Concise Identities
draft-birkholz-core-coid-01

Abstract

There is an increased demand of trustworthy claim sets -- a set of system entity characteristics tied to an entity via signatures -- in order to provide information. Claim sets represented via CBOR Web Tokens (CWT) can compose a variety of evidence suitable for constrained-node networks and to support secure device automation. This document focuses on sets of identifiers and attributes that are tied to a system entity and are typically used to compose identities appropriate for Constrained RESTful Environment (CoRE) authentication needs.

Status of This Memo

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

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

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

This Internet-Draft will expire on July 6, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
This document is subject to BCP 78 and the IETF Trust’s Legal
Provisions Relating to IETF Documents
(https://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1.  Introduction ........................................ 3
   1.1.  Terminology .................................. 4
2.  Claims in a Concise Identity .......................... 4
   2.1.  iss: CWT issuer ................................ 5
   2.2.  sub: CWT subject ............................... 5
   2.3.  aud: CWT audience ............................. 5
   2.4.  exp: CWT expiration time ...................... 5
   2.5.  nbf: CWT start of validity ................... 5
   2.6.  iat: CWT time of issue ....................... 5
   2.7.  cti: CWT ID .................................. 6
   2.8.  cnf: CWT proof-of-possession key claim ........ 6
3.  Signature Envelope .................................. 6
4.  Processing Rules .................................... 6
5.  IANA Considerations .................................. 6
6.  Security Considerations .............................. 6
7.  References .......................................... 6
   7.1.  Normative References .......................... 6
   7.2.  Informative References ........................ 7
Appendix A.  Common Terminology on Identity Documents ..... 8
   A.1.  Terms Specified in IEEE 802.1AR ................ 8
   A.2.  Terms Specified in RFC 4949 .................... 8
Appendix B.  Concise Identities and Trust Relationships ... 9
Appendix C.  Concise Identity (CoID) CDDL Data Definition based
   on RFC 5280 ......................................... 10
Appendix D.  Concise Secure Device Identifier (CoDeID) based on
   IEEE 802.1AR-2018 .................................... 10
   D.1.  The Intended Use of DevIDs .................... 10
   D.2.  DevID Flavors ................................ 11
   D.3.  Privacy ....................................... 12
   D.4.  Concise DevID CDDL data definition (sans COSE header) . 12
Appendix E.  Attic .................................... 14
   E.1.  Examples of claims taken from IEEE 802.1AR identifiers . 14
   E.1.1.  7.2.1 version ............................... 14
   E.1.2.  7.2.2 serialNumber .......................... 14
   E.1.3.  7.2.3 signature ............................. 14
E.1.4. 7.2.4 issuerName ........................................... 15
E.1.5. 7.2.5 authorityKeyIdentifier ........................... 15
E.1.6. 7.2.7.1 notBefore ....................................... 15
E.1.7. 7.2.7.2 notAfter ......................................... 15
E.1.8. 7.2.8 subject ............................................. 15
E.1.9. 7.2.10 subjectPublicKeyInfo ............................ 15
E.1.10. 7.2.11 signatureAlgorithm ............................. 15
E.1.11. 7.2.12 signatureValue ................................... 15
E.2.  Examples of claims taken from X.509 certificates ..... 16
E.2.1. 2.5.29.35 - Authority Key Identifier .................. 16
E.2.2. 2.5.29.14 - Subject Key Identifier ..................... 16
E.2.3. 2.5.29.15 - Key Usage .................................... 16
E.2.4. 2.5.29.37 - Extended Key Usage ......................... 16
E.2.5. 1.3.6.1.5.5.7.1.1 - Authority Information Access .... 16
E.2.6. 1.3.6.1.4.1.311.20.2 - Certificate Template Name  
           Domain Controller (Microsoft) ........................ 16
Appendix F. Graveyard ........................................... 16
F.1.  7.2.9 subjectAltName  .................................... 17
F.2.  7.2.13 extensions ......................................... 17
F.3.  2.5.29.31 - CRL Distribution Points .................... 17
F.4.  2.5.29.17 - Subject Alternative Name ................... 17
F.5.  2.5.29.19 - Basic Constraints ............................ 17
Acknowledgements ............................................... 17
Authors’ Addresses .............................................. 17

1.  Introduction

X.509 certificates [RFC5280] and Secure Device Identifier  
[IEEE-802.1AR] are ASN.1 encoded Identity Documents and intended to  
be tied to a system entity uniquely identified via these Identity  
Documents. An Identity Document - in general, a public-key  
certificate - can be conveyed to other system entities in order to  
prove or authenticate the identity of the owner of the Identity  
Document. Trust in the proof can be established by mutual trust of  
the provider and assessor of the identity in a third party  
verification (TVP) provided, for example, by a certificate authority  
(CA) or its subsidiaries (sub CA).

The evidence a certificate comprises is typically composed of a set  
of claims that is signed using secret keys issued by a (sub) CA. The  
core set of claims included in a certificate - its attributes - are  
well defined in the X.509v3 specifications and IEEE 802.1AR.

This document summarizes the core set of attributes and provides a  
corresponding list of claims using concise integer labels to be used  
in claim sets for CBOR Web Tokens (CWT) [RFC8392]. A resulting  
Concise Identity (CoID) is able to represent a signed set of claims  
that composes an Identity as defined in [RFC4949].
The objective of using CWT as a basis for the signed claim sets defined in this document is to gain more flexibility and at the same time more rigorously defined semantics for the signed claim sets. In addition, the benefits of using CBOR, COSE, and the corresponding CWT structure accrue, including more compact encoding and a simpler implementation in contrast to classical ASN.1 (DER/BER/PEM) structures and the X.509 complexity and uncertainty that has accreted since X.509 was released 29 years ago. One area where both the compactness and the definiteness are highly desirable is in Constrained-Node Networks [RFC7228], which may also make use of the Constrained Application Protocol (CoAP, [RFC7252]); however, the area of application of Concise Identities is not limited to constrained-node networks.

The present version of this document is a strawman that attempts to indicate the direction the work is intended to take. Not all inspirations this version takes from X.509 maybe need to be taken.

1.1. Terminology

This document uses terminology from [RFC8392] and therefore also [RFC7519], as well as from [RFC8152]. Specifically, we note:

Assertion:

Claim: A piece of information asserted about a subject. A claim is represented as a name/value pair consisting of a Claim Name and a Claim Value.

Claims are grouped into claims sets (represented here by a CWT), which need to be interpreted as a whole. Note that this usage is a bit different from idiomatic English usage, where a claim would stand on its own.

(Note that the current version of this draft is not very explicit about the relationship of identities and identifiers. To be done in next version.)

2. Claims in a Concise Identity

A Concise Identity (CoID) is a CBOR Web Token [RFC8392] with certain claims present. It can be signed in a number of ways, including a COSE_Sign1 data object [RFC8152].
2.1. **iss**: CWT issuer

Optional: identifies the principal that is the claimant for the claims in the CoID ([RFC8392] Section 3.1.1, cf. Section 4.1.1 in [RFC7519]).

- Note that this is a StringOrURI (if it contains a ":" it needs to be a URI)
- For the "string" case (no ":"), there is no way to extract meaningful components from the string
- Make it a URI if it needs to be structured (not for routine retrieval, unless specified so by an application)
- If this URI looks like an HTTP or HTTPS URI then something retrievable by humans should exist there.
- Alternatively, some arithmetic can be applied to the URI (extract origin, add /.well-known/...) to find relevant information.

2.2. **sub**: CWT subject

Optional: identifies the principal that is the subject for the claims in the CoID ([RFC8392] Section 3.1.2, cf. Section 4.1.2 in [RFC7519]).

2.3. **aud**: CWT audience

Optional: identifies the recipients that the CoID is intended for ([RFC8392] Section 3.1.4, cf. Section 4.1.4 in [RFC7519]).

2.4. **exp**: CWT expiration time

Optional: the time on or after which the CoID must no longer be accepted for processing ([RFC8392] Section 3.1.4, cf. Section 4.1.4 in [RFC7519]).

2.5. **nbf**: CWT start of validity

Optional: the time before which the CoID must not be accepted for processing ([RFC8392] Section 3.1.5, cf. Section 4.1.5 in [RFC7519]).

2.6. **iat**: CWT time of issue

Optional: the creation time of the CoID ([RFC8392] Section 3.1.6, cf. Section 4.1.6 in [RFC7519]).
2.7.  cti: CWT ID

The "cti" (CWT ID) claim provides a unique identifier for the CoID ([RFC8392] Section 3.1.7, cf. "jti" in Section 4.1.7 in [RFC7519]).

CWT IDs are intended to be unique within an application, so they need to be either coordinated between issuers or based on sufficient randomness (e.g., 112 bits or more).

2.8.  cnf: CWT proof-of-possession key claim

The "cnf" claim identifies the key that can be used by the subject for proof-of-possession and provides parameters to identify the CWT Confirmation Method ([I-D.ietf-ace-cwt-proof-of-possession] Section 3.1).

3.  Signature Envelope

The signature envelope [TBD: need not actually be envelope, may be detached, too] carries additional information, e.g., the signature, as well as the identification of the signature algorithm employed (COSE: alg). Additional information may pertain to the signature (as opposed to the claims being signed), e.g., a key id (COSE: kid) may be given in the header of the signature.

4.  Processing Rules

(TBD: This should contain some discussion of the processing rules that apply for CoIDs. Some of this will just be pointers to [I-D.ietf-oauth-jwt-bcp].)

5.  IANA Considerations

This document makes no requests of IANA

6.  Security Considerations

7.  References

7.1.  Normative References

[I-D.ietf-ace-cwt-proof-of-possession]
7.2. Informative References

[IEEE-802.1AR]

[RFC4949]

[RFC5652]

[RFC7228]

[RFC7252]
Appendix A. Common Terminology on Identity Documents

To illustrate the purpose and intent of Identity Documents, typically, terms, such as certificates, certificate chains/paths and trust anchors, are used. To provide more context and for the convenience of the reader, three sources of definitions are highlighted in this section.

A.1. Terms Specified in IEEE 802.1AR

1. a certificate is "a digitally signed object that binds information identifying an entity that possesses a secret private key to the corresponding public key."

2. a certificate chain is "an ordered list of intermediate certificates that links an end entity certificate (e.g., a DevID certificate) to a trust anchor."

3. a trust anchor is "a Certificate Authority that is trusted and for which the trusting party holds information, usually in the form of a self-signed certificate issued by the trust anchor".

A.2. Terms Specified in RFC 4949

1. a public-key certificate is "a digital certificate that binds a system entity’s identifier to a public key value, and possibly to additional, secondary data items; i.e., a digitally signed data structure that attests to the ownership of a public key."

2. a certification path is "a linked sequence of one or more public-key certificates [...] that enables a certificate user to verify the signature on the last certificate in the path, and thus enables the user to obtain (from that last certificate) a certified public key, or certified attributes, of the system entity that is the subject of that last certificate."

3. a trust anchor is "a CA that is the subject of a trust anchor certificate or otherwise establishes a trust anchor key. Correspondingly, a trust anchor has a trust anchor certificate that "is a public-key certificate that is used to provide the first public key in a certification path."


1. a public-key certificate is "the public key of an entity, together with some other information, rendered unforgeable by digital signature with the private key of the certification authority (CA) that issued it."
2. A certification path is "an ordered list of one or more public-key certificates, starting with a public-key certificate signed by the trust anchor, and ending with the end-entity public-key certificate to be validated. All intermediate public-key certificates, if any, are certification authority (CA) certificates in which the subject of the preceding public-key certificate is the issuer of the following public-key certificate".

3. A trust anchor is "an entity that is trusted by a relying party and used for validating public-key certificates".

Appendix B. Concise Identities and Trust Relationships

Following the terminology highlighted above, Concise Identities are signed CBOR Web Tokens that compose public-key Identity Documents based on asymmetric key pairs, potentially including additional assertions: claims that are secondary data items.

In the context of certification paths, the "last certificate" in the certification path is the Identity Document that resides on the system component, which presents its Identity Document to relying parties in order to be authenticated. The "first certificate" in the certification path resides on the trust anchor.

In order to be able to rely on the trust put into the Identity Document presented to relying parties, these have to put trust into two assumptions first:

- the corresponding trust anchor (certificate) is trusted. In consequence, the consumer of the Identity Document requires a basis for decision whether to rely on the trust put in the trust anchor certificate, or not (e.g. via policies or a known certification paths).

- the secret key included in the system component that is presenting its Identity Document is protected. In consequence, the secret key has to be stored in a shielded location. Type and quality of the protection or shielding or even its location are assertions that can be included as secondary data items in the Identity Document.

In summary, a path of trust relationships between a system component’s Identity Document and a trusted authority’s Identity Document is required to enable transitive trust in the system component that presents the Identity Document.
Appendix C. Concise Identity (CoID) CDDL Data Definition based on RFC 5280

COSE MUST be used to sign this CoID template flavor.

"signatureAlgorithm" and "signature" are not part of the CoID map but of the COSE envelope.

CoID = { version: uint .range 1..3 ; (8)
issuer: text, ; iss(1)
subject: text / bytes, ; sub(2)
notAfter: uint, ; exp(4)
notBefore: uint ; nbf(5)
serialNumber: uint,; (7)
subjectPublicKeyInfo: [ algorithm: COSE-Algorithm-Value,
subjectPublicKey: bytes,
], ;(9)
? extensions: [ + [ extension-id: uint / registeredID,
extension-value: any,
? criticality: bool,
]
], ;(0)
}

COSE-Algorithm-Value = uint .size 0..2 / nint .size 0..2
registeredID = [ + uint ] ; OID

extensions = 0
issuer = 1
subject = 2
notAfter = 4
notBefore = 5
serialNumber = 7
version = 8
subjectPublicKeyInfo = 9

Appendix D. Concise Secure Device Identifier (CoDeID) based on IEEE 802.1AR-2018

This section illustrates the context and background of Secure Device Identifiers.

D.1. The Intended Use of DevIDs

IEEE 802.1AR Secure Device Identifier are a specific subset of X.509 Identity Documents that are intended to "authenticate a device’s identity", where the corresponding Identity Document is "cryptographically bound to that device". In this context,
"cryptographically bound" means that the Identity Document is "constructed using cryptographic operations to combine a secret with other arbitrary data objects such that it may be proven that the result could only be created by an entity having knowledge of the secret."

While the intent of using X.509 Identity Documents as Device Identifiers starts to blur the line between authentication and authorization, the specification of IEEE 802.1AR Identity Documents provides a meaningful subset of assertions that can be used to identify one or more system components. The following CDDL data definition maps the semantics of an RFC 5280 Public Key Infrastructure Certificate Profile, which provides the basis for the Secure Device Identifier semantics. Both are mapped to a CWT representation.

