Workgroup: Interpeer Project
Published: 14 July 2023
Author: J. Finkhaeuser
Interpeer

CAProck Compact Wire Encoding

Abstract

[CAPROCK] is a distributed authorization scheme based on cryptographic capabilities ([I-D.draft-jfinkhaeuser-caps-for-distributed-auth]). This document describes a compact wire encoding for CAProck capabilities, suitable for 0-RTT transmission.

The RFC Editor will remove this note

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://specs.interpeer.org/draft-jfinkhaeuser-caprock-enc-compact/.

Discussion of this document takes place on the Interpeer mailing list interpeer@lists.interpeer.io, which is archived at https://lists.interpeer.io/pipermail/interpeer/. Subscribe at https://lists.interpeer.io/mailman/listinfo/interpeer.

Source for this draft and an issue tracker can be found at https://codeberg.org/interpeer/specs.

Status of This Memo

Drafts are working documents of the Interpeer Project. The list of current Drafts is at https://specs.interpeer.org/. Drafts may be updated, replaced, or obsoleted by other documents at any time. It is inadvisable to use Drafts as reference material or to cite them other than as "work in progress."

Copyright Notice

Copyright (c) Interpeer gUG and the persons identified as the document authors. This document is licensed under CC BY-SA 4.0.

Table of Contents

1. Introduction 3
2. Conventions and Definitions 3
3. CAProck Compact Wire Encoding 3
  3.1. Self-Describing Binary Format 4
    3.1.1. Alternative Encoding Approaches 4
    3.1.2. Version Compatibility 4
  3.2. Token Layout 5
  3.3. Token Fields 7
    3.3.1. Token Header 8
    3.3.2. Token Type 9
    3.3.3. Identifiers 9
    3.3.4. Issuer Identifier 11
    3.3.5. Sequence Number 11
    3.3.6. Scope 11
    3.3.7. Claims 13
    3.3.8. Signature 14
    3.3.9. Tag Values 14
4. Referenced Encodings 16
  4.1. Variable-Length Integer fields (ULEB128) 16
  4.2. TAI64 Labels 16
5. Related Considerations 16
  5.1. IANA Considerations 17
6. References 17
  6.1. Normative References 17
  6.2. Informative References 17
Appendix A. Scheme for Tag Values 18
Acknowledgments 22

Author's Address 22

1. Introduction

This document describes a compact, self-describing wire format for [CAPROCK] tokens. In this context, compact means "small enough", specifically to fit into a 0-RTT handshake.

0-RTT handshakes aim to cut down on initial overhead. But in datagram oriented protocols, handshakes in which each peer may send more than a single datagram suffer from additional issues: what if a single of these datagrams does not arrive?

Mitigation techniques against this are widely employed and resemble stream oriented protocols, but all serve to increase the protocol complexity, and thus risk inviting more issues.

This encoding is chosen to be small enough to instead ensure that a capability token can be transmitted in a 0-RTT message.

2. Conventions and Definitions

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 order to respect inclusive language guidelines from [NIST.IR.8366] and [I-D.draft-knodel-terminology-10], this document uses plural pronouns.

3. CAProck Compact Wire Encoding

While the use-case of this encoding is for 0-RTT transmission, it may be re-used in any scenario with similar size constraints.

Simply stating the encoding should result in a size "small enough for 0-RTT does not define this target size very well. Consider also that IP fragmentation stall handshakes if individual fragments take longer to deliver than others.

The 0-RTT goal, then, should be understood as making an encoded capability token small enough to be transmitted in a single packet fitting into a single data link frame along the entire path from source to destination.

Being a network of networks, it is impossible to guarantee this for all data link types. For example, using [LoRaWAN], a payload may need to be as small as 51 octets, which is unlikely to contain multiple digests and a cryptographic signature without aggressive truncation. Here, of course, using approaches like [SCHC] may help (see e.g. [SCHC-LORA]).

SCHC then provides for a reasonable bound: an SCHC window is 630 octets, while most home internet equipment such as based on wired or wireless IEEE 802 standards typically provides for larger transmissions.

For the purposes of this document, "small enough" then means fitting comfortably (i.e. with room to spare) into a SCHC window.

3.1. Self-Describing Binary Format

To fulfil the space requirements above, we choose a self-describing binary format. Self-describing formats balance space savings with having an upgrade path.

Each field in the format we choose is prefixed with a field type; for fixed sized fields, this immediately tells a parser where to expect the next value. Variable sized fields require an additional length prefix.

In small encodings such as the one in this document, the field type can usually fit into a single octet, wasting very little space over the raw field data. At the same time, adding new field types in updated specifications is a relatively painless endeavour.

3.1.1. Alternative Encoding Approaches

Alternative encodings abound, but a quick comparison against the common [ASN.1] and [JSON] encodings is warranted as stand-ins for similar classes of encodings.

