

CoRE Working Group
Internet-Draft
Intended status: Informational
Expires: January 7, 2020

H. Birkholz
Fraunhofer SIT
C. Bormann
Universitaet Bremen TZI
M. Pritikin
Cisco
R. Moskowitz
Huawei
July 06, 2019

Concise Identities
draft-birkholz-core-coid-02

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 January 7, 2020.

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
1.2. Requirements Language	4
2. Claims in a Concise Identity	5
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	6
2.6. iat: CWT time of issue	6
2.7. cti: CWT ID	6
2.8. cnf: CWT proof-of-possession key claim	6
3. The Essential Qualities of the Subject Claim	6
4. Signature Envelope	7
5. Processing Rules	7
6. IANA Considerations	8
7. Security Considerations	8
8. References	8
8.1. Normative References	8
8.2. Informative References	9
Appendix A. Common Terminology on Identity Documents	10
A.1. Terms Specified in IEEE 802.1AR	10
A.2. Terms Specified in RFC 4949	10
A.3. Terms specified in ISO/IEC 9594-8:2017	11
Appendix B. Concise Identities and Trust Relationships	11
Appendix C. Concise Identity (CoID) CDDL Data Definition based on RFC 5280	12
Appendix D. Concise Secure Device Identifier (CoDeID) based on IEEE 802.1AR-2018	12
D.1. The Intended Use of DevIDs	13
D.2. DevID Flavors	13
D.3. Privacy	14
D.4. Concise DevID CDDL data definition (sans COSE header)	14
Appendix E. Concise Attribute Documents	16
Appendix F. Attic	16
F.1. Examples of claims taken from IEEE 802.1AR identifiers	16

F.1.1.	7.2.1 version	17
F.1.2.	7.2.2 serialNumber	17
F.1.3.	7.2.3 signature	17
F.1.4.	7.2.4 issuer Name	17
F.1.5.	7.2.5 authoritykeyidentifier	17
F.1.6.	7.2.7.1 notBefore	17
F.1.7.	7.2.7.2 notAfter	18
F.1.8.	7.2.10 subjectPublicKeyInfo	18
F.1.9.	7.2.11 signatureAlgorithm	18
F.1.10.	7.2.12 signatureValue	18
F.2.	Examples of claims taken from X.509 certificates	18
F.2.1.	2.5.29.35 - Authority Key Identifier	18
F.2.2.	2.5.29.14 - Subject Key Identifier	18
F.2.3.	2.5.29.15 - Key Usage	18
F.2.4.	2.5.29.37 - Extended key usage	19
F.2.5.	1.3.6.1.5.5.7.1.1 - Authority Information Access	19
F.2.6.	1.3.6.1.4.1.311.20.2 - Certificate Template Name Domain Controller (Microsoft)	19
Appendix G.	Graveyard	19
G.1.	7.2.9 subjectAltName	19
G.2.	7.2.13 extensions	19
G.3.	2.5.29.31 - CRL Distribution Points	19
G.4.	2.5.29.17 - Subject Alternative Name	19
G.5.	2.5.29.19 - Basic Constraints	20
Acknowledgements	20
Authors' Addresses	20

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: A statement made by an entity without accompanying evidence of its validity [X.1252].

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.)

1.2. Requirements Language

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. 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]).

- o Note that this is a StringOrURI (if it contains a ":" it needs to be a URI)
- o For the "string" case (no ":"), there is no way to extract meaningful components from the string
- o Make it a URI if it needs to be structured (not for routine retrieval, unless specified so by an application)
- o If this URI looks like an HTTP or HTTPS URI then something retrievable by humans should exist there.
- o 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. The Essential Qualities of the Subject Claim

As highlighted above, the base definition of the representation of the "sub claim" is already covered by [RFC8392] and [RFC7519].

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

In certain cases, the subject of a CoID needs to be an X.500 Distinguished Name in its full glory. These are sequences of relative names, where each relative name has a relative name type and a (text string) value.

```
dn-subject = [* (relativenametype, relativenamevalue)]
```

(RFC 5280 does not appear to specify how many DN components must be in a DN, so this uses a zero or more quantity.)

Any RelativeDistinguishedName values that are SETs of more than one AttributeTypeAndValue are translated into a sequence of pairs of the nametype used and each of the namevalues, lexicographically sorted.

To be able to map these to CBOR, we define labels for the relative name types listed in Section 4.1.2.4 of [RFC5280]:

(Note that unusual relative name types could be represented as OIDs; this would probably best be done by reviving the currently dormant [I-D.bormann-cbor-tags-oid].)

```
relativenametype = &(
  country: 1
  organization: 2
  organizational-unit: 3
  distinguished-name-qualifier: 4
  state-or-province-name: 5
  common-name: 6
  serial-number: 7
  locality: 8
  title: 9
  surname: 10
  given-name: 11
  initials: 12
  pseudonym: 13
  generation-qualifier: 14
)
```

The relative name values for these types are always expressed as CBOR text strings, i.e., in UTF-8:

```
relativenamevalue = text
```

4. 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.

5. 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].)

6. IANA Considerations

This document makes no requests of IANA

7. Security Considerations

8. References

8.1. Normative References

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

Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", draft-ietf-ace-cwt-proof-of-possession-06 (work in progress), February 2019.

[I-D.ietf-oauth-jwt-bcp]

Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", draft-ietf-oauth-jwt-bcp-06 (work in progress), June 2019.

[I-D.ietf-rats-eat]

Mandyam, G., Lundblade, L., Ballesteros, M., and J. O'Donoghue, "The Entity Attestation Token (EAT)", draft-ietf-rats-eat-01 (work in progress), July 2019.

[I-D.tschofenig-rats-psa-token]

Tschofenig, H., Frost, S., Brossard, M., and A. Shaw, "Arm's Platform Security Architecture (PSA) Attestation Token", draft-tschofenig-rats-psa-token-01 (work in progress), April 2019.

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

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.

[RFC5755] Farrell, S., Housley, R., and S. Turner, "An Internet Attribute Certificate Profile for Authorization", RFC 5755, DOI 10.17487/RFC5755, January 2010, <<https://www.rfc-editor.org/info/rfc5755>>.

- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <<https://www.rfc-editor.org/info/rfc8392>>.
- [X.1252] "ITU-T X.1252 (04/2010)", n.d..

8.2. Informative References

- [I-D.bormann-cbor-tags-oid]
Bormann, C. and S. Leonard, "Concise Binary Object Representation (CBOR) Tags and Techniques for Object Identifiers, UUIDs, Enumerations, Binary Entities, Regular Expressions, and Sets", draft-bormann-cbor-tags-oid-06 (work in progress), March 2017.
- [IEEE-802.1AR]
"ISO/IEC/IEEE International Standard for Information technology -- Telecommunications and information exchange between systems -- Local and metropolitan area networks -- Part 1AR: Secure device identity", IEEE standard, DOI 10.1109/ieeestd.2014.6739984, n.d..
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <<https://www.rfc-editor.org/info/rfc4949>>.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <<https://www.rfc-editor.org/info/rfc5652>>.
- [RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <<https://www.rfc-editor.org/info/rfc7228>>.

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

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 ([...] 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".

A.3. Terms specified in ISO/IEC 9594-8:2017

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:

- o 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).
- o 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 LDevIDs 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,
    ], ;(9)
  signatureAlgorithm: COSE-Algorithm-Value ; 802.1ar-2018 states
    ; this must be identical
    ; to cose sig-alg (rm?)
  authorityKeyIdentifier: bytes, ; all, non-critical,
  ? subjectKeyIdentifier: bytes, ; only intermediates, non-critical
  ? keyUsage : [ bitmask: bytes .size 1,
    ? criticality: bool,
    ]
  ? subjectAltName: text / iPAddress / 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

iPAddress = 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

authorityKeyIdentifier = 11

subjectKeyIdentifier = 12

keyUsage = 13 ; could move to COSE header?

subjectAltName = 14

HardwareModuleName = 15

Appendix E. Concise Attribute Documents

Additional well-defined sets of characteristics can be bound to Identity Documents [RFC5280] or Secure Device Identifiers [IEEE-802.1AR]. CDDL specifications to define these can be found in the corresponding appendices above and the Profile for X.509 Internet Attribute Certificates is defined in [RFC5755].

Essentially, various existing CWT specializations, such as the Entity Attestation Tokens [I-D.ietf-rats-eat] and the Platform Security Architecture Tokens [I-D.tschofenig-rats-psa-token] already compose a type of Attribute Certificates today. In order to bridge the gap between these already existing Concise Attribute Documents and binding them to traditional X.509 Identity Documents (pub-key certificates), a sub claim referencing the corresponding Identity Document has to be included in the signed CBOR Web Token (flavor). The mechanics of how to handle the corresponding key material is also defined in [RFC5755] (and this document will elaborate on these in future versions).

With respect to Concise Identity Documents the dn-subject claim should be used. If a Concise Attribute Certificate has to refer to a traditional ASN.1 encoded X.509 Identity document the subject claim should be used. This procedure provides a migration path from ASN.1 encoded Identity documents [RFC5280] to CBOR encoded Concise Identity documents that allows to bind Concise Attribute Documents, such as EAT or PSA Tokens to both kinds of certificates. In an ideal scenario CBOR encoding in the form of [RFC8392] is used both for Concise Identity Documents and Concise Attribute Documents. The alternate uses of subject claims or dn-subject claims addresses the fact that the vast majority of constrained node devices still use an ASN.1 encoding and simplified interoperability between CBOR encoded and ASN.1 encoded documents is still of essence today.

Appendix F. Attic

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

F.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.]

F.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.

F.1.2. 7.2.2 serialNumber

Covered by cti claim.

F.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.

F.1.4. 7.2.4 issuer Name

Covered by iss claim.

F.1.5. 7.2.5 authoritykeyidentifier

Covered by COSE kid in signature, if needed.

F.1.6. 7.2.7.1 notBefore

Covered by nbf claim.

F.1.7. 7.2.7.2 notAfter

Covered by exp claim.

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

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

F.1.8. 7.2.10 subjectPublicKeyInfo

Covered by cnf claim.

F.1.9. 7.2.11 signatureAlgorithm

In COSE_Sign1 envelope.

F.1.10. 7.2.12 signatureValue

In COSE_Sign1 envelope.

F.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.

F.2.1. 2.5.29.35 - Authority Key Identifier

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

F.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]).

F.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].

F.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.

F.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.

F.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 G. Graveyard

Items and Content that was already discarded.

G.1. 7.2.9 subjectAltName

(See "sub").

G.2. 7.2.13 extensions

Extensions are handled by adding CWT claims to the CWT.

G.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.)

G.4. 2.5.29.17 - Subject Alternative Name

Additional names for the Subject.

These may be an "OtherName", i.e. a mystery 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.

G.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

Network Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: September 12, 2019

F. Gont
SI6 Networks / UTN-FRH
I. Arce
Quarkslab
March 11, 2019

Security and Privacy Implications of Numeric Identifiers Employed in
Network Protocols
draft-gont-predictable-numeric-ids-03

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.

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.

This document may not be modified, and derivative works of it may not be created, and it may not be published except as an Internet-Draft.

Table of Contents

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. Common Algorithms for Identifier Generation	11
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. Common Vulnerabilities Associated with Identifiers	23
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	

failure)	23
9.4. Category #4: Uniqueness, monotonically increasing within context (hard failure)	24
10. Security and Privacy Requirements for Identifiers	25
11. IANA Considerations	26
12. Security Considerations	26
13. Acknowledgements	26
14. References	26
14.1. Normative References	26
14.2. Informative References	27
Authors' Addresses	31

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:

- o Predictable TCP sequence numbers
- o Predictable transport protocol numbers
- o Predictable IPv4 or IPv6 Fragment Identifiers
- o Predictable IPv6 IIDs
- o 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 implementators. 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 definitely 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:

- o Protocol specifications which under-specify the requirements for their identifiers
- o Protocol specifications that over-specify their identifiers
- o 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:

- o That vulnerabilities related to how the values for some identifiers are generated and assigned have affected implementations for an extremely long period of time.
- o That such vulnerabilities, even when addressed for a given protocol version, were later reintroduced in new versions or new implementations of the same protocol.
- o 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:

[Gont2011] describes how to perform idle scan attacks in IPv6.

