Stream: Internet Engineering Task Force (IETF)
RFC: 9711
Category: Standards Track
Published: January 2025
ISSN: 2070-1721
Authors:
L. Lundblade, *Security Theory LLC*
G. Mandyam, *Qualcomm Technologies Inc.*
J. O'Donoghue, *Qualcomm Technologies Inc.*
C. Wallace, *Red Hound Software, Inc.*

RFC 9711
The Entity Attestation Token (EAT)

Abstract

An Entity Attestation Token (EAT) provides an attested claims set that describes the state and characteristics of an entity, a device such as a smartphone, an Internet of Things (IoT) device, network equipment, or such. This claims set is used by a relying party, server, or service to determine the type and degree of trust placed in the entity.

An EAT is either a CBOR Web Token (CWT) or a JSON Web Token (JWT) with attestation-oriented claims.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9711.

Copyright Notice

Table of Contents

1. Introduction 6
1.1. Entity Overview 8
1.2. EAT as a Framework 8
1.3. Operating Model and RATS Architecture 9
1.3.1. Relationship between Evidence and Attestation Results 10
2. Terminology 10
3. Top-Level Token Definition 12
4. The Claims 13
4.1. eat_nonce (EAT Nonce) Claim 14
4.2. Claims Describing the Entity 14
4.2.1. ueid (Universal Entity ID) Claim 14
4.2.1.1. Rules for Creating UEIDs 15
4.2.1.2. Rules for Consuming UEIDs 16
4.2.2. sueids (Semipermanent UEIDs) Claim 17
4.2.3. oemid (Hardware OEM ID) Claim 17
4.2.3.1. Random Number-Based OEM ID 18
4.2.3.2. IEEE-Based OEM ID 18
4.2.3.3. IANA Private Enterprise Number-Based OEM ID 18
4.2.4. hwmodel (Hardware Model) Claim 19
4.2.5. hwversion (Hardware Version) Claim 20
4.2.6. swname (Software Name) Claim 20
4.2.7. swversion (Software Version) Claim 20
4.2.8. oemboot (OEM Authorized Boot) Claim 21
4.2.9. dbgstat (Debug Status) Claim 21
4.2.9.1. Enabled 22
4.2.9.2. Disabled 22

4.2.9.3. Disabled Since Boot 22
4.2.9.4. Disabled Permanently 22
4.2.9.5. Disabled Fully and Permanently 22
4.2.10. location (Location) Claim 23
4.2.11. uptime (Uptime) Claim 24
4.2.12. bootcount (Boot Count) Claim 24
4.2.13. bootseed (Boot Seed) Claim 24
4.2.14. dloas (Digital Letters of Approval) Claim 25
4.2.15. manifests (Software Manifests) Claim 25
4.2.16. measurements (Measurements) Claim 27
4.2.17. measres (Software Measurement Results) Claim 27
4.2.18. submods (Submodules) Claim 29
4.2.18.1. Submodule Claims-Set 32
4.2.18.2. Detached Submodule Digest 32
4.2.18.3. Nested Tokens 32
4.3. Claims Describing the Token 33
4.3.1. iat (Timestamp) Claim 33
4.3.2. eat_profile (EAT Profile) Claim 33
4.3.3. intuse (Intended Use) Claim 34
5. Detached EAT Bundles 34
6. Profiles 35
6.1. Format of a Profile Document 36
6.2. Full and Partial Profiles 36
6.3. List of Profile Issues 36
6.3.1. Use of JSON, CBOR, or Both 36
6.3.2. CBOR Map and Array Encoding 37
6.3.3. CBOR String Encoding 37
6.3.4. CBOR Preferred Serialization 37
6.3.5. CBOR Tags 37
6.3.6. COSE/JOSE Protection 37

6.3.7. COSE/JOSE Algorithms 38
6.3.8. Detached EAT Bundle Support 38
6.3.9. Key Identification 38
6.3.10. Endorsement Identification 38
6.3.11. Freshness 38
6.3.12. Claims Requirements 39
6.4. The Constrained Device Standard Profile 39
7. Encoding and Collected CDDL 40
7.1. Claims-Set and CDDL for CWT and JWT 40
7.2. Encoding Data Types 41
7.2.1. Common Data Types 41
7.2.2. JSON Interoperability 41
7.2.3. Labels 42
7.2.4. CBOR Interoperability 42
7.3. Collected CDDL 42
7.3.1. Payload CDDL 42
7.3.2. CBOR-Specific CDDL 47
7.3.3. JSON-Specific CDDL 47
8. Privacy Considerations 48
8.1. UEID and SUEID Privacy Considerations 48
8.2. Location Privacy Considerations 48
8.3. Boot Seed Privacy Considerations 49
8.4. Replay Protection and Privacy 49
9. Security Considerations 49
9.1. Claim Trustworthiness 49
9.2. Key Provisioning 50
9.2.1. Transmission of Key Material 50
9.3. Freshness 50
9.4. Multiple EAT Consumers 50
9.5. Detached EAT Bundle Digest Security Considerations 51

9.6. Verification Keys 51
10. IANA Considerations 51
10.1. Reuse of CBOR and JSON Web Token (CWT and JWT) Claims Registries 51
10.2. CWT and JWT Claims Registered by This Document 51
10.3. UEID URNs Registered by This Document 56
10.4. CBOR Tag for Detached EAT Bundle Registered by This Document 56
10.5. Intended Use Registry 56
11. References 57
11.1. Normative References 57
11.2. Informative References 59
Appendix A. Examples 61
A.1. Claims Set Examples 61
A.1.1. Simple TEE Attestation 61
A.1.2. Submodules for Board and Device 64
A.1.3. EAT Produced by an Attestation Hardware Block 65
A.1.4. Key / Key Store Attestation 65
A.1.5. Software Measurements of an IoT Device 66
A.1.6. Attestation Results in JSON 68
A.1.7. JSON-Encoded Token with Submodules 69
A.2. Signed Token Examples 70
A.2.1. Basic CWT Example 70
A.2.2. CBOR-Encoded Detached EAT Bundle 71
A.2.3. JSON-Encoded Detached EAT Bundle 73
Appendix B. UEID Design Rationale 74
B.1. Collision Probability 74
B.2. No Use of UUID 77
Appendix C. EAT Relation to IEEE.802.1AR Secure Device Identity (DevID) 77
C.1. DevID Used with EAT 78
C.2. How EAT Provides an Equivalent Secure Device Identity 78
C.3. An X.509 Format EAT 78

C.4. Device Identifier Permanence 79
Appendix D. CDDL for CWT and JWT 79
Appendix E. New Claim Design Considerations 81
E.1. Interoperability and Relying Party Orientation 81
E.2. Operating System and Technology Neutral 81
E.3. Security Level Neutral 82
E.4. Reuse of Extant Data Formats 82
E.5. Proprietary Claims 82
Appendix F. Endorsements and Verification Keys 82
F.1. Identification Methods 83
F.1.1. COSE/JWS Key ID 83
F.1.2. JWS and COSE X.509 Header Parameters 83
F.1.3. CBOR Certificate COSE Header Parameters 84
F.1.4. Claim-Based Key Identification 84
Contributors 84
Authors' Addresses 85

1. Introduction

An Entity Attestation Token (EAT) is a message made up of claims about an entity. An entity may be a device, some hardware, or some software. The claims are ultimately used by a relying party who decides if and how it will interact with the entity. The relying party may choose to trust, not trust, or partially trust the entity. For example, partial trust may be allowing a monetary transaction only up to a limit.

The security model and goal for attestation are unique and are not the same as those for other security standards such as server authentication, user authentication, and secured messaging. To give an example of one aspect of the difference, consider the association and life cycle of key material. For authentication, keys are associated with a user or service and are set up by actions performed by a user or an operator of a service. For attestation, the keys are associated with specific devices and are configured by device manufacturers. Since the reader is assumed to be familiar with the goals and security model for attestation as described in "Remote ATtestation procedureS (RATS) Architecture" [RFC9334], they are not repeated here.

This document defines some common claims that are potentially of broad use. EAT additionally allows proprietary claims and for further claims to be standardized. Here are some examples:

- Make and model of manufactured consumer device
- Make and model of a chip or processor, particularly for a security-oriented chip
- Identification and measurement of the software running on a device
- Configuration and state of a device
- Environmental characteristics of a device such as its Global Positioning System (GPS) location
- Formal certifications received

EAT is constructed to support a wide range of use cases.

No single set of claims can accommodate all use cases, so EAT is constructed as a framework for defining specific attestation tokens for specific use cases. In particular, EAT provides a profile mechanism to be able to clearly specify the claims needed, the cryptographic algorithms that should be used, and other characteristics for a particular token and use case. Section 6 describes profile contents and provides a profile that is suitable for constrained device use cases.

The entity's EAT implementation generates the claims and typically signs them with an attestation key. It is responsible for protecting the attestation key. Some EAT implementations will use components with very high resistance to attack such as Trusted Platform Modules or Secure Elements. Others may rely solely on simple software defenses.

Nesting of tokens and claims sets is accommodated for composite devices that have multiple subsystems.

An EAT may be encoded in either JavaScript Object Notation (JSON) [RFC8259] or Concise Binary Object Representation (CBOR) [RFC8949] as needed for each use case. EAT is built on the CBOR Web Token (CWT) [RFC8392] and JSON Web Token (JWT) [RFC7519] and inherits all their characteristics and their security mechanisms. Like CWT and JWT, EAT does not imply any message flow.

The following is a very simple example. It is presented in JSON format for easy reading, but it could also be CBOR. Only the Claims-Set, the payload for the JWT, is shown.

```
{
    "eat_nonce": "MIDBNH28iioisjPy",
    "ueid":      "AgAEizrK3Q",
    "oemid":     76543,
    "swname":    "Acme IoT OS",
    "swversion": "3.1.4"
}
```

This example has a nonce for freshness. This nonce is the base64url encoding of a 12-byte random binary byte string. The ueid (Universal Entity ID) is effectively a serial number uniquely identifying the device. This ueid is the base64url encoding of a 48-bit Media Access Control

(MAC) address preceded by the type byte 0x02. The oemid (Hardware OEM ID) identifies the manufacturer using a Private Enterprise Number (PEN) [PEN]. The software is identified by a simple string name and version. It could be identified by a full manifest, but this is a minimal example.

1.1. Entity Overview

This document uses the term "entity" to refer to the target of an EAT. Most of the claims defined in this document are claims about an entity. An entity is equivalent to a target environment in an attester as defined in [RFC9334].

Layered attestation and composite devices, as described in [RFC9334], are supported by a submodule mechanism (see Section 4.2.18). Submodules allow nesting of EATs and of Claims-Sets so that such hierarchies can be modeled.

An entity is the same as a "system component", as defined in the Internet Security Glossary [RFC4949].

Note that [RFC4949] defines "entity" and "system entity" as synonyms, and that they may be a person or organization in addition to being a system component. In the EAT context, "entity" never refers to a person or organization. The hardware and software that implement a website server or service may be an entity in the EAT sense, but the organization that operates, maintains, or hosts the website is not an entity.

Some examples of entities:

- A Secure Element
- A Trusted Execution Environment (TEE)
- A network card in a router
- A router, perhaps with each network card in the router being a submodule
- An IoT device
- An individual process
- An app on a smartphone
- A smartphone with many submodules for its many subsystems
- A subsystem in a smartphone such as the modem or the camera

An entity may have strong security defenses against hardware-invasive attacks. It may also have low security, i.e., having no special security defenses. There is no minimum security requirement to be an entity.

1.2. EAT as a Framework

EAT is a framework that is used for defining attestation tokens for specific use cases; it is not used for specific token definition. While EAT is based on and compatible with CWT and JWT, it can also be described as:

- An identification and type system for claims in Claims-Sets

- Definitions of common attestation-oriented claims
- Claims defined in Concise Data Definition Language (CDDL) and serialized using CBOR or JSON
- Security envelopes based on CBOR Object Signing and Encryption (COSE) and JSON Object Signing and Encryption (JOSE)
- The nesting of claims sets and tokens to represent complex and compound devices
- A profile mechanism for specifying and identifying specific tokens for specific use cases

EAT uses name/value pairs to identify individual claims the same way as CWT and JWT. Section 4 defines common attestation-oriented claims that have been added to the "CBOR Web Token (CWT) Claims" and "JSON Web Token Claims" IANA registries. As with CWT and JWT, no claims are mandatory and claims not recognized should be ignored.

Unlike (but compatible with) CWT and JWT, EAT defines claims using CDDL [RFC8610]. In most cases, the same CDDL definition is used for both the CBOR/CWT serialization and the JSON/JWT serialization.

Like CWT and JWT, EAT uses COSE and JOSE to provide authenticity, integrity, and optionally confidentiality. EAT places no new restrictions on cryptographic algorithms, retaining all the cryptographic flexibility of CWT, COSE, JWT, and JOSE.

EAT defines a means for nesting tokens and claims sets to accommodate composite devices that have multiple subsystems and multiple attesters. Tokens with security envelopes or bare claims sets may be embedded in an enclosing token. The nested token and the enclosing token do not have to use the same encoding (e.g., a CWT may be enclosed in a JWT).

EAT adds the ability to detach claims sets and send them separately from a security-enveloped EAT that contains a digest of the detached claims set.

This document registers no media or content types for the identification of the EAT type, serialization encoding, or security envelope. The definition and registration of EAT media types are addressed in [EAT.media-types].

Finally, this document introduces the notion of an EAT profile that facilitates the creation of narrowed definitions of EATs for specific use cases in subsequent documents. One basic profile for constrained devices is normatively defined.

1.3. Operating Model and RATS Architecture

EAT follows the operational model described in Figure 1 of RATS Architecture (Section 3 of [RFC9334]). To summarize, an attester generates evidence in the form of a claims set describing various characteristics of an entity. Evidence is usually signed by a key that proves the attester and the evidence it produces are authentic. The claims set either includes a received nonce or uses some other means to assure freshness.

A verifier confirms an EAT is valid by verifying the signature and may vet some claims using reference values. The verifier then produces attestation results, which may also be represented as an EAT. The attestation results are provided to the relying party, which is the ultimate consumer of the Remote Attestation Procedure. The relying party uses the attestation results as needed for its use case, perhaps allowing an entity to access a network, a financial transaction, or such. In some cases, the verifier and relying party are not distinct entities.

1.3.1. Relationship between Evidence and Attestation Results

Any claim defined in this document or in the IANA "CBOR Web Token (CWT) Claims" or "JSON Web Token Claims" registries may be used in evidence or attestation results. The relationship of claims in attestation results to evidence is fundamentally governed by the verifier and the verifier's policy.

