version-claim = (
    device-version => tstr
)

device-version-scheme-claim = (
    device-version-scheme => $version-scheme
)


ean-type = text .regexp "[0-9]{13}"

ean-chip-version-claim = (
    ean-chip-version => ean-type
)

ean-board-version-claim = (
    ean-board-version => ean-type
)

ean-device-version-claim = (
    ean-device-version => ean-type
)

hardware-version-claims = (
    ? chip-version-claim,
    ? board-version-claim,
    ? device-version-claim,
    ? chip-version-scheme-claim,
    ? board-version-scheme-claim,
    ? device-version-scheme-claim,
    ? ean-chip-version-claim,
    ? ean-board-version-claim,
    ? ean-device-version-claim,
```

)

3.8. Software Description and Version

TODO: Add claims that reference CoSWID.

3.9. 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 done by defining four security levels as described below. This is similar to the key protection types defined by the Fast Identity Online (FIDO) Alliance ([FIDO.Registry]).

These claims describe security environment and countermeasures available on the end-entity / client device where the attestation key reside 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 should not be 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.

3 - Secure Restricted Entities at this level must meet the criteria defined by 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. Example 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 schemes 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 Entity
Originator.

3.9.1. security-level CDDL

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

security-level-claim = (
    security-level => security-level-type
)
```

3.10. 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 claimd 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.

3.10.1. secure-boot CDDL

```
secure-boot-claim = (
    secure-boot => bool
)
```

3.11. 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.

3.11.1. Enabled

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

3.11.2. 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.

3.11.3. Disabled Since Boot

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

3.11.4. 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.

3.11.5. Disabled Fully and Permanently

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

3.11.6. debug-status CDDL

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

debug-status-claim = (
    debug-status => debug-status-type
)
```

3.12. 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.

3.13. 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 Section 7.2 below.

3.13.1. location CDDL

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

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

location-claim = (
    location => location-type
)
```

3.14. 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.

3.14.1. uptime CDDL

```
uptime-claim = (
    uptime => uint
)
```

3.15. 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.

3.15.1. intended-use CDDL

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

intended-use-claim = (
  intended-use => intended-use-type
)
```

3.16. The Submodules Part of a Token (submods)

Some devices are complex, having many subsystems or submodules. A
mobile phone is a good example. It may have several connectivity
submodules 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 each these can be grouped together in a submodule.

The submods part of a token are in a single map/object with many entries, one per submodule. There is only one submods map 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 claim set rather than an individual claim. This submods part of a token allows what might be called recursion. It allows claim sets inside of claim sets inside of claims sets...

3.16.1. Two Types of Submodules

Each entry in the submod map is one of two types:

  *A non-token submodule that is a map or object directly containing claims for the submodule.

  *A nested EAT that is a fully formed, independently signed EAT token

3.16.1.1. Non-token Submodules

This is simply a map or object containing claims about the submodule.

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.

It is signed/encrypted along with the rest of the token and thus the claims are secured by the same Attester with the same signing key as the rest of the token.

If a token is in CBOR format (a CWT or a UCCS), all non-token submodules must be CBOR format. If a token in in JSON format (a JWT), all non-token submodules must be in JSON format.

When decoding, this type of submodule is recognized from the other type by being a data item of type map for CBOR or type object for JSON.

3.16.1.2. Nested EATs

This type of submodule is a fully formed secured EAT as defined in this document except that it MUST NOT be a UCCS or an unsecured JWT. A nested token that is one that is always secured using COSE or JOSE, usually by an independent Attester. When the surrounding EAT is a CWT or secured JWT, the nested token becomes securely bound with the other claims in the surrounding token.

It is allowed to have a CWT as a submodule in a JWT and vice versa,
but this SHOULD be avoided unless necessary.

3.16.1.2.1. Surrounding EAT is CBOR format

They type of an EAT nested in a CWT is determined by whether the
CBOR type is a text string or a byte string. If a text string, then
it is a JWT. If a byte string, then it is a CWT.

A CWT nested in a CBOR-format token is always wrapped by a byte
string for easier handling with standard CBOR decoders and token
processing APIs that will typically take a byte buffer as input.

Nested CWTs may be either a CWT CBOR tag or a CWT Protocol Message.
COSE layers in nested CWT EATs MUST be a COSE_Tagged_Message, never
a COSE_Untagged_Message. If a nested EAT has more than one level of
COSE, for example one that is both encrypted and signed, a
COSE_Tagged_message must be used at every level.

3.16.1.2.2. Surrounding EAT is JSON format

When a CWT is nested in a JWT, it must be as a 55799 tag in order to
distinguish it from a nested JWT.

When a nested EAT in a JWT is decoded, first remove the base64url
encoding. Next, check to see if it starts with the bytes 0xd9d9f7.
If so, then it is a CWT as a JWT will never start with these four
bytes. If not if it is a JWT.

Other than the 55799 tag requirement, tag usage for CWT's nested in
a JSON format token follow the same rules as for CWTs nested in
CBOR-format tokens. It may be a CWT CBOR tag or a CWT Protocol
Message and COSE_Tagged_Message MUST be used at all COSE layers.

3.16.1.3. Unsecured JWTs and UCCS Tokens as Submodules

To incorporate a UCCS token as a submodule, it MUST be as a non-
token submodule. This can be accomplished inserting the content of
the UCCS Tag into the submodule map. The content of a UCCS tag is
exactly a map of claims as required for a non-token submodule. If
the UCCS is not a UCCS tag, then it can just be inserted into the
submodule map directly.

The definition of a nested EAT type of submodule is that it is one
that is secured (signed) by an Attester. Since UCCS tokens are
unsecured, they do not fulfill this definition and must be non-token
submodules.

To incorporate an Unsecured JWT as a submodule, the null-security
JOSE wrapping should be removed. The resulting claims set should be
inserted as a non-token submodule.

To incorporate a UCCS token in a surrounding JSON token, the UCCS
token claims should be translated from CBOR to JSON. To incorporate
an Unsecured JWT into a surrounding CBOR-format token, the null-
security JOSE should be removed and the claims translated from JSON
to CBOR.

3.16.2. 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 and age.

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

3.16.3. 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.

3.16.4. 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.

3.16.5. submods CDDL