D.2. DevID Flavors

In order to provide consistent semantics for the claims as defined below, understanding the distinction of IDevIDs (mandatory representation capabilities) and LDevIDs (recommended representation capabilities) is of the essence.

Both flavors of Secure Device Identifiers share most of their assertion semantics (claim sets).

IDevIDs are the _initially_ Secure Device Identifiers that "are normally created during manufacturing or initial provisioning" and are "installed on the device by the manufacturer". IDevIDs are intended to be globally unique and to be stored in a way that protects it from modification (typically, a shielded location). It is important to note that a potential segregation of a manufacturer into separate supply chain/tree entities is not covered by the 802.1AR specification.

LDevIDs are the _local significant_ Secure Device Identifiers that are intended to be "unique in the local administrative domain in which the device is used". In essence, LDevIDs "can be created at any time [after IDevID provisioning], in accordance with local policies". An "LDevID is bound to the device in a way that makes it infeasible for it to be forged or transferred to a device with a different IDevID without knowledge of the private key used to effect the cryptographic binding".
D.3. Privacy

The exposition of IDevID Identity Documents enables global unique identification of a system component. To mitigate the obvious privacy concerns, IDevIDs may also be used as the sole identifier (by disabling the IDevID) to assure the privacy of the user of a DevID and the equipment in which it is installed.

D.4. Concise DevID CDDL data definition (sans COSE header)

COSE MUST be used to sign this DevID flavor, if represented via CoID.

"signature" and "signatureValue" are not part of the CoID map but of the COSE envelope.

"AlgorithmIdentifier" and corresponding "algorithm" and "parameters" should be part of the COSE envelope.
CoDeID = { version: 3, ; (8)
    serialNumber: uint, (7)
    issuer: text, iss(1)
    notAfter: uint, exp(4)
    notBefore: uint, nbf(5)
    subject: text / URI, sub(2)
    subjectPublicKeyInfo: [ algorithm: COSE-Algorithm-Value,
        subjectPublicKey: bytes,
    ], sub(9)
    signatureAlgorithm: COSE-Algorithm-Value
    ; 802.1ar-2018 states
    authorityKeyId: bytes, ; all, non-critical
? subjectKeyId: bytes, ; only intermediates, non-critical
? keyUsage: [ bitmask: bytes .size 1,
    ? criticality: bool,
    ]
? subjectAltName: text / IPv4Address / registeredID,
? HardwareModuleName: [ hwType: registeredID,
    hwSerialNum: bytes,
    ],
? extensions: [ + [ extension-id: uint,
        extension-value: any,
        ? criticality: bool,
    ],
    ]
}

COSE-Algorithm-Value = uint .size 0..2 / nint .size 0..2
IPv4Address = bytes .size 4 / bytes .size 16
registeredID = [ + uint ] ; OID

extensions = 0
issuer = 1
subject = 2
notAfter = 4
notBefore = 5
serialNumber = 7
version = 8
subjectPublicKeyInfo = 9
signatureAlgorithm = 10
authorityKeyId = 11
subjectKeyId = 12
keyUsage = 13 ; could move to COSE header?
subjectAltName = 14
HardwareModuleName = 15
Appendix E. Attic

Notes and previous content that will be pruned in next versions.

E.1. Examples of claims taken from IEEE 802.1AR identifiers

This appendix briefly discusses common fields in a X.509 certificate or an IEEE 802.1AR Secure Device Identifier and relates them to claims in a CoID.

The original purpose of X.509 was only to sign the association between a name and a public key. In principle, if something else needs to be signed as well, CMS [RFC5652] is required. This principle has not been strictly upheld over time; this is demonstrated by the growth of various extensions to X.509 certificates that might or might not be interpreted to carry various additional claims.

This document details only the claim sets for CBOR Web Tokens that are necessary for authentication. The plausible integration or replacement of ASN.1 formats in enrollment protocols, [D]TLS handshakes and similar are not in scope of this document.

Subsections in this appendix are marked by the ASN.1 Object Identifier (OID) typically used for the X.509 item. [TODO: Make this true; there are still some section numbers.]

E.1.1. 7.2.1 version

The version field is typically not employed usefully in an X.509 certificate, except possibly in legacy applications that accept original (pre-v3) X.509 certificates.

Generally, the point of versioning is to deliberately inhibit interoperability (due to semantic meaning changes). CoIDs do not employ versioning. Where future work requires semantic changes, these will be expressed by making alternate kinds of claims.

E.1.2. 7.2.2 serialNumber

Covered by cti claim.

E.1.3. 7.2.3 signature

The signature, as well as the identification of the signature algorithm, are provided by the COSE container (e.g., COSE_Sign1) used to sign the CoID’s CWT.
E.1.4.  7.2.4 issuer Name

Covered by iss claim.

E.1.5.  7.2.5 authoritykeyidentifier

Covered by COSE kid in signature, if needed.

E.1.6.  7.2.7.1 notBefore

Covered by nbf claim.

E.1.7.  7.2.7.2 notAfter

Covered by exp claim.

For Secured Device identifiers, this claim is typically left out.

- get a new one whenever you think you need it ("normal path")
- nonced ocsp? might benefit from a more lightweight freshness verification of existing signed assertion - exploration required!
- (first party only verifiable freshness may be cheaper than third-party verifiable?)

E.1.8.  7.2.8 subject

Covered by sub claim.

Note that if claim sets need to be made about multiple subjects, the favored approach in CoID is to create multiple CoIDs, one each per subject.

E.1.9.  7.2.10 subjectPublicKeyInfo

Covered by cnf claim.

E.1.10.  7.2.11 signatureAlgorithm

In COSE_Sign1 envelope.

E.1.11.  7.2.12 signatureValue

In COSE_Sign1 envelope.
E.2. Examples of claims taken from X.509 certificates

Most claims in X.509 certificates take the form of certificate extensions. This section reviews a few common (and maybe not so common) certificate extensions and assesses their usefulness in signed claim sets.

E.2.1. 2.5.29.35 - Authority Key Identifier

Used in certificate chaining. Can be mapped to COSE "kid" of the issuer.

E.2.2. 2.5.29.14 - Subject Key Identifier

Used in certificate chaining. Can be mapped to COSE "kid" in the "cnf" (see Section 3.4 of [I-D.ietf-ace-cwt-proof-of-possession]).

E.2.3. 2.5.29.15 - Key Usage

Usage information for a key claim that is included in the signed claims. Can be mapped to COSE "key_ops" [TBD: Explain details].

E.2.4. 2.5.29.37 - Extended key usage

Can include additional usage information such as 1.3.6.1.5.5.7.3.1 for TLS server certificates or 1.3.6.1.5.5.7.3.2 for TLS client certificates.

E.2.5. 1.3.6.1.5.5.7.1.1 - Authority Information Access

More information about the signer. May include a pointer to signers higher up in the certificate chain (1.3.6.1.5.5.7.48.2), typically in the form of a URI to their certificate.

E.2.6. 1.3.6.1.4.1.311.20.2 - Certificate Template Name Domain Controller (Microsoft)

This is an example for many ill-defined extensions that are on some arcs of the OID space somewhere.

E.g., the UCS-2 string (ASN.1 BMPString) "IPSECIntermediateOffline"

Appendix F. Graveyard

Items and Content that was already discarded.
F.1. 7.2.9 subjectAltName

(See "sub").

F.2. 7.2.13 extensions

Extensions are handled by adding CWT claims to the CWT.

F.3. 2.5.29.31 - CRL Distribution Points

Usually URIs of places where a CRL germane to the certificate can be obtained. Other forms of validating claim sets may be more appropriate than CRLs for the applications envisaged here.

(Might be replaced by a more general freshness verification approach later. For example one could define a generic "is this valid" request to an authority.)

F.4. 2.5.29.17 - Subject Alternative Name

Additional names for the Subject.

These may be an "OtherName", i.e. a mistery blob "defined by" an ASN.1 OID such as 1.3.6.1.4.1.9.21.2.3, or one out of a few formats such as URIs (which may, then, turn out not to be really URIs). Naming subjects obviously is a major issue that needs attention.

F.5. 2.5.29.19 - Basic Constraints

Can identify the key claim as that for a CA, and can limit the length of a certificate path. Empty in all the examples analyzed.

Any application space can define new fields / claims as appropriate and use them. There is no need for the underlying structure to define an additional extension method for this. Instead, they can use the registry as defined in Section 9.1 of [RFC8392].

Acknowledgements

Authors' Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de
Carsten Bormann  
Universitaet Bremen TZI  
Postfach 330440  
Bremen  D-28359  
Germany  
Phone: +49-421-218-63921  
Email: cabo@tzi.org

Max Pritikin  
Cisco  
Email: pritikin@cisco.com

Robert Moskowitz  
Huawei  
Oak Park, MI  48237  
Email: rgm@labs.htt-consult.com
Abstract

This document performs an analysis of the security and privacy implications of different types of "numeric identifiers" used in IETF protocols, and tries to categorize them based on their interoperability requirements and the associated failure severity when such requirements are not met. It describes a number of algorithms that have been employed in real implementations to meet such requirements and analyzes their security and privacy properties. Additionally, it provides advice on possible algorithms that could be employed to satisfy the interoperability requirements of each identifier type, while minimizing the security and privacy implications, thus providing guidance to protocol designers and protocol implementers. Finally, it provides recommendations for future protocol specifications regarding the specification of the aforementioned numeric identifiers.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.
1. Introduction .................................................. 3
2. Terminology ................................................... 4
3. Threat Model ................................................... 5
4. Issues with the Specification of Identifiers .................. 5
5. Timeline of Vulnerability Disclosures Related to Some Sample
   Identifiers .................................................. 6
   5.1. IPv4/IPv6 Identification ................................. 6
   5.2. TCP Initial Sequence Numbers (ISNs) .................... 7
6. Protocol Failure Severity ....................................... 9
7. Categorizing Identifiers ....................................... 9
   8.1. Category #1: Uniqueness (soft failure) ................. 11
     8.1.1. Simple Randomization Algorithm ..................... 11
     8.1.2. Another Simple Randomization Algorithm ............ 12
   8.2. Category #2: Uniqueness (hard failure) .................. 13
   8.3. Category #3: Uniqueness, constant within context (soft-
       failure) ................................................. 13
   8.4. Category #4: Uniqueness, monotonically increasing within
       context (hard failure) ................................... 14
     8.4.1. Predictable Linear Identifiers Algorithm .......... 14
     8.4.2. Per-context Counter Algorithm ....................... 16
     8.4.3. Simple Hash-Based Algorithm ........................ 18
     8.4.4. Double-Hash Algorithm .............................. 20
     8.4.5. Random-Increments Algorithm ....................... 21
   9.1. Category #1: Uniqueness (soft failure) ................. 23
   9.2. Category #2: Uniqueness (hard failure) ................. 23
   9.3. Category #3: Uniqueness, constant within context (soft
1. Introduction

Network protocols employ a variety of numeric identifiers for different protocol entities, ranging from DNS Transaction IDs (TxIDs) to transport protocol numbers (e.g., TCP ports) or IPv6 Interface Identifiers (IIDs). These identifiers usually have specific properties that must be satisfied such that they do not result in negative interoperability implications (e.g., uniqueness during a specified period of time), and associated failure severities when such properties are not met, ranging from soft to hard failures.

For more than 30 years, a large number of implementations of the TCP/IP protocol suite have been subject to a variety of attacks, with effects ranging from Denial of Service (DoS) or data injection, to information leakage that could be exploited for pervasive monitoring [RFC7528]. The root of these issues has been, in many cases, the poor selection of identifiers in such protocols, usually as a result of an insufficient or misleading specification. While it is generally trivial to identify an algorithm that can satisfy the interoperability requirements for a given identifier, there exists practical evidence that doing so without negatively affecting the security and/or privacy properties of the aforementioned protocols is prone to error.

For example, implementations have been subject to security and/or privacy issues resulting from:

- Predictable TCP sequence numbers
- Predictable transport protocol numbers
- Predictable IPv4 or IPv6 Fragment Identifiers
- Predictable IPv6 IIDs
- Predictable DNS TxIDs
Recent history indicates that when new protocols are standardized or new protocol implementations are produced, the security and privacy properties of the associated identifiers tend to be overlooked and inappropriate algorithms to generate identifier values are either suggested in the specification or selected by implementors. As a result, we believe that advice in this area is warranted.

This document contains a non-exhaustive survey of identifiers employed in various IETF protocols, and aims to categorize such identifiers based on their interoperability requirements, and the associated failure severity when such requirements are not met. Subsequently, it analyzes several algorithms that have been employed in real implementation to meet such requirements and analyzes their security and privacy properties, and provides advice on possible algorithms that could be employed to satisfy the interoperability requirements of each category, while minimizing the associated security and privacy implications. Finally, it provides recommendations for future protocol specifications regarding the specification of the aforementioned numeric identifiers.

2. Terminology

Identifier:
A data object in a protocol specification that can be used to definitively distinguish a protocol object (a datagram, network interface, transport protocol endpoint, session, etc) from all other objects of the same type, in a given context. Identifiers are usually defined as a series of bits and represented using integer values. We note that different identifiers may have additional requirements or properties depending on their specific use in a protocol. We use the term "identifier" as a generic term to refer to any data object in a protocol specification that satisfies the identification property stated above.

Failure Severity:
The consequences of a failure to comply with the interoperability requirements of a given identifier. Severity considers the worst potential consequence of a failure, determined by the system damage and/or time lost to repair the failure. In this document we define two types of failure severity: "soft" and "hard".

Hard Failure:
A hard failure is a non-recoverable condition in which a protocol does not operate in the prescribed manner or it operates with excessive degradation of service. For example, an established TCP connection that is aborted due to an error condition constitutes, from the point of view of the transport protocol, a hard failure,
since it enters a state from which normal operation cannot be recovered.

Soft Failure:
A soft failure is a recoverable condition in which a protocol does not operate in the prescribed manner but normal operation can be resumed automatically in a short period of time. For example, a simple packet-loss event that is subsequently recovered with a retransmission can be considered a soft failure.

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

3. Threat Model
Throughout this document, we assume an attacker does not have physical or logical device to the device(s) being attacked. We assume the attacker can simply send any traffic to the target devices, to e.g. sample identifiers employed by such devices.

4. Issues with the Specification of Identifiers
While assessing protocol specifications regarding the use of identifiers, we found that most of the issues discussed in this document arise as a result of one of the following:

- Protocol specifications which under-specify the requirements for their identifiers
- Protocol specifications that over-specify their identifiers
- Protocol implementations that simply fail to comply with the specified requirements

A number of protocol implementations (too many of them) simply overlook the security and privacy implications of identifiers. Examples of them are the specification of TCP port numbers in [RFC0793], the specification of TCP sequence numbers in [RFC0793], or the specification of the DNS TxID in [RFC1035].