A common use case is for the verifier and its policy to perform checks, calculations, and processing with evidence as the input to produce a summary result in attestation results that indicates the overall health and status of the entity. For example, measurements in evidence may be compared to reference values, the results of which are represented as a simple pass/fail in attestation results.

It is also possible that some claims in the evidence will be forwarded unmodified to the relying party in attestation results. This forwarding is subject to the verifier's implementation and policy. The relying party should be aware of the verifier's policy to know what checks it has performed on claims it forwards.

The verifier may modify claims it forwards, for example, to implement a privacy preservation functionality. It is also possible the verifier will put claims in the attestation results that give details about the entity that it has computed or looked up in a database. For example, the verifier may be able to put an "oemid" claim in the attestation results by performing a lookup based on a "ueid" claim (e.g., serial number) it received in evidence.

This specification does not establish any normative rules for the verifier to follow, as these are a matter of local policy. It is up to each relying party to understand the processing rules of each verifier to know how to interpret claims in attestation results.

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.

In this document, the structure of data is specified in CDDL [RFC8610] [RFC9165].

The examples in Appendix A use CBOR diagnostic notation defined in Section 8 of [RFC8949] and Appendix G of [RFC8610].

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

base64url encoding: base64 encoding using the URL- and filename-safe character set defined in Section 5 of [RFC4648], with all trailing '=' characters omitted and without the inclusion of any line breaks, whitespace, or other additional characters [RFC7515].

Claim: A piece of information asserted about a subject. A claim is represented as a value and either a name or a 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. (The term "Claim Key" comes from CWT. This document, like COSE [RFC9052], uses the term "label" to refer to CBOR map keys to avoid confusion with cryptographic keys.)

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

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

This document reuses terminology from RATS Architecture [RFC9334]; note that EAT does not capitalize RATS terms like “evidence” for easier readability:

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 evidence about an attester and produces attestation results to be used by a relying party.

Relying Party: A role performed by an entity that depends on the validity of information about an attester for purposes of reliably applying application-specific actions. Compare: relying party [RFC4949].

Evidence: A set of claims generated by an attester to be appraised by a verifier. 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 evidence. Reference values are sometimes referred to in other documents as "known-good values", "golden measurements", or "nominal values". These terms typically assume comparison for equality, whereas here, reference values might be more general and be used in any sort of comparison.

Endorsement: A secure statement that an endorser vouches for the integrity of an attester's various capabilities such as claims collection and evidence signing.

This document reuses terminology from CDDL [RFC8610]:

Group Socket: The mechanism by which a CDDL definition is extended, as described in [RFC8610] and [RFC9165].

3. Top-Level Token Definition

An "EAT" is an encoded (serialized) message, the purpose of which is to transfer a Claims-Set between two parties. An EAT **MUST** contain a Claims-Set. In this document, an EAT is always a CWT or JWT.

An EAT **MUST** have authenticity and integrity protection. CWT and JWT provide that in this document.

Further documents may define other encodings and security mechanisms for EAT.

The identification of a protocol element as an EAT follows the general conventions used for CWTs and JWTs. Identification depends on the protocol carrying the EAT. In some cases, it may be by media type (e.g., in an HTTP Content-Type field). In other cases, it may be through use of CBOR tags. There is no fixed mechanism across all use cases.

This document also defines another message, the detached EAT bundle (see Section 5), which holds a collection of detached claims sets and an EAT that provides integrity and authenticity protection for them. Detached EAT bundles can be either CBOR or JSON encoded.

The following CDDL defines the top-level `$CBOR-Tagged-Token`, `$EAT-CBOR-Untagged-Token`, and `$EAT-JSON-Token-Formats` sockets (see Section 3.9 of [RFC8610]), enabling future token formats to be defined. Any new format that plugs into one or more of these sockets **MUST** be defined by an IETF Standards Action [RFC8126]. Of particular use may be a token type that provides no direct authenticity or integrity protection for use with transport mechanisms that do provide the necessary security services [UCCS].

Nesting of EATs is allowed and defined in Section 4.2.18.3. This includes the nesting of an EAT that is in a different format than the enclosing EAT, i.e., the nested EAT may be encoded using CBOR and the enclosing EAT encoded using JSON or vice versa. The definition of Nested-Token references the CDDL defined in this section. When new token formats are defined, the means for identification in a nested token **MUST** also be defined.

The top-level CDDL type for CBOR-encoded EATs is EAT-CBOR-Token and for JSON-encoded EATs is EAT-JSON-Token (while CDDL and CDDL tools provide enough support for shared definitions of most items in this document, they do not provide enough support for this sharing at the top level).

```
EAT-CBOR-Token = $CBOR-Tagged-Token / $EAT-CBOR-Untagged-Token

$CBOR-Tagged-Token /= CWT-Tagged-Message
$CBOR-Tagged-Token /= BUNDLE-Tagged-Message

$EAT-CBOR-Untagged-Token /= CWT-Untagged-Message
$EAT-CBOR-Untagged-Token /= BUNDLE-Untagged-Message
```

```
EAT-JSON-Token = $EAT-JSON-Token-Formats

$EAT-JSON-Token-Formats /= JWT-Message
$EAT-JSON-Token-Formats /= BUNDLE-Untagged-Message
```

4. The Claims

This section describes new claims defined for attestation that have been added to the IANA "CBOR Web Token (CWT) Claims" [IANA.CWT.Claims] and "JSON Web Token Claims" [IANA.JWT.Claims] registries.

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

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

The set of claims that an EAT must contain to be considered valid is context dependent and is outside the scope of this specification. Specific applications of EATs will require implementations to understand and process some claims in particular ways. However, in the absence of such requirements, all claims that are not understood by implementations **MUST** be ignored.

CDDL, along with a text description, is used to define each claim independent of encoding. Each claim is defined as a CDDL group. In "Encoding and Collected CDDL" (Section 7), the CDDL groups turn into CBOR map entries and JSON name/value pairs.

Each claim defined in this document is added to the `$$Claims-Set-Claims` group socket. Claims defined by other specifications **MUST** also be added to the `$$Claims-Set-Claims` group socket.

All claims in an EAT **MUST** use the same encoding except where otherwise explicitly stated (e.g., in a CBOR-encoded token, all claims must be encoded with CBOR).

This specification provides a CDDL definition for most of the elements defined in [RFC7519] and [RFC8392]. These definitions are in Appendix D and are not normative.

Each claim described has a unique text string and integer that identifies it. CBOR-encoded tokens **MUST** only use the integer for claim keys. JSON-encoded tokens **MUST** only use the text string for claim names.

4.1. eat_nonce (EAT Nonce) Claim

In JSON, an EAT nonce is either a text string or an array of text strings. In CBOR, an EAT nonce is either a byte string or an array of byte strings. The array option supports multistage EAT verification and consumption.

A claim named "nonce" was defined for JWT and registered with IANA in the "JSON Web Token Claims" registry, but it **MUST NOT** be used because it does not support multiple nonces. No previous "nonce" claim was defined for CWT. To distinguish from the previously defined JWT "nonce" claim, this claim is named "eat_nonce" in JSON-encoded EATs. The CWT nonce defined here is intended for general purpose use and retains the "Nonce" claim name instead of an EAT-specific name.

An EAT nonce **MUST** have at least 64 bits of entropy. A maximum EAT nonce size is set to limit the memory required for an implementation. All receivers **MUST** be able to accommodate the maximum size.

In CBOR, an EAT nonce is a byte string between 8 and 64 bytes in length. In JSON, an EAT nonce is a text string between 8 and 88 bytes in length.

```
$$Claims-Set-Claims //=
    (nonce-label => nonce-type / [ 2* nonce-type ])

nonce-type = JC< tstr .size (8..88), bstr .size (8..64)>
```

4.2. Claims Describing the Entity

The claims in this section describe the entity itself. They describe the entity whether they occur in evidence or occur in attestation results. See Section 1.3.1 for discussion on how attestation results relate to evidence.

4.2.1. ueid (Universal Entity ID) Claim

The "ueid" claim conveys a UEID, which identifies an individual manufactured entity such as a mobile phone, water meter, Bluetooth speaker, or networked security camera. It may identify the entire entity or a submodule. It does not identify types, models, or classes of entities. It is akin to a serial number, though it does not have to be sequential.

UEIDs **MUST** be universally and globally unique across manufacturers and countries, as described in Section 4.2.1.1. 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 entity to keep entities distinct between manufacturers).

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

There are privacy considerations for UEIDs. See Section 8.1.

A Device Identifier (DevID) URN is registered for UEIDs. See Section 10.3.

```
$$Claims-Set-Claims //= (ueid-label => ueid-type)

ueid-type = JC<base64-url-text .size (10..44) , bstr .size (7..33)>
```

4.2.1.1. Rules for Creating UEIDs

These rules are solely for the creation of UEIDs. The EAT consumer need not have any awareness of them.

A UEID is constructed of a single type byte followed by the unique bytes for that type. The type byte assures global uniqueness of a UEID even if the unique bytes for different types are accidentally the same.

UEIDS are of variable length to accommodate the types defined here as well as future-defined types.

UEIDs **SHOULD NOT** be longer than 33 bytes. If they are longer, there is no guarantee that a receiver will be able to accept them. See Appendix B.

A UEID is permanent. It **MUST NOT** change for a given entity.

The different types of UEIDs 1) accommodate different manufacturing processes, 2) accommodate small UEIDs, and 3) provide an option that does not require registration fees and central administration.

In the unlikely event that a new UEID type is needed, it **MUST** be defined in an update to this document on the Standards Track.

A manufacturer of entities **MAY** use different types for different products. They **MAY** also change from one type to another for a given product or use one type for some items of a given product and another type for others.

Type Byte	Type Name	Specification
0x01	RAND	This is a 128-, 192-, or 256-bit random number generated once and stored in the entity. 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 entity and stored. It **MUST NOT** be smaller than 128 bits. See the length analysis in Appendix B.
0x02	IEEE EUI	This makes use of the device identification scheme operated by the IEEE. An Extended Unique Identifier (EUI) can be an EUI-48, EUI-60, or EUI-64 and consists of two parts. The first part is a registered company identifier, an Organizationally Unique Identifier (OUI), an OUI-36, or a Company ID (CID). The second part ensures uniqueness within the registered company. 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 entities with multiple network interfaces may have multiple MAC addresses, there is only one UEID for an entity; changeable MAC addresses that do not meet the permanence requirements in this document **MUST NOT** be used for the UEID or Semipermanent UEID (SUEID).) See [IEEE.802-2014] and [OUI.Guide].
0x03	IMEI	This makes use of the International Mobile Equipment Identity (IMEI) scheme operated by the Global System for Mobile Communications Association (GSMA). This is a 14-digit identifier consisting of an 8-digit Type Allocation Code (TAC) and a 6-digit serial number allocated by the manufacturer, which **SHALL** be encoded as a 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 encoded value **SHALL NOT** include the Luhn checksum or Software Version Number (SVN) information. See [ThreeGPP.IMEI].

Table 1: UEID Composition Types

4.2.1.2. Rules for Consuming UEIDs

For the consumer, a UEID is solely a globally unique opaque identifier. The consumer does not and should not have any awareness of the rules and structure used to achieve global uniqueness.

All implementations **MUST** be able to receive UEIDs up to 33 bytes long. 33 bytes is the longest defined in this document and gives necessary entropy for probabilistic uniqueness.

The consumer of a UEID **MUST** treat it as a completely opaque string of bytes and **MUST NOT** make any use of its internal structure. The reasons for this are:

- UEID types vary freely from one manufacturer to the next.

- New types of UEIDs may be defined.
- The manufacturer of an entity is allowed to change from one type of UEID to another anytime they want.

For example, when the consumer receives a type 0x02 UEID, they should not use the OUI part to identify the manufacturer of the device because there is no guarantee all UEIDs will be type 0x02. Different manufacturers may use different types. A manufacturer may make some of their product with one type and others with a different type or even change to a different type for newer versions of their product. Instead, the consumer should use the "oemid" claim.

4.2.2. sueids (Semipermanent UEIDs) Claim

The "sueids" claim conveys one or more semipermanent UEIDs (SUEIDs). An SUEID has the same format, characteristics, and requirements as a UEID but **MAY** change to a different value on entity life-cycle events. An entity **MAY** have both a UEID and SUEIDs, neither, or one or the other.

Examples of life-cycle events are change of ownership, factory reset, and onboarding into an IoT device management system. It is beyond the scope of this document to specify particular types of SUEIDs and the life-cycle events that trigger their change. An EAT profile **MAY** provide this specification.

There **MAY** be multiple SUEIDs. Each has a text string label, the purpose of which is to distinguish it from others. The label **MAY** name the purpose, application, or type of the SUEID. For example, the label for the SUEID used by the XYZ Onboarding Protocol could thus be "XYZ". It is beyond the scope of this document to specify any SUEID labeling schemes. They are use case specific and **MAY** be specified in an EAT profile.

If there is only one SUEID, the claim remains a map and there still **MUST** be a label.

An SUEID provides functionality similar to an IEEE Local Device Identifier (LDevID) [IEEE.802.1AR].

There are privacy considerations for SUEIDs; see Section 8.1.

A DevID URN is registered for SUEIDs; see Section 10.3.

```
$$Claims-Set-Claims //= (sueids-label => sueids-type)

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

4.2.3. oemid (Hardware OEM ID) Claim

The "oemid" claim identifies the Original Equipment Manufacturer (OEM) of the hardware. Any of the three forms described below **MAY** be used at the convenience of the claim sender. The receiver of this claim **MUST** be able to handle all three forms.

Note that the "hwmodel" claim in Section 4.2.4, the "oemboot" claim in Section 4.2.8, and the "dbgstat" claim in Section 4.2.9 depend on this claim.

Sometimes one manufacturer will acquire or merge with another. Depending on the situation and use case, newly manufactured devices may continue to use the old OEM ID or switch to a new one. This is left to the discretion of the manufacturers, but they should consider how it affects the above-mentioned claims and the attestation ecosystem for their devices. The considerations are the same for all three forms of this claim.

4.2.3.1. Random Number-Based OEM ID

The random number-based OEM ID **MUST** be 16 bytes (128 bits) long.

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 entity they make. This uniquely identifies the OEM on a statistical basis and is large enough should there be ten billion companies.

In JSON-encoded tokens, this **MUST** be base64url encoded.

4.2.3.2. IEEE-Based OEM ID

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 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. Similarly, MA-M is a 28-bit value used as the first part of a MAC address, and MA-S (formerly known as OUI-36) is a 36-bit value. Many companies have already obtained an OEM ID from the IEEE registry. A CID is also a 24-bit value from the same space as an MA-L but is not for use as a MAC address. IEEE has published Guidelines for Use of EUI, OUI, and CID [OUI.Guide] and provides a lookup service [OUI.Lookup].

Companies that have more than one of these IDs or MAC address blocks **SHOULD** select one as preferred and use that for all their entities.

Commonly, these are expressed in "hexadecimal representation" as described in [IEEE.802-2014]. When this claim is encoded, the order of bytes in the bstr is 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, and 0x48.

This format is always 3 bytes in size in CBOR.

In JSON-encoded tokens, this **MUST** be base64url encoded and always 4 bytes.

4.2.3.3. IANA Private Enterprise Number-Based OEM ID

IANA maintains a registry for Private Enterprise Numbers [PEN]. A PEN is an integer that identifies an enterprise, and it may be used to construct an object identifier (OID) relative to the following OID arc that is managed by IANA: iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1).

For EAT purposes, only the integer value assigned by IANA as the PEN is relevant, not the full OID value.

In CBOR, this value **MUST** be encoded as a major type 0 integer and is typically 3 bytes. In JSON, this value **MUST** be encoded as a number.