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:

[Gont2012] notes that some major IPv6 implementations still employ predictable IPv6 Identification values.

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)

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

Table 1: Survey of Identifiers

Notes:

- (1)
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].
- (2)
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].
- (3)
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.
- (4)
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].
- (5)
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.
- (6)
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:

Cat #	Category	Sample Proto IDs
1	Uniqueness (soft failure)	IPv6 Flow L., DNS TxIDs
2	Uniqueness (hard failure)	IPv6 Frag ID, TCP ephemeral port
3	Uniqueness, constant within context (soft failure)	IPv6 IIDs
4	Uniqueness, monotonically increasing within context (hard failure)	TCP ISN

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.

```
/* 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 its 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:

- o Replace the global counter with multiple counters (initialized to a random value)

- o 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 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 and 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:

- o 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.
- o 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 `[min_id, 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 subtract 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:

```
/* 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:

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

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

```
/* 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:

- o 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.
- o 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.

- o 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.
- o 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:

- o 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.
- o 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.
- o `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

- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <<https://www.rfc-editor.org/info/rfc2460>>.

- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4086>>.
- [RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <<https://www.rfc-editor.org/info/rfc4291>>.
- [RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <<https://www.rfc-editor.org/info/rfc4862>>.
- [RFC5722] Krishnan, S., "Handling of Overlapping IPv6 Fragments", RFC 5722, DOI 10.17487/RFC5722, December 2009, <<https://www.rfc-editor.org/info/rfc5722>>.
- [RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <<https://www.rfc-editor.org/info/rfc6151>>.
- [RFC6528] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <<https://www.rfc-editor.org/info/rfc6528>>.
- [RFC7098] Carpenter, B., Jiang, S., and W. Tarreau, "Using the IPv6 Flow Label for Load Balancing in Server Farms", RFC 7098, DOI 10.17487/RFC7098, January 2014, <<https://www.rfc-editor.org/info/rfc7098>>.
- [RFC7217] Gont, F., "A Method for Generating Semantically Opaque Interface Identifiers with IPv6 Stateless Address Autoconfiguration (SLAAC)", RFC 7217, DOI 10.17487/RFC7217, April 2014, <<https://www.rfc-editor.org/info/rfc7217>>.

14.2. Informative References

- [Bellovin1989] Bellovin, S., "Security Problems in the TCP/IP Protocol Suite", Computer Communications Review, vol. 19, no. 2, pp. 32-48, 1989, <<https://www.cs.columbia.edu/~smb/papers/ipext.pdf>>.

[Bellovin2002]

Bellovin, S., "A Technique for Counting NATted Hosts",
IMW'02 Nov. 6-8, 2002, Marseille, France, 2002.

[CERT2001]

CERT, "CERT Advisory CA-2001-09: Statistical Weaknesses in
TCP/IP Initial Sequence Numbers", 2001,
<<http://www.cert.org/advisories/CA-2001-09.html>>.

[CPNI-TCP]

Gont, F., "Security Assessment of the Transmission Control
Protocol (TCP)", United Kingdom's Centre for the
Protection of National Infrastructure (CPNI) Technical
Report, 2009, <[http://www.gont.com.ar/papers/
tn-03-09-security-assessment-TCP.pdf](http://www.gont.com.ar/papers/tn-03-09-security-assessment-TCP.pdf)>.

[FIPS-SHS]

FIPS, "Secure Hash Standard (SHS)", Federal Information
Processing Standards Publication 180-4, March 2012,
<[http://csrc.nist.gov/publications/fips/fips180-4/
fips-180-4.pdf](http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf)>.

[Fyodor2004]

Fyodor, "Idle scanning and related IP ID games", 2004,
<<http://www.insecure.org/nmap/idlescan.html>>.

[Gont2011]

Gont, F., "Hacking IPv6 Networks (training course)", Hack
In Paris 2011 Conference Paris, France, June 2011.

[Gont2012]

Gont, F., "Recent Advances in IPv6 Security", BSDCan 2012
Conference Ottawa, Canada. May 11-12, 2012, May 2012.

[I-D.eddy-rfc793bis-04]

Eddy, W., "Transmission Control Protocol Specification",
draft-eddy-rfc793bis-04 (work in progress), August 2014.

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

Gont, F., "Security Assessment of the IPv6 Flow Label",
draft-gont-6man-flowlabel-security-03 (work in progress),
March 2012.

[I-D.ietf-6man-default-iids]

Gont, F., Cooper, A., Thaler, D., and S. LIU,
"Recommendation on Stable IPv6 Interface Identifiers",
draft-ietf-6man-default-iids-16 (work in progress),
September 2016.

- [I-D.ietf-6man-predictable-fragment-id-08]
Gont, F., "Security Implications of Predictable Fragment Identification Values", draft-ietf-6man-predictable-fragment-id-08 (work in progress), June 2015.
- [Joncheray1995]
Joncheray, L., "A Simple Active Attack Against TCP", Proc. Fifth Usenix UNIX Security Symposium, 1995.
- [Klein2007]
Klein, A., "OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability", 2007,
<http://www.trusteer.com/files/OpenBSD_DNS_Cache_Poisoning_and_Multiple_OS_Predictable_IP_ID_Vulnerability.pdf>.
- [Morris1985]
Morris, R., "A Weakness in the 4.2BSD UNIX TCP/IP Software", CSTR 117, AT&T Bell Laboratories, Murray Hill, NJ, 1985,
<<https://pdos.csail.mit.edu/~rtm/papers/117.pdf>>.
- [RedHat2011]
RedHat, "RedHat Security Advisory RHSA-2011:1465-1: Important: kernel security and bug fix update", 2011,
<<https://rhn.redhat.com/errata/RHSA-2011-1465.html>>.
- [RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <<https://www.rfc-editor.org/info/rfc1035>>.
- [RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, DOI 10.17487/RFC1321, April 1992,
<<https://www.rfc-editor.org/info/rfc1321>>.
- [RFC1948] Bellare, S., "Defending Against Sequence Number Attacks", RFC 1948, DOI 10.17487/RFC1948, May 1996,
<<https://www.rfc-editor.org/info/rfc1948>>.
- [RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, DOI 10.17487/RFC4963, July 2007,
