Abstract

Building on already available security formats and message transports (like PGP/MIME for email), and with the intention to stay interoperable to systems widely deployed, pretty Easy privacy (pEp) describes protocols to automatize operations (key management, key discovery, private key handling including peer-to-peer synchronization of private keys and other user data across devices) that have been seen to be barriers to deployment of end-to-end secure interpersonal messaging. pEp also introduces "Trustwords" (instead of fingerprints) to verify communication peers and proposes a trust rating system to denote secure types of communications and signal the privacy level available on a per-user and per-message level. In this document, the general design choices and principles of pEp are outlined.

Status of This Memo

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

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

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

This Internet-Draft will expire on July 13, 2018.
Copyright Notice

Copyright (c) 2018 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. Terms .......................................................... 4
3. Protocol’s core design principles ................................. 4
   3.1. Compatibility ............................................... 4
   3.2. Peer-to-Peer (P2P) ......................................... 4
   3.3. User Experience (UX) ..................................... 5
4. Identities in pEp ................................................ 6
5. Key Management .................................................. 8
   5.1. Private Keys ............................................... 9
   5.2. Key Distribution ......................................... 10
   5.3. Passphrases .............................................. 11
6. Privacy Status .................................................. 11
7. Options in pEp .................................................. 12
   7.1. Option "Passive Mode" .................................... 12
   7.2. Option "Disable Protection" ............................... 12
      7.2.1. For all communications ............................... 12
      7.2.2. For some communications ............................. 12
   7.3. Option "Extra Keys" ...................................... 12
   7.4. Option "Blacklist Keys" .................................. 13
   7.5. Establishing trust between peers .......................... 13
8. Security Considerations ......................................... 13
9. Implementation Status ........................................... 13
   9.1. Introduction .............................................. 13
   9.2. Reference implementation of pEp’s core .................... 14
   9.3. Abstract Crypto API examples ............................. 15
      9.3.1. Encrypting a message .................................. 15
      9.3.2. Decrypting a message .................................. 16
      9.3.3. Obtaining common Trustwords .......................... 17
   9.4. Current software implementing pEp ........................ 18
10. Notes ............................................................ 19
11. Acknowledgements ............................................... 19
1. Introduction

[[At this stage it is not yet clear to us how many of our implementation details should be part of new RFCs and at which places we can safely refer to already existing RFCs to make clear on which RFCs we are already relying.]]

The pretty Easy privacy (pEp) protocols are propositions to the Internet community to create software for peers to automatically encrypt, anonymize (where possible, depending on the message transport used) and verify their daily written digital communications -- this is done by building upon already existing standards and tools and automatizing all steps a user would need to carry out to engage in secure end-to-end encrypted communications without depending on centralized infrastructures.

To mitigate for Man-In-The-Middle Attacks (MITM) and as the only manual step users may carry out, Trustwords as natural language representations of two peers’ fingerprints are proposed, for peers to put trust on their communication channel.

Particularly, pEp proposes to automatize key management, key discovery and also synchronization of secret key material by an in-band peer-to-peer approach.

[[The pEp initiators had to learn from the CryptoParty movement, from which the project emerged, that step-by-step guides can be helpful to a particular set of users to engage in secure end-to-end communications, but that for a much major fraction of users it would be more convenient to have the step-by-step procedures put into actual code (as such, following a protocol) and thus automatizing the initial configuration and whole usage of cryptographic tools.]]

The Privacy by Default principles that pretty Easy privacy (pEp) introduces, are in accordance with the perspective outlined in [RFC7435] to bring Opportunistic Security in the sense of "some protection most of the time", with the subtle, but important difference that when privacy is weighted against security, the choice falls to privacy, which is why in pEp data minimization is a primary goal (e.g., omitting unnecessary email headers or encrypting the subject line).
The pEp propositions are focused on written digital communications, but not limited to asynchronous (offline) types of communications like email, but can also be implemented for message transports, which support synchronous (online) communications (e.g., for peer-to-peer networks like GNUnet). pEp’s goal is to bridge the different standardized and/or widely spread communications channels, such that users can reach their peers in the most privacy-enhancing way possible using differing IRIs/URIs.

2. Terms

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].

Terms like "TOFU" or "MITM" are used as defined in [RFC4949].