```
$$Claims-Set-Claims //= (
    oemid-label => oemid-type
)

oemid-type => oemid-pen / oemid-ieee / oemid-random

oemid-pen = int

oemid-ieee = JC<oemid-ieee-json, oemid-ieee-cbor>
oemid-ieee-cbor = bstr .size 3
oemid-ieee-json = base64-url-text .size 4

oemid-random = JC<oemid-random-json, oemid-random-cbor>
oemid-random-cbor = bstr .size 16
oemid-random-json = base64-url-text .size 24
```

4.2.4. hwmodel (Hardware Model) Claim

The "hwmodel" claim differentiates hardware models, products, and variants manufactured by a particular OEM, namely the one identified by the OEM ID in Section 4.2.3. It **MUST** be unique within a given OEM ID. The concatenation of the OEM ID and "hwmodel" gives a global identifier of a particular product. The "hwmodel" claim **MUST** only be present if an "oemid" claim described in Section 4.2.3 is present.

The granularity of the model identification is for each OEM to decide. It may be very granular, perhaps including some version information. It may be very general, perhaps only indicating top-level products.

The "hwmodel" claim is for use in protocols and not for human consumption. The format and encoding of this claim should not be human readable to discourage use other than in protocols. If this claim is to be derived from an already-in-use human-readable identifier, it can be run through a hash function.

There is no minimum length so that an OEM with a very small number of models can use a one-byte encoding. The maximum length is 32 bytes. All receivers of this claim **MUST** be able to receive this maximum size.

The receiver of this claim **MUST** treat it as a completely opaque string of bytes, even if there is some apparent naming or structure. The OEM is free to alter the internal structure of these bytes as long as the claim continues to uniquely identify its models.

```
$$Claims-Set-Claims //= (
    hardware-model-label => hardware-model-type
)

hardware-model-type = JC<base64-url-text .size (4..44),
                         bytes .size (1..32)>
```

4.2.5. hwversion (Hardware Version) Claim

The "hwversion" claim is a text string, of which the format is set by each manufacturer. The structure and sorting order of this text string can be specified using the version-scheme item from Concise Software Identification (CoSWID) [RFC9393]. It is useful to know how to sort versions so the newer ones can be distinguished from the older ones. A "hwversion" claim **MUST** only be present if a "hwmodel" claim, as described in Section 4.2.4, is present.

```
$$Claims-Set-Claims //=  (
    hardware-version-label => hardware-version-type
)

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

4.2.6. swname (Software Name) Claim

The "swname" claim contains a very simple free-form text value for naming the software used by the entity. Intentionally, no general rules or structure are set. This will make it unsuitable for use cases that wish precise naming.

If precise and rigorous naming of the software for the entity is needed, the "manifests" claim, as described in Section 4.2.15, may be used instead.

```
$$Claims-Set-Claims //= ( sw-name-label => tstr )
```

4.2.7. swversion (Software Version) Claim

The "swversion" claim makes use of the CoSWID version-scheme defined in [RFC9393] to give a simple version for the software. A "swversion" claim **MUST** only be present if a "swname" claim, as described in Section 4.2.6, is present.

The "manifests" claim (Section 4.2.15) may be used instead if this is too simple.

```
$$Claims-Set-Claims //= (sw-version-label => sw-version-type)

sw-version-type = [
    version:  tstr
    ? scheme:  $version-scheme
]
```

4.2.8. oemboot (OEM Authorized Boot) Claim

An "oemboot" claim with a value of "true" indicates that the entity booted with software authorized by the manufacturer of the entity as indicated by the "oemid" claim described in Section 4.2.3. It indicates that the firmware and operating system are fully under control of the OEM and may not be replaced by the end user or even the enterprise that owns the device. The means of control may be by cryptographic authentication of the software, the software being in Read-Only Memory (ROM), a combination of the two, or other. If this claim is present, the "oemid" claim **MUST** be present.

```
$$Claims-Set-Claims //= (oem-boot-label => bool)
```

4.2.9. dbgstat (Debug Status) Claim

The "dbgstat" claim applies to entity-wide or submodule-wide debug facilities of the entity like [JTAG] and diagnostic hardware built into chips. It applies to any software debug facilities related to privileged software that allows 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-entity public keys.

As with all claims, the absence of the "dbgstat" claim means it is not reported.

This claim is not extensible so as to provide a common interoperable description of debug status. 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 require 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 receiver of an EAT **MUST NOT** assume that debug is turned off in a submodule because there is a claim indicating it is turned off in a superior module.

An entity 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 receiver can know the debug status of the submodules since there is no inheritance.

4.2.9.1. Enabled

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

4.2.9.2. Disabled

This level indicates all debug facilities are currently disabled. It may be possible to enable them in the future. It may also be that they were enabled in the past but are currently disabled.

4.2.9.3. Disabled Since Boot

This level indicates all debug facilities are currently disabled and have been so since the entity booted/started.

4.2.9.4. Disabled Permanently

This level indicates all non-manufacturer facilities are permanently disabled such that no end user or developer can 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. If this debug state is reported, the "oemid" claim **MUST** be present.

4.2.9.5. Disabled Fully and Permanently

This level indicates that all debug facilities for the entity are permanently disabled.

```
$$Claims-Set-Claims //= ( debug-status-label => debug-status-type )

debug-status-type = ds-enabled /
                    disabled /
                    disabled-since-boot /
                    disabled-permanently /
                    disabled-fully-and-permanently

ds-enabled                     = JC< "enabled", 0 >
disabled                       = JC< "disabled", 1 >
disabled-since-boot            = JC< "disabled-since-boot", 2 >
disabled-permanently           = JC< "disabled-permanently", 3 >
disabled-fully-and-permanently =
                       JC< "disabled-fully-and-permanently", 4 >
```

4.2.10. location (Location) Claim

The "location" claim gives the geographic position of the entity from which the attestation originates. Latitude, longitude, altitude, accuracy, altitude-accuracy, heading, and speed **MUST** be as defined in the W3C Geolocation API [W3C.GeoLoc] (which, in turn, is based on [WGS84]). If the entity is stationary, the heading is 'null'. Latitude and longitude **MUST** be provided. If any other of these values are unknown, they are omitted.

The location may have been cached for a period of time before token creation. For example, it might have been minutes, hours, or more since the last contact with a satellite in the Global Navigation Satellite System (GNSS). Either the timestamp or the age data item can be used to quantify the cached period. The timestamp data item is preferred as it is a non-relative time. If the entity has no clock or the clock is unset but has a means to measure the time interval between the acquisition of the location and the token creation, the age may be reported instead. The age is in seconds.

See location-related privacy considerations in Section 8.2.

```
$$Claims-Set-Claims //= (location-label => location-type)

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

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

4.2.11. uptime (Uptime) Claim

The "uptime" claim contains the number of seconds that have elapsed since the entity or submodule was last booted.

```
$$Claims-Set-Claims //= (uptime-label => uint)
```

4.2.12. bootcount (Boot Count) Claim

The "bootcount" claim contains a count of the number of times the entity or submodule has been booted. Support for this claim requires a persistent storage on the device.

```
$$Claims-Set-Claims //= (boot-count-label => uint)
```

4.2.13. bootseed (Boot Seed) Claim

The "bootseed" claim contains a value created at system boot time that allows differentiation of attestation reports from different boot sessions of a particular entity (e.g., a certain UEID).

This value is usually public. It is not a secret and **MUST NOT** be used for any purpose where a secret seed is needed, such as seeding a random number generator.

There are privacy considerations for this claim; see Section 8.3.

```
$$Claims-Set-Claims //=  (boot-seed-label => binary-data)
```

4.2.14. dloas (Digital Letters of Approval) Claim

The "dloas" claim conveys one or more Digital Letters of Approval (DLOAs). A DLOA [DLOA] is a document that describes a certification that an entity has received. Examples of certifications represented by a DLOA include those issued by GlobalPlatform [GP-Example] and those based on Common Criteria [CC-Example]. 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. Verifiers **MUST NOT** issue this claim unless the entity has received the certification indicated by the DLOA.

This claim **MAY** contain more than one DLOA. If multiple DLOAs are present, verifiers **MUST NOT** issue this claim unless the entity has received all of the certifications.

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

This claim **MUST** be encoded as an array with either two or three elements. The first element **MUST** be the URL for the registrar. The second element **MUST** be a platform label indicating which platform was certified. If the DLOA applies to an application, then the third element is added, which **MUST** be an application label. The method of constructing the registrar URL, platform label, and possibly application label is specified in [DLOA].

The retriever of a DLOA **MUST** follow the recommendation in [DLOA] and use Transport Layer Security (TLS) or some other means to be sure the DLOA registrar they are accessing is authentic. The platform and application labels in the claim indicate the correct DLOA for the entity.

```
$$Claims-Set-Claims //= (
    dloas-label => [ + dloa-type ]
)

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

4.2.15. manifests (Software Manifests) Claim

The "manifests" claim contains descriptions of software present on the entity. These manifests are installed on the entity when the software is installed or are created as part of the installation process. Installation is anything that adds software to the entity, possibly factory installation, the user installing elective applications, and so on. The defining characteristic of a manifest is that it is created by the software manufacturer. The purpose of this claim is to relay unmodified manifests to the verifier and possibly to the relying party.

Some manifests are signed by their software manufacturer independently, and some are not because either they do not support signing or the manufacturer chose not to sign them. For example, a CoSWID might be signed independently before it is included in an EAT. When signed manifests are put into an EAT, the manufacturer's signature **SHOULD** be included even though an EAT's signature will also cover the manifest. This allows the receiver to directly verify the manufacturer-originated manifest.

This claim allows multiple manifest formats. For example, the manifest may be a CBOR-encoded CoSWID, an XML-encoded Software Identification (SWID) tag, or other. Identification of the type of manifest is always by a Constrained Application Protocol (CoAP) Content-Format identifier [RFC7252]. If there is no CoAP identifier registered for a manifest format, one **MUST** be registered.

This claim **MUST** be an array of one or more manifests. Each manifest in the claim **MUST** be an array of two. The first item in the array of two **MUST** be a CoAP Content-Format identifier. The second item is **MUST** be the actual manifest.

In JSON-encoded tokens, the manifest, whatever encoding it is, **MUST** be placed in a text string. When a non-text encoded manifest such as a CBOR-encoded CoSWID is put in a JSON-encoded token, the manifest **MUST** be base64 encoded.

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

A CoSWID manifest **MUST** be a payload CoSWID, not an evidence CoSWID. These are defined in [RFC9393].

This claim is extensible for use of manifest formats beyond those mentioned in this document. No particular manifest format is preferred. For manifest interoperability, an EAT profile, as defined in Section 6, should be used to specify which manifest format(s) is allowed.

```
$$Claims-Set-Claims //= (
    manifests-label => manifests-type
)

manifests-type = [+ manifest-format]

manifest-format = [
    content-type:   coap-content-format,
    content-format: JC< $manifest-body-json,
                        $manifest-body-cbor >
]

$manifest-body-cbor /= bytes .cbor untagged-coswid
$manifest-body-json /= base64-url-text
```

4.2.16. measurements (Measurements) Claim

The "measurements" claim contains descriptions, lists, evidence, or measurements of the software that exists on the entity or on any other measurable subsystem of the entity (e.g., hash of sections of a file system or non-volatile memory). The defining characteristic of this claim is that its contents are created by processes on the entity that inventory, measure, or otherwise characterize the software on the entity. The contents of this claim do not originate from the manufacturer of the measurable subsystem (e.g., developer of a software library).

This claim can be a CoSWID [RFC9393]. When the CoSWID format is used, it **MUST** be an evidence CoSWID, not a payload CoSWID.

Formats other than CoSWID **MAY** be used. The format is identified by a CoAP Content-Format identifier, which is the same for the "manifests" claim in Section 4.2.15.

```
$$Claims-Set-Claims //= (
    measurements-label => measurements-type
)

measurements-type = [+ measurements-format]

measurements-format = [
    content-type:   coap-content-format,
    content-format: JC< $measurements-body-json,
                        $measurements-body-cbor >
]

$measurements-body-cbor /= bytes .cbor untagged-coswid
$measurements-body-json /= base64-url-text
```

4.2.17. measres (Software Measurement Results) Claim

The "measres" claim is a general-purpose structure for reporting the comparison of measurements to expected reference values. This claim provides a simple standard way to report the result of a comparison as a success, a failure, not run, or absent.

It is the nature of measurement systems to be specific to the operating system, software, and hardware of the entity that is being measured. It is not possible to standardize what is measured and how it is measured across platforms, OSes, software, and hardware. The recipient must obtain the information about what was measured and what it indicates for the characterization of the security of the entity from the provider of the measurement system. What this claim provides is a standard way to report basic success or failure of the measurement. In some use cases, it is valuable to know if measurements succeeded or failed in a general way even if the details of what was measured are not characterized.

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 "measurements" claim (Section 4.2.16) to reference values.

This claim **MAY** also be generated on the entity if the entity has the ability for one subsystem to measure and evaluate 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 entity, attestation target, or submodule, multiple results can be reported. For example, it may be desirable to report the results for measurements of the file system, chip configuration, installed software, running software, and so on.

Note that this claim is not for reporting the overall result of a verifier. It is solely for reporting the result of comparison to reference values.

An individual measurement result (individual-result) is an array consisting of two elements, an identifier of the measurement (result-id), and an enumerated type of the result (result). Different measurement systems will measure different things and perhaps measure the same thing in different ways. It is up to each measurement system to define identifiers (result-id) for the measurements it reports.

Each individual measurement result is part of a group that may contain many individual results. Each group has a text string that names it, typically the name of the measurement scheme or system.

The claim itself consists of one or more groups.

The values for the results enumerated type are as follows:

1 -- comparison success: The comparison to reference values was successful.

2 -- comparison failure: The comparison was completed but did not compare correctly to the reference values.