<<https://www.rfc-editor.org/info/rfc4963>>.
- [RFC5927] Gont, F., "ICMP Attacks against TCP", RFC 5927, DOI 10.17487/RFC5927, July 2010,
<<https://www.rfc-editor.org/info/rfc5927>>.

- [RFC6056] Larsen, M. and F. Gont, "Recommendations for Transport-Protocol Port Randomization", BCP 156, RFC 6056, DOI 10.17487/RFC6056, January 2011, <<https://www.rfc-editor.org/info/rfc6056>>.
- [RFC7528] Higgs, P. and J. Piesing, "A Uniform Resource Name (URN) Namespace for the Hybrid Broadcast Broadband TV (HbbTV) Association", RFC 7528, DOI 10.17487/RFC7528, April 2015, <<https://www.rfc-editor.org/info/rfc7528>>.
- [RFC7707] Gont, F. and T. Chown, "Network Reconnaissance in IPv6 Networks", RFC 7707, DOI 10.17487/RFC7707, March 2016, <<https://www.rfc-editor.org/info/rfc7707>>.
- [RFC7721] Cooper, A., Gont, F., and D. Thaler, "Security and Privacy Considerations for IPv6 Address Generation Mechanisms", RFC 7721, DOI 10.17487/RFC7721, March 2016, <<https://www.rfc-editor.org/info/rfc7721>>.
- [RFC7739] Gont, F., "Security Implications of Predictable Fragment Identification Values", RFC 7739, DOI 10.17487/RFC7739, February 2016, <<https://www.rfc-editor.org/info/rfc7739>>.
- [Sanfilippo1998a]
Sanfilippo, S., "about the ip header id", Post to Bugtraq mailing-list, Mon Dec 14 1998, <<http://seclists.org/bugtraq/1998/Dec/48>>.
- [Sanfilippo1998b]
Sanfilippo, S., "Idle scan", Post to Bugtraq mailing-list, 1998, <<http://www.kyuzz.org/antirez/papers/dumbscan.html>>.
- [Sanfilippo1999]
Sanfilippo, S., "more ip id", Post to Bugtraq mailing-list, 1999, <<http://www.kyuzz.org/antirez/papers/moreipid.html>>.
- [Shimomura1995]
Shimomura, T., "Technical details of the attack described by Markoff in NYT", Message posted in USENET's comp.security.misc newsgroup Message-ID: <3g5gkl\$5jl@ariel.sdsc.edu>, 1995, <<http://www.gont.com.ar/docs/post-shimomura-usenet.txt>>.

[Silbersack2005]

Silbersack, M., "Improving TCP/IP security through randomization without sacrificing interoperability", EuroBSDCon 2005 Conference, 2005, <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4542&rep=rep1&type=pdf>>.

[SUSE2011]

SUSE, "SUSE Security Announcement: Linux kernel security update (SUSE-SA:2011:046)", 2011, <<http://lists.opensuse.org/opensuse-security-announce/2011-12/msg00011.html>>.

[Ubuntu2011]

Ubuntu, "Ubuntu: USN-1253-1: Linux kernel vulnerabilities", 2011, <<http://www.ubuntu.com/usn/usn-1253-1/>>.

[USCERT2001]

US-CERT, "US-CERT Vulnerability Note VU#498440: Multiple TCP/IP implementations may use statistically predictable initial sequence numbers", 2001, <<http://www.kb.cert.org/vuls/id/498440>>.

[Zalewski2001]

Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis", 2001, <<http://lcamtuf.coredump.cx/oldtcp/tcpseq.html>>.

[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, <<http://lcamtuf.coredump.cx/ipfrag.txt>>.

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>

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: March 5, 2020

M. McCain
FLM
M. Lee
TI
N. Welch
Google
September 2, 2019

Distributing OpenPGP Key Fingerprints with Signed Keylist Subscriptions
draft-mccain-keylist-05

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 from the correct keyserver(s), 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 a 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 March 5, 2020.

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	2
1.1. Requirements Notation	3
1.2. Terminology	3
1.3. Note to Readers	3
2. Functions and Procedures	4
2.1. Subscribing to Keylists	4
2.2. Automatic Updates	4
2.3. Cryptographic Verification of Keylists	6
3. Data Element Formats	6
3.1. Keylist	6
3.2. Signature	7
3.3. Well-Known URL	8
4. Implementation Status	8
5. Security Benefits	8
6. Relation to Other Technologies	8
6.1. Web Key Directories	9
6.2. OPENPGPKEY DNS Records	9
7. Security Considerations	9
8. IANA Considerations	9
9. References	9
9.1. Normative References	9
9.2. URIs	10
Authors' Addresses	10

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 a 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 fingerprint type SHOULD be V4).

The term "keylist" is defined as a list of OpenPGP public key fingerprints accessible via a URI in the format specified in Section 3. Keylists SHOULD be treated as public documents, however a system administrator MAY choose, for example, to restrict access to a keylist to a specific subnet or private network.

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](https://github.com/firstlookmedia/Keylist-RFC) [1].

We are still considering whether this Draft is better for the Experimental or Informational track. All feedback is appreciated.

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. If a current copy (i.e. not from local cache) cannot be obtained, the client SHOULD abort the update for this keylist and notify the user. The client SHOULD continue the update for other keylists to which it is subscribed, notwithstanding also failing the criteria described in this section.
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. If a current copy cannot be obtained, the client SHOULD abort the update and notify the user. The client SHOULD continue the update for other keylists to which it is subscribed, notwithstanding also failing the criteria described in this section.
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. The client SHOULD continue the update for other keylists to which it is subscribed, notwithstanding also failing the criteria described in this section.
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 either the key entry, the keylist (see Section 3) or, if the keylist specifies no keyserver, from the user's default keyserver. If the public key cannot be found for a particular fingerprint, the client MUST NOT abort the entire update process; instead, it SHOULD notify the user that the key retrieval failed but otherwise merely skip updating the key and continue. If the key

is already present and not revoked, refresh it from the keyserver determined in the same manner as above. If it is present and revoked, do nothing for that particular key.

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 a HKP keyserver from which the OpenPGP keys in the keylist should be retrieved. Each PGP key listed in the keylist **MAY** have a "keyserver" JSON key; if a PGP key in the keylist specifies a HKP keyserver that is different from the one described in the metadata object, the PGP key-specific keyserver should be used to retrieve that particular key (and not the key listed in the metadata object).

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" (the name of the PGP key's owner), "email" (an email of the PGP key's owner), "keyserver" (a HKP keyserver from which the key should be retrieved), and "comment" key-value pairs, as well as any other key-value pairs.

The following is an example of a valid keylist.