On the other hand, there are a number of protocol specifications that over-specify some of their associated protocol identifiers. For example, [RFC4291] essentially results in link-layer addresses being embedded in the IPv6 Interface Identifiers (IIDs) when the interoperability requirement of uniqueness could be achieved in other ways that do not result in negative security and privacy implications [RFC7721]. Similarly, [RFC2460] suggests the use of a global counter
for the generation of Fragment Identification values, when the interoperability properties of uniqueness per {Src IP, Dst IP} could be achieved with other algorithms that do not result in negative security and privacy implications.

Finally, there are protocol implementations that simply fail to comply with existing protocol specifications. For example, some popular operating systems (notably Microsoft Windows) still fail to implement randomization of transport protocol ephemeral ports, as specified in [RFC6056].

5. Timeline of Vulnerability Disclosures Related to Some Sample Identifiers

This section contains a non-exhaustive timeline of vulnerability disclosures related to some sample identifiers and other work that has led to advances in this area. The goal of this timeline is to illustrate:

- That vulnerabilities related to how the values for some identifiers are generated and assigned have affected implementations for an extremely long period of time.

- That such vulnerabilities, even when addressed for a given protocol version, were later reintroduced in new versions or new implementations of the same protocol.

- That standardization efforts that discuss and provide advice in this area can have a positive effect on protocol specifications and protocol implementations.

5.1. IPv4/IPv6 Identification

December 1998:
[Sanfilippo1998a] finds that predictable IPv4 Identification values can be leveraged to count the number of packets sent by a target node. [Sanfilippo1998b] explains how to leverage the same vulnerability to implement a port-scanning technique known as dumb/idle scan. A tool that implements this attack is publicly released.

November 1999:
[Sanfilippo1999] discusses how to leverage predictable IPv4 Identification to uncover the rules of a number of firewalls.

November 1999:
[Bellovin2002] explains how the IPv4 Identification field can be exploited to count the number of systems behind a NAT.
December 2003:
[Zalewski2003] explains a technique to perform TCP data injection attack based on predictable IPv4 identification values which requires less effort than TCP injection attacks performed with bare TCP packets.

November 2005:
[Silbersack2005] discusses shortcoming in a number of techniques to mitigate predictable IPv4 Identification values.

October 2007:
[Klein2007] describes a weakness in the pseudo random number generator (PRNG) in use for the generation of the IP Identification by a number of operating systems.

June 2011:

November 2011:
Linux mitigates predictable IPv6 Identification values
[RedHat2011] [SUSE2011] [Ubuntu2011].

December 2011:
[I-D.ietf-6man-predictable-fragment-id-08] describes the security implications of predictable IPv6 Identification values, and possible mitigations.

May 2012:

June 2015:
[I-D.ietf-6man-predictable-fragment-id-08] notes that some popular host and router implementations still employ predictable IPv6 Identification values.

5.2. TCP Initial Sequence Numbers (ISNs)

September 1981:
[RFC0793], suggests the use of a global 32-bit ISN generator, whose lower bit is incremented roughly every 4 microseconds. However, such an ISN generator makes it trivial to predict the ISN that a TCP will use for new connections, thus allowing a variety of attacks against TCP.

February 1985:
[Morris1985] was the first to describe how to exploit predictable TCP ISNs for forging TCP connections that could then be leveraged for trust relationship exploitation.

April 1989:
[Bellovin1989] discussed the security implications of predictable ISNs (along with a range of other protocol-based vulnerabilities).

February 1995:
[Shimomura1995] reported a real-world exploitation of the attack described in 1985 (ten years before) in [Morris1985].

May 1996:
[RFC1948] was the first IETF effort, authored by Steven Bellovin, to address predictable TCP ISNs. The same concept specified in this document for TCP ISNs was later proposed for TCP ephemeral ports [RFC6056], TCP Timestamps, and eventually even IPv6 Interface Identifiers [RFC7217].

March 2001:
[Zalewski2001] provides a detailed analysis of statistical weaknesses in some ISN generators, and includes a survey of the algorithms in use by popular TCP implementations.

May 2001:
Vulnerability advisories [CERT2001] [USCERT2001] are released regarding statistical weaknesses in some ISN generators, affecting popular TCP/IP implementations.

March 2002:
[Zalewski2002] updates and complements [Zalewski2001]. It concludes that "while some vendors [...] reacted promptly and tested their solutions properly, many still either ignored the issue and never evaluated their implementations, or implemented a flawed solution that apparently was not tested using a known approach". [Zalewski2002].

February 2012:
[RFC6528], after 27 years of Morris’ original work [Morris1985], formally updates [RFC0793] to mitigate predictable TCP ISNs.

August 2014:
[I-D.eddy-rfc793bis-04], the upcoming revision of the core TCP protocol specification, incorporates the algorithm specified in [RFC6528] as the recommended algorithm for TCP ISN generation.
6. Protocol Failure Severity

Section 2 defines the concept of "Failure Severity" and two types of failures that we employ throughout this document: soft and hard.

Our analysis of the severity of a failure is performed from the point of view of the protocol in question. However, the corresponding severity on the upper application or protocol may not be the same as that of the protocol in question. For example, a TCP connection that is aborted may or may not result in a hard failure of the upper application: if the upper application can establish a new TCP connection without any impact on the application, a hard failure at the TCP protocol may have no severity at the application level. On the other hand, if a hard failure of a TCP connection results in excessive degradation of service at the application layer, it will also result in a hard failure at the application.

7. Categorizing Identifiers

This section includes a non-exhaustive survey of identifiers, and proposes a number of categories that can accommodate these identifiers based on their interoperability requirements and their failure modes (soft or hard)

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Interoperability Requirements</th>
<th>Failure Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv6 Frag ID</td>
<td>Uniqueness (for IP address pair)</td>
<td>Soft/Hard (1)</td>
</tr>
<tr>
<td>IPv6 IID</td>
<td>Uniqueness (and constant within IPv6 prefix) (2)</td>
<td>Soft (3)</td>
</tr>
<tr>
<td>TCP SEQ</td>
<td>Monotonically-increasing</td>
<td>Hard (4)</td>
</tr>
<tr>
<td>TCP eph. port</td>
<td>Uniqueness (for connection ID)</td>
<td>Hard</td>
</tr>
<tr>
<td>IPv6 Flow L.</td>
<td>Uniqueness</td>
<td>None (5)</td>
</tr>
<tr>
<td>DNS TxID</td>
<td>Uniqueness</td>
<td>None (6)</td>
</tr>
</tbody>
</table>

Table 1: Survey of Identifiers

Notes:
While a single collision of Fragment ID values would simply lead to a single packet drop (and hence a "soft" failure), repeated collisions at high data rates might trash the Fragment ID space, leading to a hard failure [RFC4963].

While the interoperability requirements are simply that the Interface ID results in a unique IPv6 address, for operational reasons it is typically desirable that the resulting IPv6 address (and hence the corresponding Interface ID) be constant within each network [I-D.ietf-6man-default-iids] [RFC7217].

While IPv6 Interface IDs must result in unique IPv6 addresses, IPv6 Duplicate Address Detection (DAD) [RFC4862] allows for the detection of duplicate Interface IDs/addresses, and hence such Interface ID collisions can be recovered.

In theory there are no interoperability requirements for TCP sequence numbers, since the TIME-WAIT state and TCP’s "quiet time" take care of old segments from previous incarnations of the connection. However, a widespread optimization allows for a new incarnation of a previous connection to be created if the Initial Sequence Number (ISN) of the incoming SYN is larger than the last sequence number seen in that direction for the previous incarnation of the connection. Thus, monotonically-increasing TCP sequence numbers allow for such optimization to work as expected [RFC6528].

The IPv6 Flow Label is typically employed for load sharing [RFC7098], along with the Source and Destination IPv6 addresses. Reuse of a Flow Label value for the same set {Source Address, Destination Address} would typically cause both flows to be multiplexed into the same link. However, as long as this does not occur deterministically, it will not result in any negative implications.

DNS TxIDs are employed, together with the Source Address, Destination Address, Source Port, and Destination Port, to match DNS requests and responses. However, since an implementation knows which DNS requests were sent for that set of {Source Address, Destination Address, Source Port, and Destination Port, DNS TxID}, a collision of TxID would result, if anything, in a small performance penalty (the response would be discarded when it
is found that it does not answer the query sent in the corresponding DNS query).

Based on the survey above, we can categorize identifiers as follows:

<table>
<thead>
<tr>
<th>Cat #</th>
<th>Category</th>
<th>Sample Proto IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uniqueness (soft failure)</td>
<td>IPv6 Flow L., DNS TxIDs</td>
</tr>
<tr>
<td>2</td>
<td>Uniqueness (hard failure)</td>
<td>IPv6 Frag ID, TCP ephemeral port</td>
</tr>
<tr>
<td>3</td>
<td>Uniqueness, constant within context (soft failure)</td>
<td>IPv6 IIDs</td>
</tr>
<tr>
<td>4</td>
<td>Uniqueness, monotonically increasing within context (hard failure)</td>
<td>TCP ISN</td>
</tr>
</tbody>
</table>

Table 2: Identifier Categories

We note that Category #4 could be considered a generalized case of category #3, in which a monotonically increasing element is added to a constant (within context) element, such that the resulting identifiers are monotonically increasing within a specified context. That is, the same algorithm could be employed for both #3 and #4, given appropriate parameters.

8. Common Algorithms for Identifier Generation

The following subsections describe common algorithms found for Protocol ID generation for each of the categories above.

8.1. Category #1: Uniqueness (soft failure)

8.1.1. Simple Randomization Algorithm
/* Ephemeral port selection function */
id_range = max_id - min_id + 1;
next_id = min_id + (random() % id_range);
count = next_id;

do {
    if(check_suitable_id(next_id))
        return next_id;

    if (next_id == max_id) {
        next_id = min_id;
    } else {
        next_id++;
    }
    count--;
} while (count > 0);

return ERROR;

Note:
random() is a function that returns a pseudo-random unsigned
integer number of appropriate size. Note that the output needs to
be unpredictable, and typical implementations of POSIX random()
function do not necessarily meet this requirement. See [RFC4086]
for randomness requirements for security.

The function check_suitable_id() can check, when possible, whether
this identifier is e.g. already in use. When already used, this
algorithm selects the next available protocol ID.

All the variables (in this and all the algorithms discussed in
this document) are unsigned integers.

8.1.2. Another Simple Randomization Algorithm

The following pseudo-code illustrates another algorithm for selecting
a random identifier in which, in the event the identifier is found to
be not suitable (e.g., already in use), another identifier is
selected randomly:
id_range = max_id - min_id + 1;
next_id = min_id + (random() % id_range);
count = id_range;

do {
    if(check_suitable_id(next_id))
        return next_id;

    next_id = min_id + (random() % id_range);
    count--;
} while (count > 0);

return ERROR;

This algorithm might be unable to select an identifier (i.e., return "ERROR") even if there are suitable identifiers available, when there are a large number of identifiers "in use".

8.2. Category #2: Uniqueness (hard failure)

One of the most trivial approaches for achieving uniqueness for an identifier (with a hard failure mode) is to implement a linear function. As a result, all of the algorithms described in Section 8.4 are of use for complying the requirements of this identifier category.

8.3. Category #3: Uniqueness, constant within context (soft-failure)

The goal of this algorithm is to produce identifiers that are constant for a given context, but that change when the aforementioned context changes.

Keeping one value for each possible "context" may in many cases be considered too onerous in terms of memory requirements. As a workaround, the following algorithm employs a calculated technique (as opposed to keeping state in memory) to maintain the constant identifier for each given context.

In the following algorithm, the function F() provides (statelessly) a constant identifier for each given context.
/ Protocol ID selection function */
id_range = max_id - min_id + 1;

counter = 0;
do {
    offset = F(CONTEXT, counter, secret_key);
    next_id = min_id + (offset % id_range);
    if(check_suitable_id(next_id))
        return next_id;
    counter++;
} while (counter <= MAX_RETRIES);
return ERROR;

The function F() provides a "per-CONTEXT" constant identifier for a given context. 'offset' may take any value within the storage type range since we are restricting the resulting identifier to be in the range [min_id, max_id] in a similar way as in the algorithm described in Section 8.1.1. Collisions can be recovered by incrementing the 'counter' variable and recomputing F().

The function F() should be a cryptographic hash function like SHA-256 [FIPS-SHS]. Note: MD5 [RFC1321] is considered unacceptable for F() [RFC6151]. CONTEXT is the concatenation of all the elements that define a given context. For example, if this algorithm is expected to produce identifiers that are unique per network interface card (NIC) and SLAAC autoconfiguration prefix, the CONTEXT should be the concatenation of e.g. the interface index and the SLAAC autoconfiguration prefix (please see [RFC7217] for an implementation of this algorithm for the generation of IPv6 IIDs).

The secret should be chosen to be as random as possible (see [RFC4086] for recommendations on choosing secrets).

8.4. Category #4: Uniqueness, monotonically increasing within context (hard failure)

8.4.1. Predictable Linear Identifiers Algorithm

One of the most trivial ways to achieve uniqueness with a low identifier reuse frequency is to produce a linear sequence. This obviously assumes that each identifier will be used for a similar period of time.
For example, the following algorithm has been employed in a number of operating systems for selecting IP fragment IDs, TCP ephemeral ports, etc.

```c
/* Initialization at system boot time. Could be random */
next_id = min_id;
id_inc = 1;

/* Identifier selection function */
count = max_id - min_id + 1;

do {
    if (next_id == max_id) {
        next_id = min_id;
    } else {
        next_id = next_id + id_inc;
    }

    if (check_suitable_id(next_id))
        return next_id;

    count--;
} while (count > 0);
return ERROR;
```

Note:
`check_suitable_id()` is a function that checks whether the resulting identifier is acceptable (e.g., whether it's in use, etc.).

For obvious reasons, this algorithm results in predictable sequences. If a global counter is used (such as "next_id" in the example above), a node that learns one protocol identifier can also learn or guess values employed by past and future protocol instances. On the other hand, when the value of increments is known (such as "1" in this case), an attacker can sample two values, and learn the number of identifiers that were generated in-between.

Where identifier reuse would lead to a hard failure, one typical approach to generate unique identifiers (while minimizing the security and privacy implications of predictable identifiers) is to obfuscate the resulting protocol IDs by either:

- Replace the global counter with multiple counters (initialized to a random value)
Randomizing the "increments"

Avoiding global counters essentially means that learning one identifier for a given context (e.g., one TCP ephemeral port for a given (src IP, Dst IP, Dst Port)) is of no use for learning or guessing identifiers for a different context (e.g., TCP ephemeral ports that involve other peers). However, this may imply keeping one additional variable/counter per context, which may be prohibitive in some environments. The choice of id_inc has implications on both the security and privacy properties of the resulting identifiers, but also on the corresponding interoperability properties. On one hand, minimizing the increments (as in "id_inc = 1" in our case) generally minimizes the identifier reuse frequency, albeit at increased predictability. On the other hand, if the increments are randomized predictability of the resulting identifiers is reduced, and the information leakage produced by global constant increments is mitigated.