ASN.1 is a telecommunications standard for wire encoding of arbitrary data. It works by prefixing data with a type identifier, that provides sufficient information to a parser to extract the encapsulated data and advance to the next field. However, ASN.1 does not encode any field semantics. Therefore, the order of fields in an ASN.1 encoding is of paramount importance, as semantics can only be represented by a specific field order.

To work around this, ASN.1 often encodes tuples of data, where the first entry specifies an application defined type for the second. In this manner, flexibility is restored. However, to do this, ASN.1 needs to encode the fact that a tuple is present, an ASN.1 data type for each tuple field, etc.

By contrast, JSON essentially always encodes semantics. In a JSON object, both field names and field values are encoded. JSON is intended to be very generic, so this approach makes sense. As names are strings, however, they consume a lot of unnecessary space when the number of possible fields is strictly limited, as in the case of encoding CAProck tokens.

A self-describing format therefore combines benefits from both approaches to encoding: as semantics typically imply a data type, the most compact encoding possible is achieved, while retaining some flexibility for extensions.

3.1.2. Version Compatibility

A downside of using a self-describing binary format is that while the format can *encode* extenions quite well, *decoding* them requires knowlege of which specific extensions might be used.

Versioning field formats helps retain backward compatibility in these scenarios. However, forward compatibility is a goal that is sacrificed in the pursuit of compactness, as it can only be maintained by taking an approach closer to that of ASN.1. For encodings that retain forward compatibility, consider approaches such as taken by ASN.1 or JSON.

3.2. Token Layout

In order to achieve optimal compactness, the encoding in this document makes the simplifying assumption that a specific version of a specific field type has a fixed layout. Therefore, tags in the encoding often indicate both the semantics as well as the data type of the field following it. This rule is not strictly adhered to, however, such as when there is a choice of different data types for the same field.

Field tags are often variable length integer fields (see Section 4.1), but in some cases, single octets are used instead when a large amount of variation is not expected.

Because some fields have variable sizes, the layout below is approximate and mostly describes the relative position of fields to each other.