```
{
  "metadata": {
    "signature_uri": "https://www.example.com/keylist.json.asc",
    "comment": "This is an example of a keylist file"
  },
  "keys": [
    {
      "fingerprint": "927F419D7EC82C2F149C1BD1403C2657CD994F73",
      "name": "Micah Lee",
      "email": "micah.lee@theintercept.com",
      "comment": "Each key can have a comment"
    },
    {
      "fingerprint": "1326CB162C6921BF085F8459F3C78280DDBF52A1",
      "name": "R. Miles McCain",
      "email": "0@rmrm.io",
      "keyserver": "https://keys.openpgp.org/"
    },
    {
      "fingerprint": "E0BE0804CF04A65C1FC64CC4CAD802E066046C02",
      "name": "Nat Welch",
      "email": "nat.welch@firstlook.org"
    }
  ]
}
```

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 expected host of the well-known location for a keylist may not always be self-evident.

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:

- o The ability for new keys to be quickly distributed across an organization.
- o Removing the complexity of key distribution from end users, allowing them to focus on the content of their communications rather than on key management.
- o 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

- [RFC1928] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC 1928, DOI 10.17487/RFC1928, March 1996, <<https://www.rfc-editor.org/info/rfc1928>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/info/rfc2818>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <<https://www.rfc-editor.org/info/rfc4880>>.
- [RFC7929] Wouters, P., "DNS-Based Authentication of Named Entities (DANE) Bindings for OpenPGP", RFC 7929, DOI 10.17487/RFC7929, August 2016, <<https://www.rfc-editor.org/info/rfc7929>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

9.2. URIs

[1] <https://github.com/firstlookmedia/keylist-rfc>

Authors' Addresses

R. Miles McCain
First Look Media

Email: ietf@sendmiles.email
URI: <https://rmm.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>

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 23 July 2020

A. Rundgren
Independent
B. Jordan
Broadcom
S. Erdtman
Spotify AB
20 January 2020

JSON Canonicalization Scheme (JCS)
draft-rundgren-json-canonicalization-scheme-17

Abstract

Cryptographic operations like hashing and signing need the data to be expressed in an invariant format so that the operations are reliably repeatable. One way to address this is to create a canonical representation of the data. Canonicalization also permits data to be exchanged in its original form on the "wire" while cryptographic operations performed on the canonicalized counterpart of the data in the producer and consumer end points, generate consistent results.

This document describes the JSON Canonicalization Scheme (JCS). The JCS specification defines how to create a canonical representation of JSON data by building on the strict serialization methods for JSON primitives defined by ECMAScript, constraining JSON data to the I-JSON subset, and by using deterministic property sorting.

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 23 July 2020.

Copyright Notice

Copyright (c) 2020 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
2. Terminology	4
3. Detailed Operation	4
3.1. Creation of Input Data	4
3.2. Generation of Canonical JSON Data	5
3.2.1. Whitespace	5
3.2.2. Serialization of Primitive Data Types	5
3.2.2.1. Serialization of Literals	6
3.2.2.2. Serialization of Strings	6
3.2.2.3. Serialization of Numbers	7
3.2.3. Sorting of Object Properties	7
3.2.4. UTF-8 Generation	9
4. IANA Considerations	9
5. Security Considerations	9
6. Acknowledgements	10
7. References	10
7.1. Normative References	10
7.2. Informative References	11
Appendix A. ES6 Sample Canonicalizer	12
Appendix B. Number Serialization Samples	13
Appendix C. Canonicalized JSON as "Wire Format"	15
Appendix D. Dealing with Big Numbers	15
Appendix E. String Subtype Handling	16
E.1. Subtypes in Arrays	18
Appendix F. Implementation Guidelines	18
Appendix G. Open Source Implementations	19
Appendix H. Other JSON Canonicalization Efforts	20
Appendix I. Development Portal	20
Appendix J. Document History	20
Authors' Addresses	22

1. Introduction

This document describes the JSON Canonicalization Scheme (JCS). The JCS specification defines how to create a canonical representation of JSON [RFC8259] data by building on the strict serialization methods for JSON primitives defined by ECMAScript [ES6], constraining JSON data to the I-JSON [RFC7493] subset, and by using deterministic property sorting. The output from JCS is a "Hashable" representation of JSON data that can be used by cryptographic methods. The subsequent paragraphs outline the primary design considerations.

Cryptographic operations like hashing and signing need the data to be expressed in an invariant format so that the operations are reliably repeatable. One way to accomplish this is to convert the data into a format that has a simple and fixed representation, like Base64Url [RFC4648]. This 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, using canonicalization enables a JSON Object to remain a JSON Object even after being signed. This can simplify system design, documentation, and logging.

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

Seasoned XML developers may recall difficulties getting XML signatures to validate. This was usually due to different interpretations of the quite intricate XML canonicalization rules as well as of the equally complex Web Services security standards. The reasons why JCS should not suffer from similar issues are:

- o The absence of a namespace concept and default values.
- o Constraining data to the I-JSON [RFC7493] subset. This eliminates the need for specific parsers for dealing with canonicalization.
- o JCS compatible serialization of JSON primitives is currently supported by most Web browsers as well as by Node.js [NODEJS],
- o The full JCS specification is currently supported by multiple Open Source implementations (see Appendix G). See also Appendix F for implementation guidelines.

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. The reader is assumed to be knowledgeable in ECMAScript including the "JSON" object.

2. Terminology

Note that this document is not on the IETF standards track. However, a conformant implementation is supposed to adhere to the specified behavior for security and interoperability reasons. This text uses BCP 14 to describe that necessary behavior.

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 details related to creating a canonical JSON representation, and how they are addressed by JCS.

Appendix F describes the RECOMMENDED way of adding JCS support to existing JSON tools.

3.1. Creation of Input Data

Data to be canonically serialized is usually created by:

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

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

- o JSON Objects MUST NOT exhibit duplicate property names.
- o JSON String data MUST be expressible as Unicode [UNICODE].

- o 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, it is RECOMMENDED to represent such numbers as JSON Strings, see Appendix D for details on how this can be performed 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 the possibility of rearranging certain character sequences, referred to as "Unicode Normalization" (<https://www.unicode.org/reports/tr15/>), JCS compliant string processing does not take this in consideration. That is, all components involved in a scheme depending on JCS, MUST preserve Unicode string data "as is".

3.2. Generation of Canonical JSON Data

The following subsections describe the steps required to create 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 tokens MUST NOT be emitted.

3.2.2. Serialization of Primitive Data Types

Assume a JSON object as follows is parsed:

```
{
  "numbers": [333333333.33333329, 1E30, 4.50,
              2e-3, 0.00000000000000000000000000000001],
  "string": "\u20ac$\u000F\u000a'\u0042\u0022\u005c\\\"/\"",
  "literals": [null, true, false]
}
```

If the parsed data is subsequently serialized 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 the original data:

```
{ "numbers": [3333333333.3333333, 1e+30, 4.5, 0.002, 1e-27], "string":  
"$\u000f\nA'B\"\\\"\\\"/\", \"literals\": [null, true, false] }
```

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 in the example, numbers are subject to rounding as well.

The following subsections describe the serialization of primitive JSON data types according to JCS. This part is identical to that of ES6. In the (unlikely) event that a future version of ECMAScript would invalidate any of the following serialization methods, it will be up to the developer community to either stick to this specification or create a new specification.

3.2.2.1. Serialization of Literals

In accordance with JSON [RFC8259], the literals "null", "true", and "false" MUST be serialized as null, true, and false respectively.

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 (see section 24.3.2.2 of [ES6]):

- o 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.
- o 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: since invalid Unicode data like "lone surrogates" (e.g. U+DEAD) may lead to interoperability issues including broken signatures, occurrences of such data MUST cause a compliant JCS implementation to terminate with an appropriate error.

3.2.2.3. Serialization of Numbers

ES6 builds on the IEEE-754 [IEEE754] double precision standard for representing JSON Number data. Such 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. For implementers of JCS compliant number serialization, Google's implementation in V8 [V8] may serve as a reference. Another compatible number serialization reference implementation is Ryu [RYU], that is used by the JCS open source Java implementation mentioned in Appendix G. Appendix B holds a set of IEEE-754 sample values and their corresponding JSON serialization.

Note: since "NaN" (Not a Number) and "Infinity" are not permitted in JSON, occurrences of "NaN" or "Infinity" MUST cause a compliant JCS implementation to terminate with an appropriate error.

3.2.3. Sorting of Object Properties

Although the previous step normalized the representation of primitive JSON data types, the result would not yet 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":"$\u000f\nA'B\"\\\"\\\"/\"}
```

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

- o JSON Object properties MUST be sorted recursively, which means that JSON child Objects MUST have their properties sorted as well.
- o JSON Array data MUST also be scanned for the presence of JSON Objects (if an object is found then its properties MUST be sorted), 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:

- o The sorting process is applied to property name strings in their

"raw" (unescaped) form. That is, a newline character is treated as U+000A.

- o 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.
- o 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. In addition, JSON only supports escape sequences expressed as UTF-16 code units making knowledge and handling of such data a necessity anyway. 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.

The following test data can be used for verifying the correctness of the sorting scheme in a JCS implementation. JSON test data:

```
{  
  "\u20ac": "Euro Sign",  
  "\r": "Carriage Return",  
  "\ufb33": "Hebrew Letter Dalet With Dagesh",  
  "1": "One",  
  "\ud83d\ude00": "Emoji: Grinning Face",  
  "\u0080": "Control",  
  "\u00f6": "Latin Small Letter O With Diaeresis"  
}
```

Expected argument order after sorting property strings:

```
"Carriage Return"  
"One"  
"Control"  
"Latin Small Letter O With Diaeresis"  
"Euro Sign"  
"Emoji: Grinning Face"  
"Hebrew Letter Dalet With Dagesh"
```