8.4.2. Per-context Counter Algorithm

One possible way to achieve similar (or even lower) identifier reuse frequency while still avoiding predictable sequences would be to employ a per-context counter, as opposed to a global counter. Such an algorithm could be described as follows:
/* Initialization at system boot time. Could be random */
id_inc = 1;

/* Identifier selection function */
count = max_id - min_id + 1;

if(lookup_counter(CONTEXT) == ERROR){
    create_counter(CONTEXT);
}

next_id = lookup_counter(CONTEXT);

do {
    if (next_id == max_id) {
        next_id = min_id;
    } else {
        next_id = next_id + id_inc;
    }

    if (check_suitable_id(next_id)){
        store_counter(CONTEXT, next_id);
        return next_id;
    }

    count--;
} while (count > 0);

store_counter(CONTEXT, next_id);
return ERROR;

NOTE:

lookup_counter() returns the current counter for a given context, or an error condition if such a counter does not exist.

create_counter() creates a counter for a given context, and initializes such counter to a random value.

store_counter() saves (updates) the current counter for a given context.

check_suitable_id() is a function that checks whether the resulting identifier is acceptable (e.g., whether its in use, etc.).

Essentially, whenever a new identifier is to be selected, the algorithm checks whether there there is a counter for the
corresponding context. If there is, such counter is incremented to obtain the new identifier, and the new identifier updates the corresponding counter. If there is no counter for such context, a new counter is created an initialized to a random value, and used as the new identifier.

This algorithm produces a per-context counter, which results in one linear function for each context. Since the origin of each "line" is a random value, the resulting values are unknown to an off-path attacker.

This algorithm has the following drawbacks:

- If, as a result of resource management, the counter for a given context must be removed, the last identifier value used for that context will be lost. Thus, if subsequently an identifier needs to be generated for such context, that counter will need to be recreated and reinitialized to random value, thus possibly leading to reuse/collision of identifiers.

- If the identifiers are predictable by the destination system (e.g., the destination host represents the context), a vulnerable host might possibly leak to third parties the identifiers used by other hosts to send traffic to it (i.e., a vulnerable Host B could leak to Host C the identifier values that Host A is using to send packets to Host B). Appendix A of [RFC7739] describes one possible scenario for such leakage in detail.

8.4.3. Simple Hash-Based Algorithm

The goal of this algorithm is to produce monotonically-increasing sequences, with a randomized initial value, for each given context. For example, if the identifiers being generated must be unique for each {src IP, dst IP} set, then each possible combination of {src IP, dst IP} should have a corresponding "next_id" value.

Keeping one value for each possible "context" may in many cases be considered too onerous in terms of memory requirements. As a workaround, the following algorithm employs a calculated technique (as opposed to keeping state in memory) to maintain the random offset for each possible context.

In the following algorithm, the function F() provides (statelessly) a random offset for each given context.
/* Initialization at system boot time. Could be random. */
counter = 0;

/* Protocol ID selection function */
id_range = max_id - min_id + 1;
offset = F(CONTEXT, secret_key);
count = id_range;

do {
    next_id = min_id + (counter + offset) % id_range;
    counter++;
    if(check_suitable_id(next_id))
        return next_id;
    count--;
} while (count > 0);

return ERROR;

The function F() provides a "per-CONTEXT" fixed offset within the identifier space. Both the 'offset' and 'counter' variables may take any value within the storage type range since we are restricting the resulting identifier to be in the range \([\text{min\_id}, \text{max\_id}]\) in a similar way as in the algorithm described in Section 8.1.1. This allows us to simply increment the 'counter' variable and rely on the unsigned integer to wrap around.

The function F() should be a cryptographic hash function like SHA-256 [FIPS-SHS]. Note: MD5 [RFC1321] is considered unacceptable for F() [RFC6151]. CONTEXT is the concatenation of all the elements that define a given context. For example, if this algorithm is expected to produce identifiers that are monotonically-increasing for each set (Source IP Address, Destination IP Address), the CONTEXT should be the concatenation of these two values.

The secret should be chosen to be as random as possible (see [RFC4086] for recommendations on choosing secrets).

It should be noted that, since this algorithm uses a global counter ("counter") for selecting identifiers, if an attacker could, e.g., force a client to periodically establish a new TCP connection to an attacker-controlled machine (or through an attacker-observable routing path), the attacker could substract consecutive source port
values to obtain the number of outgoing TCP connections established
globally by the target host within that time period (up to wrap-
around issues and five-tuple collisions, of course).

8.4.4. Double-Hash Algorithm

A trade-off between maintaining a single global ‘counter’ variable
and maintaining 2**N ‘counter’ variables (where N is the width of the
result of F()) could be achieved as follows. The system would keep
an array of TABLE_LENGTH integers, which would provide a separation
of the increment of the ‘counter’ variable. This improvement could
be incorporated into the algorithm from Section 8.4.3 as follows:

```c
/* Initialization at system boot time */
for(i = 0; i < TABLE_LENGTH; i++)
  table[i] = random();

id_inc = 1;

/* Protocol ID selection function */
id_range = max_id - min_id + 1;
offset = F(CONTEXT, secret_key1);
index = G(CONTEXT, secret_key2);
count = id_range;

do {
  next_id = min_id + (offset + table[index]) % id_range;
  table[index] = table[index] + id_inc;

  if(check_suitable_id(next_id))
    return next_id;

  count--;
} while (count > 0);

return ERROR;
```

‘table[]’ could be initialized with random values, as indicated by
the initialization code in pseudo-code above. The function G() should be a cryptographic hash function. It should use the same
CONTEXT as F(), and a secret key value to compute a value between 0 and (TABLE_LENGTH-1). Alternatively, G() could take an "offset" as input, and perform the exclusive-or (XOR) operation between all the
bytes in ‘offset’. 
The array ‘table[]’ assures that successive identifiers for a given context will be monotonically-increasing. However, the increments space is separated into TABLE_LENGTH different spaces, and thus identifier reuse frequency will be (probabilistically) lower than that of the algorithm in Section 8.4.3. That is, the generation of identifier for one given context will not necessarily result in increments in the identifiers for other contexts.

It is interesting to note that the size of ‘table[]’ does not limit the number of different identifier sequences, but rather separates the "increments" into TABLE_LENGTH different spaces. The identifier sequence will result from adding the corresponding entry of ‘table[]’ to the variable ‘offset’, which selects the actual identifier sequence (as in the algorithm from Section 8.4.3).

An attacker can perform traffic analysis for any "increment space" into which the attacker has "visibility" -- namely, the attacker can force a node to generate identifiers where G(offset) identifies the target "increment space". However, the attacker’s ability to perform traffic analysis is very reduced when compared to the predictable linear identifiers (described in Section 8.4.1) and the hash-based identifiers (described in Section 8.4.3). Additionally, an implementation can further limit the attacker’s ability to perform traffic analysis by further separating the increment space (that is, using a larger value for TABLE_LENGTH) and/or by randomizing the increments.

8.4.5. Random-Increments Algorithm

This algorithm offers a middle ground between the algorithms that select ephemeral ports randomly (such as those described in Section 8.1.1 and Section 8.1.2), and those that offer obfuscation but no randomization (such as those described in Section 8.4.3 and Section 8.4.4).
/* Initialization code at system boot time. */
next_id = random();       /* Initialization value */
id_inc = 500;            /* Determines the trade-off */

/* Identifier selection function */
id_range = max_id - min_id + 1;
count = id_range;

do {
   /* Random increment */
   next_id = next_id + (random() % id_inc) + 1;

   /* Keep the identifier within acceptable range */
   next_id = min_id + (next_id % id_range);

   if(check_suitable_id(next_id))
      return next_id;
   count--;
} while (count > 0);
return ERROR;

This algorithm aims at producing a monotonically increasing sequence of identifiers, while avoiding the use of fixed increments, which would lead to trivially predictable sequences. The value "id_inc" allows for direct control of the trade-off between the level of obfuscation and the ID reuse frequency. The smaller the value of "id_inc", the more similar this algorithm is to a predictable, global monotonically-increasing ID generation algorithm. The larger the value of "id_inc", the more similar this algorithm is to the algorithm described in Section 8.1.1 of this document.

When the identifiers wrap, there is the risk of collisions of identifiers (i.e., identifier reuse). Therefore, "id_inc" should be selected according to the following criteria:

- It should maximize the wrapping time of the identifier space.
- It should minimize identifier reuse frequency.
- It should maximize obfuscation.

Clearly, these are competing goals, and the decision of which value of "id_inc" to use is a trade-off. Therefore, the value of "id_inc"
should be configurable so that system administrators can make the trade-off for themselves.

9. Common Vulnerabilities Associated with Identifiers

This section analyzes common vulnerabilities associated with the generation of identifiers for each of the categories identified in Section 7.

9.1. Category #1: Uniqueness (soft failure)

Possible vulnerabilities associated with identifiers of this category are:

- Use of trivial algorithms (e.g. global counters) that generate predictable identifiers
- Use of flawed PRNGs.

Since the only interoperability requirement for these identifiers is uniqueness, the obvious approach to generate them is to employ a PRNG. An implementer should consult [RFC4086] regarding randomness requirements for security, and consult relevant documentation when employing a PRNG provided by the underlying system.

Use algorithms other than PRNGs for generating identifiers of this category is discouraged.

9.2. Category #2: Uniqueness (hard failure)

As noted in Section 8.2 this category typically employs the same algorithms as Category #4, since a monotonically-increasing sequence tends to minimize the identifier reuse frequency. Therefore, the vulnerability analysis of Section 9.4 applies to this case.

9.3. Category #3: Uniqueness, constant within context (soft failure)

There are two main vulnerabilities that may be associated with identifiers of this category:

1. Use algorithms or sources that result in predictable identifiers
2. Employing the same identifier across contexts in which constantcy is not required

At times, an implementation or specification may be tempted to employ a source for the identifier which is known to provide unique values. However, while unique, the associated identifiers may have other
properties such as being predictable or leaking information about the node in question. For example, as noted in [RFC7721], embedding link-layer addresses for generating IPv6 IIDs not only results in predictable values, but also leaks information about the manufacturer of the network interface card.

On the other hand, using an identifier across contexts where constancy is not required can be leveraged for correlation of activities. One of the most trivial examples of this is the use of IPv6 IIDs that are constant across networks (such as IIDs that embed the underlying link-layer address).

9.4. Category #4: Uniqueness, monotonically increasing within context (hard failure)

A simple way to generalize algorithms employed for generating identifiers of Category #4 would be as follows:

```c
/* Identifier selection function */
count = max_id - min_id + 1;
do {
    linear(CONTEXT) = linear(CONTEXT) + increment();
    next_id = offset(CONTEXT) + linear(CONTEXT);
    if(check_suitable_id(next_id))
        return next_id;
    count--;
} while (count > 0);
return ERROR;
```

Essentially, an identifier (next_id) is generated by adding a linear function (linear()) to an offset value, which is unknown to the attacker, and constant for given context.

The following aspects of the algorithm should be considered:

- For the most part, it is the offset() function that results in identifiers that are unpredictable by an off-path attacker. While the resulting sequence will be monotonically-increasing, the use of an offset value that is unknown to the attacker makes the resulting values unknown to the attacker.

- The most straightforward "stateless" implementation of offset would be that in which offset() is the result of a
cryptographically-secure hash-function that takes the values that identify the context and a "secret" (not shown in the figure above) as arguments.

- Another possible (but stateful) approach would be to simply generate a random offset and store it in memory, and then look-up the corresponding context when a new identifier is to be selected. The algorithm in Section 8.4.2 is essentially an implementation of this type.

- The linear function is incremented according to increment(). In the most trivial case increment() could always return the constant "1". But it could also possibly return small integers such the increments are randomized.

Considering the generic algorithm illustrated above we can identify the following possible vulnerabilities:

- If the offset value spans more than the necessary context, identifiers could be unnecessarily predictable by other parties, since the offset value would be unnecessarily leaked to them. For example, an implementation that means to produce a per-destination counter but replaces offset() with a constant number (i.e., employs a global counter), will unnecessarily result in predictable identifiers.

- The function linear() could be seen as representing the number of identifiers that have so far been generated for a given context. If linear() spans more than the necessary context, the "increments" could be leaked to other parties, thus disclosing information about the number of identifiers that have so far been generated. For example, an implementation in which linear() is implemented as a single global counter will unnecessarily leak information the number of identifiers that have been produced.

- increment() determines how the linear() is incremented for each identifier that is selected. In the most trivial case, increment() will return the integer "1". However, an implementation may have increment() return a "small" integer value such that even if the current value employed by the generator is guessed (see Appendix A of [RFC7739]), the exact next identifier to be selected will be slightly harder to identify.

10. Security and Privacy Requirements for Identifiers

Protocol specifications that specify identifiers should:
1. Clearly specify the interoperability requirements for selecting the aforementioned identifiers.

2. Provide a security and privacy analysis of the aforementioned identifiers.

3. Recommend an algorithm for generating the aforementioned identifiers that mitigates security and privacy issues, such as those discussed in Section 9.

11. IANA Considerations

There are no IANA registries within this document. The RFC-Editor can remove this section before publication of this document as an RFC.

12. Security Considerations

The entire document is about the security and privacy implications of identifiers.

13. Acknowledgements

The authors would like to thank (in alphabetical order) Steven Bellovin, Joseph Lorenzo Hall, Gre Norcie, and Martin Thomson, for providing valuable comments on earlier versions of this document.

The authors would like to thank Diego Armando Maradona for his magic and inspiration.

14. References

14.1. Normative References


14.2. Informative References

[Bellovin1989]
Internet-Draft Predictable Numeric IDs March 2019

[Bellovin2002]

[CERT2001]

[CPNI-TCP]

[FIPS-SHS]

[Fyodor2004]

[Gont2011]

[Gont2012]

[I-D.eddy-rfc793bis-04]

[I-D.gont-6man-flowlabel-security]

[I-D.ietf-6man-default-iids]
[I-D.ietf-6man-predictable-fragment-id-08]

[Joncheray1995]

[Klein2007]

[Morris1985]

[RedHat2011]


[Silbersack2005]
Silbersack, M., "Improving TCP/IP security through randomization without sacrificing interoperability", EuroBSDCon 2005 Conference, 2005,

[SUSE2011]

[Ubuntu2011]
Ubuntu, "Ubuntu: USN-1253-1: Linux kernel vulnerabilities", 2011,

[USCERT2001]
US-CERT, "US-CERT Vulnerability Note VU#498440: Multiple TCP/IP implementations may use statistically predictable initial sequence numbers", 2001,

[Zalewski2001]
Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis", 2001,

[Zalewski2002]
Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later", 2001,
<http://lcamtuf.coredump.cx/newtcp/>.