```
; 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.

submods-part = (
    submods => submods-type
)

submods-type = { + submod-type }


; The type of a submodule which can either be a nested claim set or a
; nested separately signed token. Nested tokens are wrapped in a bstr
; or a tstr.

submod-type = (
    submod-name => eat-claim-set / nested-token
)


; When this is a bstr, the contents are an eat-token in CWT or UCCS
; format.  When this is a tstr, the contents are an eat-token in JWT
; format.

nested-token = bstr / tstr;


; Each submodule has a unique text string name.

submod-name = tstr
```

4. Endorsements and Verification Keys

TODO: fill this section in. It will discuss key IDs, endorsement ID
and such that are needed as input needed to by the Verifier to
verify the signature. This will NOT discuss the contents of an
Endorsement, just and ID/locator.

5. Encoding

This makes use of the types defined in CDDL Appendix D, Standard
Prelude.

Some of the CDDL included here is for claims that are defined in CWT
[RFC8392] or JWT [RFC7519] or are in the IANA CWT or JWT registries.
CDDL was not in use when these claims where defined.

5.1. Common CDDL Types

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

```
string-or-uri = tstr
```

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

5.2. CDDL for CWT-defined Claims

This section provides CDDL for the claims defined in CWT. It is non-
normative as [RFC8392] is the authoritative definition of these
claims.

```
$$eat-extension //= (
    ? issuer => text,
    ? subject => text,
    ? audience => text,
    ? expiration => time,
    ? not-before => time,
    ? issued-at => time,
    ? cwt-id => bytes,
)
```

```
issuer = 1
subject = 2
audience = 3
expiration = 4
not-before = 5
issued-at = 6
cwt-id = 7
```

5.3. JSON

5.3.1. JSON Labels

```
ueid /= "ueid"
nonce /= "nonce"
origination /= "origination"
oemid /= "oemid"
security-level /= "security-level"
secure-boot /= "secure-boot"
debug-status /= "debug-status"
location /= "location"
age /= "age"
uptime /= "uptime"
submods /= "submods"
timestamp /= "timestamp"

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

5.3.2. JSON Interoperability

JSON should be encoded per RFC 8610 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].

5.4. CBOR

5.4.1. CBOR Interoperability

Variations in the CBOR serializations supported in CBOR encoding and
decoding are allowed and suggests that CBOR-based protocols specify
how this variation is handled. This section specifies what formats
MUST be supported in order to achieve interoperability.