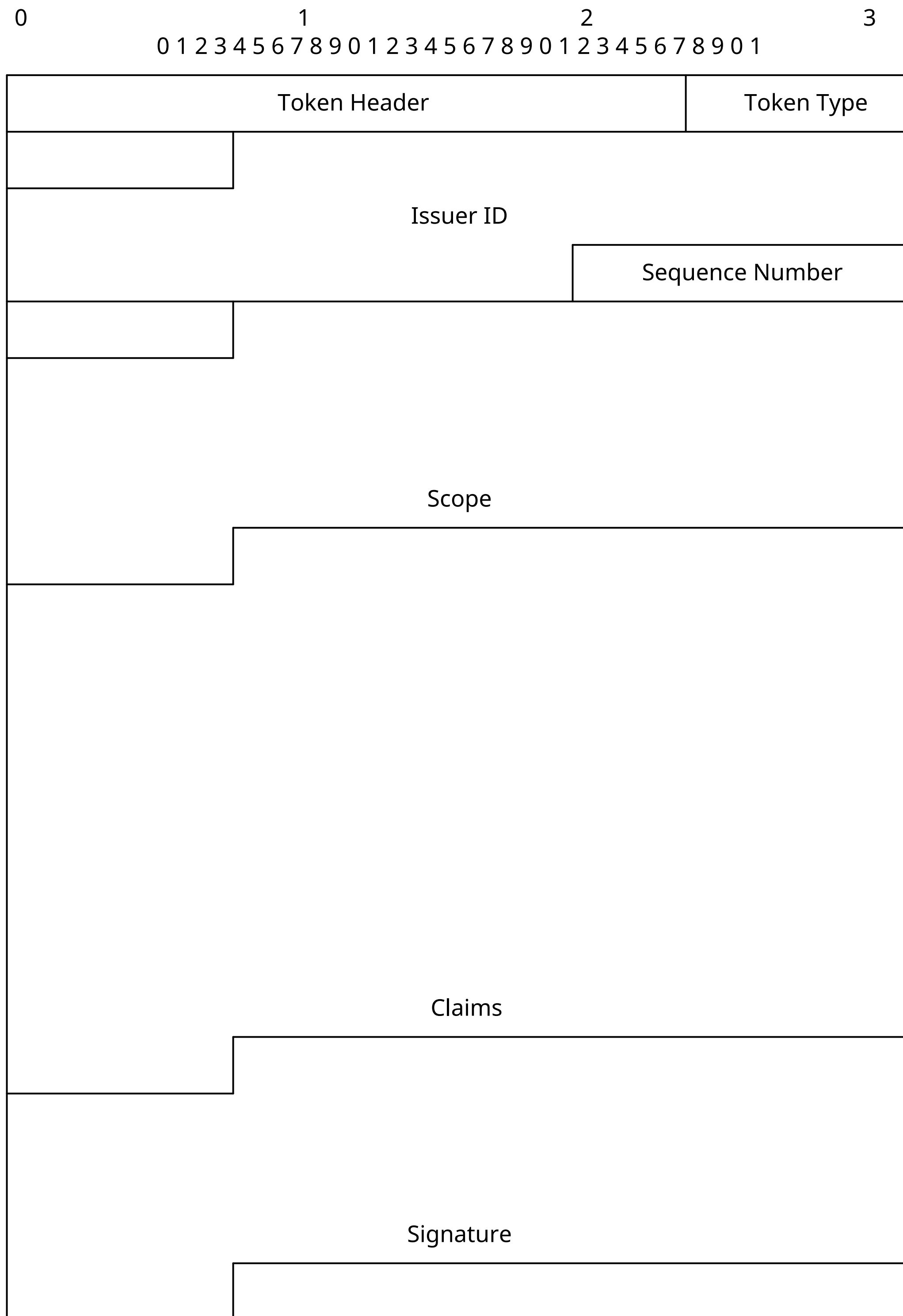

Figure 1: Token Memory Layout

Token Header: The token header indicates the that the following octets are a CAProck token, and specifies the remainder of the memory layout. This document only provides a version 1 layout. See Section 3.3.1.

Token Type: This field specifies the type of token, i.e. whether the token grants or revokes privileges specified in the claims. See Section 3.3.2.

Issuer ID: In the version 1 layout, the field that follows identifiers the token issuer. See Section 3.3.3.

Sequence Number: The sequence number for ordering tokens follows. See Section 3.3.5.

Scope: Following the above, the scope specifies the time period for which the token is valid, as well as related flags. See Section 3.3.6.

Claims: The claims encoded in this token follow. See Section 3.3.7.

Signature: The token ends with a cryptographic signature over the entire preceding contents, created with a private key associated with the issuer ID. See Section 3.3.8.

The following section provides details on each field.

Note that of the above fields starts with a field tag. That means that, in principle, fields can be re-ordered within the token. However, the token header **MUST** always be serialized first. Similarly, the signature **MUST** always be the last field.

Implementations **SHOULD** always produce the above order, while parsers **MAY** accept other field orders.

3.3. Token Fields

Token fields are prefixed by a tag that specifies the field type. The tag is encoded as a ULEB128 variable sized unsigned integer. For details on the encoding, see Section 4.1. All types in this document fit into 7 bits, so occupy a single octet in the encoding.

Implementations **SHOULD** implement ULEB128 already. If they do not, they **MUST** only use the 7 least significant bits of the field tag octet, and produce errors if the most significant bit is set.

If the field type is for a fixed sized field, the size is implied and can be listed in this document. The layout of a fixed sized field is as follows. Of course, the actual data size here is an example only.

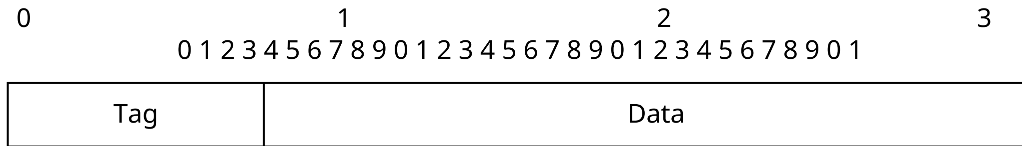

Figure 2: Fixed Sized Field Layout

Variable sized fields have a ULEB128 encoded size preceeding the data (see Section 4.1). This permits a value of size of less than 127 to be encoded in a single octet, and should represent the most common case. Nonetheless, larger sizes are permitted. Implementations **MUST** ensure to never except values larger than `2 ^ 1 6`, however, as that is the maximum token size.

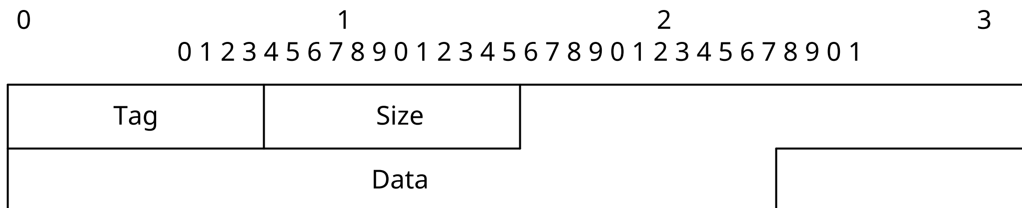

Figure 3: Variable Sized Field Layout

3.3.1. Token Header

The token header field is the token tag, followed by a token size. The size is a two octet integer in network byte order (big endian). It specifies the size of the token, in octets, from the beginning of the token header until the end of the signature.

The token tag and size together should give readers an understanding whether the token can be processed or must be skipped, and if skipped, by how much to skip. Subsequent revisions of this document **SHOULD** not modify the token header, except to update the token tag value.