- **Handshake**: The process when Alice -- e.g. in-person or via phone -- contacts Bob to verify Trustwords (or by fallback: fingerprints) is called handshake.

- **Trustwords**: A scalar-to-word representation of 16-bit numbers (0 to 65535) to natural language words. When doing a handshake, peers are shown combined Trustwords of both public keys involved to ease the comparison. [pEpTrustwords]

3. Protocol’s core design principles

3.1. Compatibility

- Be conservative (strict) in requirements for pEp implementations and how they behave between each other.

- Be liberal (accepting) in what comes in from non-pEp implementations (e.g., do not send, but support to decipher PGP/INLINE formats).

- Where pEp requires diverging from an RFC for privacy reasons (e.g., from OpenPGP propositions as defined in [RFC4880]), options SHOULD be implemented to empower the user to comply to practices already (widespread) used, either at contact level or globally.

3.2. Peer-to-Peer (P2P)

All communications and verifications in pEp implementations for pursuing secure and establishing trusted communications between peers MUST be Peer-to-Peer (P2P) in nature.
This means, there SHALL NOT be any pEp-specific central services whatsoever needed for implementers of pEp to rely on, neither for verification of peers nor for the actual encryption.

Still, implementers of pEp MAY provide options to interoperate with providers of centralized infrastructures as users MAY NOT be stopped from being able to communicate with their peers on platforms with vendor lock-in.

Trust provided by global Certificate Authorities (e.g., commercial X.509 CAs) SHALL NOT be signaled (cf. [pEpTrustRating]) as trustworthy to users of pEp (e.g., when interoperating with peers using S/MIME) by default.

3.3. User Experience (UX)

[[We are aware of the fact that usually UX requirements are not part of RFCs. However, to have massively more people engaged in secure end-to-end encryption and at the same time to avoid putting users at risk, we believe requiring certain straightforward signaling for the users to be a good idea -- in a similar way as this happens to be the case for already popular Instant Messaging services.]]

Implementers of pEp MUST take special care to not confuse users with technical terms, especially those of cryptography (e.g., "keys", "certificates" or "fingerprints") if they do not explicitly ask for them. Advanced settings MAY be available, in some cases further options MUST be available, but they SHALL NOT be unnecessarily exposed to users of pEp implementations at the first sight when using clients implementing the pEp propositions.

The authors believe widespread adoption of end-to-end cryptography is much less of an issue if the users are not hassled and visibly forced in any way to use cryptography. That is, if they can just rely on the principles of Privacy by Default.

By consequence, this means that users MUST NOT wait for cryptographic tasks (e.g., key generation or public key retrieval) to finish before being able to have their respective message clients ready to communicate. This finally means, pEp implementers MUST make sure that the ability to draft, send and receive messages is always preserved -- even if that means a message is sent out unencrypted, thus being in accordance with the Opportunistic Security approach outlined in [RFC7435].

In turn, pEp implementers MUST make sure a Privacy Status is clearly visible to the user on both contact and message level, such that
4. Identities in pEp

With pEp users MUST have the possibility to have different identities, which MUST not correlate to each other by default. On the other hand, binding of different identities MUST be supported (being for support of aliases).

[[This is the reason why in current pEp implementations for each email account a different key pair is created, which allows a user to retain different identities.]]

In particular, with pEp users MUST NOT be bound to a specific IRI/URI, but SHALL be free to choose which identity they want to expose to certain peers -- this includes support for pseudonymity and anonymity, which the authors consider to be vital for users to control their privacy.

[[It might be necessary to introduce further addressing schemes through IETF contributions or IANA registrations.]]

In the reference implementation of the pEp Engine (cf. src/pEpEngine.h), a pEp identity is defined like the following (C99):

```c
typedef struct _pEp_identity {
    char *address;              // C string with address UTF-8 encoded
    char *fpr;                  // C string with fingerprint UTF-8 encoded
    char *user_id;              // C string with user ID UTF-8 encoded
    char *username;             // C string with user name UTF-8 encoded
    PEP_comm_type comm_type;    // type of communication with this ID
    char lang[3];               // language of conversation
    // ISO 639-1 ALPHA-2, last byte is 0
    bool me;                    // if this is the local user herself/himself
    identity_flags_t flags;     // identity_flag1 | identity_flag2 | ...
} pEp_identity;
```