The assumption is that the entity is likely to be a constrained
device and relying party is likely to be a very capable server. The
approach taken is that the entity generating the token can use

whatever encoding it wants, specifically encodings that are easier to implement such as indefinite lengths. The relying party receiving the token must support decoding all encodings.

These rules cover all types used in the claims in this document. They also are recommendations for additional claims.

Canonical CBOR encoding, Preferred Serialization and Deterministically Encoded CBOR are explicitly NOT required as they would place an unnecessary burden on the entity implementation, particularly if the entity implementation is implemented in hardware.

  *Integer Encoding (major type 0, 1) - The entity may use any integer encoding allowed by CBOR. The server MUST accept all integer encodings allowed by CBOR.

  *String Encoding (major type 2 and 3) - The entity can use any string encoding allowed by CBOR including indefinite lengths. It may also encode the lengths of strings in any way allowed by CBOR. The server must accept all string encodings.

  *Major type 2, bstr, SHOULD have tag 21 to indicate conversion to base64url in case that conversion is performed.

  *Map and Array Encoding (major type 4 and 5) - The entity can use any array or map encoding allowed by CBOR including indefinite lengths. Sorting of map keys is not required. Duplicate map keys are not allowed. The server must accept all array and map encodings. The server may reject maps with duplicate map keys.

  *Date and Time - The entity should send dates as tag 1 encoded as 64-bit or 32-bit integers. The entity may not send floating-point dates. The server must support tag 1 epoch-based dates encoded as 64-bit or 32-bit integers. The entity may send tag 0 dates, however tag 1 is preferred. The server must support tag 0 UTC dates.

  *URIs - URIs should be encoded as text strings and marked with tag 32.

  *Floating Point - The entity may use any floating-point encoding. The relying party must support decoding of all types of floating-point.

  *Other types - Other types like bignums, regular expressions and such, SHOULD NOT be used. The server MAY support them but is not required to so interoperability is not guaranteed.