The token tag value also determines which fields to expect, as listed in Section 3.2.

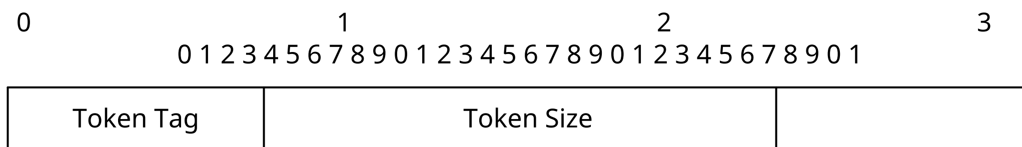

Figure 4: Token Header Layout

The tag name for this field is `TAG_TOKEN`; see Section 3.3.9 for values.

> The initial token tag and size are probably not sufficient for scanning a data stream for token boundaries. But given the size, at minimum a token can be processed or skipped as a whole. This is also why this size is not in ULEB128 encoding, like most others in this document. Readers should be able to get an indication of a token's size with minimal effort.

3.3.2. Token Type

The token type field specifies whether the token grants or revokes privileges listed in the claims. Following the tag is a single octet for the type value.

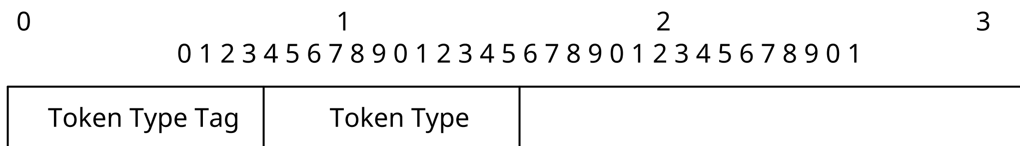

Figure 5: Token Type Layout

The tag name for this field is `TAG_TOKEN_TYPE`; see Section 3.3.9 for values.

The "grant" and "revoke" types as specified in [CAPROCK] are mapped into numeric values as below.

Type Name	Decimal	Hexadecimal
Grant	0	0x00
Revoke	1	0x01

Table 1: Token Type Values

3.3.3. Identifiers

[CAPROCK] defines identifiers for the issuer and subject as derived from public keys. The identifier scheme results in identifier sizes ranging from 28 to 64 octets (224 to 512 bits), but only a limited number of sizes are feasible. The document further defines that object identifiers should have the same size, though the semantics are application defined.

Technically, this makes identifiers variable sized fields, but the number of possible sizes is very limited. Encoding the sub-type of identifier that also specifies the size is useful after parsing, and takes the same amount of space or less than encoding the actual identifier size.

In addition to this sub-type, identifiers have a purpose, i.e. they identify issuer, subject or object. This purpose must be encoded as well. We therefore distinguish between the identifier tag, which encodes the purpose, and the identifier type tag, which encodes the format.

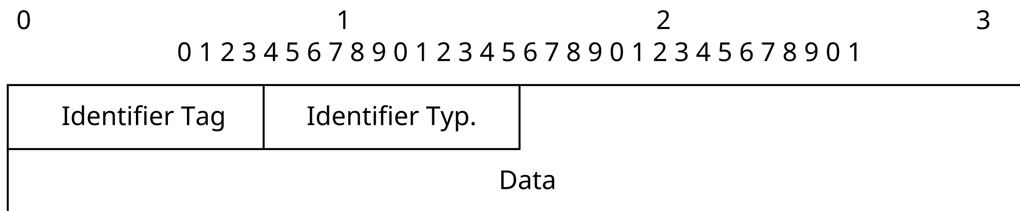

Figure 6: Identifier Layout

Due to the size limits and digest types defined in [CAPROCK], as well as the wildcard identifier and the fact that object identifiers are optional, there are a number of tag names that can apply to identifier fields. The full list and their values can be found in Section 3.3.9; below are the token names only:

Identifier Type Tag Name
TAG_ID_NONE
TAG_ID_WILDCARD
TAG_ID_RAW_32
TAG_ID_RAW_57
TAG_ID_SHA3_28
TAG_ID_SHA3_32
TAG_ID_SHA3_48
TAG_ID_SHA3_64

Table 2: Identifier Type Tag Names

The first two tag names are special. TAG_ID_NONE describes that the identifier is absent. This is only valid for object identifiers.

By contrast, TAG_ID_WILDCARD specifies that the identifier is a wildcard identifier as defined in [CAPROCK]. This value is not valid for issuer identifiers. For both types of identifiers, the data size is zero octets - no data may follow.

The other tags describe the size of the identifier data.

By contrast, the identifier tag may be one of the following names:

Identifier Tag Name	Details
TAG_ISSUER_ID	Section 3.3.4
TAG_CLAIM_SUBJECT	Section 3.3.7.1
TAG_CLAIM_OBJECT	Section 3.3.7.3

Table 3

3.3.4. Issuer Identifier

The issuer is an identifier field as described in the previous Section 3.3.3. Issuer identifers may neither be TAG_ID_NONE nor TAG_ID_WILDCARD.