3 -- comparison not run: The comparison was not run. This includes error conditions such as running out of memory.

4 -- measurement absent: The particular measurement was not available for comparison.

```
$$Claims-Set-Claims //= (
    measurement-results-label =>
        [ + measurement-results-group ] )

measurement-results-group = [
    measurement-system: tstr,
    measurement-results: [ + individual-result ]
]

individual-result = [
    result-id:  tstr / binary-data,
    result:     result-type,
]

result-type = comparison-success /
              comparison-fail /
              comparison-not-run /
              measurement-absent

comparison-success       = JC< "success",       1 >
comparison-fail          = JC< "fail",          2 >
comparison-not-run       = JC< "not-run",       3 >
measurement-absent       = JC< "absent",        4 >
```

4.2.18. submods (Submodules) Claim

Some devices are complex and have many subsystems. A mobile phone is a good example. It may have subsystems for communications (e.g., Wi-Fi and cellular), low-power audio and video playback, and multiple security-oriented subsystems such as a TEE and a Secure Element. The claims for a subsystem can be grouped together in a submodule.

Submodules may be used in either evidence or attestation results.

Because system architecture will vary greatly from use case to use case, there are no set requirements for what a submodule represents either in evidence or in attestation results. Profiles (Section 6) may wish to impose requirements. An attester that outputs evidence with submodules should document the semantics it associates with particular submodules for the verifier. Likewise, a verifier that outputs attestation results with submodules should document the semantics it associates with the submodules for the relying party.

A submodule claim is a map that holds some number of submodules. Each submodule is named by its label in the submodule claim map. The value of each entry in a submodule may be a Claims-Set, nested token, or Detached-Submodule-Digest. This allows for the submodule to serve as its own attester or not and allows for claims for each submodule to be represented directly or indirectly, i.e., detached.

A submodule may include a submodule, allowing for arbitrary levels of nesting. However, submodules do not inherit anything from the containing token and must explicitly include all claims. Submodules may contain claims that are present in any surrounding token or submodule. For example, the top level of the token may have a UEID, a submodule may have a different UEID, and a further subordinate submodule may also have a UEID.

The following subsections define the three types for representing submodules:

- A submodule Claims-Set
- The digest of a detached Claims-Set
- A nested token, which can be any EAT

The Submodule type and Nested-Token type definitions vary with the type of encoding. The definitions for CBOR-encoded EATs are as follows:

```
Nested-Token = CBOR-Nested-Token

CBOR-Nested-Token =
    JSON-Token-Inside-CBOR-Token /
    CBOR-Token-Inside-CBOR-Token

CBOR-Token-Inside-CBOR-Token = bstr .cbor $CBOR-Tagged-Token

JSON-Token-Inside-CBOR-Token = tstr

$$Claims-Set-Claims //= (submods-label => { + text => Submodule })

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

The Submodule and Nested-Token definitions for JSON-encoded EATs are as below. The definitions are necessarily different than CBOR because JSON has no tag mechanism and no byte-string type to help indicate that the nested token is CBOR.

```
Nested-Token = JSON-Selector

JSON-Selector = $JSON-Selector

$JSON-Selector /= [type: "JWT", nested-token: JWT-Message]
$JSON-Selector /= [type: "CBOR", nested-token:
  CBOR-Token-Inside-JSON-Token]
$JSON-Selector /= [type: "BUNDLE", nested-token: Detached-EAT-Bundle]
$JSON-Selector /= [type: "DIGEST", nested-token:
  Detached-Submodule-Digest]

CBOR-Token-Inside-JSON-Token = base64-url-text

$$Claims-Set-Claims //= (submods-label => { + text => Submodule })

Submodule = Claims-Set / JSON-Selector
```

The Detached-Submodule-Digest type is defined as follows:

```
Detached-Submodule-Digest = [
   hash-algorithm : text / int,
   digest         : binary-data
]
```

Nested tokens can be one of three types as defined in this document or types that are standardized in subsequent documents (e.g., [UCCS]). Nested tokens are the only mechanism by which JSON can be embedded in CBOR and vice versa.

The addition of further types is accomplished by augmenting the $CBOR-Tagged-Token socket or the $JSON-Selector socket.

When decoding a JSON-encoded EAT, the type of submodule is determined as follows. A JSON object indicates that the submodule is a Claims-Set. In all other cases, it is a JSON-Selector, which is an array of two elements that indicates whether the submodule is a nested token or a Detached-Submodule-Digest. The first element in the array indicates the type present in the second element. If the value is "JWT", "CBOR", "BUNDLE", or future-standardized token types, e.g., see [UCCS], the submodule is a nested token of the indicated type, i.e., JWT-Message, CBOR-Token-Inside-JSON-Token, Detached-EAT-Bundle, or a future type. If the value is "DIGEST", the submodule is a Detached-Submodule-Digest. Any other value indicates a standardized extension to this specification.

When decoding a CBOR-encoded EAT, the CBOR item type indicates the type of the submodule as follows. A map indicates a CBOR-encoded submodule Claims-Set. An array indicates a CBOR-encoded Detached-Submodule-Digest. A byte string indicates a CBOR-encoded CBOR-Nested-Token. A text string indicates a JSON-encoded JSON-Selector. Where JSON-Selector is used in a CBOR-encoded EAT, the "DIGEST" type and corresponding Detached-Submodule-Digest type **MUST NOT** be used.

The type of a CBOR-encoded nested token is always determined by the CBOR tag encountered after the byte string wrapping is removed in a CBOR-encoded enclosing token or after the base64 wrapping is removed in a JSON-encoded enclosing token.

The type of JSON-encoded nested token is always determined by the string name in JSON-Selector and is always "JWT", "BUNDLE", or a new name standardized outside this document for a further type (e.g., "UCCS"). This string name may also be "CBOR" to indicate the nested token is CBOR encoded.

"JWT": The second array item **MUST** be a JWT formatted according to [RFC7519].

"CBOR": The second array item **MUST** be some base64url-encoded CBOR that is a tag, typically a CWT or CBOR-encoded detached EAT bundle.

"BUNDLE": The second array item **MUST** be a JSON-encoded detached EAT bundle as defined in this document.

"DIGEST": The second array item **MUST** be a JSON-encoded Detached-Submodule-Digest as defined in this document.

As noted elsewhere, additional EAT types may be defined by a Standards Action. New type specifications **MUST** address the integration of the new type into the submodule claim type for submodules.

4.2.18.1. Submodule Claims-Set

The Claims-Set type provides a means of representing claims from a submodule that does not have its own attesting environment, i.e., it has no keys distinct from the attester producing the surrounding token. Claims are represented as a Claims-Set. Submodule claims represented in this way are secured by the same mechanism as the enclosing token (e.g., it is signed by the same attestation key).

The encoding of a submodule Claims-Set **MUST** be the same as the encoding of the surrounding EAT, e.g., all submodule Claims-Sets in a CBOR-encoded token must be CBOR encoded.

4.2.18.2. Detached Submodule Digest

The Detached-Submodule-Digest type is similar to a submodule Claims-Set, except a digest of the Claims-Set is included in the claim with the Claims-Set contents conveyed separately. The separately conveyed Claims-Set is called a "detached claims set". The input to the digest algorithm is the CBOR or the JSON-encoded Claims-Set for the submodule. There is no byte string wrapping or base64 encoding.

The data type for this type of submodule is an array consisting of two data items: an algorithm identifier and a byte string containing the digest. The hash algorithm identifier is always from the "COSE Algorithms" registry [IANA.COSE.Algorithms]. Either the integer or string identifier may be used. The hash algorithm identifier is never from any other algorithm registry.

A detached EAT bundle, as described in Section 5, may be used to convey detached claims sets and the EAT containing the corresponding detached digests. However, EAT does not require the use of a detached EAT bundle. Any other protocols may be used to convey detached claims sets and the EAT containing the corresponding detached digests. If detached Claims-Sets are modified in transit, then validation can fail.

4.2.18.3. Nested Tokens

The CBOR-Nested-Token and JSON-Selector types provide a means of representing claims from a submodule that has its own attesting environment, i.e., it has keys distinct from the attester producing the surrounding token. Claims are represented in a signed EAT token.

Inclusion of a signed EAT as a claim cryptographically binds the EAT 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 need not use the same encoding as the enclosing token. This enables composite devices to be built without regards to the encoding used by components. Thus, a CBOR-encoded EAT can have a JSON-encoded EAT as a nested token and vice versa.

4.3. Claims Describing the Token

The claims in this section provide metadata about the token they occur in. They do not describe the entity. They may appear in evidence or attestation results.

4.3.1. iat (Timestamp) Claim

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 of floating-point for this claim, whereas EAT disallows the use of floating-point. An EAT token **MUST NOT** contain an "iat" claim in floating-point format. Any recipient of a token with a floating-point format "iat" claim **MUST** consider it an error.

A 64-bit integer representation of the CBOR epoch-based time [RFC8949] used by this claim 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.

4.3.2. eat_profile (EAT Profile) Claim

See Section 6 for the detailed description of an EAT profile.

The "eat_profile" claim identifies an EAT profile by either a Uniform Resource Identifier (URI) 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 "eat_profile" claim is to uniquely identify the profile even if it is a private profile.

The OID is always absolute and never relative.

See Section 7.2.1 for OID and URI encoding.

```
$$Claims-Set-Claims //= (profile-label => general-uri / general-oid)
```

4.3.3. intuse (Intended Use) Claim

EATs may be employed in the context of several different applications. The "intuse" 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 utilizes EAT. The possible values are in the "Entity Attestation Token (EAT) Intended Uses" registry defined in Section 10.5.

```
$$Claims-Set-Claims //= ( intended-use-label => intended-use-type )

intended-use-type = JC< text, int>
```

5. Detached EAT Bundles

A detached EAT bundle is a message to convey an EAT plus detached claims sets secured by that EAT. It is a top-level message like a CWT or JWT. It can occur in any place that a CWT or JWT occurs, for example, as a submodule nested token as defined in Section 4.2.18.3.

A detached EAT bundle may be either CBOR or JSON encoded.

A detached EAT bundle consists of two parts.

The first part is an encoded EAT that:

- **MUST** have at least one submodule that is a detached submodule digest as defined in Section 4.2.18.2
- **MAY** be either CBOR or JSON encoded and does not have to be the same as the encoding of the bundle
- **MAY** be a CWT, JWT, or some future-defined token type, but it **MUST NOT** be a detached EAT bundle
- **MUST** be authenticity and integrity protected

The same mechanism for distinguishing the type for nested token submodules is employed here.

The second part is a map/object that:

- **MUST** be a Claims-Set
- **MUST** use the same encoding as the bundle
- **MUST** be wrapped in a byte string when the encoding is CBOR and be base64url encoded when the encoding is JSON

For a CBOR-encoded detached EAT bundle, tag 602 can be used to identify it. The standard 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 that 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.

```
BUNDLE-Messages = BUNDLE-Tagged-Message / BUNDLE-Untagged-Message

BUNDLE-Tagged-Message   = #6.602(BUNDLE-Untagged-Message)
BUNDLE-Untagged-Message = Detached-EAT-Bundle

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

json-wrapped-claims-set = base64-url-text

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

6. Profiles

EAT makes normative use of CBOR, JSON, COSE, JOSE, CWT, and JWT. Most of these have implementation options to accommodate a range of use cases.

For example, COSE does not require a particular set of cryptographic algorithms so as to accommodate different usage scenarios and evolution of algorithms over time. Section 10 of [RFC9052] describes the profiling considerations for COSE.

The use of encryption is optional for both CWT and JWT. Section 8 of [RFC7519] describes implementation requirements and recommendations for JWT.

Similarly, CBOR provides indefinite-length encoding, which is not commonly used but is valuable for very constrained devices. For EAT itself, in a particular use case some claims will be used and others will not. Section 4 of [RFC8949] describes serialization considerations for CBOR.

For example, a mobile phone use case may require the device make and model and may prohibit UEID and location for privacy reasons. The general EAT standard retains all this flexibility because it too is aimed to accommodate a broad range of use cases.

It is necessary to explicitly narrow these implementation options to guarantee interoperability. EAT chooses one general and explicit mechanism, the profile, to indicate the choices made for these implementation options for all aspects of the token.

Below is a list of the various issues that should be addressed by a profile.

The "eat_profile" claim in Section 4.3.2 provides a unique identifier for the profile a particular token uses.

A profile can apply to evidence, attestation results, or both.

6.1. Format of a Profile Document

A profile document does not have to be in any particular format. It may be simple text, something more formal, or a combination of both.

A profile may define, and possibly register, one or more new claims if needed. A profile may also reuse one or more already defined claims either as is or with values constrained to a subset or subrange.

6.2. Full and Partial Profiles

For a "full" profile, the receiver will be able to decode and verify every possible EAT sent when a sender and receiver both adhere to it. For a "partial" profile, there are still some protocol options left undecided.

For example, a profile that allows the use of signing algorithms by the sender that the receiver is not required to support is a partial profile. The sender might choose a signing algorithm that some receivers do not support.

Full profiles **MUST** be complete such that a complying receiver can decode, verify, and check for freshness for every EAT created by a complying sender. Full profiles do not need to require the receiver to fully handle every claim in an EAT from a complying sender. Profile specifications may assume the receiver has access to the necessary verification keys or may go into specific detail on the means to access verification keys.

The "eat_profile" claim **MUST NOT** be used to identify partial profiles.

While fewer profiles are preferable, sometimes several may be needed for a use case. One approach to handling variation in devices might be to define several full profiles that are variants of each other. It is relatively easy and inexpensive to define profiles as they do not have to be published on the Standards Track and do not have to be registered anywhere. For example, flexibility for post-quantum algorithms can be handled as follows. First, define a full profile for a set of non-post-quantum algorithms for current use. Then, when post-quantum algorithms are settled, define another full profile derived from the first.

6.3. List of Profile Issues

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

6.3.1. Use of JSON, CBOR, or Both

A profile should specify whether CBOR, JSON, or both may be sent. A profile should specify that the receiver can accept all encodings that the sender is allowed to send.

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

6.3.2. CBOR Map and Array Encoding

A profile should specify whether definite-length arrays/maps, indefinite-length arrays/maps, or both may be sent. A profile should specify that the receiver accepts all length encodings that the sender is allowed to send.

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

For most use cases, specifying that only definite-length arrays/maps may be sent is suitable.

6.3.3. CBOR String Encoding

A profile should specify whether definite-length strings, indefinite-length strings, or both may be sent. A profile should specify that the receiver accepts all types of string encodings that the sender is allowed to send.

For most use cases, specifying that only definite-length strings may be sent is suitable.

6.3.4. CBOR Preferred Serialization

A profile should specify whether or not CBOR preferred serialization must be sent or not. A profile should specify that the receiver accepts preferred and/or non-preferred serialization, so it will be able to accept anything sent by the sender.