Note: for the purpose of obtaining a deterministic property order, sorting on UTF-8 or UTF-32 encoded data would also work, but the outcome for JSON data like above would differ and thus be incompatible with this specification. However, in practice, property names are rarely defined outside of 7-bit ASCII making it possible to sort on string data in UTF-8 or UTF-32 format without conversions to UTF-16 and still be compatible with JCS. If this is a viable option or not depends on the environment JCS is 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 2c 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 2c 22 73 74 72 69 6e 67 22 3a 22 e2 82 ac 24 5c 75 30 30  
30 66 5c 6e 41 27 42 5c 22 5c 5c 5c 5c 5c 22 2f 22 7d
```

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 crucial to perform 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 described in

Appendix F, applications MUST perform the following operations before acting upon received data:

1. Parse the JSON data and verify that it adheres to I-JSON.
2. Verify the data for correctness according to the conventions defined by the ecosystem where it is to be used. This also includes locating the property holding the signature data.
3. Verify the signature.

If any of these steps fail, the operation in progress MUST be aborted.

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, Tim Bray, Ben Campbell, Adrian Farrell, Richard Gibson, Bron Gondwana, John-Mark Gurney, John Levine, Mark Miller, Matthew Miller, Mike Jones, 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

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

- [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<https://www.rfc-editor.org/info/rfc7493>>.
- [ES6] Ecma International, "ECMAScript 2015 Language Specification", June 2015, <<https://www.ecma-international.org/ecma-262/6.0/index.html>>.
- [IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", August 2008, <<http://grouper.ieee.org/groups/754/>>.
- [UNICODE] The Unicode Consortium, "The Unicode Standard, Version 12.1.0", May 2019, <<https://www.unicode.org/versions/Unicode12.1.0/>>.

7.2. Informative References

- [RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <<https://www.rfc-editor.org/info/rfc7638>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [JSONCOMP] A. Rundgren, "'Comparable' JSON - Work in progress", <<https://tools.ietf.org/html/draft-rundgren-comparable-json-04>>.
- [V8] Google LLC, "Chrome V8 Open Source JavaScript Engine", <<https://developers.google.com/v8/>>.
- [RYU] Ulf Adams, "Ryu floating point number serializing algorithm", <<https://github.com/ulfjack/ryu>>.
- [NODEJS] "Node.js", <<https://nodejs.org>>.
- [KEYBASE] "Keybase", <https://keybase.io/docs/api/1.0/canonical_packings#json>.
- [OPENAPI] "The OpenAPI Initiative", <<https://www.openapis.org/>>.
- [XMLDSIG] W3C, "XML Signature Syntax and Processing Version 1.1", <<https://www.w3.org/TR/xmlsig-core1/>>.

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.

IEEE-754	JSON Representation	Comment
0000000000000000	0	Zero
8000000000000000	0	Minus zero
0000000000000001	5e-324	Min pos number
8000000000000001	-5e-324	Min neg number
7fefffffffffffffff	1.7976931348623157e+308	Max pos number
ffefffffffffffffffff	-1.7976931348623157e+308	Max neg number
4340000000000000	9007199254740992	Max pos int (1)
c340000000000000	-9007199254740992	Max neg int (1)
4430000000000000	295147905179352830000	~2**68 (2)

7fffffffffffffffffff		NaN	(3)
7ff0000000000000		Infinity	(3)
44b52d02c7e14af5		9.999999999999997e+22	
44b52d02c7e14af6		1e+23	
44b52d02c7e14af7		1.0000000000000001e+23	
444b1ae4d6e2ef4e		999999999999999700000	
444b1ae4d6e2ef4f		999999999999999900000	
444b1ae4d6e2ef50		1e+21	
3eb0c6f7a0b5ed8c		9.999999999999997e-7	
3eb0c6f7a0b5ed8d		0.000001	
41b3de4355555553		333333333.3333332	
41b3de4355555554		333333333.33333325	
41b3de4355555555		333333333.3333333	
41b3de4355555556		333333333.3333334	
41b3de4355555557		333333333.33333343	
becbf647612f3696		-0.0000033333333333333333	
43143ff3c1cb0959		1424953923781206.2	Round to even (4)

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) Value out range, not permitted in JSON. See Section 3.2.2.3.

- (4) This number is exactly 1424953923781206.25 but will after the "Note 2" rule mentioned in Section 3.2.2.3 be truncated and rounded to the closest even value.

For a more exhaustive validation of a JCS number serializer, you may test against a file (currently) available in the development portal (see Appendix I), containing a large set of sample values. Another option is running V8 [V8] as a live reference together with a program generating a substantial amount of random IEEE-754 values.

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:

```
{
  "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:

```
{
  "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

```
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 "BigInt" 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:

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

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

```
var object = JSON.parse(JSON_object_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:

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

Note that the "BigInt" data type is currently only natively supported by V8 [V8].

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

```
{"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 ECMAScript as shown below:

```
// "BigInt" requires the following code to become JSON serializable
BigInt.prototype.toJSON = function() {
  return this.toString();
};

// JSON parsing using a "stream" based method
var object = JSON.parse(JSON_object_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, presumably 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:

```
// 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 data types, custom parsing and serialization code may be required 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 introduced through externally supplied canonicalizer software acting as a post processor to existing JSON serializers. This arrangement also relieves the JCS implementer from having to deal with how underlying data is to be represented in JSON.

The post processor concept enables 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:

- * JavaScript: <https://www.npmjs.com/package/canonicalize>
- * Java: <https://github.com/erdtman/java-json-canonicalization>

- * Go: <https://github.com/cyberphone/json-canonicalization/tree/master/go>
- * .NET/C#: <https://github.com/cyberphone/json-canonicalization/tree/master/dotnet>
- * Python: <https://github.com/cyberphone/json-canonicalization/tree/master/python3>

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://tools.ietf.org/html/draft-staykov-hu-json-canonical-form-00>
- * <https://gibson042.github.io/canonicaljson-spec/>
- * http://wiki.laptop.org/go/Canonical_JSON

The listed efforts all build on text level JSON to JSON transformations. The primary feature of text level canonicalization is that it can be made neutral to the flavor of JSON used. However, such schemes also imply major changes to the JSON parsing process which is a likely hurdle for adoption. Albeit at the expense of certain JSON and application constraints, JCS was designed to be compatible with existing JSON tools.

Appendix I. Development Portal

The JCS specification is currently developed at:
<https://github.com/cyberphone/ietf-json-canon>.

JCS source code and extensive test data is available at:
<https://github.com/cyberphone/json-canonicalization>

Appendix J. Document History

[[This section to be removed by the RFC Editor before publication as an RFC]]

Version 00-06:

- * See IETF diff listings.

Version 07:

- * Initial conversion to XML RFC version 3.
- * Changed intended status to "Informational".
- * Added UTF-16 test data and explanations.

Version 08:

- * Updated Abstract.
- * Added a "Note 2" number serialization sample.
- * Updated Security Considerations.
- * Tried to clear up the JSON input data section.
- * Added a line about Unicode normalization.
- * Added a line about serialiation of structured data.
- * Added a missing fact about "BigInt" (V8 not ES6).

Version 09:

- * Updated initial line of Abstract and Introduction.
- * Added note about breaking ECMAScript changes.
- * Minor language nit fixes.

Version 10-12:

- * Language tweaks.

Version 13:

- * Reorganized Section 3.2.2.3.

Version 14:

- * Improved introduction + some minor changes in security considerations, aknowlegdgements, and unicode normalization.
- * Generalized data representation issues by updating Appendix F.

Version 15:

- * Minor nits, reverted the IEEE-754 table to ASCII.

- * Added a bit more meat to the IEEE-754 table.
- * Changed all <artwork> to: type="ascii-art" and removed name="".

Version 16:

- * Updated section 2 according to AD's wish.

Version 17:

- * Updated section 2 after IESG input.
- * Author affiliation update.