[Zalewski2003]
Zalewski, M., "A new TCP/IP blind data injection technique?", 2003,

Authors' Addresses
Fernando Gont
SI6 Networks / UTN-FRH
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Phone: +54 11 4650 8472
Email: fgont@si6networks.com
URI:   http://www.si6networks.com

Ivan Arce
Quarkslab

Email: iarce@quarkslab.com
URI:   https://www.quarkslab.com
Distributing OpenPGP Key Fingerprints with Signed Keylist Subscriptions
draft-mccain-keylist-04

Abstract

This document specifies a system by which an OpenPGP client may subscribe to an organization’s public keylist to keep its keystore up-to-date with correct keys, even in cases where the keys correspond to multiple (potentially uncontrolled) domains. Ensuring that all members or followers of an organization have their colleagues’ most recent PGP public keys is critical to maintaining operational security. Without the most recent keys’ fingerprints and a source of trust for those keys (as this document specifies), users must manually update and sign each others’ keys — a system that is untenable in larger organizations. This document proposes an experimental format for the keylist file as well as requirements for clients who wish to implement this experimental keylist subscription functionality.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 7, 2019.
1. Introduction

This document specifies a system by which clients may subscribe to cryptographically signed `keylists' of public key fingerprints. The public keys do not necessarily all correspond to a single domain. This system enhances operational security by allowing seamless key rotation across entire organizations without centralized public key
hosting. To enable cross-client compatibility, this document provides an experimental format for the keylist, its cryptographic verification, and the method by which it is retrieved by the client. The user interface by which a client provides this functionality to the user is out of scope, as is the process by which the client retrieves public keys. Other non-security-related implementation details are also out of scope.

1.1. Requirements Notation

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

1.2. Terminology

This document uses the terms "OpenPGP", "public key", "private key", "signature", and "fingerprint" as defined by OpenPGP Message Format [RFC4880].

The term "keylist" is defined as a list of OpenPGP public key fingerprints and accessible via a URI. The exact format of this data is specified in Section 3. Keylists SHOULD be treated as public documents; although a system administrator MAY choose, for example, to restrict access to a keylist to a specific subnet.

An "authority key" is defined as the OpenPGP secret key used to sign a particular keylist. Every keylist has a corresponding authority key, and every authority key has at least one corresponding keylist. A single authority key SHOULD NOT be used to sign multiple keylists.

To be "subscribed" to a keylist means that a program will retrieve that keylist on a regular interval. After retrieval, that program will perform an update to an internal OpenPGP keystore.

A "client" is a program that allows the user to subscribe to keylists. A client may be an OpenPGP client itself or a separate program that interfaces with an OpenPGP client to update its keystore.

1.3. Note to Readers

RFC Editor: please remove this section prior to publication.

Development of this Internet draft takes place on GitHub at firstlookmedia/Keylist-RFC [1].
We are still considering whether this Draft is better for the Experimental or Informational track.

2. Functions and Procedures

As new keys are created and other keys are revoked, it is critical that all members of an organization have the most recent set of keys available on their computers. Keylists enable organizations to publish a directory of OpenPGP keys that clients can use to keep their internal keystores up-to-date.

2.1. Subscribing to Keylists

A single client may subscribe to any number of keylists. When a client first subscribes to a keylist, it SHOULD update or import every key present in the keylist into its local keystore. Keylist subscriptions SHOULD be persistent -- that is, they should be permanently stored by the client to enable future automatic updates.

To subscribe to a keylist, the client must be aware of the keylist URI (see [RFC3986]), and the fingerprint of the authority key used to sign the keylist. The protocol used to retrieve the keylist and its signature SHOULD be HTTPS (see [RFC2818]), however other implementation MAY be supported. A client implementing keylist functionality MUST support the retrieval of keylists and signatures over HTTPS. All other protocols are OPTIONAL.

A client MUST NOT employ a trust-on-first-use (TOFU) model for determining the fingerprint of the authority public key; the authority public key fingerprint must be explicitly provided by the user.

The process by which the client stores its keylist subscriptions is out of scope, as is the means by which subscription functionality is exposed to the end-user.

The client MAY provide the option to perform all its network activity over a SOCKS5 proxy (see [RFC1928]).

2.2. Automatic Updates

The primary purpose of keylists is to enable periodic updates of OpenPGP clients' internal keystores. We RECOMMEND that clients provide automatic 'background' update functionality; we also recognize that automatic background updates are not possible in every application (specifically cross-platform CLI tools).
When automatic background updates are provided, we RECOMMEND that clients provide a default refresh interval of less than one day, however we also RECOMMEND that clients allow the user to select this interval. The exact time at which updates are performed is not critical.

To perform an update, the client MUST perform the following steps on each keylist to which it is subscribed. The steps SHOULD be performed in the given order.

1. Obtain a current copy of the keylist from its URI.

2. Obtain a current copy of the keylist’s signature data from its URI, which is included in the keylist data format specified in Section 3.

3. Using the keylist and the keylist’s signature, cryptographically verify that the keylist was signed using the authority key. If the signature does not verify, the client MUST abort the update of this keylist and SHOULD alert the user. The client SHOULD NOT abort the update of other keylists to which it is subscribed, unless they too fail signature verification.

4. Validate the format of the keylist according to Section 3. If the keylist is in an invalid format, the client MUST abort the update this keylist and SHOULD alert the user.

5. For each fingerprint listed in the keyfile, if a copy of the associated public key is not present in the client’s local keystore, retrieve it from the keyserver specified by the keylist (see Section 3) or, if the keylist specifies no keyserver, from any keyserver. If the key is already present and not revoked, refresh it from a keyserver. If it is present and revoked, do nothing.

2.3. Cryptographic Verification of Keylists

To ensure authenticity of a keylist during an update, the client MUST verify that the keylist’s data matches its cryptographic signature, and that the public key used to verify the signature matches the authority key fingerprint given by the user.

For enhanced security, it is RECOMMENDED that keylist operators sign each public key listed in their keylist with the authority private key. This way, an organization can have an internal trust relationship without requiring members of the organization to certify each other’s public keys.
3. Data Element Formats

The following are format specifications for the keylist file and its signature file.

3.1. Keylist

The keylist MUST be a valid JavaScript Object Notation (JSON) Data Interchange Format [RFC8259] object with specific keys and values, as defined below. Note that unless otherwise specified, 'key' in this section refers to JSON keys -- not OpenPGP keys.

To encode metadata, the keylist MUST have a "metadata" root key with an object as the value ("metadata object"). The metadata object MUST contain a "signature_uri" key whose value is the URI string of the keylist's signature file. All metadata keys apart from "signature_uri" are OPTIONAL.

The metadata object MAY contain a "keyserver" key with the value of the URI string of the keyservers from which the OpenPGP keys in the keylist should be retrieved.

The metadata object MAY contain a "comment" key with the value of any string. The metadata object MAY also contain other arbitrary key-value pairs.

The keylist MUST have a "keys" key with an array as the value. This array contains a list of OpenPGP key fingerprints and metadata about them. Each item in the array MUST be an object. Each of these objects MUST have a "fingerprint" key with the value of a string that contains the full 40-character hexadecimal public key fingerprint, as defined in OpenPGP Message Format [RFC4880]. Any number of space characters ( ' ', U+0020) MAY be included at any location in the fingerprint string. These objects MAY contain "name", "email", and "comment" key-value pairs, as well as any other key-value pairs relevant.

The following is an example of a valid keylist.
3.2. Signature

The signature file MUST be an ASCII-armored ‘detached signature’ of the keylist file, as defined in OpenPGP Message Format [RFC4880].

3.3. Well-Known URL

Keylists SHOULD NOT be well-known resources [RFC4880]. To subscribe to a keylist, the client must be aware not only of the keylist’s location, but also of the fingerprint of the authority public key used to sign the keylist. Furthermore, because keylists can reference public keys from several different domains, the host of the well-known location for a keylist may not always be clear.

4. Implementation Status

GPG Sync, an open source program created by one of the authors, implements this experimental standard. GPG Sync is used by First Look Media and the Freedom of the Press Foundation to keep OpenPGP keys in sync across their organizations, as well as to publish their employee’s OpenPGP keys to the world. These organizations collectively employ more than 200 people and have used the system described in this document successfully for multiple years.
GPG Sync’s existing code can be found at
<https://github.com/firstlookmedia/gpgsync>

First Look Media’s keylist file can be found at
<https://github.com/firstlookmedia/gpgsync-firstlook-fingerprints>

5. Security Benefits

The keylist subscription functionality defined in this document provides a number of security benefits, including:

- The ability for new keys to be quickly distributed across an organization.
- Removing the complexity of key distribution from end users, allowing them to focus on the content of their communications rather than on key management.
- The ability for an organization to prevent the spread of falsely attributed keys by centralizing the public key discovery process within their organization without centralized public key hosting.

6. Relation to Other Technologies

6.1. Web Key Directories

Unlike Web Key Directories, keylists are not domain specific. A keylist might contain public key fingerprints for email addresses across several different domains. Moreover, keylists only provide references to public keys by way of fingerprints; Web Key Directories provide the public keys themselves.

6.2. OPENPGPKEY DNS Records

A keylist MAY reference public keys corresponding to email addresses across several different domains. Because managing OPENPGPKEY DNS Records [RFC7929] for a particular domain requires control of that domain, the OPENPGPKEY DNS Record system is not suitable for cases in which keys are strewn about several different domains, including ones outside of the control of an organization’s system administrators.

7. Security Considerations

There is a situation in which keylist subscriptions could pose a potential security threat. If both the authority key and the keylist distribution system were to be compromised, it would be possible for an attacker to distribute any key of their choosing to the subscribers of the keylist. The potential consequences of this
attack are limited, however, because the attacker cannot remove or modify the keys already present on subscribers’ systems.

Some organizations may wish to keep their keylists private. While this may be achievable by serving keylists at URIs only accessible from specific subnets, keylists are designed to be public documents. As such, clients may leak the contents of keylists to keyservers -- this specification ensures to the best of its ability the integrity of keylists, but not the privacy of keylists.

8. IANA Considerations

This document has no actions for IANA.

9. References

9.1. Normative References


9.2. URIs


Authors’ Addresses

R. Miles McCain
First Look Media

Email: ietf@sendmiles.email
URI: https://rmrm.io

Micah Lee
The Intercept

Email: micah.lee@theintercept.com
URI: https://micahflee.com/

Nat Welch
Google

Email: nat@natwelch.com
URI: https://natwelch.com
CBOR Profile of X.509 Certificates
draft-raza-ace-cbor-certificates-02

Abstract

This document specifies a CBOR encoding and profiling of X.509 public key certificate suitable for Internet of Things (IoT) deployments. The full X.509 public key certificate format and commonly used ASN.1 encoding is overly verbose for constrained IoT environments. Profiling together with CBOR encoding reduces the certificate size significantly with associated known performance benefits.

The CBOR certificates are compatible with the existing X.509 standard, enabling the use of profiled and compressed X.509 certificates without modifications in the existing X.509 standard.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 26, 2019.
1. Introduction

One of the challenges with deploying a Public Key Infrastructure (PKI) for the Internet of Things (IoT) is the size and encoding of X.509 public key certificates, since those are not optimized for constrained environments [RFC7228]. More compact certificate representations are desirable. Due to the current PKI usage of X.509 certificates, keeping X.509 compatibility is necessary at least for a transition period. However, the use of a more compact encoding with the Concise Binary Object Representation (CBOR) [I-D.ietf-cbor-7049bis] reduces the certificate size significantly which has known performance benefits in terms of decreased communication overhead, power consumption, latency, storage, etc.
CBOR is a data format designed for small code size and small message size. CBOR builds on the JSON data model but extends it by e.g. encoding binary data directly without base64 conversion. In addition to the binary CBOR encoding, CBOR also has a diagnostic notation that is readable and editable by humans. The Concise Data Definition Language (CDDL) [RFC8610] provides a way to express structures for protocol messages and APIs that use CBOR. [RFC8610] also extends the diagnostic notation.

CBOR data items are encoded to or decoded from byte strings using a type-length-value encoding scheme, where the three highest order bits of the initial byte contain information about the major type. CBOR supports several different types of data items, in addition to integers (int, uint), simple values (e.g. null), byte strings (bstr), and text strings (tstr), CBOR also supports arrays of data items and maps of pairs of data items. For a complete specification and examples, see [I-D.ietf-cbor-7049bis] and [RFC8610].

This document specifies the CBOR certificate profile, which is a CBOR based encoding and compression of the X.509 certificate format. The profile is based on previous work on profiling of X.509 certificates for Internet of Things deployments [X.509-IoT] which retains backwards compatibility with X.509, and can be applied for lightweight certificate based authentication with e.g. DTLS [RFC6347] or EDHOC [I-D.selander-ace-cose-ecdhe]. The same profile can be used for "native" CBOR encoded certificates, which further optimizes the performance in constrained environments but are not backwards compatible with X.509, see Section 6.

Other work has looked at reducing size of X.509 certificates. The purpose of this document is to stimulate a discussion on CBOR based certificates. Further optimizations of this profile are known and will be included in future versions.

o Terminology (#terminology)

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

This specification makes use of the terminology in [RFC7228].
2. X.509 Certificate Profile

This profile is inspired by [RFC7925] and mandates further restrictions to enable reduction of certificate size. In this section we list the required fields in an X.509 certificate needed by devices in IoT deployments. The corresponding ASN.1 schema is given in Appendix B.

In order to comply with this certificate profile, the following restrictions MUST be applied:

- **Version number.** The X.509 standard has not moved beyond version 3 since 2008. With the introduction of certificate extensions new certificate fields can be added without breaking the format, making version changes less likely. Therefore this profile fixes the version number to 3.

- **Serial number.** The serial number together with the identity of the CA is the unique identifier of a certificate. The serial number MUST be an unsigned integer.

- **Signature algorithm.** For the CBOR profile, the signature algorithm is by default assumed to be ECDSA with SHA256.

- **Issuer.** Used to identify the issuing CA through a sequence of name-value pairs. This profile is restricting this to one pair, common name and associated string value. The common name MUST uniquely identify the CA. Other fields MUST NOT be used.

- **Validity.** The following representation MUST be used: UTCTime-format, YYMMDDhhmmss. This is the most compact format allowed by the X.509 standard.

- **Subject.** The subject section has the same format as the issuer, identifying the receiver of the public key through a sequence of name-value pairs. This sequence is in the profile restricted to a single pair, subject name and associated (unique) value. For an IoT-device, the MAC-derived EUI-64 serves this purpose well.