A relational example (in SQL) used in current pEp implementations:
CREATE TABLE pgp_keypair (  
    fpr text primary key,  
    created integer,  
    expires integer,  
    comment text,  
    flags integer default 0  
);  

CREATE TABLE person (  
    id text primary key,  
    username text not null,  
    main_key_id text    
        references pgp_keypair (fpr)  
        on delete set null,  
    lang text,  
    comment text,  
    device_group text  
);  

CREATE TABLE identity (  
    address text,  
    user_id text  
        references person (id)  
        on delete cascade,  
    main_key_id text  
        references pgp_keypair (fpr)  
        on delete set null,  
    comment text,  
    flags integer default 0,  
    primary key (address, user_id)  
);  

The public key’s fingerprint (denoted as fpr) as part of a pEp identity MUST always be the full fingerprint.

Notable differences of how terms and concepts used differ between pEp and OpenPGP:

<table>
<thead>
<tr>
<th>pEp</th>
<th>OpenPGP</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>(no concept)</td>
<td>ID for a person, i.e. a contact</td>
</tr>
<tr>
<td>username + address</td>
<td>uid</td>
<td>comparable only for email</td>
</tr>
<tr>
<td>fpr</td>
<td>fingerprint</td>
<td>used as key ID in pEp</td>
</tr>
<tr>
<td>(no concept)</td>
<td>Key ID</td>
<td>does not exist in pEp</td>
</tr>
</tbody>
</table>
5. Key Management

Key management in pEp MUST be automatized in order to achieve the goal of widespread adoption of secure communications.

A pEp implementation MUST make sure cryptographic keys for end-to-end cryptography are generated for every identity configured (or instantly upon its configuration) if no secure cryptographic setup can be found. Users SHALL NOT be stopped from communicating -- this also applies for initial situations where cryptographic keys are not generated fast enough. This process MUST be carried out in the background so the user is not stopped from communicating.

There is the pEp Trust Rating system in [pEpTrustRating] describing which kind of encryption MUST be considered reliable and is thus secure enough for usage in pEp implementations. This also applies for keys already available for the given identity. If the available keys are considered unsecure (e.g., insufficient key length), pEp implementers are REQUIRED to generate new keys for use with the respective identity.

As example for the rating of communication types, the definition of the data structure by the pEp Engine reference implementation (cf. src/pEpEngine.h) is provided:

```
typedef enum _PEP_comm_type {
    PEP_ct_unknown = 0,

    // range 0x01 to 0x09: no encryption, 0x0a to 0x0e: nothing reasonable
    PEP_ct_no_encryption = 0x01, // generic
    PEP_ct_no_encrypted_channel = 0x02,
    PEP_ct_key_not_found = 0x03,
    PEP_ct_key_expired = 0x04,
    PEP_ct_key_revoked = 0x05,
    PEP_ct_key_b0rken = 0x06,
    PEP_ct_my_key_not_included = 0x09,
    PEP_ct_security_by_obscurity = 0x0a,
    PEP_ct_b0rken_crypto = 0x0b,
    PEP_ct_key_too_short = 0x0c,

    PEP_ct_compromized = 0x0e, // known compromised connection
    PEP_ct_mistrusted = 0x0f, // known mistrusted key

    // range 0x10 to 0x3f: unconfirmed encryption
    PEP_ct_unconfirmed_encryption = 0x10, // generic
```
Internet-Draft Pretty Easy privacy (pEp): Privacy by Default January 2018

PEP_ct_OpenPGP_weak_unconfirmed = 0x11, // RSA 1024 is weak

PEP_ct_to_be_checked = 0x20, // generic
PEP_ct_SMIME_unconfirmed = 0x21,
PEP_ct_CMS_unconfirmed = 0x22,

PEP_ct_strong_but_unconfirmed = 0x30, // generic
PEP_ct_OpenPGP_unconfirmed = 0x38, // key at least 2048 bit RSA or EC
PEP_ct_OTR_unconfirmed = 0x3a,

// range 0x40 to 0x7f: unconfirmed encryption and anonymization

PEP_ct_unconfirmed_enc_anon = 0x40, // generic
PEP_ct_pEp_unconfirmed = 0x7f,

PEP_ct_confirmed = 0x80, // this bit decides if trust is confirmed

// range 0x81 to 0x8f: reserved
// range 0x90 to 0xbf: confirmed encryption

PEP_ct_confirmed_encryption = 0x90, // generic
PEP_ct_OpenPGP_weak = 0x91, // RSA 1024 is weak (unused)