6.3.5. CBOR Tags

The profile should specify whether the token should be a CWT tag or not.

When COSE protection is used, the profile should specify whether COSE tags are used or not. Note that [RFC8392] requires COSE tags be used in a CWT tag.

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

6.3.6. COSE/JOSE Protection

COSE and JOSE have several options for signed, MACed, and encrypted messages. It may be an Unsecured JWT as described in Section 6 of [RFC7519]. It is possible to implement no protection, sign only, MAC only, sign then encrypt, and so on. All combinations allowed by COSE, JOSE, JWT, and CWT are allowed by EAT.

A profile should specify all signing, encryption, and MAC message formats that may be sent. For example, a profile might allow only COSE_Sign1 to be sent. As another example, a profile might allow COSE_Sign and COSE_Encrypt to be sent to carry multiple signatures for post quantum cryptography and to use encryption to provide confidentiality.

A profile should specify that the receiver accepts all message formats that are allowed to be sent.

When both signing and encryption are allowed, a profile should specify which is applied first.

6.3.7. COSE/JOSE Algorithms

See "Application Profiling Considerations" (Section 10 of [RFC9052]) for a discussion on the selection of cryptographic algorithms and related issues.

The profile **MAY** require the protocol or system using EAT to provide an algorithm negotiation mechanism.

If not, the profile document should list a set of algorithms for each COSE and JOSE message type allowed by the profile per Section 6.3.6. The verifier should implement all of them. The attester may implement any of them it wishes, possibly just one for each message type.

If detached submodule digests are used, the profile should address the determination of the hash algorithm(s) for the digests.

6.3.8. Detached EAT Bundle Support

A profile should specify whether or not a detached EAT bundle (Section 5) can be sent. A profile should specify that a receiver accepts a detached EAT bundle if the sender is allowed to send it.

6.3.9. Key Identification

A profile should specify what must be sent to identify the verification, decryption, or MAC key(s). If multiple methods of key identification may be sent, a profile should require the receiver to support them all.

Appendix F describes a number of methods for identifying verification keys. When encryption is used, there are further considerations. In some cases, key identification may be very simple, and in other cases, multiple components may be involved. For example, it may be simple through the use of a COSE key ID, or it may be complex through the use of an X.509 certificate hierarchy.

While not always possible, a profile should specify, or make reference to, a full end-to-end specification for key identification. For example, a profile should specify in full detail how COSE key IDs are to be created, their life cycle, and such rather than just specifying that a COSE key ID be used. For example, a profile should specify the full details of an X.509 hierarchy including extension processing, algorithms allowed, and so on rather than just saying X.509 certificates are used.

6.3.10. 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, whereas key identification is not.

6.3.11. Freshness

Security considerations (see Section 9.3) require a mechanism to provide freshness. This may be the EAT nonce claim in Section 4.1 or some claim or mechanism defined outside this document. Several options are described in "Freshness" (Section 10 of [RFC9334]). A profile should specify which freshness mechanism or mechanisms can be used.

If the EAT nonce claim is used, a profile should specify whether multiple nonces may be sent. If a profile allows multiple nonces to be sent, it should require the receiver to process multiple nonces.

6.3.12. Claims Requirements

A profile may define new claims that are not defined in this document.

This document requires that an EAT receiver must accept tokens with claims it does not understand. A profile for a specific use case may reverse this and allow a receiver to reject tokens with claims it does not understand. A profile for a specific use case may specify that specific claims are prohibited.

A profile for a specific use case may modify this and specify that some claims are required.

A profile may constrain the definition of claims that are defined in this document or elsewhere. For example, a profile may require the EAT nonce to be a certain length or the "location" claim to always include the altitude.

Some claims are "pluggable" in that they allow different formats for their content. The "manifests" claim (Section 4.2.15) and the "measurements" claim (Section 4.2.16) are examples of this, allowing the use of CoSWID and other formats. A profile should specify which formats are allowed to be sent, with the assumption that the corresponding CoAP content types have been registered. A profile should require the receiver to accept all formats that are allowed to be sent.

Further, if there is variation within a format that is allowed, the profile should specify which variations can be sent. For example, there are variations in the CoSWID format. A profile might require the receiver to accept all variations that are allowed to be sent.

6.4. The Constrained Device Standard Profile

It is anticipated that there will be many profiles defined for EAT for many different use cases. This section gives a normative definition of one profile that is good for many constrained device use cases.

The identifier for this profile is "urn:ietf:rfc:rfc9711".

Issue	Profile Definition
CBOR/JSON	CBOR **MUST** be used.
CBOR Encoding	Definite-length maps and arrays **MUST** be used.
CBOR Encoding	Definite-length strings **MUST** be used.
CBOR Serialization	Preferred serialization **MUST** be used.
COSE Protection	COSE_Sign1 **MUST** be used.

Issue	Profile Definition
Algorithms	The receiver **MUST** accept ES256, ES384, and ES512; the sender **MUST** send one of these.
Detached EAT Bundle Usage	Detached EAT bundles **MUST NOT** be sent with this profile.
Verification Key Identification	Either the COSE key identifier (kid) or the UEID **MUST** be used to identify the verification key. If both are present, the kid takes precedence. (It is assumed the receiver has access to a database of trusted verification keys, which allows a lookup of the verification key ID; the key format and means of distribution are beyond the scope of this profile.)
Endorsements	This profile contains no endorsement identifier.
Freshness	A new single unique nonce **MUST** be used for every token request.
Claims	No requirement is made for the presence or absence of claims other than requiring an EAT nonce. As per general EAT rules, the receiver **MUST NOT** error out on claims it does not understand.

Table 2: Constrained Device Profile Definition

Any profile with different requirements than those above **MUST** have a different profile identifier.

Note that many claims can be present for tokens conforming to this profile, even claims not defined in this document. Note also that even slight deviation from the above requirements is considered a different profile that **MUST** have a different identifier. For example, if a kid (key identifier) or UEID is not used for key identification, it is not in conformance with this profile. As another example, requiring the presence of some claim is also not in conformance and requires another profile.

Derivations of this profile are encouraged. For example, another profile may be simply defined as "The Constrained Device Standard Profile" plus the requirement for the presence of claim xxxx and claim yyyy.

7. 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 cannot (e.g., tags) or that are expressed differently (e.g., labels), there is some CDDL that is specific to the encoding.

7.1. 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. 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, and JWT. 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, or JWT in CDDL without it. The CDDL definition of Claims-Set here is applicable to EAT, CWT, and JWT.

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 encoded in either CBOR or JSON, never a mixture.

CDDL for the seven claims defined by [RFC8392] and [RFC7519] is also specified in this document.

7.2. Encoding Data Types

The following subsections use the types defined in "Standard Prelude" (Appendix D of [RFC8610]).

7.2.1. Common Data Types

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

For CBOR-encoded tokens, OIDs are specified using the CDDL type name "oid" from [RFC9090]. They are encoded without the tag number. For JSON-encoded tokens, OIDs are text strings in the common form of "nn.nn.nn...".

Unless explicitly indicated, URIs are not the URI tag defined in [RFC8949]. They are just text strings that contain a URI conforming to the format defined in [RFC3986].

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

binary-data = JC< base64-url-text, bstr>

base64-url-text = tstr .regexp "[A-Za-z0-9_-]+"

general-oid = JC< json-oid, ~oid >

json-oid = tstr .regexp "([0-2])((\\.0)|(\\.[1-9][0-9]*))*"

general-uri = JC< text, ~uri >

coap-content-format = uint .le 65535
```

7.2.2. JSON Interoperability

JSON should be encoded per Appendix E of [RFC8610]. 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 in Section 2 of [RFC7519].

- string-or-uri -- **MUST** be encoded as StringOrURI as described in Section 2 of [RFC7519].
- uri -- **MUST** be a URI [RFC3986].
- oid -- **MUST** be encoded as a string using the well-established dotted-decimal notation (e.g., the text "1.2.250.1") [RFC4517].

The CDDL generic "JC<>" is used in most places where there is a variance between CBOR and JSON. The first argument is the CDDL for JSON, and the second is CDDL for CBOR.

7.2.3. Labels

Most map labels, Claims-Keys, Claim-Names, and enumerated-type values are integers for CBOR-encoded tokens and strings for JSON-encoded tokens. When this is the case, the JC<> CDDL construct is used to give both the integer and string values.

7.2.4. CBOR Interoperability

CBOR allows data items to be serialized in more than one form to accommodate a variety of use cases. This is addressed in Section 6.

7.3. Collected CDDL

See [EAT-GitHub] for additional information and stub files, when using the CDDL presented in this section to validate EAT protocol messages.

7.3.1. Payload CDDL

The payload CDDL defines all the EAT claims that are added to the main definition of a Claims-Set in Appendix D. Claims-Set is the payload for CWT, JWT, and potentially other token types. This is for both CBOR and JSON. When there is variation between CBOR and JSON, the JC<> CDDL generic defined in Appendix D is used. Note that the JC<> generic uses the CDDL ".feature" control operator defined in [RFC9165].

This CDDL uses, but does not define, Submodule or nested tokens because the definition for these types varies between CBOR and JSON and the JC<> generic cannot be used to define it. The submodule claim is the one place where a CBOR token can be nested inside a JSON token and vice versa. Encoding-specific definitions are provided in the following sections.

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

binary-data = JC< base64-url-text, bstr>

base64-url-text = tstr .regexp "[A-Za-z0-9_-]+"

general-oid = JC< json-oid, ~oid >

json-oid = tstr .regexp "([0-2])((\\.0)|(\\.[1-9][0-9]*))*"

general-uri = JC< text, ~uri >

coap-content-format = uint .le 65535
```

```
$$Claims-Set-Claims //=
    (nonce-label => nonce-type / [ 2* nonce-type ])

nonce-type = JC< tstr .size (8..88), bstr .size (8..64)>


$$Claims-Set-Claims //= (ueid-label => ueid-type)

ueid-type = JC<base64-url-text .size (10..44) , bstr .size (7..33)>

$$Claims-Set-Claims //= (sueids-label => sueids-type)

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

$$Claims-Set-Claims //= (
    oemid-label => oemid-pen / oemid-ieee / oemid-random
)

oemid-pen = int

oemid-ieee = JC<oemid-ieee-json, oemid-ieee-cbor>
oemid-ieee-cbor = bstr .size 3
oemid-ieee-json = base64-url-text .size 4

oemid-random = JC<oemid-random-json, oemid-random-cbor>
oemid-random-cbor = bstr .size 16
oemid-random-json = base64-url-text .size 24


$$Claims-Set-Claims //=  (
    hardware-version-label => hardware-version-type
)

hardware-version-type = [
    version:  tstr,
    ? scheme:  $version-scheme
]

$$Claims-Set-Claims //= (
    hardware-model-label => hardware-model-type
)

hardware-model-type = JC<base64-url-text .size (4..44),
                         bytes .size (1..32)>

$$Claims-Set-Claims //= ( sw-name-label => tstr )

$$Claims-Set-Claims //= (sw-version-label => sw-version-type)

sw-version-type = [
    version:  tstr
    ? scheme:  $version-scheme
]

$$Claims-Set-Claims //= (oem-boot-label => bool)
```

```
$$Claims-Set-Claims //= ( debug-status-label => debug-status-type )

debug-status-type = ds-enabled /
                    disabled /
                    disabled-since-boot /
                    disabled-permanently /
                    disabled-fully-and-permanently

ds-enabled                     = JC< "enabled", 0 >
disabled                       = JC< "disabled", 1 >
disabled-since-boot            = JC< "disabled-since-boot", 2 >
disabled-permanently           = JC< "disabled-permanently", 3 >
disabled-fully-and-permanently =
                       JC< "disabled-fully-and-permanently", 4 >

$$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          = JC< "latitude",          1 >
longitude         = JC< "longitude",         2 >
altitude          = JC< "altitude",          3 >
accuracy          = JC< "accuracy",          4 >
altitude-accuracy = JC< "altitude-accuracy", 5 >
heading           = JC< "heading",           6 >
speed             = JC< "speed",             7 >
timestamp         = JC< "timestamp",         8 >
age               = JC< "age",               9 >

$$Claims-Set-Claims //= (uptime-label => uint)

$$Claims-Set-Claims //=  (boot-seed-label => binary-data)

$$Claims-Set-Claims //= (boot-count-label => uint)

$$Claims-Set-Claims //= ( intended-use-label => intended-use-type )

intended-use-type = JC< text, int>

$$Claims-Set-Claims //= (
    dloas-label => [ + dloa-type ]
)

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

```
]

$$Claims-Set-Claims //= (profile-label => general-uri / general-oid)

$$Claims-Set-Claims //= (
    manifests-label => manifests-type
)

manifests-type = [+ manifest-format]

manifest-format = [
    content-type:   coap-content-format,
    content-format: JC< $manifest-body-json,
                        $manifest-body-cbor >
]

$manifest-body-cbor /= bytes .cbor untagged-coswid
$manifest-body-json /= base64-url-text


$$Claims-Set-Claims //= (
    measurements-label => measurements-type
)

measurements-type = [+ measurements-format]

measurements-format = [
    content-type:   coap-content-format,
    content-format: JC< $measurements-body-json,
                        $measurements-body-cbor >
]

$measurements-body-cbor /= bytes .cbor untagged-coswid
$measurements-body-json /= base64-url-text


$$Claims-Set-Claims //= (
    measurement-results-label =>
        [ + measurement-results-group ] )

measurement-results-group = [
    measurement-system: tstr,
    measurement-results: [ + individual-result ]
]

individual-result = [
    result-id:  tstr / binary-data,
    result:     result-type,
]

result-type = comparison-success /
              comparison-fail /
              comparison-not-run /
              measurement-absent

comparison-success       = JC< "success",       1 >
comparison-fail          = JC< "fail",          2 >
comparison-not-run       = JC< "not-run",       3 >
```

```
measurement-absent       = JC< "absent",        4 >



Detached-Submodule-Digest = [
   hash-algorithm : text / int,
   digest         : binary-data
]


BUNDLE-Messages = BUNDLE-Tagged-Message / BUNDLE-Untagged-Message

BUNDLE-Tagged-Message   = #6.602(BUNDLE-Untagged-Message)
BUNDLE-Untagged-Message = Detached-EAT-Bundle