- **Subject public key info.** For the IoT devices, elliptic curve cryptography based algorithms have clear advantages. For the IoT profile the public key algorithm is by default assumed to be prime256v1.

- **Issuer Unique ID and Subject Unique ID.** These fields are optional in X.509 and MUST NOT be used with the CBOR profile.
Extensions. Extensions consist of three parts; an OID, a boolean telling if it is critical or not, and the value. To maintain forward compatibility, the CBOR profile does not restrict the use of extensions. By the X.509-standard, any device must be able to process eight extensions types. Since only four of them are critical for IoT, this profile is making the other four optional. Still mandatory to be understood are:

* Key Usage
* Subject Alternative Name
* Basic Constraints
* Extended Key Usage

Certificate signature algorithm. This field duplicates the info present in the signature algorithm field. By default assumed to be ECDSA with SHA256.

Certificate Signature. The field corresponds to the signature done by the CA private key. For the CBOR profile, this is restricted to ECDSA type signatures with a signature length of 64 bits.

3. CBOR Encoding

This section specifies the CBOR certificates, which are the result of the CBOR encoding and lossless compression of the X.509 certificate profile of the previous section. The CDDL representation is given in Appendix A.

The encoding and compression has several components including: ASN.1 and base64 encoding is replaced with CBOR encoding, static fields are elided, and compression of elliptic curve points. The field encodings and associated savings are listed below. Combining these different components reduces the certificate size significantly, see Figure 1.

* Version number. The version number field is omitted in the encoding. This saves 5 bytes.
* Serial number. The serial number is encoded as an unsigned integer. Encoding overhead is reduced by one byte.
* Signature algorithm. If the signature algorithm is the default it is omitted in the encoding, otherwise encoded as a one byte COSE identifier. This saves 11 or 12 bytes.
o Issuer. Since the profile only allows the common name type, the common name type specifier is omitted. In total, the issuer field encoding overhead goes from 13 bytes to one byte.

o Validity. The time is encoded as UnixTime in integer format. The validity is represented as a ’not before’–’not after’ pair of integer. This reduces the size from 32 to 11 bytes.

o Subject. An IoT subject is identified by a EUI-64, in turn based on a 48bit unique MAC id. This is encoded using only 7 bytes using CBOR. This is a reduction down from 36 bytes for the corresponding ASN.1 encoding.

o Subject public key info. If the algorithm identifier is the default, it is omitted, otherwise encoded as a one-byte COSE identifier. For the allowed ECC type keys, one of the public key ECC curve point elements can be calculated from the other, hence only one of the curve points is needed (point compression, see [PointCompression]). These actions together, for the default algorithm, reduce size from 91 to 35 bytes.

o Extensions. Minor savings are achieved by the compact CBOR encoding. In addition, the relevant X.509 extension OIDs always start with 0x551D, hence these two bytes can be omitted.

o Certificate signature algorithm. This algorithm field is always the same as the above signature algorithm, and is omitted in the encoding.

o Signature. Since the signature algorithm and resulting signature length are known, padding and extra length fields which are present in the ASN.1 encoding are omitted. The overhead for encoding the 64-bit signature value is reduced from 11 to 2 bytes.

4. Deployment settings

CBOR certificates can be deployed with legacy X.509 certificates and CA infrastructure. In order to verify the signature, the CBOR certificate is used to recreate the original X.509 data structure to be able to verify the signature.

For the currently used DTLS v1.2 protocol, where the handshake is sent unencrypted, the actual encoding and compression can be done at different locations depending on the deployment setting. For example, the mapping between CBOR certificate and standard X.509 certificate can take place in a 6LoWPAN border gateway which allows the server side to stay unmodified. This case gives the advantage of the low overhead of a CBOR certificate over a constrained wireless
links. The conversion to X.509 within an IoT device will incur a computational overhead, however, this is negligible compared to the reduced communication overhead.

For the setting with constrained server and server-only authentication, the server only needs to be provisioned with the CBOR certificate and does not perform the conversion to X.509. This option is viable when client authentication can be asserted by other means.

For DTLS v1.3, because certificates are encrypted, the proposed encoding needs to be done fully end-to-end, through adding the encoding/decoding functionality to the server. A new certificate format or new certificate compression scheme needs to be added. While that requires changes on the server side, we believe it to be in line with other proposals utilizing cbor encoding for communication with resource constrained devices.

5. Expected Certificate Sizes

The profiling size saving mainly comes from enforcing removal of issuer and subject info fields besides the common name. The encoding savings are presented above in Section 3, for a sample certificate given in Appendix C resulting in the numbers shown in Figure 1.

After profiling, all duplicated information has been removed, and remaining text strings are minimal in size. Therefore no further size reduction can be reached with general compression mechanisms. (In practice the size might even grow slightly due to the compression encoding information, as illustrated in the table below.)

<table>
<thead>
<tr>
<th>Certificate Size</th>
<th>X.509 Profiled</th>
<th>CBOR Encoded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>313</td>
<td>144</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Certificate Size</th>
<th>X.509 Profiled</th>
<th>CBOR Encoded</th>
<th>Zlib</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>313</td>
<td>144</td>
<td>319</td>
</tr>
</tbody>
</table>

Figure 1: Comparing Sizes of Certificates (bytes)
6. Native CBOR Certificates

Further performance improvements can be achieved with the use of native CBOR certificates. In this case the signature is calculated over the CBOR encoded structure rather than the ASN.1 encoded structure. This removes entirely the need for ASN.1 and reduces the processing in the authenticating devices.

This solution applies when the devices are only required to authenticate with a set of native CBOR certificate compatible servers, which may become a preferred approach for future deployments. The mapping between X.509 and CBOR certificates enables a migration path between the backwards compatible format and the fully optimized format.

7. Security Considerations

The CBOR profiling of X.509 certificates does not change the security assumptions needed when deploying standard X.509 certificates but decreases the number of fields transmitted, which reduces the risk for implementation errors.

Conversion between the certificate formats can be made in constant time to reduce risk of information leakage through side channels.

The current version of the format hardcodes the signature algorithm which does not allow for crypto agility. A COSE crypto algorithm can be specified with small overhead, and this changed is proposed for a future version of the draft.

8. Privacy Considerations

The mechanism in this draft does not reveal any additional information compared to X.509.

Because of difference in size, it will be possible to detect that this profile is used.

The gateway solution described in Section 4 requires unencrypted certificates.

9. IANA Considerations

This document registers a compression algorithm in the registry entitled "Certificate Compression Algorithm IDs", under the "Transport Layer Security (TLS) Extensions" heading (see [I-D.ietf-tls-certificate-compression]).
<table>
<thead>
<tr>
<th>Algorithm Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD</td>
<td>cbor-iot</td>
</tr>
</tbody>
</table>

10.  References

10.1.  Normative References

[I-D.ietf-cbor-7049bis]

[I-D.ietf-tls-certificate-compression]


10.2.  Informative References

[I-D.selander-ace-cose-ecdhe]

[PointCompression]

Appendix A.  CBOR Certificate, CDDL

certificate = [
  serial_number : uint,
  issuer : text,
  validity : [notBefore: int, notAfter: int],
  subject : text / bytes
  public_key : bytes
  ? extensions : [+ extension],
  signature : bytes
  ? signature_alg + public_key_info : bytes
]

extension = [
  oid : int,
  ? critical : bool,
  value : bytes
]

Appendix B.  X.509 Certificate Profile, ASN.1

IOTCertificate DEFINITIONS EXPLICIT TAGS ::= BEGIN

Certificate ::= SEQUENCE {
  tbsCertificate   TBSCertificate,


signatureAlgorithm  SignatureIdentifier,
signature            BIT STRING
}

TBSCertificate ::= SEQUENCE {
    version          \[0\] INTEGER {v3(2)},
    serialNumber      INTEGER (1..MAX),
    signature         SignatureIdentifier,
    issuer            Name,
    validity          Validity,
    subject           Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    extensions        \[3\] Extensions OPTIONAL
}

SignatureIdentifier ::= SEQUENCE {
    algorithm     OBJECT IDENTIFIER (ecdsa-with-SHA256)
Name  ::= SEQUENCE SIZE (1) OF DistinguishedName
DistinguishedName ::= SET SIZE (1) OF CommonName
CommonName ::= SEQUENCE {
    type          OBJECT IDENTIFIER (id-at-commonName),
    value         UTF8String
    -- For a CA, value is CA name, else EUI-64 in format
    -- "01-23-54-FF-FE-AB-CD-EF"
}

Validity ::= SEQUENCE {
    notBefore    UTCTime,
    notAfter     UTCTime
}

SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm     AlgorithmIdentifier,
    subjectPublicKey          BIT STRING
}

AlgorithmIdentifier ::= SEQUENCE {
    algorithm     OBJECT IDENTIFIER (id-ecPublicKey),
    parameters    OBJECT IDENTIFIER (prime256v1)
}

Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension

Extension ::= SEQUENCE {
    extnId        OBJECT IDENTIFIER,
    critical      BOOLEAN DEFAULT FALSE,
    extnValue     OCTET STRING
}
Appendix C. Certificate Example

This section shows an example of an X.509 profiled certificate before CBOR encoding.

Certificate:
Data:
  Version: 3 (0x2)
  Serial Number: DEC (HEX)
  Signature Algorithm: ecdsa-with-SHA256
  Issuer: <23 byte issuer ID>
  Validity
    Not Before: <not_before_ts>
    Not After : <not_after_ts>
  Subject: <23 byte UID>
  Subject Public Key Info:
    Public Key Algorithm: id-ecPublicKey
    Public-Key: (256 bit)
    pub:
      ... ... ...
    ASN1 OID: prime256v1
    NIST CURVE: P-256
X509v3 extensions:
  X509v3 Basic Constraints: critical
  CA:FALSE
  X509v3 Key Usage:
    Digital Signature, Key Encipherment
  Signature Algorithm: ecdsa-with-SHA256
    ... ... ...
Authors’ Addresses

Shahid Raza
RISE AB
Email: shahid.raza@ri.se

Joel Hoeglund
RISE AB
Email: joel.hoglund@ri.se

Goeran Selander
Ericsson AB
Email: goran.selander@ericsson.com

John Mattsson
Ericsson AB
Email: john.mattsson@ericsson.com

Martin Furuhed
Nexus Group
Email: martin.furuheed@nexusgroup.com
Abstract

Cryptographic operations like hashing and signing requires that the original data does not change during serialization or parsing. One way addressing this issue is creating a canonical form of the data. Canonicalization also permits data to be exchanged in its original form on the "wire" while still being subject to secure cryptographic operations. The JSON Canonicalization Scheme (JCS) provides canonicalization support for data in the JSON format by building on the strict serialization methods for JSON primitives defined by ECMAScript, constraining JSON data to the I-JSON subset, and through a deterministic property sorting scheme.

Status of This Memo

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

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

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

This Internet-Draft will expire on November 10, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction

Cryptographic operations like hashing and signing requires that the original data does not change during serialization or parsing. One way of accomplishing this is converting the data into a format that...
has a simple and fixed representation like Base64Url [RFC4648], which
is how JWS [RFC7515] addressed this issue.

Another solution is to create a canonical version of the data,
similar to what was done for the XML Signature [XMLDSIG] standard.
The primary advantage with a canonicalizing scheme is that data can
be kept in its original form. This is the core rationale behind JCS.
Put another way: by using canonicalization a JSON Object may remain a
JSON Object even after being signed which simplifies system design,
documentation and logging.

To avoid "reinventing the wheel", JCS relies on serialization of JSON
primitives compatible with ECMAScript (aka JavaScript) beginning with
version 6 [ES6], hereafter referred to as "ES6".

Seasoned XML developers recalling difficulties getting signatures to
validate (usually due to different interpretations of the quite
intricate XML canonicalization rules as well as of the equally
extensive Web Services security standards), may rightfully wonder why
JCS would not suffer from similar issues. The reasons are twofold:

- The absence of a namespace concept and default values, as well as
  constraining data to the I-JSON subset eliminate the need for
  specific parsers for dealing with canonicalization.

- JCS compatible serialization of JSON primitives is supported by
  most current Web browsers and as well as by Node.js [NODEJS],
  while the full JCS specification is supported by multiple Open
  Source implementations (see Appendix G). See also Appendix F.

In summary the JCS specification describes how serialization of JSON
primitives compliant with ES6 combined with a deterministic property
sorting scheme can be used for creating "Hashable" representations of
JSON data intended for consumption by cryptographic methods.

JCS is compatible with some existing systems relying on JSON
canonicalization such as JWK Thumbprint [RFC7638] and Keybase
[KEYBASE].

For potential uses outside of cryptography see [JSONCOMP].

The intended audiences of this document are JSON tool vendors, as
well as designers of JSON based cryptographic solutions.
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.

3. Detailed Operation

This section describes the different issues related to creating a canonical JSON representation, and how they are addressed by JCS.

3.1. Creation of Input Data

In order to serialize JSON data, one needs data that is adapted for JSON serialization. This is usually achieved by:

- Parsing previously generated JSON data.
- Programmatically creating data.

Irrespective of the method used, the data to be serialized MUST be compatible with I-JSON [RFC7493], which implies the following:

- JSON Objects MUST NOT exhibit duplicate property names.
- JSON String data MUST be expressible as Unicode [UNICODE].
- JSON Number data MUST be expressible as IEEE-754 [IEEE754] double precision values. For applications needing higher precision or longer integers than offered by IEEE-754 double precision, Appendix D outlines how such requirements can be supported in an interoperable and extensible way.

An additional constraint is that parsed JSON String data MUST NOT be altered during subsequent serializations. For more information see Appendix E.

Note: although the Unicode standard offers a possibility combining certain characters into one, referred to as "Unicode Normalization" (https://www.unicode.org/reports/tr15/ [1]), such functionality MUST be delegated to the application layer.
3.2. Generation of Canonical JSON Data

The following subsections describe the steps required for creating a canonical JSON representation of the data elaborated on in the previous section.

Appendix A shows sample code for an ES6 based canonicalizer, matching the JCS specification.

3.2.1. Whitespace

Whitespace between JSON elements MUST NOT be emitted.

3.2.2. Serialization of Primitive Data Types

Assume that you parse a JSON object like the following:

```json
{
  "numbers": [333333333.33333329, 1E30, 4.50, 2e-3, 0.000000000000000000000000001],
  "string": "\u20ac$\u000f\u000aA'B"\\"/",
  "literals": [null, true, false]
}
```

If you subsequently serialize the parsed data using a serializer compliant with ES6’s "JSON.stringify()", the result would (with a line wrap added for display purposes only), be rather divergent with respect to representation of data:

```json
{"numbers":[333333333.3333333,1e+30,4.5,0.002,1e-27],"string": "EURO$\u000f\nA'B"\\"/","literals":[null,true,false]}
```

Note: EURO denotes a single Euro character (Unicode: U+20AC), which not being ASCII, is currently not displayable in RFCs.