The tag name for this field is TAG_ISSUER_ID; see Section 3.3.9 for values.

3.3.5. Sequence Number

Similar to identifiers, the sequence number is technically a variable sized field, but for reasons of compactness has its own encoding. It is a tag, followed by a ULEB128 encoded variable sized integer (see Section 4.1).

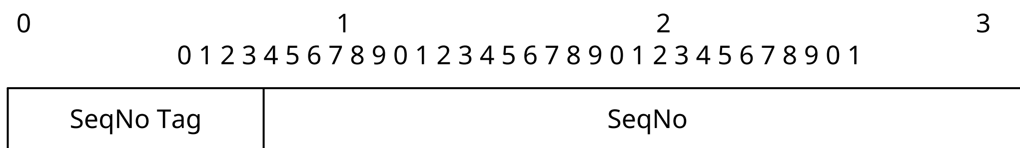

Figure 7: Identifier Layout

The tag name for this field is TAG_SEQUENCE_NO; see Section 3.3.9 for values.

3.3.6. Scope

The scope is a compound field, made up of subfields. These fields describe the circumstances under which to apply the enclosed claims (Section 3.3.7). As far as [CAPROCK] is concerned, this is a (potentially open-ended) time span, as well as a policy flag to determine how to handle tokens outside of that time span.

The scope is prefixed by its own tag, after which follow the individual scope related fields.

The tag name for this field is TAG_SCOPE; see Section 3.3.9 for values.

3.3.6.1. Scope From

The scope's from field consists of its tag, followed by a TAI64 label (Section 4.2):

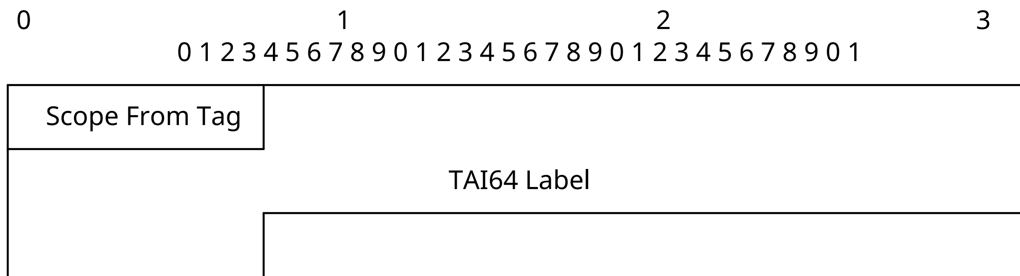

Figure 8: Scope From Layout

The field must always contain a timestamp, as per the [CAPROCK] specification. Validators **MUST** reject values out of range for TAI64 as invalid.

The tag name for this field is `TAG_SCOPE_FROM`; see Section 3.3.9 for values.

3.3.6.2. Scope To

The scope's to field is nearly identical to the scope's from field, except that

1. It has a different tag value, and
2. it may be empty (see Section 4.2 for how that is encoded).

The tag name for this field is `TAG_SCOPE_TO`; see Section 3.3.9 for values.

3.3.6.3. Scope Expiry Policy

The scope expiry policy in [CAPROCK] specifies an `issuer` and a `local` policy, which is a number of choices that fit comfortable into a single octet.

The scope expiry policy field therefore consists of a single octet field tag, and a single octed value.

The tag name for this field is `TAG_SCOPE_EXPIRY_POLICY`; see Section 3.3.9 for values.

The policy values may be extended in future documents. For now, only the two mentioned previously are supported. Implementations **SHOULD** produce warnings when encountering unsupported policies, and **MUST** treat tokens with such policies as invalid.

Policy Name	Decimal	Hexadecimal
Issuer	0	0x00
Local	1	0x01

Table 4: Scope Expiry Policy Values

3.3.7. Claims

Much like the scope field, the claims field is a compound field. It contains a list of claims, each of which contains three fields. Following the tag, a ULEB128 encoded size is encoded (see Section 4.1), providing the number of claims that follow.

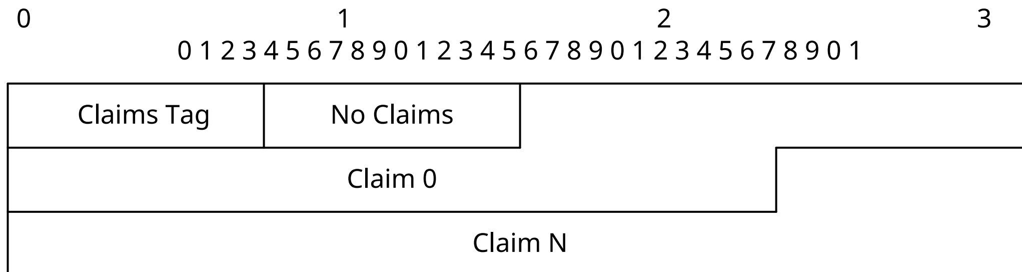

Figure 9: Claims Layout

The tag name for this field is TAG_CLAIMS; see Section 3.3.9 for values.