Authors' Addresses

Anders Rundgren
Independent
Montpellier
France

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

Bret Jordan
Broadcom
1320 Ridder Park Drive
San Jose, CA 95131
United States of America

Email: bret.jordan@broadcom.com

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

Email: erdtman@spotify.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: 11 July 2020

D. Schinazi
Google LLC
8 January 2020

The MASQUE Protocol
draft-schinazi-masque-02

Abstract

This document describes MASQUE (Multiplexed Application Substrate over QUIC Encryption). MASQUE is a framework that allows concurrently running multiple networking applications inside an HTTP/3 connection. For example, MASQUE can allow a QUIC client to negotiate proxying capability with an HTTP/3 server, and subsequently make use of this functionality while concurrently processing HTTP/3 requests and responses.

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. Discussion of this work is encouraged to happen on the MASQUE IETF mailing list masque@ietf.org (<mailto:masque@ietf.org>) or on the GitHub repository which contains the draft: <https://github.com/DavidSchinazi/masque-drafts> (<https://github.com/DavidSchinazi/masque-drafts>).

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 11 July 2020.

Copyright Notice

Copyright (c) 2020 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	2
1.1. Conventions and Definitions	3
2. MASQUE Negotiation	3
3. MASQUE Applications	3
3.1. HTTP Proxy	3
3.2. DNS over HTTPS	4
3.3. QUIC Proxying	4
3.4. UDP Proxying	4
3.5. IP Proxying	4
3.6. Service Registration	5
4. Security Considerations	5
5. IANA Considerations	5
6. References	5
6.1. Normative References	5
6.2. Informative References	6
Acknowledgments	6
Author's Address	6

1. Introduction

This document describes MASQUE (Multiplexed Application Substrate over QUIC Encryption). MASQUE is a framework that allows concurrently running multiple networking applications inside an HTTP/3 connection (see [HTTP3]). For example, MASQUE can allow a QUIC client to negotiate proxying capability with an HTTP/3 server, and subsequently make use of this functionality while concurrently processing HTTP/3 requests and responses.

MASQUE Negotiation is performed using HTTP mechanisms, but MASQUE applications can subsequently leverage QUIC [QUIC] features without using HTTP.

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. Discussion of this work is encouraged to happen on the MASQUE IETF mailing list masque@ietf.org (<mailto:masque@ietf.org>) or on the GitHub repository which contains

the draft: <https://github.com/DavidSchinazi/masque-drafts>
(<https://github.com/DavidSchinazi/masque-drafts>).

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. MASQUE Negotiation

In order to negotiate the use of the MASQUE protocol, the client starts by sending a MASQUE request in the HTTP data of an HTTP POST request to `"/.well-known/masque/initial"`. The client can use this to request specific MASQUE applications and advertise support for MASQUE extensions. The MASQUE server indicates support for MASQUE by sending an HTTP status code 200 response, and can use the data to inform the client of which MASQUE applications are now in use, and various configuration parameters.

Both the MASQUE negotiation initial request and its response carry a list of type-length-value fields. The type field is a number corresponding to a MASQUE application, and is encoded as a QUIC variable-length integer. The length field represents the length in bytes of the value field, encoded as a QUIC variable-length integer. The contents of the value field are defined by its corresponding MASQUE application. When parsing, endpoints MUST ignore unknown MASQUE applications.

3. MASQUE Applications

As soon as the server has accepted the client's MASQUE initial request, it can advertise support for MASQUE Applications, which will be multiplexed over this HTTP/3 connection.

3.1. 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. The proxy applies back-pressure to streams in both directions.

3.2. DNS over HTTPS

The client can send DNS queries using DNS over HTTPS [DOH] to the MASQUE server.

3.3. QUIC Proxying

By leveraging QUIC client connection IDs, a MASQUE server can act as a QUIC proxy while only using one UDP port. The server informs the client of a scheme for client connection IDs (for example, random of a minimum length or vended by the MASQUE server) and then the server can forward those packets to further web servers.

This mechanism can elide the connection IDs on the link between the client and MASQUE server by negotiating a mapping between `DATAGRAM_IDs` and the tuple (client connection ID, server connection ID, server IP address, server port).

Compared to UDP proxying, this mode has the advantage of only requiring one UDP port to be open on the MASQUE server, and can lower the overhead on the link between client and MASQUE server by compressing connection IDs.

3.4. 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.

3.5. 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.

3.6. Service Registration

MASQUE can be used to make a home server accessible on the wide area. The home server authenticates to the MASQUE server and registers a domain name it wishes to serve. The MASQUE server can then forward any traffic it receives for that domain name (by inspecting the TLS Server Name Indication (SNI) extension) to the home server. This received traffic is not authenticated and it allows non-modified clients to communicate with the home server without knowing it is not colocated with the MASQUE server.

To help obfuscate the home server, deployments can use Encrypted Server Name Indication [ESNI]. That will require the MASQUE server sending the cleartext SNI to the home server.

4. Security Considerations

Here be dragons. TODO: slay the dragons.

5. IANA Considerations

This document will request IANA to register the `"/.well-known/masque/"` URI (expert review) <https://www.iana.org/assignments/well-known-uris/well-known-uris.xhtml> (<https://www.iana.org/assignments/well-known-uris/well-known-uris.xhtml>).

This document will request IANA to create a new MASQUE Applications registry which governs a 62-bit space of MASQUE application types.

6. References

6.1. Normative References

- [HTTP3] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-24, 4 November 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-http-24.txt>>.
- [QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-24, 3 November 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-24.txt>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [DOH] Hoffman, P. and P. McManus, "DNS Queries over HTTPS (DoH)", RFC 8484, DOI 10.17487/RFC8484, October 2018, <<https://www.rfc-editor.org/info/rfc8484>>.

6.2. Informative References

- [ESNI] Rescorla, E., Oku, K., Sullivan, N., and C. Wood, "Encrypted Server Name Indication for TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-esni-05, 4 November 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-tls-esni-05.txt>>.

Acknowledgments

This proposal was inspired directly or indirectly by prior work from many people. The author would like to thank Nick Harper, Christian Huitema, Marcus Ihlar, Eric Kinnear, Mirja Kuehlewind, Brendan Moran, Lucas Pardue, Tommy Pauly, Zaheduzzaman Sarker, Ben Schwartz, and Christopher A. Wood for their input.

Author's Address

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

Email: dschinazi.ietf@gmail.com