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

json-wrapped-claims-set = base64-url-text

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



nonce-label                = JC< "eat_nonce",    10 >
ueid-label                 = JC< "ueid",         256 >
sueids-label               = JC< "sueids",       257 >
oemid-label                = JC< "oemid",        258 >
hardware-model-label       = JC< "hwmodel",      259 >
hardware-version-label     = JC< "hwversion",    260 >
uptime-label               = JC< "uptime",       261 >
oem-boot-label             = JC< "oemboot",      262 >
debug-status-label         = JC< "dbgstat",      263 >
location-label             = JC< "location",     264 >
profile-label              = JC< "eat_profile",  265 >
submods-label              = JC< "submods",      266 >
boot-count-label           = JC< "bootcount",    267 >
boot-seed-label            = JC< "bootseed",     268 >
dloas-label                = JC< "dloas",        269 >
sw-name-label              = JC< "swname",       270 >
sw-version-label           = JC< "swversion",    271 >
manifests-label            = JC< "manifests",    272 >
measurements-label         = JC< "measurements", 273 >
measurement-results-label  = JC< "measres" ,     274 >
intended-use-label         = JC< "intuse",       275 >
```

7.3.2. CBOR-Specific CDDL

```
EAT-CBOR-Token = $CBOR-Tagged-Token / $EAT-CBOR-Untagged-Token

$CBOR-Tagged-Token /= CWT-Tagged-Message
$CBOR-Tagged-Token /= BUNDLE-Tagged-Message

$EAT-CBOR-Untagged-Token /= CWT-Untagged-Message
$EAT-CBOR-Untagged-Token /= BUNDLE-Untagged-Message


Nested-Token = CBOR-Nested-Token

CBOR-Nested-Token =
    JSON-Token-Inside-CBOR-Token /
    CBOR-Token-Inside-CBOR-Token

CBOR-Token-Inside-CBOR-Token = bstr .cbor $CBOR-Tagged-Token

JSON-Token-Inside-CBOR-Token = tstr

$$Claims-Set-Claims //= (submods-label => { + text => Submodule })

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

7.3.3. JSON-Specific CDDL

```
EAT-JSON-Token = $EAT-JSON-Token-Formats

$EAT-JSON-Token-Formats /= JWT-Message
$EAT-JSON-Token-Formats /= BUNDLE-Untagged-Message


Nested-Token = JSON-Selector

JSON-Selector = $JSON-Selector

$JSON-Selector /= [type: "JWT", nested-token: JWT-Message]
$JSON-Selector /= [type: "CBOR", nested-token:
  CBOR-Token-Inside-JSON-Token]
$JSON-Selector /= [type: "BUNDLE", nested-token: Detached-EAT-Bundle]
$JSON-Selector /= [type: "DIGEST", nested-token:
  Detached-Submodule-Digest]

CBOR-Token-Inside-JSON-Token = base64-url-text

$$Claims-Set-Claims //= (submods-label => { + text => Submodule })

Submodule = Claims-Set / JSON-Selector
```

8. Privacy Considerations

Certain EAT claims can be used to track the owner of an entity; therefore, implementations should consider privacy-preserving options dependent on the usage of the EAT. For example, the location claim might be suppressed in EATs sent to unauthenticated consumers.

8.1. UEID and SUEID Privacy Considerations

A UEID is usually not privacy-preserving. Relying parties receiving tokens from a particular entity will be able to know that the tokens are from the same entity and identify the entity issuing those tokens.

Thus, the use of the claim may violate privacy policies. In other usage situations, a UEID will not be allowed for certain products such as 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 it is generated.

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

- The entity obtains explicit permission from the user of the entity to use the UEID/SUEID; this may be through a prompt or 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 use case. It is used only by one relying party.
- The entity 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 entity, then the entity generates a UEID just for that relying party by hashing a proofed relying party ID with the main entity UEID/SUEID.

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

8.2. Location Privacy Considerations

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

8.3. Boot Seed Privacy Considerations

The "bootseed" claim is effectively a stable entity identifier within a given boot epoch. Therefore, it is not suitable for use in attestation schemes that are privacy-preserving.

8.4. Replay Protection and Privacy

EAT defines the EAT nonce claim for replay protection and token freshness. The nonce claim is based on a value usually derived remotely (outside of the entity). This claim might be used to extract and convey personally identifying information either inadvertently or by intention. For instance, an implementor may choose a nonce equivalent to a username associated with the device (e.g., account login). If the token is inspected by a third party, then this information could be used to identify the source of the token or an account associated with the token. To avoid the conveyance of privacy-related information in the nonce claim, it should be derived using a salt that originates from a true and reliable random number generator or any other source of randomness that would still meet the target system requirements for replay protection and token freshness.

9. Security Considerations

The security considerations provided in Section 8 of [RFC8392] and of Section 11 of [RFC7519] apply to EAT in its CWT and JWT form, respectively. Moreover, Section 12 of [RFC9334] is also applicable to implementations of EAT. In addition, implementors should consider the information in the following subsections.

9.1. Claim Trustworthiness

This specification defines semantics for each claim. It does not require any particular level of security in the implementation of the claims or even for the attester itself. Such specification is far beyond the scope of this document, which is about a message format not the security level of an implementation.

The receiver of an EAT knows the trustworthiness of the claims in it by understanding the implementation made by the attester vendor and/or understanding the checks and processing performed by the verifier.

For example, this document states that a UEID is permanent and that it must not change, but it does not describe any security requirements or a level of defense to prevent an attacker from changing the UEID.

The degree of security will vary from use case to use case. In some cases, the receiver may only need to know something of the implementation such as that it was implemented in a TEE. In other cases, the receiver may require the attester to be certified by a particular certification program. Or perhaps the receiver is content with very little security.

9.2. Key Provisioning

Private key material can be used to sign and/or encrypt the EAT or 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 manufacturer 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.

9.2.1. 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 (e.g., key wrapping) may be necessary. The transmission itself may be performed electronically, but it 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.

9.3. Freshness

All EAT use **MUST** provide a freshness mechanism to prevent replay and related attacks. The extensive discussions in [RFC9334] on freshness, as well as the security considerations, apply here. One option to provide freshness is the EAT nonce claim (Section 4.1).

9.4. 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, and 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 **MUST** leverage an equivalent communication security protocol (e.g., 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, the nested EAT itself is encrypted and cryptographically verifiable (due to its COSE envelope) by a downstream consumer (unlike the previous example where a claims set without a COSE envelope is sent to a

downstream consumer). Therefore, TLS 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.

9.5. Detached EAT Bundle Digest Security Considerations

A detached EAT bundle is composed of a nested EAT and a claims set as per Section 5. Although the attached claims set is vulnerable to modification in transit, any modification can be detected by the receiver through the associated digest, which is a claim fully contained within an EAT. Moreover, the digest itself can only be derived using an appropriate COSE hash algorithm, implying that an attacker cannot induce false detection of modified detached claims because the algorithms in the COSE registry are assumed to be of sufficient cryptographic strength.

9.6. Verification Keys

In all cases, there must be some way that the verification key itself is 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 "Key Identification" of JSON Web Signature (Section 6 of [RFC7515]).

10. IANA Considerations

10.1. 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 reused. No new IANA registry is created.

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

Appendix E describes some considerations when defining new claims.

10.2. CWT and JWT Claims Registered by This Document

Per this specification, the following values have been added to the "JSON Web Token Claims" registry established by [RFC7519] and the "CBOR Web Token (CWT) Claims" registry established by [RFC8392]. Each entry below has been added to both registries.

The "Claim Description", "Change Controller", and "Reference" fields are common and equivalent for the JWT and CWT registries. The "Claim Key" and "Claim Value Type" fields are for the CWT registry only. The "Claim Name" field is as defined for the CWT registry, not the JWT registry. The "JWT Claim Name" field is equivalent to the "Claim Name" field in the JWT registry.

IANA has registered the following claims.

Claim Name: Nonce
Claim Description: Nonce
JWT Claim Name: eat_nonce
Claim Key: 10
Claim Value Type: bstr or array
Change Controller: IETF
Reference: RFC 9711

Claim Name: UEID
Claim Description: Universal Entity ID
JWT Claim Name: ueid
CWT Claim Key: 256
Claim Value Type: bstr
Change Controller: IETF
Reference: RFC 9711

Claim Name: SUEIDs
Claim Description: Semipermanent UEIDs
JWT Claim Name: sueids
CWT Claim Key: 257
Claim Value Type: map
Change Controller: IETF
Reference: RFC 9711

Claim Name: Hardware OEM ID
Claim Description: Hardware OEM ID
JWT Claim Name: oemid
Claim Key: 258
Claim Value Type: bstr or int
Change Controller: IETF
Reference: RFC 9711

Claim Name: Hardware Model
Claim Description: Model identifier for hardware
JWT Claim Name: hwmodel
Claim Key: 259

Claim Value Type: bstr
Change Controller: IETF
Reference: RFC 9711

Claim Name: Hardware Version
Claim Description: Hardware Version Identifier
JWT Claim Name: hwversion
Claim Key: 260
Claim Value Type: array
Change Controller: IETF
Reference: RFC 9711

Claim Name: Uptime
Claim Description: Uptime
JWT Claim Name: uptime
Claim Key: 261
Claim Value Type: uint
Change Controller: IETF
Reference: RFC 9711

Claim Name: OEM Authorized Boot
Claim Description: Indicates whether the software booted was OEM authorized
JWT Claim Name: oemboot
Claim Key: 262
Claim Value Type: bool
Change Controller: IETF
Reference: RFC 9711

Claim Name: Debug Status
Claim Description: The status of debug facilities
JWT Claim Name: dbgstat
Claim Key: 263
Claim Value Type: uint
Change Controller: IETF
Reference: RFC 9711

Claim Name: Location
Claim Description: The geographic location
JWT Claim Name: location
Claim Key: 264
Claim Value Type: map
Change Controller: IETF
Reference: RFC 9711

Claim Name: EAT Profile
Claim Description: The EAT profile followed
JWT Claim Name: eat_profile
Claim Key: 265
Claim Value Type: uri or oid
Change Controller: IETF
Reference: RFC 9711

Claim Name: Submodules Section
Claim Description: The section containing submodules
JWT Claim Name: submods
Claim Key: 266
Claim Value Type: map
Change Controller: IETF
Reference: RFC 9711

Claim Name: Boot Count
Claim Description: The number of times the entity or submodule has been booted
JWT Claim Name: bootcount
Claim Key: 267
Claim Value Type: uint
Change Controller: IETF
Reference: RFC 9711

Claim Name: Boot Seed
Claim Description: Identifies a boot cycle
JWT Claim Name: bootseed
Claim Key: 268
Claim Value Type: bstr
Change Controller: IETF
Reference: RFC 9711

Claim Name: DLOAs
Claim Description: Certifications received as Digital Letters of Approval
JWT Claim Name: dloas
Claim Key: 269
Claim Value Type: array
Change Controller: IETF
Reference: RFC 9711

Claim Name: Software Name
Claim Description: The name of the software running in the entity

JWT Claim Name: swname
Claim Key: 270
Claim Value Type: tstr
Change Controller: IETF
Reference: RFC 9711

Claim Name: Software Version
Claim Description: The version of software running in the entity
JWT Claim Name: swversion
Claim Key: 271
Claim Value Type: array
Change Controller: IETF
Reference: RFC 9711

Claim Name: Software Manifests
Claim Description: Manifests describing the software installed on the entity
JWT Claim Name: manifests
Claim Key: 272
Claim Value Type: array
Change Controller: IETF
Reference: RFC 9711

Claim Name: Measurements
Claim Description: Measurements of the software, memory configuration, and such on the entity
JWT Claim Name: measurements
Claim Key: 273
Claim Value Type: array
Change Controller: IETF
Reference: RFC 9711

Claim Name: Software Measurement Results
Claim Description: The results of comparing software measurements to reference values
JWT Claim Name: measres
Claim Key: 274
Claim Value Type: array
Change Controller: IETF
Reference: RFC 9711

Claim Name: Intended Use
Claim Description: The intended use of the EAT
JWT Claim Name: intuse
Claim Key: 275

Claim Value Type: uint
Change Controller: IETF
Reference: RFC 9711

10.3. UEID URNs Registered by This Document

IANA has registered the following new subtypes in the "DEV URN Subtypes" registry [IANA.DEV-URNs] under the "Device Identification" registry group; see [RFC9039].

Subtype	Description	Reference
ueid	Universal Entity ID	RFC 9711
sueid	Semipermanent Universal Entity ID	RFC 9711

Table 3: UEID URN Registration

The ABNF [RFC5234] [RFC7405] for these two URNs is as follows, where b64ueid is the base64url-encoded binary byte string for the UEID or SUEID:

```
body =/ ueidbody
ueidbody = %s"ueid:" b64ueid
```

10.4. CBOR Tag for Detached EAT Bundle Registered by This Document

In the "CBOR Tags" registry [IANA.cbor-tags], IANA has allocated the following tag from the Specification Required range, with the present document as the reference.

Tag	Data Item	Semantics	Reference
602	array	Detached EAT Bundle	RFC 9711, Section 5

Table 4: Detached EAT Bundle Tag Registration

10.5. Intended Use Registry

IANA has created a new registry titled "Entity Attestation Token (EAT) Intended Uses" under the new "Remote Attestation Procedures (RATS)" registry group. The registry uses the Expert Review registration procedure [RFC8126].

Guidelines for designated experts:

- Each intended use should be clearly described so a user knows what it means.
- Each intended use should be distinct from others that are registered.
- Point squatting is discouraged.

The three columns for the registry are:

1. Value: This is a unique integer that is used to identify the intended use in CBOR-encoded tokens.
2. Description: This is one or more text paragraphs that sufficiently define what the intended use means. It may also be a reference to another document.
3. Reference: This field contains a reference to the defining specification.

The following 5 values represent the initial content of the registry. Note that 0 will be marked as "reserved" for the CBOR value, and the maximum CBOR value for assignment is 255.

1 -- Generic: Generic attestation describes an application where the EAT consumer requires 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 "intuse" 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 entity security state of the entity prior to provisioning.

4 -- Certificate Issuance: Certification Authorities (CAs) may require attestation results (which in a background check model might require receiving evidence to be passed to a verifier) to make decisions about the issuance of certificates. 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) application. More precisely, a PoP transaction is intended to provide the recipient with cryptographically verifiable proof that the sender has possession of a key. This kind of attestation may be necessary to verify the security state of the entity storing the private key used in a PoP application.

11. References

11.1. Normative References

[DLOA] GlobalPlatform, "GlobalPlatform Card: Digital Letter of Approval", Public Release Version 1.0, Document Reference: GPC_SPE_095, November 2015, <https://globalplatform.org/wp-content/uploads/2015/12/GPC_DigitalLetterOfApproval_v1.0.pdf>.

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

[IANA.COSE.Algorithms] IANA, "COSE Algorithms", <https://www.iana.org/assignments/cose>.

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

[IANA.DEV-URNs] IANA, "DEV URN Subtypes", <https://www.iana.org/assignments/device-identification>.

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

[PEN] IANA, "Private Enterprise Numbers (PENs)", <https://www.iana.org/assignments/enterprise-numbers/>.