Note that the claim sizes above are for illustration purposes only. Each claim must contain exactly one subject, one predicate and one object field. There is no additional tag preceeding each claim; the parser has enough information with the number of claims and the required claim fields to parse the claims field.

3.3.7.1. Claim Subject

The subject is an identifier field as described in Section 3.3.3. Subject identifiers may not be TAG_ID_NONE, and implementations **MUST** reject such claims.

The tag name for this field is TAG_CLAIM_SUBJECT; see Section 3.3.9 for values.

3.3.7.2. Claim Predicate

The predicate is a variable sized field, the contents of which are not further specified here. An appropriate privilege scheme may encode any values here for application defined privileges.

The tag name for this field is TAG_CLAIM_PREDICATE; see Section 3.3.9 for values.

3.3.7.3. Claim Object

Identifiers for objects also follow the identifier scheme from Section 3.3.3. Note that object identifiers may contain any value, so long as they respect the identifier size bounds.

The tag name for this field is TAG_CLAIM_OBJECT; see Section 3.3.9 for values.

3.3.8. Signature

Finally, the signature field is a cryptographic signature over the preceeding token fields, from the first octet of the token header to the last octet before the signature field tag.

Note that [CAPROCK] specifies which cryptographic keys to accept. Some of these produce only a single type of signature, whilst others allow a choice of digest algorithms to use. The signature size, therefore, is dependent on the algorithm choice.

Signature Tag Name
TAG_SIG_RAW_32
TAG_SIG_RAW_57
TAG_SIG_SHA2_28
TAG_SIG_SHA2_32
TAG_SIG_SHA2_48
TAG_SIG_SHA2_64
TAG_SIG_SHA3_28
TAG_SIG_SHA3_32
TAG_SIG_SHA3_48
TAG_SIG_SHA3_64

Table 5: Signature Digest Tag Names

For the tag values, see Section 3.3.9.

3.3.9. Tag Values

The full list of tag values is given below. There is method to the seemingly arbitrary values; for an explanation and more complete table, see Appendix A.

Tag Name	Decimal Value	Hexadecimal
TAG_TOKEN	32	0x20
TAG_TOKEN_TYPE	36	0x24

Tag Name	Decimal Value	Hexadecimal
TAG_ISSUER_ID	40	0x28
TAG_SEQUENCE_NO	44	0x2c
TAG_SCOPE	48	0x30
TAG_SCOPE_FROM	52	0x34
TAG_SCOPE_TO	64	0x40
TAG_SCOPE_EXPIRY_POLICY	68	0x44
TAG_CLAIMS	72	0x48
TAG_CLAIM_SUBJECT	76	0x4c
TAG_CLAIM_PREDICATE	80	0x50
TAG_CLAIM_OBJECT	84	0x54
TAG_ID_NONE	8	0x08
TAG_ID_WILDCARD	12	0x0c
TAG_ID_RAW_32	5	0x05
TAG_ID_RAW_57	29	0x1d
TAG_ID_SHA3_28	3	0x03
TAG_ID_SHA3_32	7	0x07
TAG_ID_SHA3_48	23	0x17
TAG_ID_SHA3_64	39	0x27
TAG_SIG_RAW_32	69	0x45
TAG_SIG_RAW_57	93	0x5d
TAG_SIG_SHA2_28	66	0x42

Tag Name	Decimal Value	Hexadecimal
TAG_SIG_SHA2_32	70	0x46
TAG_SIG_SHA2_48	86	0x56
TAG_SIG_SHA2_64	102	0x66
TAG_SIG_SHA3_28	67	0x43
TAG_SIG_SHA3_32	71	0x47
TAG_SIG_SHA3_48	87	0x57
TAG_SIG_SHA3_64	103	0x67

Table 6: Tag Values

4. Referenced Encodings

A small number of encodings are defined as normative references, but may require some additional context.

4.1. Variable-Length Integer fields (ULEB128)

The Little Endian Base 128 (LEB128) encoding has no authoritative specification. It is used in a variety of open source projects. The earliest documented case we found is in the [DWARF] standard for debugging file format used by various compilers and debugers.

We use unsigned LEB128 (ULEB128) as defined in the DWARF standard here.

4.2. TAI64 Labels

TAI64 labels are a rarely used, but very simple format for encoding timestamps in a compact fashion. The encoding is defined in [TAI64]. In terms of binary encoding, it is a 64 bit signed integer in network byte order (big endian), representing seconds relative to the beginning of 1970 TAI.