The reason for the difference between the parsed data and its serialized counterpart, is due to a wide tolerance on input data (as defined by JSON [RFC8259]), while output data (as defined by ES6), has a fixed representation. As can be seen by the example, numbers are subject to rounding as well.

The following subsections describe serialization of primitive JSON data types according to JCS. This part is identical to that of ES6.
3.2.2.1. Serialization of Literals

The JSON literals "null", "true", and "false" present no challenge since they already have a fixed definition in JSON [RFC8259].

3.2.2.2. Serialization of Strings

For JSON String data (which includes JSON Object property names as well), each Unicode code point MUST be serialized as described below (also matching Section 24.3.2.2 of [ES6]):

- If the Unicode value falls within the traditional ASCII control character range (U+0000 through U+001F), it MUST be serialized using lowercase hexadecimal Unicode notation (\uhhhh) unless it is in the set of predefined JSON control characters U+0008, U+0009, U+000A, U+000C or U+000D which MUST be serialized as \b, \t, \n, \f and \r respectively.

- If the Unicode value is outside of the ASCII control character range, it MUST be serialized "as is" unless it is equivalent to U+005C (\) or U+0022 ("), which MUST be serialized as \\ and \" respectively.

Finally, the resulting sequence of Unicode code points MUST be enclosed in double quotes (").

Note: some JSON systems permit the use of invalid Unicode data including "lone surrogates" (e.g. U+DEAD). Since this leads to interoperability issues including broken signatures, occurrences of such data MUST cause the JCS algorithm to terminate with an error indication.

3.2.2.3. Serialization of Numbers

JSON Number data MUST be serialized according to Section 7.1.12.1 of [ES6] including the "Note 2" enhancement.

Due to the relative complexity of this part, the algorithm itself is not included in this document. However, the specification is fully implemented by for example Google's V8 [V8]. The open source Java implementation mentioned in Appendix G uses a recently developed number serialization algorithm called Ryu [RYU].

Note: since NaN (Not a Number) and Infinity are not permitted in JSON, occurrences of such values MUST cause the JCS algorithm to terminate with an error indication.

3.2.3. Sorting of Object Properties

Although the previous step indeed normalized the representation of primitive JSON data types, the result would not qualify as "canonical" since JSON Object properties are not in lexicographic (alphabetical) order.

Applied to the sample in Section 3.2.2, a properly canonicalized version should (with a line wrap added for display purposes only), read as:

```
{"literals":null,true,false,"numbers":333333333.3333333,1e+30,4.5,0.002,1e-27,"string":"EURO\n\nA'B"\\"/"}
```

Note: EURO denotes a single Euro character (Unicode: U+20AC), which not being ASCII, is currently not displayable in RFCs.

The rules for lexicographic sorting of JSON Object properties according to JCS are as follows:

- JSON Object properties MUST be sorted in a recursive manner which means that possible JSON child Objects MUST have their properties sorted as well.

- JSON Array data MUST also be scanned for presence of JSON Objects (and applying associated property sorting), but array element order MUST NOT be changed.

When a JSON Object is about to have its properties sorted, the following measures MUST be adhered to:

- The sorting process is applied to property name strings in their "raw" (unescaped) form. That is, a newline character is treated as U+000A.

- Property name strings to be sorted are formatted as arrays of UTF-16 [UNICODE] code units. The sorting is based on pure value comparisons, where code units are treated as unsigned integers, independent of locale settings.

- Property name strings either have different values at some index that is a valid index for both strings, or their lengths are different, or both. If they have different values at one or more index positions, let k be the smallest such index; then the string
whose value at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string. If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

In plain English this means that property names are sorted in ascending order like the following:

```
"
"a"
"aa"
"ab"
```

The rationale for basing the sorting algorithm on UTF-16 code units is that it maps directly to the string type in ECMAScript (featured in Web browsers and Node.js), Java and .NET. Systems using another internal representation of string data will need to convert JSON property name strings into arrays of UTF-16 code units before sorting. The conversion from UTF-8 or UTF-32 to UTF-16 is defined by the Unicode [UNICODE] standard.

Note: for the purpose of obtaining a deterministic property order, sorting on UTF-8 or UTF-32 encoded data would also work, but the result would differ and thus be incompatible with this specification. However, in practice property names rarely go outside of 7-bit ASCII making it possible sorting on the UTF-8 byte level and still be compatible with JCS. If this is a viable option or not depends on the environment JCS is supposed to be used in.

### 3.2.4. UTF-8 Generation

Finally, in order to create a platform independent representation, the result of the preceding step MUST be encoded in UTF-8.

Applied to the sample in Section 3.2.3 this should yield the following bytes here shown in hexadecimal notation:

```
7b 22 6c 69 74 65 72 61 6c 73 22 3a 5b 6e 75 6c 6c 2c 74 72 75 65 2c 66 61 6c 73 65 5d 22 6e 75 6d 62 65 72 73 22 3a 5b 33 33 33 33 33 33 33 33 33 2e 33 33 33 33 33 33 33 2c 31 65 2b 33 30 2c 34 2e 35 2c 30 2e 30 30 32 2c 31 65 2d 32 37 5d 22 73 74 72 69 6e 67 22 3a 5c 5c 5c 5c 5c 5c 2f 5d
```

This data is intended to be usable as input to cryptographic methods.
4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

It is vital performing "sanity" checks on input data to avoid overflowing buffers and similar things that could affect the integrity of the system.

When JCS is applied to signature schemes like the one in Appendix F, applications MUST perform the following operations before acting upon received data:

1. Parse the JSON data
2. Verify the data for correctness
3. Verify the signature

6. Acknowledgements

Building on ES6 Number serialization was originally proposed by James Manger. This ultimately led to the adoption of the entire ES6 serialization scheme for JSON primitives.

Other people who have contributed with valuable input to this specification include Scott Ananian, Ben Campbell, Richard Gibson, Bron Gondwana, John-Mark Gurney, Mike Jones, Mike Miller, Mark Nottingham, Mike Samuel, Jim Schaad, Robert Tupelo-Schneck and Michal Wadas.

For carrying out real world concept verification, the software and support for number serialization provided by Ulf Adams, Tanner Gooding and Remy Oudompheng was very helpful.

7. References

7.1. Normative References


7.2. Informal References


7.3. URIs

[1] https://www.unicode.org/reports/tr15/
[8] https://gibson042.github.io/canonicaljson-spec/

Appendix A. ES6 Sample Canonicalizer

Below is an example of a JCS canonicalizer for usage with ES6 based systems:

// Since the primary purpose of this code is highlighting //
// the core of the JCS algorithm, error handling and //
// UTF-8 generation were not implemented //
var canonicalize = function(object) {
    var buffer = '';
    serialize(object);
    return buffer;
}

function serialize(object) {
    if (object === null || typeof object !== 'object' ||
        object.toJSON != null) {
        // Primitive type or toJSON - Use ES6/JSON
        buffer += JSON.stringify(object);
    } else if (Array.isArray(object)) {
        // Array - Maintain element order
        buffer += '[';
        let next = false;
        object.forEach((element) => {
            if (next) {
                buffer += ',';
            }
            next = true;
            // Array element - Recursive expansion
            serialize(element);
        });
        buffer += ']';;
    } else {
        // Object - Sort properties before serializing
        buffer += '{';
        let next = false;
        Object.keys(object).sort().forEach((property) => {
            if (next) {
                buffer += ',';
            }
            next = true;
            // Property names are strings - Use ES6/JSON
            buffer += JSON.stringify(property);
        });
    }
}
buffer += ':';

///////--------------------------------------
// Property value - Recursive expansion //
///////--------------------------------------
serialize(object[property]);
}
buffer += '}'

Appendix B. Number Serialization Samples

The following table holds a set of ES6 compatible Number serialization samples, including some edge cases. The column "IEEE-754" refers to the internal ES6 representation of the Number data type which is based on the IEEE-754 [IEEE754] standard using 64-bit (double precision) values, here expressed in hexadecimal.

<table>
<thead>
<tr>
<th>IEEE-754</th>
<th>JSON Representation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>8000000000000000</td>
<td>0</td>
<td>Minus zero</td>
</tr>
<tr>
<td>0000000000000001</td>
<td>5e-324</td>
<td>Min pos number</td>
</tr>
<tr>
<td>8000000000000001</td>
<td>-5e-324</td>
<td>Min neg number</td>
</tr>
<tr>
<td>7fefffffffffffff</td>
<td>1.7976931348623157e+308</td>
<td>Max pos number</td>
</tr>
<tr>
<td>ffefffffffffffff</td>
<td>-1.7976931348623157e+308</td>
<td>Max neg number</td>
</tr>
<tr>
<td>4340000000000000</td>
<td>9007199254740992</td>
<td>Max pos integer (1)</td>
</tr>
<tr>
<td>c3400000000000000</td>
<td>-9007199254740992</td>
<td>Max neg integer (1)</td>
</tr>
<tr>
<td>4430000000000000</td>
<td>29514790517935283000</td>
<td>2**68 (2)</td>
</tr>
<tr>
<td>7fffffff</td>
<td>NaN</td>
<td>NaN (3)</td>
</tr>
<tr>
<td>7ff00000000000000</td>
<td>Infinity</td>
<td>(3)</td>
</tr>
<tr>
<td>44b52d02c7e14af5</td>
<td>9.999999999999997e+22</td>
<td></td>
</tr>
<tr>
<td>44b52d02c7e14af6</td>
<td>1e+23</td>
<td></td>
</tr>
</tbody>
</table>
Notes:

(1) For maximum compliance with the ES6 "JSON" object values that are to be interpreted as true integers SHOULD be in the range -9007199254740991 to 9007199254740991. However, how numbers are used in applications do not affect the JCS algorithm.

(2) Although a set of specific integers like 2**68 could be regarded as having extended precision, the JCS/ES6 number serialization algorithm does not take this in consideration.

(3) Invalid. See Section 3.2.2.3.

Appendix C. Canonicalized JSON as "Wire Format"

Since the result from the canonicalization process (see Section 3.2.4), is fully valid JSON, it can also be used as "Wire Format". However, this is just an option since cryptographic schemes based on JCS, in most cases would not depend on that externally supplied JSON data already is canonicalized.

In fact, the ES6 standard way of serializing objects using "JSON.stringify()" produces a more "logical" format, where properties
are kept in the order they were created or received. The example below shows an address record which could benefit from ES6 standard serialization:

```json
{
    "name": "John Doe",
    "address": "2000 Sunset Boulevard",
    "city": "Los Angeles",
    "zip": "90001",
    "state": "CA"
}
```

Using canonicalization the properties above would be output in the order "address", "city", "name", "state" and "zip", which adds fuzziness to the data from a human (developer or technical support), perspective. Canonicalization also converts JSON data into a single line of text, which may be less than ideal for debugging and logging.

Appendix D. Dealing with Big Numbers

There are several issues associated with the JSON Number type, here illustrated by the following sample object:

```json
{
    "giantNumber": 1.4e+9999,
    "payMeThis": 26000.33,
    "int64Max": 9223372036854775807
}
```

Although the sample above conforms to JSON [RFC8259], applications would normally use different native data types for storing "giantNumber" and "int64Max". In addition, monetary data like "payMeThis" would presumably not rely on floating point data types due to rounding issues with respect to decimal arithmetic.

The established way handling this kind of "overloading" of the JSON Number type (at least in an extensible manner), is through mapping mechanisms, instructing parsers what to do with different properties based on their name. However, this greatly limits the value of using the JSON Number type outside of its original somewhat constrained, JavaScript context. The ES6 "JSON" object does not support mappings to JSON Number either.

Due to the above, numbers that do not have a natural place in the current JSON ecosystem MUST be wrapped using the JSON String type. This is close to a de-facto standard for open systems. This is also applicable for other data types that do not have direct support in JSON, like "DateTime" objects as described in Appendix E.
Aided by a system using the JSON String type; be it programmatic like

```javascript
var obj = JSON.parse('{"giantNumber": "1.4e+9999"}');
var biggie = new BigNumber(obj.giantNumber);
```

or declarative schemes like OpenAPI [OPENAPI], JCS imposes no limits on applications, including when using ES6.

Appendix E. String Subtype Handling

Due to the limited set of data types featured in JSON, the JSON String type is commonly used for holding subtypes. This can depending on JSON parsing method lead to interoperability problems which MUST be dealt with by JCS compliant applications targeting a wider audience.

Assume you want to parse a JSON object where the schema designer assigned the property "big" for holding a "BigInteger" subtype and "time" for holding a "DateTime" subtype, while "val" is supposed to be a JSON Number compliant with JCS. The following example shows such an object:

```json
{
    "time": "2019-01-28T07:45:10Z",
    "big": "055",
    "val": 3.5
}
```

Parsing of this object can accomplished by the following ES6 statement:

```javascript
var object = JSON.parse(JSON-data-featured-as-a-string);
```

After parsing the actual data can be extracted which for subtypes also involve a conversion step using the result of the parsing process (an ECMAScript object) as input:

```javascript
... = new Date(object.time); // Date object
... = BigInt(object.big); // Big integer
... = object.val; // JSON/JS number
```

Canonicalization of "object" using the sample code in Appendix A would return the following string:

```json
{"big":"055","time":"2019-01-28T07:45:10Z",val:3.5}
```
Although this is (with respect to JCS) technically correct, there is another way parsing JSON data which also can be used with ES6 as shown below:

```javascript
// Currently required to make BigInt JSON serializable
BigInt.prototype.toJSON = function() {
    return this.toString();
};

// JSON parsing using a "stream" based method
var object = JSON.parse(JSON-data-featured-as-a-string,
    (k,v) => k == 'time' ? new Date(v) : k == 'big' ? BigInt(v) : v
);```

If you now apply the canonicalizer in Appendix A to "object", the following string would be generated:

```
{"big":"55","time":"2019-01-28T07:45:10.000Z","val":3.5}
```

In this case the string arguments for "big" and "time" have changed with respect to the original, presumable making an application depending on JCS fail.

The reason for the deviation is that in stream and schema based JSON parsers, the original "string" argument is typically replaced on-the-fly by the native subtype which when serialized, may exhibit a different and platform dependent pattern.

That is, stream and schema based parsing MUST treat subtypes as "pure" (immutable) JSON String types, and perform the actual conversion to the designated native type in a subsequent step. In modern programming platforms like Go, Java and C# this can be achieved with moderate efforts by combining annotations, getters and setters. Below is an example in C#/Json.NET showing a part of a class that is serializable as a JSON Object:

```csharp
// The "pure" string solution uses a local
// string variable for JSON serialization while
// exposing another type to the application
[JsonProperty("amount")]
private string _amount;