PEP_ct_to_be_checked_confirmed = 0xa0, // generic
PEP_ct_SMIME = 0xa1,
PEP_ct_CMS = 0xa2,

PEP_ct_strong_encryption = 0xb0, // generic
PEP_ct_OpenPGP = 0xb8, // key at least 2048 bit RSA or EC
PEP_ct_OTR = 0xba,

// range 0xc0 to 0xff: confirmed encryption and anonymization

PEP_ct_confirmed_enc_anon = 0xc0, // generic
PEP_ct_pEp = 0xff
}
PEP_comm_type;

5.1. Private Keys

Private keys in pEp implementations MUST always be held on the end
user's device(s): pEp implementers SHALL NOT rely on private keys
stored in centralized remote locations. This also applies for key
storages where the private keys are protected with sufficiently long
passphrases. It MUST be considered a violation of pEp’s P2P design
principle to rely on centralized infrastructures. This also applies
for pEp implementations created for applications not residing on a
user’s device (e.g., web-based MUAs). In such cases, pEp
implementations MUST be done in a way the locally-held private key can neither be directly accessed nor leaked to the outside world.

[[It is particularly important that browser add-ons implementing pEp functionality do not obtain their cryptographic code from a centralized (cloud) service, as this must be considered a centralized attack vector allowing for backdoors, negatively impacting privacy.]]

As a decentralized proposition, there is a pEp Key Synchronization protocol. [pEpKeySync] It outlines how pEp implementers can distribute their private keys in a secure and trusted manner: this allows Internet users to read their messages across their different devices, when sharing a common address (e.g., the same email account).

5.2. Key Distribution

Implementers of pEp are REQUIRED to attach the identity’s public key to any outgoing message. However, this MAY be omitted if you previously received a message encrypted with the public key of the receiver.

The sender’s public key MUST be sent encrypted whenever possible, i.e. when a public key of the receiving peer is available.

If no encryption key is available for the recipient, the sender’s public key MUST be sent unencrypted. In either case, this approach ensures that message clients (e.g., MUAs which at least implement OpenPGP) do not need to have pEp implemented to see a user’s public key. Such peers thus have the chance to (automatically) import the sender’s public key.

If there is already a known public key from the sender of a message and it is still valid and not expired, new keys SHALL not be used for future communication to avoid a MITM attack unless they are signed by the previous key. Messages SHALL always be encrypted with the receiving peer’s oldest public key, as long as it is valid and not expired.

Implementers of pEp SHALL make sure that public keys attached to messages (e.g, in email) are not displayed to the user. This ensures, they do not get confused by a file they cannot potentially deal with.

Metadata (e.g., email headers) SHALL NOT be made to announce a user’s public key by the Privacy by Default principles. This must be considered unnecessary information leakage, potentially affecting privacy -- also depending on a country’s data retention laws.
Furtherly, this affects interoperability to existing users (e.g., in the OpenPGP field) which have no notion of such header fields and thus lose the ability to import any such keys distributed this way. It SHOULD, though, be supported to obtain other users’ public keys by extracting them from respective header fields, in case such approaches get widespread.

Keyservers or generally intermediate approaches to obtain a peer’s public key SHALL NOT be used by default. On the other hand, the user MAY be given the option to opt-in for remote locations to obtain keys, considering the widespread adoption of such approaches for key distribution.

Keys generated or obtained by implementations SHALL NOT be uploaded to any (intermediate) keystore locations without the user’s explicit will.

5.3. Passphrases

Passphrases to protect an user’s private key MUST be supported by pEp implementations, but SHALL NOT be enforced by default. That is, if a pEp implementation finds a suitable (i.e., secure enough) cryptographic setup, which uses passphrases, pEp implementations MUST provide a way to unlock the key. However, if a new key pair is generated for a given identity no passphrase SHALL be put in place. The authors assume that the enforcement of secure (i.e., unique and long enough) passphrases would massively reduce the users of pEp (by hassling them) -- and in turn provide little to no additional privacy for the common cases of passive monitoring being carried out by corporations and state-level actors.

6. Privacy Status

For end-users, the most important component of pEp, which MUST be made visible on a per-recipient and per-message level, is the Privacy Status.