TAI64 reserves values of `2^63` and larger for future use. For the purposes of this document's encoding, we use the value of `(2^64)-1` to indicate a timestamp field without value.

5. Related Considerations

This document adds no considerations related to [RFC8280] or [BCP72] over the base [CAPROCK] document.

5.1. IANA Considerations

This document has no IANA actions.

6. References

6.1. Normative References

[BCP72] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/rfc/rfc3552>.

[CAPROCK] Finkhaeuser, J., "CAProck Distributed Authorization Scheme", n.d., <https://specs.interpeer.io/draft-jfinkhaeuser-caprock-auth-scheme/>.

[DWARF] DWARF Standards Committee, "DWARF Debugging Format Version 5 Standard", 13 February 2017, <https://dwarfstd.org/Dwarf5Std.php>.

[I-D.draft-jfinkhaeuser-caps-for-distributed-auth] Finkhäuser, J. and S. P. ISEP, "Capabilities for Distributed Authorization", Work in Progress, Internet-Draft, draft-jfinkhaeuser-caps-for-distributed-auth-01, 1 June 2023, <https://datatracker.ietf.org/doc/html/draft-jfinkhaeuser-caps-for-distributed-auth-01>.

[NIST.IR.8366] Miller, K., Alderman, D., Carnahan, L., Chen, L., Foti, J., Goldstein, B., Hogan, M., Marshall, J., Reczek, K., Rioux, N., Theofanos, M., and D. Wollman, "Guidance for NIST staff on using inclusive language in documentary standards", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.ir.8366, April 2021, <https://doi.org/10.6028/nist.ir.8366>.

[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/rfc/rfc2119>.

[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/rfc/rfc8174>.

[TAI64] Bernstein, D. J., "TAI64, TAI64N, and TAI64NA", July 1997, <https://cr.yp.to/libtai/tai64.html>.

6.2. Informative References

[ASN.1] International Telephone and Telegraph Consultative Committee, "Abstract Syntax Notation One (ASN.1): Specification of basic notation", CCITT Recommendation X.680, July 2002.

[I-D.draft-knodel-terminology-10] Knodel, M. and N. ten Oever, "Terminology, Power, and Inclusive Language in Internet-Drafts and RFCs", Work in Progress, Internet-Draft, draft-knodel-terminology-10, 11 July 2022, <https://datatracker.ietf.org/doc/html/draft-knodel-terminology-10>.

[ISOC-FOUNDATION] Internet Society Foundation, "Internet Society Foundation", n.d., <https://www.isocfoundation.org/>.

[JSON] 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/rfc/rfc8259>.

[LoRaWAN] LoRa Alliance, Inc., "LoRaWAN™ 1.1 Specification", 2017, <https://lora-alliance.org/resource_hub/lorawan-specification-v1-1/>.

[RFC8280] ten Oever, N. and C. Cath, "Research into Human Rights Protocol Considerations", RFC 8280, DOI 10.17487/RFC8280, October 2017, <https://www.rfc-editor.org/rfc/rfc8280>.

[SCHC] Minaburo, A., Toutain, L., Gomez, C., Barthel, D., and JC. Zuniga, "SCHC: Generic Framework for Static Context Header Compression and Fragmentation", RFC 8724, DOI 10.17487/RFC8724, April 2020, <https://www.rfc-editor.org/rfc/rfc8724>.

[SCHC-LORA] Muñoz, R., Saez Hidalgo, J., Canales, F., Dujovne, D., and S. Céspedes, "SCHC over LoRaWAN Efficiency: Evaluation and Experimental Performance of Packet Fragmentation", MDPI AG, Sensors vol. 22, no. 4, pp. 1531, DOI 10.3390/s22041531, February 2022, <https://doi.org/10.3390/s22041531>.

Appendix A. Scheme for Tag Values

The use of the ULEB128 encoding makes it desirable to define only tag values up to 127 (7 bits), otherwise multiple octets may be used. At the same time, the multi-byte encoding does permit for future extensions.

This document defines a number of tags for identifiers, which all have one of a set of specific sizes; they're either raw public keys or digests thereof (see [CAPROCK]). Similarly, the signatures used in this document relate to the output sizes of digests from the same algorithms.

The resulting number of permutations of digest algorithms and sizes for both types of tag, and adding the number of additional tags, easily permits encoding in 7 bits. However, a scheme may be desirable that permits decoders to determine the identifier/signature sizes with little branching.

Examining the digest sizes first, we note that they fit into 7 bits. 64, the largest digest size, sets the most significant of 7 bits. We can further note that being all sizes (except for 57) being divisible by 4, none have the two least significant bits set.

Digest Size (octets)	Digest Size (binary)
28	0 0 0 1 1 1 0 0
32	0 0 1 0 0 0 0 0
48	0 0 1 1 0 0 0 0
57	0 0 1 1 1 0 0 1
64	0 1 0 0 0 0 0 0

Table 7: Digest Sizes

These two least significant bits can encode up to four distinct values. As luck would have it, we have four different categories of tag to deal with: those unrelated to digest sizes, those using raw digests, those using SHA-2 and those using SHA-3.