[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>.

[RFC4517] Legg, S., Ed., "Lightweight Directory Access Protocol (LDAP): Syntaxes and Matching Rules", RFC 4517, DOI 10.17487/RFC4517, June 2006, <https://www.rfc-editor.org/info/rfc4517>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

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

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/rfc7405>.

[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>.

[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>.

[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>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[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>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

[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>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/info/rfc9052>.

[RFC9090] Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", RFC 9090, DOI 10.17487/RFC9090, July 2021, <https://www.rfc-editor.org/info/rfc9090>.

[RFC9165] Bormann, C., "Additional Control Operators for the Concise Data Definition Language (CDDL)", RFC 9165, DOI 10.17487/RFC9165, December 2021, <https://www.rfc-editor.org/info/rfc9165>.

[RFC9334] Birkholz, H., Thaler, D., Richardson, M., Smith, N., and W. Pan, "Remote ATtestation procedureS (RATS) Architecture", RFC 9334, DOI 10.17487/RFC9334, January 2023, <https://www.rfc-editor.org/info/rfc9334>.

[RFC9393] Birkholz, H., Fitzgerald-McKay, J., Schmidt, C., and D. Waltermire, "Concise Software Identification Tags", RFC 9393, DOI 10.17487/RFC9393, June 2023, <https://www.rfc-editor.org/info/rfc9393>.

[ThreeGPP.IMEI] 3GPP, "Numbering, addressing and identification", 3GPP TS 23.003, Version 19, September 2024, <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=729>.

[W3C.GeoLoc] Cáceres, M. and R. Grant, "Geolocation", W3C Recommendation, September 2024, <https://www.w3.org/TR/geolocation/>.

[WGS84] National Geospatial-Intelligence Agency (NGA), "Department of Defense World Geodetic System 1984: Its Definition and Relationships with Local Geodetic Systems", NGA.STND.0036_1.0.0_WGS84, July 2014, <https://nsgreg.nga.mil/doc/view?i=4085>.

11.2. Informative References

[BirthdayAttack] Wikipedia, "Birthday attack", October 2024, <https://en.wikipedia.org/w/index.php?title=Birthday_attack&oldid=1249270346>.

[CBOR.Certs] 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-13, 3 March 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-cose-cbor-encoded-cert-13>.

[CC-Example] Eurosmart, "Secure Sub-System in System-on-Chip (3S in SoC) Protection Profile", Version 1.8, October 2023, <https://commoncriteriaportal.org/nfs/ccpfiles/files/ppfiles/pp0117V2b_pdf.pdf>.

[EAT-GitHub] "Entity Attestation Token IETF Draft Standard", commit 62c726b, January 2024, <https://github.com/ietf-rats-wg/eat>.

[EAT.media-types] Lundblade, L., Birkholz, H., and T. Fossati, "EAT Media Types", Work in Progress, Internet-Draft, draft-ietf-rats-eat-media-type-12, 3 November 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-eat-media-type-12>.

[GP-Example] GlobalPlatform, "GlobalPlatform Technology: TEE Certification Process", Public Release Version 2.0, Document Reference: GP_PRO_023, January 2021, <https://globalplatform.org/wp-content/uploads/2021/01/GP_TEECertificationProcess_v2.0_PublicRelease.pdf>.

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

[IEEE.802-2014] IEEE, "IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture", IEEE Std 802-2014, DOI 10.1109/IEEESTD.2014.6847097, June 2014, <https://ieeexplore.ieee.org/document/6847097>.

[IEEE.802.1AR] IEEE, "IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity", IEEE Std 802.1AR-2018, DOI 10.1109/IEEESTD.2018.8423794, August 2018, <https://ieeexplore.ieee.org/document/8423794>.

[JTAG] IEEE, "IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture", IEEE Std 1149.7-2009, DOI 10.1109/IEEESTD.2010.5412866, February 2010, <https://ieeexplore.ieee.org/document/5412866>.

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

[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>.

[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>.

[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>.

[RFC9360] Schaad, J., "CBOR Object Signing and Encryption (COSE): Header Parameters for Carrying and Referencing X.509 Certificates", RFC 9360, DOI 10.17487/RFC9360, February 2023, <https://www.rfc-editor.org/info/rfc9360>.

[RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <https://www.rfc-editor.org/info/rfc9562>.

[UCCS] 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-12, 3 November 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-rats-uccs-12>.

Appendix A. Examples

Most examples are shown as a Claims-Set that would be a payload for a CWT, a JWT, a detached EAT bundle, or future token types. The signing is left off so the Claims-Set is easier to see. Some examples of signed tokens are also given.

A.1. Claims Set Examples

A.1.1. Simple TEE Attestation

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

```
/ This is an EAT payload that describes a simple TEE. /

{
    / eat_nonce /       10: h'48df7b172d70b5a18935d0460a73dd71',
    / oemboot /        262: true,
    / dbgstat /        263: 2, / disabled-since-boot /
    / manifests /      272: [
                              [
                               258, / CoAP Content ID for CoSWID    /

                               / This is a byte-string-wrapped      /
                               / payload CoSWID. It gives the TEE   /
                               / software name, the version, and    /
                               / the name of the file it is in.     /
                               / {0: "3a24",                        /
                               /  12: 1,                            /
                               /   1: "Acme TEE OS",                /
                               /  13: "3.1.4",                      /
                               /   2: [{31: "Acme TEE OS", 33: 1},  /
                               /       {31: "Acme TEE OS", 33: 2}], /
                               /   6: {                             /
                               /       17: {                        /
                               /           24: "acme_tee_3.exe"     /
                               /       }                            /
                               /    }                               /
                               /  }                                 /
                               h' a60064336132340c01016b
                                  41636d6520544545204f530d65332e31
                                  2e340282a2181f6b41636d6520544545
                                  204f53182101a2181f6b41636d652054
                                  4545204f5318210206a111a118186e61
                                  636d655f7465655f332e657865'
                              ]
                             ]
}
```

```
/ This is a payload CoSWID created by the software (SW) vendor. All /
/ this does is name the TEE SW, name its version, and list 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.1.2. Submodules for Board and Device

```
/ This example shows use of submodules to give information  /
/ about the chip, board, and overall device.                /
/                                                           /
/ The main attestation is associated with the chip          /
/ containing the CPU and running the main OS. It is what    /
/ has the keys and produces the token.                      /
/                                                           /
/ The board is made by a different vendor than the chip;    /
/ perhaps it is some generic IoT board.                     /
/                                                           /
/ The device is some specific appliance that is made by a   /
/ different vendor than either the chip or the board.       /
/                                                           /
/ Here, the board and device submodules aren't the typical  /
/ target environments as described by RATS Architecture     /
/ (RFC 9334), but they are a valid use of submodules.       /

{
    / eat_nonce /       10: h'e253cabedc9eec24ac4e25bcbeaf7765',
    / ueid /           256: h'0198f50a4ff6c05861c8860d13a638ea',
    / oemid /          258: h'894823', / IEEE OUI format OEM ID /
    / hwmodel /        259: h'549dcecc8b987c737b44e40f7c635ce8'
                             / Hash of chip model name /,
    / hwversion /      260: ["1.3.4", 1], / Multipartnumeric  /
    / swname /         270: "Acme OS",
    / swversion /      271: ["3.5.5", 1],
    / oemboot /        262: true,
    / dbgstat /        263: 3, / permanent-disable  /
    / timestamp (iat) /  6: 1526542894,
    / submods / 266: {
        / A submodule to hold some claims about the circuit board /
        "board" :  {
            / oemid /     258: h'9bef8787eba13e2c8f6e7cb4b1f4619a',
            / hwmodel /   259: h'ee80f5a66c1fb9742999a8fdab930893'
                                 / Hash of board module name /,
            / hwversion / 260: ["2.0a", 2] / multipartnumeric+sfx /
        },

        / A submodule to hold claims about the overall device /
        "device" :  {
            / oemid /     258: 61234, / PEN Format OEM ID /
            / hwversion / 260: ["4.0", 1] / Multipartnumeric /
        }
    }
}
```

A.1.3. EAT Produced by an Attestation Hardware Block

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

{
    / eat_nonce /       10: h'd79b964ddd5471c1393c8888',
    / ueid /           256: h'0198f50a4ff6c05861c8860d13a638ea',
    / oemid /          258: 64242, / Private Enterprise Number /
    / oemboot /        262: true,
    / dbgstat /        263: 3, / disabled-permanently /
    / hwversion /      260: [ "3.1", 1 ] / Type is multipartnumeric /
}
```

A.1.4. 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 (HLOS), for      /
/ example, a Trusted Execution Environment (TEE). The key      /
/ store is the attester.                                       /
/                                                              /
/ There is some attestation of the HLOS, just version and      /
/ boot and debug status. It is a Claims-Set submodule because  /
/ it has a 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 as this is a hypothetical example,      /
/ not part of a standard protocol.                             /


{
    / eat_nonce /       10: h'99b67438dba40743266f70bf75feb1026d5134
                             97a229bfe8',
    / oemboot /        262: true,
    / dbgstat /        263: 2, / disabled-since-boot /
    / manifests /      272: [
                              [ 258, / CoAP Content ID. /
                                h'a600683762623334383766
                                  0c000169436172626f6e6974650d6331
                                  2e320e0102a2181f75496e6475737472
                                  69616c204175746f6d6174696f6e1821
                                  02'
                               ]
                               / Above is an encoded CoSWID     /
                               / with the following data        /
```

```
                                /   SW Name: "Carbonite"         /
                                /   SW Vers: "1.2"               /
                                /   SW Creator:                  /
                                /      "Industrial Automation"   /
                             ],
    / exp /              4: 1634324274, / 2021-10-15T18:57:54Z /
    / iat /              6: 1634317080, / 2021-10-15T16:58:00Z /
                   -80000 : "fingerprint",
                   -80001 : { / The key -- A COSE_Key  /
                / kty /       1: 2, / EC2, elliptic 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 /        266 : {
                           "HLOS" : { / submod for high-level OS /
          / eat_nonce /         10: h'8b0b28782a23d3f6',
            / oemboot /        262: true,
            / manifests /      272: [
                                  [ 258, / CoAP Content ID. /
                                      h'a600687337
                                        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.1.5. Software Measurements of an IoT Device

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

```
/ This EAT payload 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.            /

{
    / eat_nonce / 10: h'5e19fba4483c7896',
    / oemboot /  262: true,
    / dbgstat /  263: 2, / disabled-since-boot /
    / oemid /    258: h'8945ad', / IEEE CID based /
    / ueid /     256: h'0198f50a4ff6c05861c8860d13a638ea',
    / submods /  266: {
                        "OS" : {
        / oemboot /         262: true,
        / dbgstat /         263: 2, / disabled-since-boot /
        / measurements /    273: [
                                   [
                                     258, / CoAP Content ID         /

                                     / This is a byte-string-wrapped /
                                     / evidence CoSWID. It has       /
                                     / hashes of the main files of   /
                                     / the IoT OS.  /
                                     h'a600663463613234350c
                                       17016d41636d6520522d496f542d4f
                                       530d65332e312e3402a2181f724163
                                       6d652042617365204174746573746 5
                                       7218210103a11183a318187161636d
                                       655f725f696f745f6f732e65786514
                                       1a0044b349078201582005f6b327c1
                                       73b4192bd2c3ec248a292215eab456
                                       611bf7a783e25c1782479905a31818
                                       6d7265736f75726365732e72736314
                                       1a000c38b10782015820c142b9aba4
                                       280c4bb8c75f716a43c99526694caa
                                       be529571f5569bb7dc542f98a31818
                                       6a636f6d6d6f6e2e6c6962141a0023
                                       3d3b0782015820a6a9dcdfb3884da5
                                       f884e4e1e8e8629958c2dbc7027414
                                       43a913e34de9333be6'
                                   ]
                               ]
                           }
                       }
}
```

```
/ This is an evidence CoSWID created for the "Acme R-IoT-OS"     /
/ that is created by the "Acme Base Attester" (both fictitious   /
/ 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.1.6. Attestation Results in JSON

This is a JSON-encoded payload 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 such as debug status directly through to the relying party. The verifier also knows the reference values for the measured software components and is able to check them. It informs the relying party that they were correct in the "measres" claim. "Trustus Verifications" is the name of the service that verifies the software component measurements.

```
{
   "eat_nonce": "jkd8KL-8xQk",
   "oemboot": true,
   "dbgstat": "disabled-since-boot",
   "oemid": "iUWt",
   "ueid": "AZj1Ck_2wFhhyIYNE6Y4",
   "swname": "Acme R-IoT-OS",
   "swversion": [
      "3.1.4"
   ],
   "measres": [
      [
         "Trustus Measurements",
         [
            [
               "all",
               "success"
            ]
         ]
      ]
   ]
}
```

A.1.7. JSON-Encoded Token with Submodules

The lines in this example are wrapped per [RFC8792].

```
=============== NOTE: '\' line wrapping per RFC 8792 ================

{
   "eat_nonce": "lI-IYNE6Rj6O",
   "ueid": "AJj1Ck_2wFhhyIYNE6Y46g==",
   "secboot": true,
   "dbgstat": "disabled-permanently",
   "iat": 1526542894,
   "submods": {
      "Android App Foo": {
         "swname": "Foo.app"
      },
      "Secure Element Eat": [
         "CBOR",
         "2D3ShEOhASagWGaoCkiUj4hg0TpGPhkBAFABmPUKT_bAWGHIhg0TpjjqGQ\
ECGfryGQEFBBkBBvUZAQcDGQEEgmMzLjEBGQEKoWNURUWCL1gg5c-V_ST6txRGdC3VjU\
Pa4XjlX-K5QpGpKRCC_8JjWgtYQPaQywOIZ3-mJKN3X9fLxOhAnsmBa-MvpHRzOw-Ywn\
-67bvJljuctezAPD41s6_At7NbSV3qwJlxIuqGfwe41es="
      ],
      "Linux Android": {
         "swname": "Android"
      },
      "Subsystem J": [
         "JWT",
         "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJKLUF0dGVzd\
GVyIiwiaWF0IjoxNjUxNzc0ODY4LCJleHAiOm51bGwsImF1ZCI6IiIsInN1YiI6IiJ9.\
gjw4nFMhLpJUuPXvMPzK1GMjhyJq2vWXg1416XKszwQ"
      ]
   }
}
```

A.2. Signed Token Examples

A.2.1. Basic CWT Example

This is a simple CWT-format token signed with the Elliptic Curve Digital Signature Algorithm (ECDSA).