By colors, symbols and texts a user SHALL immediately understand how private

- a communication channel with a given peer was or ought to be and
- a given message was or ought to be.

The Privacy Status in its most general form MUST be expressed with traffic lights semantics (and respective symbols and texts), whereas the three colors yellow, green and red can be applied for any peer or message -- like this immediately indicating how secure and
trustworthy (and thus private) a communication was or ought to be considered. In cases no (special) Privacy Status can be inferred for peers or messages, no color (or the gray color) MUST be shown and respective texts -- being "unknown" or "unreliable" -- MUST be shown.

The detailed Privacy Status as an end-user element of the pEp Trust Rating system with all its states and respective representations to be followed is outlined in [pEpTrustRating].

7. Options in pEp

[[Just a selection; not yet complete.]]

7.1. Option "Passive Mode"

For situations where it might not be desirable to attach the sender’s public key for outgoing messages (which is the default), a "Passive Mode" option MUST be made available to avoid this.

7.2. Option "Disable Protection"

7.2.1. For all communications

Implementers of pEp MUST provide an option "Disable Protection" for the user’s will to disable any outgoing encryption and signing. This option SHALL not affect the user’s ability to decipher already received or sent messages.

7.2.2. For some communications

For users to disable protection for some situations, i.e. at contact or message level, pEp implementers MUST provide an option. This allows users to disable outgoing encryption and signing for peers or individual messages.

7.3. Option "Extra Keys"

For environments where there is the need to send messages to further locations, pEp implementers MAY provide an "Extra Keys" option where further recipients (by public key) can be specified. With the user’s will, each outgoing message MUST then be sent encrypted to any of those extra (implicit) recipients. Message clients SHOULD save and show only one message as sent to the explicit recipient(s), so as to not confuse users.
7.4. Option "Blacklist Keys"

An option "Blacklist Keys" MUST be provided for an advanced user to be able to disable keys which the user does not want to be used anymore for any new communications. However, the keys SHALL NOT be deleted. It MUST still be possible to verify and decipher past communications.

7.5. Establishing trust between peers

In pEp, Trustwords [pEpTrustwords] are used for users to compare the authenticity of peers in order to mitigate for MITM attacks.

By default, Trustwords MUST be used to represent two peers’ fingerprints of their public keys in pEp implementations.

In order to retain compatibility with peers not using pEp implementations (e.g., Mail User Agents (MUAs) with OpenPGP implementations without Trustwords), it is REQUIRED that pEp implementers give the user the choice to show both peers’ fingerprints instead of just their common Trustwords.

8. Security Considerations

By attaching the sender’s public key to outgoing messages, Trust on First Use (TOFU) is established, which can lead for MITM attacks to succeed. Cryptographic key subversion is considered Pervasive Monitoring (PM) according to [RFC7258]. Those attacks can be mitigated by having the involved users comparing their common Trustwords. This possibility MUST be made easily accessible to pEp users in the user interface implementation. If for compatibility reasons (e.g., with OpenPGP users) no Trustwords can be used, then an comparably easy way to verify the respective public key fingerprints MUST be implemented.

Devices themselves SHOULD be made encrypted, as the use of passphrases for private keys is not advised.

9. Implementation Status

9.1. Introduction

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation
here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

9.2. Reference implementation of pEp’s core

The pEp Foundation provides a reference implementation of pEp’s core principles and functionalities, which go beyond the documentation status of this Internet-Draft. [pEpCore]

pEp’s reference implementation is composed of pEp Engine and pEp Adapters (or bindings), alongside with some libraries which pEp Engine relies on to function on certain platforms (like a NetPGP fork we maintain for the iOS platform).

The pEp engine is a Free Software library encapsulating implementations of:

- Key Management
  - Key Management in pEp engine is based on GnuPG key chains (NetPGP on iOS). Keys are stored in an OpenPGP compatible format and can be used for different crypto implementations.

- Trust Rating
  - pEp engine is sporting a two phase trust rating system. In phase one there is a rating based on channel, crypto and key security named "comm_types". In phase 2 these are mapped to user representable values which have attached colors to present them in traffic light semantics.