[JsonIgnore]
public decimal Amount {
    get { return decimal.Parse(_amount); } 
    set { _amount = value.ToString(); } 
}
```

In an application "Amount" can be accessed as any other property while it is actually represented by a quoted string in JSON contexts.

Note: the example above also addresses the constraints on numeric data implied by I-JSON (the C# "decimal" data type has quite different characteristics compared to IEEE-754 double precision).

E.1. Subtypes in Arrays

Since the JSON Array construct permits mixing arbitrary JSON elements, custom parsing and serialization code must normally be used to cope with subtypes anyway.

Appendix F. Implementation Guidelines

The optimal solution is integrating support for JCS directly in JSON serializers (parsers need no changes). That is, canonicalization would just be an additional "mode" for a JSON serializer. However, this is currently not the case. Fortunately JCS support can be performed through externally supplied canonicalizer software, enabling signature creation schemes like the following:

1. Create the data to be signed.
2. Serialize the data using existing JSON tools.
3. Let the external canonicalizer process the serialized data and return canonicalized result data.
4. Sign the canonicalized data.
5. Add the resulting signature value to the original JSON data through a designated signature property.
6. Serialize the completed (now signed) JSON object using existing JSON tools.

A compatible signature verification scheme would then be as follows:

1. Parse the signed JSON data using existing JSON tools.
2. Read and save the signature value from the designated signature property.
3. Remove the signature property from the parsed JSON object.
4. Serialize the remaining JSON data using existing JSON tools.
5. Let the external canonicalizer process the serialized data and return canonicalized result data.

6. Verify that the canonicalized data matches the saved signature value using the algorithm and key used for creating the signature.

A canonicalizer like above is effectively only a "filter", potentially usable with a multitude of quite different cryptographic schemes.

Using a JSON serializer with integrated JCS support, the serialization performed before the canonicalization step could be eliminated for both processes.

Appendix G. Open Source Implementations

The following Open Source implementations have been verified to be compatible with JCS:

- Java: https://github.com/erdman/java-json-canonicalization [3]

Appendix H. Other JSON Canonicalization Efforts

There are (and have been) other efforts creating "Canonical JSON". Below is a list of URLs to some of them:

- https://gibson042.github.io/canonicaljson-spec/ [8]

In contrast to JCS which is a serialization scheme, the listed efforts build on text level JSON to JSON transformations.
Appendix I. Development Portal

The JCS specification is currently developed at:

The most recent "editors’ copy" can be found at:

JCS source code and test data is available at:
https://github.com/cyberphone/json-canonicalization [12]

Authors’ Addresses

Anders Rundgren
Independent
Montpellier
France

Email: anders.rundgren.net@gmail.com
URI: https://www.linkedin.com/in/andersrundgren/

Bret Jordan
Symantec Corporation
350 Ellis Street
Mountain View CA 94043
USA

Email: bret_jordan@symantec.com

Samuel Erdtman
Spotify AB
Birger Jarlsgatan 61, 4tr
Stockholm 113 56
Sweden

Email: erdtman@spotify.com
The MASQUE Protocol
draft-schinazi-masque-00

Abstract

This document describes MASQUE (Multiplexed Application Substrate over QUIC Encryption). MASQUE is a mechanism that allows co-locating and obfuscating networking applications behind an HTTPS web server. The currently prevalent use-case is to allow running a VPN server that is indistinguishable from an HTTPS server to any unauthenticated observer. We do not expect major providers and CDNs to deploy this behind their main TLS certificate, as they are not willing to take the risk of getting blocked, as shown when domain fronting was blocked. An expected use would be for individuals to enable this behind their personal websites via easy to configure open-source software.

This document is a straw-man proposal. It does not contain enough details to implement the protocol, and is currently intended to spark discussions on the approach it is taking. As we have not yet found a home for this work, discussion is encouraged to happen on the GitHub repository which contains the draft: https://github.com/DavidSchinazi/masque-drafts [1].

Status of This Memo

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

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

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

This Internet-Draft will expire on September 1, 2019.
1. Introduction

This document describes MASQUE (Multiplexed Application Substrate over QUIC Encryption). MASQUE is a mechanism that allows co-locating and obfuscating networking applications behind an HTTPS web server. The currently prevalent use-case is to allow running a VPN server that is indistinguishable from an HTTPS server to any unauthenticated observer. We do not expect major providers and CDNs to deploy this
behind their main TLS certificate, as they are not willing to take the risk of getting blocked, as shown when domain fronting was blocked. An expected use would be for individuals to enable this behind their personal websites via easy to configure open-source software.

This document is a straw-man proposal. It does not contain enough details to implement the protocol, and is currently intended to spark discussions on the approach it is taking. As we have not yet found a home for this work, discussion is encouraged to happen on the GitHub repository which contains the draft: https://github.com/DavidSchinazi/masque-drafts [2].

MASQUE leverages the efficient head-of-line blocking prevention features of the QUIC transport protocol [I-D.ietf-quic-transport] when MASQUE is used in an HTTP/3 [I-D.ietf-quic-http] server. MASQUE can also run in an HTTP/2 server [RFC7540] but at a performance cost.

1.1. 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.

2. Requirements

This section describes the goals and requirements chosen for the MASQUE protocol.

2.1. Invisibility of VPN Usage

An authenticated client using the VPN appears to observers as a regular HTTPS client. Observers only see that HTTP/3 or HTTP/2 is being used over an encrypted channel. No part of the exchanges between client and server may stick out. Note that traffic analysis is currently considered out of scope.

2.2. Invisibility of the Server

To anyone without private keys, the server is indistinguishable from a regular web server. It is impossible to send an unauthenticated probe that the server would reply to differently than if it were a normal web server.
2.3. Fallback to HTTP/2 over TLS over TCP

When QUIC is blocked, MASQUE can run over TCP and still satisfy previous requirements. Note that in this scenario performance may be negatively impacted.

3. Overview of the Mechanism

The server runs an HTTPS server on port 443, and has a valid TLS certificate for its domain. The client has a public/private key pair, and the server maintains a list of authorized MASQUE clients, and their public key. (Alternatively, clients can also be authenticated using a shared secret.) The client starts by establishing a regular HTTPS connection to the server (HTTP/3 over QUIC or HTTP/2 over TLS 1.3 [RFC8446] over TCP), and validates the server's TLS certificate as it normally would for HTTPS. If validation fails, the connection is aborted. The client then uses a TLS keying material exporter [RFC5705] with label "EXPORTER-masque" and no context to generate a 32-byte key. This key is then used as a nonce to prevent replay attacks. The client sends an HTTP CONNECT request for "/.well-known/masque/initial" with the :protocol pseudo-header field set to "masque", and a "Masque-Authentication:" header. The MASQUE authentication header differs from the HTTP "Authorization" header in that it applies to the underlying connection instead of being per-request. It can use either a shared secret or asymmetric authentication. The asymmetric variant uses authentication method "PublicKey", and it transmits a signature of the nonce with the client's public key encoded in base64 format, followed by other information such as the client username and signature algorithm OID. The symmetric variant uses authentication method "HMAC" and transmits an HMAC of the nonce with the shared secret instead of a signature. For example this header could look like:

```
Masque-Authentication: PublicKey u="am9obi5kb2U=":a=1.3.101.112;
 s="SW5zZXJ0IHNpZzZ5hdHVyZSBvZ2VsYXJ5IjIzNzU5MiBiaXRzIGZvciBFZDI1NTE5IQ=="
```

```
Masque-Authentication: HMAC u="am9obi5kb2U=":a=2.16.840.1.101.3.4.2.3;
 s="SW5zZXJ0IHNpZzZ5hdHVyZSBvZ2VsYXJ5IjIzNzU5MiBiaXRzIGZvciBFZDI1NTE5IQ=="
```

Figure 1: MASQUE Authentication Format Example

When the server receives this CONNECT request, it verifies the signature and if that fails responds with code "405 Method Not Allowed", making sure its response is the same as what it would return for any unexpected CONNECT request. If the signature
verifies, the server responds with code "101 Switching Protocols", and from then on this HTTP stream is now dedicated to the MASQUE protocol. That protocol provides a reliable bidirectional message exchange mechanism, which is used by the client and server to negotiate what protocol options are supported and enabled by policy, and client VPN configuration such as IP addresses. When using QUIC, this protocol also allows endpoints to negotiate the use of QUIC extensions, such as support for the DATAGRAM extension [I-D.pauly-quic-datagram].

4. Mechanisms the Server Can Advertise to Authenticated Clients

Once a server has authenticated the client’s MASQUE CONNECT request, it advertises services that the client may use. These services allow for example varying degrees of proxying services to help a client obfuscate the ultimate destination of their traffic.

4.1. HTTP Proxy

The client can make proxied HTTP requests through the server to other servers. In practice this will mean using the CONNECT method to establish a stream over which to run TLS to a different remote destination.

4.2. DNS over HTTPS

The client can send DNS queries using DNS over HTTPS (DoH) [RFC8484] to the MASQUE server.

4.3. UDP Proxying

In order to support WebRTC or QUIC to further servers, clients need a way to relay UDP onwards to a remote server. In practice for most widely deployed protocols other than DNS, this involves many datagrams over the same ports. Therefore this mechanism implements that efficiently: clients can use the MASQUE protocol stream to request an UDP association to an IP address and UDP port pair. In QUIC, the server would reply with a DATAGRAM_ID that the client can then use to have UDP datagrams sent to this remote server. Datagrams are then simply transferred between the DATAGRAMs with this ID and the outer server. There will also be a message on the MASQUE protocol stream to request shutdown of a UDP association to save resources when it is no longer needed. When running over TCP, the client opens a new stream with a CONNECT request to the "masque-udp-proxy" protocol and then sends datagrams encapsulated inside the stream with a two-byte length prefix in network byte order. The target IP and port are sent as part of the URL query. Resetting that stream instructs the server to release any associates resources.
4.4. IP Proxying

For the rare cases where the previous mechanisms are not sufficient, proxying can be performed at the IP layer. This would use a different DATAGRAM_ID and IP datagrams would be encoded inside it without framing. Over TCP, a dedicated stream with two byte length prefix would be used. The server can inspect the IP datagram to look for the destination address in the IP header.

4.5. Path MTU Discovery

In the main deployment of this mechanism, QUIC will be used between client and server, and that will most likely be the smallest MTU link in the path due to QUIC header and authentication tag overhead. The client is responsible for not sending overly large UDP packets and notifying the server of the low MTU. Therefore PMTUD is currently seen as out of scope of this document.

5. Security Considerations

Here be dragons. TODO: slay the dragons.

6. IANA Considerations

We will need to register:

- the TLS keying material exporter label "EXPORTER-masque" (spec required)
  https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#exporter-labels [3]
- the new HTTP header "Masque-Authentication"
- the "/.well-known/masque/" URI (expert review)
- The "masque" and "masque-udp-proxy" extended HTTP CONNECT protocols

We will also need to define the MASQUE control protocol and that will be likely to define new registries of its own.
7. References

7.1. Normative References

[I-D.ietf-quic-http]

[I-D.ietf-quic-transport]

[I-D.pauly-quic-datagram]


7.2. Informative References

[I-D.ietf-httpbis-http2-secondary-certs]

[I-D.pardue-httpbis-http-network-tunnelling]

[I-D.schwartz-httpbis-helium]
Schwartz, B., "Hybrid Encapsulation Layer for IP and UDP Messages (HELIUM)", draft-schwartz-httpbis-helium-00 (work in progress), June 2018.

[I-D.sullivan-tls-post-handshake-auth]


7.3. URIs

[3] https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#exporter-labels
Acknowledgments

This proposal was inspired directly or indirectly by prior work from many people. In particular, this work is related to [I-D.schwartz-httpbis-helium] and [I-D.pardue-httpbis-http-network-tunnelling]. The mechanism used to run the MASQUE protocol over HTTP/2 streams was inspired by [RFC8441]. Using the OID for the signature algorithm was inspired by Signature Authentication in IKEv2 [RFC7427].

The author would like to thank Christophe A., an inspiration and true leader of VPNs.

Design Justifications

Using an exported key as a nonce allows us to prevent replay attacks (since it depends on randomness from both endpoints of the TLS connection) without requiring the server to send an explicit nonce before it has authenticated the client. Adding an explicit nonce mechanism would expose the server as it would need to send these nonces to clients that have not been authenticated yet.

The rationale for a separate MASQUE protocol stream is to allow server-initiated messages. If we were to use HTTP semantics, we would only be able to support the client-initiated request-response model. We could have used WebSocket for this purpose but that would have added wire overhead and dependencies without providing useful features.

There are many other ways to authenticate HTTP, however the authentication used here needs to work in a single client-initiated message to meet the requirement of not exposing the server.

The current proposal would also work with TLS 1.2, but in that case TLS false start and renegotiation must be disabled, and the extended master secret and renegotiation indication TLS extensions must be enabled.

If the server or client want to hide that HTTP/2 is used, the client can set its ALPN to an older version of HTTP and then use the Upgrade header to upgrade to HTTP/2 inside the TLS encryption.

The client authentication used here is similar to how Token Binding [RFC8471] operates, but it has very different goals. MASQUE does not use token binding directly because using token binding requires
sending the token_binding TLS extension in the TLS ClientHello, and that would stick out compared to a regular TLS connection.

TLS post-handshake authentication [I-D.sullivan-tls-post-handshake-auth] is not used by this proposal because that requires sending the "post_handshake_auth" extension in the TLS ClientHello, and that would stick out from a regular HTTPS connection.

Client authentication could have benefited from Secondary Certificate Authentication in HTTP/2 [I-D.ietf-httpbis-http2-secondary-certs], however that has two downsides: it requires the server advertising that it supports it in its SETTINGS, and it cannot be sent unprompted by the client, so the server would have to request authentication. Both of these would make the server stick out from regular HTTP/2 servers.

MASQUE proposes a new client authentication method (as opposed to reusing something like HTTP basic authentication) because HTTP authentication methods are conceptually per-request (they need to be repeated on each request) whereas the new method is bound to the underlying connection (be it QUIC or TLS). In particular, this allows sending QUIC DATAGRAM frames without authenticating every frame individually. Additionally, HMAC and asymmetric keying are preferred to sending a password for client authentication since they have a tighter security bound. Going into the design rationale, HMACs (and signatures) need some data to sign, and to avoid replay attacks that should be a fresh nonce provided by the remote peer. Having the server provide an explicit nonce would leak the existence of the server so we use TLS keying material exporters as they provide us with a nonce that contains entropy from the server without requiring explicit communication.

Author’s Address

David Schinazi  
Google LLC  
1600 Amphitheatre Parkway  
Mountain View, California 94043  
United States of America  

Email: dschinazi.ietf@gmail.com