5.5. Collected CDDL

```
; This is the top-level definition of the claims in EAT tokens.  To
; form an actual EAT Token, this claim set is enclosed in a COSE, JOSE
; or UCCS message.
; TO-DO:  Add intended-use claim

eat-claim-set = {
    ? ueid-claim,
    ? nonce-claim,
    ? origination-claim,
    ? oemid-claim,
    ? hardware-version-claims,
    ? security-level-claim,
    ? secure-boot-claim,
    ? debug-status-claim,
    ? location-claim,
    ? uptime-claim,
    ? submods-part,
    * $$eat-extension,
}


; This is the top-level definition of an EAT Token.  It is a CWT, JWT
; or UCSS where the payload is an eat-claim-set. A JWT_Message is what
; is defined by JWT in RFC 7519. (RFC 7519 doesn't use CDDL so a there
; is no actual CDDL definition of JWT_Message).

eat-token = EAT_Tagged_Message / EAT_Untagged_Message / JWT_Message


; This is CBOR-format EAT token in the CWT or UCCS format that is a
; tag.   COSE_Tagged_message is defined in RFC 8152.  Tag 601 is
; proposed by the UCCS draft, but not yet assigned.

EAT_Tagged_Message = #6.61(COSE_Tagged_Message) / #6.601(eat-claim-set)


; This is a CBOR-format EAT token that is a CWT or UCSS that is not a
; tag COSE_Tagged_message and COSE_Untagged_Message are defined in RFC
; 8152.

EAT_Untagged_Message = COSE_Tagged_Message / COSE_Untagged_Message / UCC


; This is an "unwrapped" UCCS tag. Unwrapping a tag means to use the
; definition of its content without the preceding type 6 tag
; integer. Since a UCCS is nothing but a tag for an unsecured CWT
; claim set, unwrapping reduces to a bare eat-claim-set.

UCCS_Untagged_Message = eat-claim-set
```

```
; The following Claim Keys (labels) are temporary. They are not
; assigned by IANA

nonce = 10
ueid = 11
origination = 12
oemid = 13
security-level = 14
secure-boot = 15
debug-status = 16
location = 17
uptime = 19
submods = 20

chip-version = 21
board-version = 22
device-version = 23
chip-version-scheme = 24
board-version-scheme = 25
device-version-scheme = 26
ean-chip-version = 27
ean-board-version = 28
ean-device-version = 29

string-or-uri = tstr

time-int = #6.1(int)

$$eat-extension //= (
    ? issuer => text,
    ? subject => text,
    ? audience => text,
    ? expiration => time,
    ? not-before => time,
    ? issued-at => time,
    ? cwt-id => bytes,
)

issuer = 1
subject = 2
audience = 3
expiration = 4
not-before = 5
issued-at = 6
cwt-id = 7


debug-status-type = &(
    enabled: 0,
```

```
        disabled: 1,
        disabled-since-boot: 2,
        disabled-permanently: 3,
        disabled-fully-and-permanently: 4
)

debug-status-claim = (
        debug-status => debug-status-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
longitude = 2
altitude = 3
accuracy = 4
altitude-accuracy = 5
heading = 6
speed = 7
timestamp = 8
age = 9

location-claim = (
        location => location-type
)

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

nonce-claim = (
        nonce => nonce-type / [ 2* nonce-type ]
)

oemid-claim = (
        oemid => bstr
)

; copied from CoSWID
; TODO: how to properly make reference to CoSWID and have tool validate

   $version-scheme /= multipartnumeric
```

```
    $version-scheme /= multipartnumeric-suffix
    $version-scheme /= alphanumeric
    $version-scheme /= decimal
    $version-scheme /= semver
    $version-scheme /= uint / text
    multipartnumeric = 1
    multipartnumeric-suffix = 2
    alphanumeric = 3
    decimal = 4
    semver = 16384


chip-version-claim = (
    chip-version => tstr
)

chip-version-scheme-claim = (
    chip-version-scheme => $version-scheme
)

board-version-claim = (
    board-version => tstr
)

board-version-scheme-claim = (
    board-version-scheme => $version-scheme
)

device-version-claim = (
    device-version => tstr
)

device-version-scheme-claim = (
    device-version-scheme => $version-scheme
)


ean-type = text .regexp "[0-9]{13}"

ean-chip-version-claim = (
    ean-chip-version => ean-type
)

ean-board-version-claim = (
    ean-board-version => ean-type
)

ean-device-version-claim = (
    ean-device-version => ean-type
)
```

```
hardware-version-claims = (
    ? chip-version-claim,
    ? board-version-claim,
    ? device-version-claim,
    ? chip-version-scheme-claim,
    ? board-version-scheme-claim,
    ? device-version-scheme-claim,
    ? ean-chip-version-claim,
    ? ean-board-version-claim,
    ? ean-device-version-claim,
)


origination-claim = (
    origination => string-or-uri
)

secure-boot-claim = (
    secure-boot => bool
)

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

security-level-claim = (
    security-level => security-level-type
)

; 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.

submods-part = (
    submods => submods-type
)

submods-type = { + submod-type }


; The type of a submodule which can either be a nested claim set or a
; nested separately signed token. Nested tokens are wrapped in a bstr
; or a tstr.

submod-type = (
    submod-name => eat-claim-set / nested-token
```

```
)


; When this is a bstr, the contents are an eat-token in CWT or UCCS
; format.  When this is a tstr, the contents are an eat-token in JWT
; format.

nested-token = bstr / tstr;


; Each submodule has a unique text string name.

submod-name = tstr



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

ueid-claim = (
      ueid => ueid-type
)

uptime-claim = (
     uptime => uint
)

ueid /= "ueid"
nonce /= "nonce"
origination /= "origination"
oemid /= "oemid"
security-level /= "security-level"
secure-boot /= "secure-boot"
debug-status /= "debug-status"
location /= "location"
age /= "age"
uptime /= "uptime"
submods /= "submods"
timestamp /= "timestamp"

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

6. IANA Considerations

6.1. Reuse of CBOR Web Token (CWT) Claims Registry

Claims defined for EAT are compatible with those of CWT so the CWT
Claims Registry is re used. No new IANA registry is created. All EAT
claims should be registered in the CWT and JWT Claims Registries.

6.2. 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.

6.2.1. 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.

6.2.2. 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.

6.2.3. 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.

6.2.4. 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.

6.2.5. 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.

6.3. Claims Registered by This Document

   *Claim Name: UEID

   *Claim Description: The Universal Entity ID

*JWT Claim Name: N/A

*Claim Key: 8

*Claim Value Type(s): byte string

*Change Controller: IESG

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

TODO: add the rest of the claims in here

7. 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.

7.1. UEID 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 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.

There are several strategies that can be used to still be able to
put UEID's in tokens:

  *The device obtains explicit permission from the user of the
   device to use the UEID. 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.

  *The UEID 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 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.

Note that some of these privacy preservation strategies result in multiple UEIDs per device. Each UEID 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.

7.2. 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.

8. 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.

8.1. 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.

8.1.1. 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.

8.2. 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.

8.3. 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.

9. References

9.1. Normative References

[CoSWID]   "Concise Software Identification Tags", November 2020,
            <https://tools.ietf.org/html/draft-ietf-sacm-coswid-16>.

[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 2019,
            <https://fidoalliance.org/specs/fido-uaf-v1.0-
            fd-20191115/fido-allowed-AROE-v1.0-fd-20191115.html>.

[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>.

[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>.

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

[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>.

[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>.

[TIME_T] The Open Group Base Specifications, "Vol. 1: Base Definitions, Issue 7", Section 4.15 'Seconds Since the Epoch', IEEE Std 1003.1, 2013 Edition, 2013, <http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_15>.

[UCCS.Draft] Birkholz, H., "A CBOR Tag for Unprotected CWT Claims Sets", 2020, <https://tools.ietf.org/html/draft-birkholz-rats-uccs-01>.

[WGS84]   National Imagery and Mapping Agency, "National Imagery and Mapping Agency Technical Report 8350.2, Third Edition", 2000, <http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf>.

9.2. Informative References

[ASN.1]
International Telecommunication Union, "Information Technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 1994.

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

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

[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>.

[FIDO.Registry] The FIDO Alliance, "FIDO Registry of Predefined Values", December 2019, <https://fidoalliance.org/specs/common-specs/fido-registry-v2.1-ps-20191217.html>.

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

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

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

[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>.

[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>.

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

[Webauthn] Worldwide Web Consortium, "Web Authentication: A Web API
            for accessing scoped credentials", 2016.

Appendix A. Examples

A.1. Very Simple EAT

This is shown in CBOR diagnostic form. Only the payload signed by
COSE is shown.