- Abstract Crypto API
  - The Abstract Crypto API is providing functions to encrypt and decrypt data or full messages without requiring an application programmer to understand the different formats and standards.
Message Transports

pEp engine will support a growing list of Message Transports to support any widespread text messaging system including email, SMS, XMPP and many more.

pEp engine is written in C99. It is not meant to be used in application code directly. Instead, pEp engine is coming together with a list of software adapters for a variety of programming languages and development environments, which are:

- pEp COM Server Adapter
- pEp JNI Adapter
- pEp JSON Adapter
- pEp ObjC (and Swift) Adapter
- pEp Python Adapter
- pEp Qt Adapter

9.3. Abstract Crypto API examples

[[Just a selection; more functionality available.]]

The following code excerpts are from the pEp Engine reference implementation, to be found in src/message_api.h.

9.3.1. Encrypting a message
// encrypt_message() - encrypt message in memory
//
// parameters:
//      session (in)     session handle
//      src (in)         message to encrypt
//      extra (in)       extra keys for encryption
//      dst (out)        pointer to new encrypted message or NULL on failure
//      enc_format (in)  encrypted format
//      flags (in)       flags to set special encryption features

// return value:
//      PEP_STATUS_OK           on success
//      PEP_KEY_NOT_FOUND       at least one of the receipient keys could not be found
//      PEP_KEY_HAS_AMBIG_NAME  at least one of the receipient keys has an ambiguous name
//      PEP_GET_KEY_FAILED      cannot retrieve key
//      PEP_UNENCRYPTED         no recipients with usable key,
//                              message is left unencrypted,
//                              and key is attached to it

// caveat:
//      the ownershop of src remains with the caller
//      the ownership of dst goes to the caller

DYNAMIC_API PEP_STATUS encrypt_message(
    PEP_SESSION session,
    message *src,
    stringlist_t *extra,
    message **dst,
    PEP_enc_format enc_format,
    PEP_encrypt_flags_t flags
);

9.3.2. Decrypting a message
// decrypt_message() - decrypt message in memory
//
//  parameters:
//      session (in)   session handle
//      src (in)       message to decrypt
//      dst (out)      pointer to new decrypted message or NULL on failure
//      keylist (out)  stringlist with keyids
//      rating (out)   rating for the message
//      flags (out)    flags to signal special decryption features
//
//  return value:
//      error status
//      or PEP_DECRYPTED if message decrypted but not verified
//      or PEP_STATUS_OK on success
//
// caveat:
//      the ownership of src remains with the caller
//      the ownership of dst goes to the caller
//      the ownership of keylist goes to the caller
//      if src is unencrypted this function returns PEP_UNENCRYPTED and sets
//      dst to NULL
DYNAMIC_API PEP_STATUS decrypt_message(
    PEP_SESSION session,
    message *src,
    message **dst,
    stringlist_t **keylist,
    PEP_rating *rating,
    PEP_decrypt_flags_t *flags
);

9.3.3. Obtaining common Trustwords
// get_trustwords() - get full trustwords string for a *pair* of identities
//
// parameters:
//    session (in)  session handle
//    id1 (in)      identity of first party in communication - fpr can’t be NULL
//    id2 (in)      identity of second party in communication - fpr can’t be NULL
//    lang (in)     C string with ISO 639-1 language code
//    words (out)   pointer to C string with all trustwords UTF-8 encoded, separated by a blank each
//    wsize (out)   length of full trustwords string
//    full (in)     if true, generate ALL trustwords for these identities. else, generate a fixed-size subset. (TODO: fixed-minum entropy)

// return value:
//    PEP_STATUS_OK            trustwords retrieved
//    PEP_OUT_OF_MEMORY        out of memory
//    PEP_TRUSTWORD_NOT_FOUND  at least one trustword not found

// caveat:
//    the word pointer goes to the ownership of the caller
//    the caller is responsible to free() it (on Windoze use pEp_free())

DYNAMIC_API PEP_STATUS get_trustwords(
    PEP_SESSION session, const pEp_identity* id1, const pEp_identity* id2,
    const char* lang, char **words, size_t *wsize, bool full
);

9.4. Current software implementing pEp

The following software implementing the pEp protocols (to varying degrees) already exists; it does not yet go beyond implementing pEp for email, which is to be described nearer in [pEpEmail]:

- pEp for Outlook as addon for Microsoft Outlook, production [pEpForOutlookSrc]
- pEp for Android (based on a fork of the K9 MUA), beta [pEpForAndroidSrc]
- Enigmail/pEp as addon for Mozilla Thunderbird, early beta [EnigmailpEpSrc]
- pEp for iOS (implemented in a new MUA), alpha [pEpForiOSSrc]
pEp for Android, iOS and Outlook are provided by pEp Security, a
commercial entity specializing in end-user software implementing pEp
while Enigmail/pEp is pursued as community project, supported by the
pEp Foundation.