```
/ This is a full CWT-format token. The payload is the /
/ attestation hardware block in Appendix A.1.3 of    /
/ RFC 9711. The main structure that is visible is   /
/ that of the COSE_Sign1.                           /

61( 18( [
    h'A10126',                          / protected headers  /
    {},                              / empty unprotected headers /
    h'A60A4CD79B964DDD5471C1393C88881901005001
      98F50A4FF6C05861C8860D13A638EA19010219FA
      F2190106F5190107031901048263332E3101',        / payload /
    h'9B9B2F5E470000F6A20C8A4157B5763FC45BE759
      9A5334028517768C21AFFB845A56AB557E0C8973
      A07417391243A79C478562D285612E292C622162
      AB233787'                                   / signature /
] ) )
```

A.2.2. CBOR-Encoded Detached EAT Bundle

In this detached EAT bundle, the main token is produced by a hardware (HW) attestation block. The detached Claims-Set is produced by a TEE and is largely identical to the simple TEE example in Appendix A.1.1. 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 detached EAT bundle itself can be assembled by untrusted software.

```
/ This is a detached EAT bundle tag. /

602([

    / The 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 part of the detached EAT bundle.   /
    / The COSE payload is as follows:                           /
    / { /
    /      10: h'948F8860D13A463E', /
    /     256: h'0198F50A4FF6C05861C8860D13A638EA', /
    /     258: 64242, /
    /     262: true, /
    /     263: 3, /
    /     260: ["3.1", 1], /
    /     266: { /
    /         "TEE": [ /
    /             -16, /
    /              h'ab86f765643aabfd09c84eebe150b7f6  /
    /                1bc24804cee75e90c5f99cb850fe808f' /
    /         ] /
    /     } /
    /   }  /
    h'D83DD28443A10126A05866A80A48948F8860D13A463E1901
      00500198F50A4FF6C05861C8860D13A638EA19010219FAF2
      19010504190106F5190107031901048263332E310119010A
      A163544545822F58208DEF652F47000710D9F466A4C666E2
      09DD74F927A1CEA352B03143E188838ABE5840F690CB0388
      677FA624A3775FD7CBC4E8409EC9816BE32FA474733B0F98
      C27FBAEDBBC9963B9CB5ECC03C3E35B3AFC0B7B35B495DEA
      C0997122EA867F07B8D5EB',
    {
       / A CBOR-encoded byte-string-wrapped EAT claims-set. /
       / It contains claims for a simple TEE attestation.   /
       "TEE" : h'a40a5048df7b172d70b5a18935d0460a73dd7119
                 0106f5190107021901108182190102 5858a60064
                 336132340c01016b41636d6520544545204f530d
                 65332e312e340282a2181f6b41636d6520544545
                 204f53182101a2181f6b41636d6520544545204f
                 5318210206a111a118186e61636d655f7465655f
                 332e657865'
    }
 ])
```

```
/ This example contains a submodule that is a detached digest, /
/ which is the hash of a Claims-Set conveyed outside this      /
/ token. It is also an example of a token from an attestation  /
/ hardware block.                                              /

{
    / eat_nonce /       10: h'3515744961254b41a6cf9c02',
    / ueid /           256: h'0198f50a4ff6c05861c8860d13a638ea',
    / oemid /          258: 64242, / Private Enterprise Number /
    / oemboot /        262: true,
    / dbgstat /        263: 3, / disabled-permanently /
    / hwversion /      260: [ "3.1", 1 ], / multipartnumeric /
    / submods/         266: {
                                "TEE": [ / detached digest submod /
                                           -16, / SHA-256 /
                                           h'ab86f765643aabfd09c8
                                             4eebe150b7f61bc24804
                                             cee75e90c5f99cb850fe
                                             808f'
                                       ]
                           }
}
```

A.2.3. JSON-Encoded Detached EAT Bundle

In this bundle, there are two detached Claims-Sets: "Audio Subsystem" and "Graphics Subsystem". The JWT at the start of the bundle has detached signature submodules with hashes that cover these two Claims-Sets. The JWT itself is protected using the Hashed Message Authentication Code (HMAC) with a key of "xxxxxx".

The lines in this example are wrapped per [RFC8792].

```
=============== NOTE: '\' line wrapping per RFC 8792 ================

[
    [
        "JWT",
        "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlYXRfbm9uY2UiOiJ5dT\
c2Tk44SXVWNmUiLCJzdWJtb2RzIjp7IkF1ZGlvIFN1YnN5c3RlbSI6WyJESUdFU1QiLF\
siU0hBLTI1NiIsIkZSRW4yVlR3aTk5cWNNRVFzYmxtTVFnM2I1b2ZYUG5OM1BJYW5CME\
5RT3MiXV0sIkdyYXBoaWNzIFN1YnN5c3RlbSI6WyJESUdFU1QiLFsiU0hBLTI1NiIsIk\
52M3NqUVU3Q1Z0RFRka0RTUlhWcFZDNUNMVFBCWmVQWWhTLUhoVlZWMXMiXV19fQ.FYs\
7R-TKhgAk85NyCOPQlbtGGjFM_3chnhBEOuM6qCo"
    ],
    {
        "Audio Subsystem" : "ewogICAgImVhdF9ub25jZSI6ICJsSStJWU5FNlJ\
qNk8iLAogICAgInVlaWQiOiAiQWROSlU0b1lYdFVwQStIeDNqQTcvRFEiCiAgICAib2V\
taWQiOiAiaVVXdCIsCiAgICAib2VtYm9vdCI6IHRydWUsIAogICAgInN3bmFtZSI6ICJ\
BdWRpbyBQcm9jZXNzb3IgT1MiCn0K",
        "Graphics Subsystem" : "ewogICAgImVhdF9ub25jZSI6ICJZWStJWU5F\
NlJqNk8iLAogICAgInVlaWQiOiAiQWVUTUlRQ1NVMnhWQmtVdGlndHU3bGUiCiAgICAi\
b2VtaWQiOiA3NTAwMCwKICAgICJvZW1ib290IjogdHJ1ZSwgCiAgICAic3duYW1lIjog\
IkdyYXBoaWNzIE9TIgp9Cg"
    }
]
```

Appendix B. UEID Design Rationale

B.1. Collision Probability

This calculation is to determine the probability of a collision of type 0x01 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 database is highly speculative perhaps involving nanorobots for every person, livestock animals, and domesticated birds. It is included to round out the analysis.

Note that the items counted here certainly do not have IP addresses and are not individually connected to the network. They may be connected to internal buses, via serial links, via 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: Entity Database Size Examples

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)}
```

For this calculation:

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 software, 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})	$2 * 10^{-09}$	$8 * 10^{-29}$	$5 * 10^{-49}$

Database Size	128-bit UEID	192-bit UEID	256-bit UEID
100 quadrillion (10^{17})	$2 * 10^{-05}$	$8 * 10^{-25}$	$5 * 10^{-45}$

Table 6: UEID Size Options

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 scenario 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 table gives the time interval until there is a probability of a collision, which is based on there being one tenth of the number of states per year as the number of records in the database.

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

For this calculation:

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 7: UEID Collision Probability

Clearly, 128 bits is enough for the near future, thus the requirement that type 0x01 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 scenario. 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 Universally Unique Identifier (UUID) [RFC9562] 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 computing platforms. In particular, hardware randomness sources were introduced in CPUs between 2010 and 2015. Operating systems and cryptographic libraries make use of this hardware. Consequently, there is little need for protocols to construct random numbers from multiple sources on their own.

Version 4 UUIDs do allow for the use of such cryptographic-quality random numbers, but they 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.

The design of UUID accommodates the construction of a unique identifier by the combination of several identifiers that separately do not provide sufficient uniqueness. The design philosophy underlying UEID assumes that this construction is no longer needed, in particular because cryptographic-quality random number generators are readily available. Therefore, hardware, software, and/or manufacturing processes can implement UEID in a simple and direct way.

Note also that a type 2 UEID (EUI/MAC) is only 7 bytes whereas a UUID is 16.

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

This section describes several distinct ways in which an IEEE Initial Device Identifier (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 here. The intent is that IDevIDs and LDevIDs can be used with any network protocol or message format. In these protocols and message formats, the DevID secret is used to sign a nonce or similar to prove the association of the DevID certificates with the device.

By contrast, EAT standardizes a message format that is sent to a relying party, the very thing that is not defined in [IEEE.802.1AR]. Nor does EAT 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, it provides 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 described above, EAT standardizes a message format, but [IEEE.802.1AR] does not. Vice versa, EAT does not define a device implementation, but DevID does.

Hence, EAT can be the message format that a DevID is used with. The DevID secret becomes the attestation key used to sign EATs, and 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. For this use, it is a good idea for the serial numbers to be the same 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 such as OEM ID are equivalent to the secure device identity that is 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 such as GPS location.

DevID secures the device identity fields by embedding them in a certificate and signing it. The certificate is created once during manufacturing and remains unchanged.

So in one case, the signing of the identity happens on the device, and in the other case, it happens in a manufacturing facility. However, 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 an X.509 certificate.

If that X.509 certificate is an IDevID or LDevID, it 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 relying 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. For EAT, permanence is not defined in terms of resistance to attacks. Instead, it is defined in the context of operational functionality and the device life cycle.

Appendix D. CDDL for CWT and JWT

[RFC8392] was published before CDDL was available and thus is specified in prose, not CDDL. In the following example, CDDL specifies CWT as it is needed to complete this specification. This CDDL also covers the Claims-Set for JWT.

Note that Section 4.3.1 requires that the "iat" claim be the type ~time-int (Section 7.2.1), not the type ~time when it is used in an EAT as floating-point values are not allowed for the "iat" claim in EAT.

The COSE-related types in this CDDL are defined in [RFC9052].

This, however, is NOT a normative or standard definition of CWT or JWT in CDDL. The prose in CWT and JWT remains the normative definition. See also [UCCS].

```
Claims-Set = {
    * $$Claims-Set-Claims
    * Claim-Label .feature "extended-claims-label" => any
}
Claim-Label = int / text
string-or-uri = text

$$Claims-Set-Claims //= ( iss-claim-label => string-or-uri  )
$$Claims-Set-Claims //= ( sub-claim-label => string-or-uri  )
$$Claims-Set-Claims //= ( aud-claim-label => string-or-uri  )
$$Claims-Set-Claims //= ( exp-claim-label => ~time )
$$Claims-Set-Claims //= ( nbf-claim-label => ~time )
$$Claims-Set-Claims //= ( iat-claim-label => ~time )
$$Claims-Set-Claims //= ( cti-claim-label => bytes )

iss-claim-label = JC<"iss", 1>
sub-claim-label = JC<"sub", 2>
aud-claim-label = JC<"aud", 3>
exp-claim-label = JC<"exp", 4>
nbf-claim-label = JC<"nbf", 5>
iat-claim-label = JC<"iat", 6>
cti-claim-label = CBOR-ONLY<7>  ; jti in JWT: different name and text

JSON-ONLY<J> = J .feature "json"
CBOR-ONLY<C> = C .feature "cbor"

JC<J,C> = JSON-ONLY<J> / CBOR-ONLY<C>
```

```
; A JWT message is either a JSON Web Signature (JWS) or a JSON Web
; Encryption (JWE) in compact serialization form with the payload
; as a Claims-Set. Compact serialization is the protected headers,
; payload, and signature that are each b64url-encoded and separated
; by a ".". This CDDL simply matches the top-level syntax of a JWS
; or JWE as 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_-]+"


; Note that the payload of a JWT is defined in the CDDL description
; of claims-set. That definition is common to CBOR and JSON.
```

```
; This is some CDDL describing a CWT at the top level. This is
; not normative. RFC 8392 is the normative definition of CWT.

CWT-Messages = CWT-Tagged-Message / CWT-Untagged-Message

; The payload of the COSE_Message is always a Claims-Set.

; The contents of a CWT tag must always be a COSE tag.
CWT-Tagged-Message = #6.61(COSE_Tagged_Message)

; An untagged CWT may be a COSE tag or not.
CWT-Untagged-Message = COSE_Messages
```

Appendix E. New Claim Design Considerations

The following are design considerations that may be helpful to take into account when creating new EAT claims. This is the product of discussion in the RATS Working Group.

EAT reuses the CWT and JWT claims registries. There is no registry exclusively for EAT claims. This is not an update to the expert review criteria for the JWT and CWT claims registries as that would be an overreach for this document.

E.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 as it requires 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.

E.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 as well as 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 software 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.

E.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 cases. For example, an app on a mobile device may generate EATs on its own.

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

E.4. Reuse of Extant Data Formats

Where possible, claims should use data items, identifiers, and formats that are already standardized. 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 encoding 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 Certificate Revocation Lists (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.

E.5. Proprietary Claims

It is not always possible or convenient to achieve the above goals, so the definition and use of proprietary claims is an option.

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.

Appendix F. 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 such as Direct Anonymous Attestation (DAA).

RATS Architecture [RFC9334] 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, and it may contain implied claims, i.e., 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 such as 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, etc.) of a particular end-to-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.

See also the security considerations in Section 9.6.

F.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.

F.1.1. COSE/JWS Key ID

The COSE standard header parameter for Key ID (kid) may be used; see [RFC9052] 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 Key Derivation Function (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.

F.1.2. JWS and COSE X.509 Header Parameters

COSE X.509 [RFC9360] and JSON Web Signature [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, and it is up to the attestation system design and the verifier to determine which.

F.1.3. CBOR Certificate COSE Header Parameters

Compressed X.509 and CBOR Native certificates are defined by CBOR Certificates [CBOR.Certs]. 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, etc.).

F.1.4. Claim-Based Key Identification

For some attestation systems, a claim may be reused as a key identifier. For example, the UEID uniquely identifies the entity and therefore can work well as a key identifier or endorsement identifier.

An advantage of this is that key identification requires no additional bytes in the EAT and makes the EAT smaller.

A disadvantage of this is 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.

Contributors

Many thanks to the following for their contributions to earlier draft versions of this document:

Henk Birkholz
Fraunhofer SIT
Email: henk.birkholz@sit.fraunhofer.de

Thomas Fossati
Arm Limited
Email: thomas.fossati@arm.com

Miguel Ballesteros

Michael Richardson
Sandelman Software Works
Email: mcr+ietf@sandelman.ca

Patrick Uiterwijk

Mathias Brossard

Hannes Tschofenig
Arm Limited
Email: hannes.tschofenig@arm.com

Paul Crowley

Authors' Addresses

Laurence Lundblade
Security Theory LLC
Email: lgl@securitytheory.com

Giridhar Mandyam
Email: giridhar.mandyam@gmail.com

Jeremy O'Donoghue
Qualcomm Technologies Inc.
279 Farnborough Road
Farnborough
GU14 7LS
United Kingdom
Phone: +44 1252 363189
Email: jodonogh@qti.qualcomm.com

Carl Wallace
Red Hound Software, Inc.
Email: carl@redhoundsoftware.com