```
{
   / issuer /            1: "joe",
   / nonce /            10: h'948f8860d13a463e8e',
   / UEID /             11: h'0198f50a4ff6c05861c8860d13a638ea4fe2fa',
   / secure-boot /      15: true,
   / debug-disable /    16: 3, / permanent-disable /
   / timestamp (iat) /   6: 1(1526542894),
   / chip-version /     21: "1.4a",
   / chip-version-scheme / 24: 2 / multipartnumeric+suffix /
}
```

A.2. Example with Submodules, Nesting and Security Levels

```
{
    / nonce /                10: h'948f8860d13a463e8e',
    / UEID /                 11: h'0198f50a4ff6c05861c8860d13a638ea4fe2
    / secure-boot /          15: true,
    / debug-disable /        16: 3, / permanent-disable  /
    / timestamp (iat) /       6: 1(1526542894),
    / security-level /       14: 3, / secure restricted OS /
    / submods / 20: {
        / first submod, an Android Application /
        "Android App Foo" :  {
            / security-level /  14: 1 / unrestricted /
        },

        / 2nd submod, A nested EAT from a secure element /
        "Secure Element Eat" :
            / an embedded EAT, bytes of which are not shown /
            h'420123',

        / 3rd submod, information about Linux Android /
        "Linux Android": {
            / security-level /  14: 1 / unrestricted /
        }
    }
}
```

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 3

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 4

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
quadrillion (10^15)	8 seconds	10^14 years	10^34 years
100 quadrillion (10^17)	8 microseconds	10^11 years	10^31 years

Table 5

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. 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.

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

*Added UEID design rationale appendix

C.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

C.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

C.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

C.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

C.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

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

*CDDL format issues resolved

*Corrected reference to Location Privacy section

Authors' Addresses

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

Laurence Lundblade
Security Theory LLC

Email: lgl@island-resort.com

Miguel Ballesteros
Qualcomm Technologies Inc.
5775 Morehouse Drive
San Diego, California
United States of America

Phone: +1 858 651 4299
Email: mballest@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