10. Notes

The pEp logo and "pretty Easy privacy" are registered trademarks
owned by pEp Foundation in Switzerland, a tax-free, non-commercial
entity.

Primarily, we want to ensure the following:

- Software using the trademarks MUST be backdoor-free.
- Software using the trademarks MUST be accompanied by a serious
  (detailed) code audit carried out by a reputable third-party, for
  any proper release.

The pEp Foundation will help to support any community-run (non-commercial)
project with the latter, be it organizationally or financially.

Through this, the foundation wants to make sure that software using
the pEp trademarks is as safe as possible from a security and privacy
point of view.

11. Acknowledgements

Special thanks to Bernie Hoeneisen, Enrico Tomae, Stephen Farrel,
Brian Trammell and Neal Walfield for roughly reviewing first versions
of this Internet-Draft and providing valuable feedback and patches.

[[Much more general acknowledgements to follow.]]

12. References

12.1. 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,

Thayer, "OpenPGP Message Format", RFC 4880,
DOI 10.17487/RFC4880, November 2007,

Birk, et al. Expires July 13, 2018
Internet-Draft pretty Easy privacy (pEp): Privacy by Default January 2018


12.2. Informative References


[pEpKeySync]

[pEpTrustRating]

[pEpTrustwords]

Authors’ Addresses

Volker Birk
pEp Foundation
Email: vb@pep-project.org

Hernani Marques
pEp Foundation
Email: hernani.marques@pep.foundation

Shelburn
pEp Foundation
Email: shelburn@pep.foundation

Sandro Koechli
pEp Security
Email: sandro@pep-security.net
Abstract

Building on already available security formats and message transports (like PGP/MIME for email), and with the intention to stay interoperable to systems widely deployed, pretty Easy privacy (pEp) describes protocols to automatize operations (key management, key discovery, private key handling including peer-to-peer synchronization of private keys and other user data across devices) that have been seen to be barriers to deployment of end-to-end secure interpersonal messaging. pEp also relies on "Trustwords" (as a word mapping of of fingerprints) to verify communication peers and proposes a trust rating system to denote secure types of communications and signal the privacy level available on a per-user and per-message level. In this document, the general design choices and principles of pEp are outlined.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 29, 2018.

Copyright Notice

Copyright (c) 2018 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. Terms .................................................................. 4
3. Protocol’s Core Design Principles ............................... 4
  3.1. Privacy by Default ........................................... 4
  3.2. Interoperability .............................................. 5
  3.3. Peer-to-Peer (P2P) .......................................... 5
  3.4. User Experience (UX) ....................................... 6
4. pEp identity system ................................................ 6
  4.1. Terms ................................................................ 7
  4.1.1. Key .......................................................... 7
  4.1.2. User .......................................................... 7
  4.1.3. Address ..................................................... 7
  4.1.4. Identity ..................................................... 7
  4.2. Example: Difference between pEp and OpenPGP .......... 8
5. Key Management ..................................................... 8
  5.1. Private Keys ................................................... 9
  5.2. Public Key Distribution ..................................... 9
  5.3. Passphrases ................................................... 10
6. Privacy Status ........................................................ 10
7. Options in pEp ....................................................... 11
  7.1. Option "Passive Mode" ...................................... 11
  7.2. Option "Disable Protection" ................................. 11
  7.2.1. For all communications ................................ 11
  7.2.2. For some communications ............................... 11
  7.3. Option "Extra Keys" ......................................... 12
  7.4. Option "Blacklist Keys" ..................................... 12
  7.5. Establishing trust between peers ......................... 12
8. Security Considerations ............................................ 12
9. Implementation Status ............................................. 13
  9.1. Introduction .................................................. 13
  9.2. Reference implementation of pEp’s core .................. 13
  9.3. Abstract Crypto API examples ............................. 14
  9.4. Current software implementing pEp ....................... 15
10. Notes ................................................................ 15
11. Acknowledgments .................................................. 16
12. References ........................................................ 16
1. Introduction

The pretty Easy privacy (pEp) protocols are propositions to the Internet community to create software for peers to automatically encrypt, anonymize (where possible, depending on the message transport used) and verify their daily written digital communications - this is done by building upon already existing standards and tools and automatizing all steps a user would need to carry out to engage in secure end-to-end encrypted communications without depending on centralized infrastructures.