However, we need an additional bit to determine whether the size is used in an identifier or in a signature. We can shift to using only 6 bits for the size by simply subtracting the lowest value, 28. In this way, the values for the least significant bits are preserved, but we use one bit less overall.

Adjusted Size (octets)	Digest Size (binary)
28 - 28 = 0	0 0 0 0 0 0 0 0
32 - 28 = 4	0 0 0 0 0 1 0 0
48 - 28 = 20	0 0 0 1 0 1 0 0
57 - 28 = 29	0 0 0 1 1 1 0 1
64 - 28 = 36	0 0 1 0 0 1 0 0

Table 8: Adjusted Digest Sizes

We can still use the least two signficiant bits to determine the digest type:

Category	Mask (binary)	Mask (decimal)
Miscellaneous	0 0 0 0 0 0 0 0	0
Raw	0 0 0 0 0 0 0 1	1
SHA-2	0 0 0 0 0 0 1 0	2

Category	Mask (binary)	Mask (decimal)
SHA-3	`0 0 0 0 0 0 1 1`	3

Table 9: Category Masks

At the same time, we can use the most significant of 7 bits as indicating whether we're dealing with an identifier or a signature.

Digest Use	Mask (binary)
Identifier	`0 0 0 0 0 0 0 0`
Signature	`0 1 0 0 0 0 0 0`

Table 10: Identifier or Signature Bit

If we apply the appropriate masks to the digest size, we can at parse time check for their presence, remove them, add 28, and have the digest size plus whichever categories and uses the masks revealed.

The "raw" category forms somewhat of an exception here. Using `0 1` as the two least significant bits means that removing the mask, the digest size would yield 56 when 57 is expected. This outlier case will require some kind of branch.

At the same time, we're free to use any value below 63 that has the two least significant bits unset as miscellaneous tokens.

Tag Name	Binary Value	Decimal Value	Hexadecimal
TAG_TOKEN	`0 0 1 0 0 0 0 0`	32	0x20
TAG_TOKEN_TYPE	`0 0 0 0 0 0 0 0`	36	0x24
TAG_ISSUER_ID	`0 0 0 0 0 0 0 0`	40	0x28
TAG_SEQUENCE_NO	`0 0 0 0 0 0 0 0`	44	0x2c
TAG_SCOPE	`0 0 0 0 0 0 0 0`	48	0x30
TAG_SCOPE_FROM	`0 0 0 0 0 0 0 0`	52	0x34
TAG_SCOPE_TO	`0 0 0 0 0 0 0 0`	64	0x40
TAG_SCOPE_EXPIRY_POLICY	`0 0 0 0 0 0 0 0`	68	0x44

Tag Name	Binary Value	Decimal Value	Hexadecimal
TAG_CLAIMS	0000 0000	72	0x48
TAG_CLAIM_SUBJECT	0000 0000	76	0x4c
TAG_CLAIM_PREDICATE	0000 0000	80	0x50
TAG_CLAIM_OBJECT	0000 0000	84	0x54
TAG_ID_NONE	0000 1000	8	0x08
TAG_ID_WILDCARD	0000 1100	12	0x0c
TAG_ID_RAW_32	0000 0101	5	0x05
TAG_ID_RAW_57	0001 1101	29	0x1d
TAG_ID_SHA3_28	0000 0011	3	0x03
TAG_ID_SHA3_32	0000 0111	7	0x07
TAG_ID_SHA3_48	0001 0111	23	0x17
TAG_ID_SHA3_64	0010 0111	39	0x27
TAG_SIG_RAW_32	0100 0101	69	0x45
TAG_SIG_RAW_57	0101 1101	93	0x5d
TAG_SIG_SHA2_28	0100 0010	66	0x42
TAG_SIG_SHA2_32	0100 0110	70	0x46
TAG_SIG_SHA2_48	0101 0110	86	0x56
TAG_SIG_SHA2_64	0110 0110	102	0x66
TAG_SIG_SHA3_28	0100 0011	67	0x43
TAG_SIG_SHA3_32	0100 0111	71	0x47
TAG_SIG_SHA3_48	0101 0111	87	0x57

Tag Name	Binary Value	Decimal Value	Hexadecimal
TAG_SIG_SHA3_64	0110 0111	103	0x67

Table 11: Tag Values With Binary Representation

Acknowledgments

Jens Finkhäuser's authorship of this document was performed as part of work undertaken under a grant agreement with the Internet Society Foundation [ISOC-FOUNDATION].

Author's Address

Jens Finkhäuser
Interpeer gUG (haftungsbeschraenkt)
Email: ietf@interpeer.io
URI: https://interpeer.io/