Particularly, pEp proposes to automatize key management, key discovery and also synchronization of secret key material by an in-band peer-to-peer approach.

To mitigate Man-In-The-Middle Attacks (MITM) and as the only manual step users may carry out, Trustwords [I-D.birk-pep-trustwords] as natural language representations of two peers’ fingerprints are proposed, for peers to put trust on their communication channel.

[[ Note: The pEp initiators had to learn from the CryptoParty movement, from which the project emerged, that step-by-step guides can be helpful to a particular set of users to engage in secure end-to-end communications, but that for a much major fraction of users it would be more convenient to have the step-by-step procedures put into actual code (as such, following a protocol) and thus automatizing the initial configuration and whole usage of cryptographic tools.]]

The Privacy by Default principles that pretty Easy privacy (pEp) introduces, are in accordance with the perspective outlined in [RFC7435] to bring Opportunistic Security in the sense of "some protection most of the time", with the subtle, but important difference that when privacy is weighted against security, the choice falls to privacy. Therefore, data minimization is a primary goal in
pEp (e.g., omitting unnecessary email headers and encrypting the subject line).

The pEp propositions are focused on (but not limited to) written digital communications and cover asynchronous (offline) types of communications like email as well as synchronous (online) types such as chat.

pEp’s goal is to bridge the different standardized and/or widely spread communications channels, such that users can reach their peers in the most privacy-enhancing way possible.

[[At this stage it is not clear to us how many of our implementation details should be part of new RFCs and at which places we can safely refer to already existing RFCs to make clear on which RFCs we are already relying.]]

2. Terms

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].

- **Handshake**: The process when Alice — e.g. in-person or via phone — contacts Bob to verify Trustwords (or by fallback: fingerprints) is called handshake. [E-D.birk-pep-handshake]

- **Trustwords**: A scalar-to-word representation of 16-bit numbers (0 to 65535) to natural language words. When doing a handshake, peers are shown combined Trustwords of both public keys involved to ease the comparison. [I-D.birk-pep-trustwords]

- **Trust on First Use (TOFU)**: cf. [RFC7435]

- **Man-in-the-middle attack (MITM)**: cf. [RFC4949]

3. Protocol’s Core Design Principles

3.1. Privacy by Default

The pEp protocols are about to ensure privacy. However, there are cases where privacy and security are contradicting, e.g., in PGP’s Web of Trust, because relations between people and trust levels are leaked. Also query privacy is not ensured in such a model when obtaining keys from remote locations. Whenever security and privacy fit together, highest security and privacy is to be reached. However, where they contradict each other, privacy is always to be
chosen over security. Though, users SHOULD have the choice to
override the default by corresponding options.

In pEp messaging (e.g., when using HTML) content SHALL NOT be
obtained from remote locations as this constitutes a privacy breach.

3.2. Interoperability

The pEp propositions seek to be interoperable with message formats
and cryptography already widespread. Seamless communication between
users of software, which implements pEp and other messaging tools for
e nd-to-end encryption, is a design goal.

Therefore:

o Be conservative (strict) in requirements for pEp implementations
  and how they behave between each other.

o Be liberal (accepting) in what comes in from non-pEp
  implementations (e.g., do not send, but support to decipher PGP/
  INLINE formats).

o Where pEp requires diverging from an RFC for privacy reasons
  (e.g., from OpenPGP propositions as defined in [RFC4880], options
  SHOULD be implemented to empower the user to comply to practices
  already (widespreadly) used (either at contact level or globally).

3.3. Peer-to-Peer (P2P)

Messaging and verification processes in pEp are designed to work
Peer-to-Peer (P2P) without intermediaries in between.

This means, there MUST NOT be any pEp-specific central services
whatsoever needed for implementers of pEp to rely on, neither for
verification of peers nor for the actual encryption.

Still, implementers of pEp MAY provide options to interoperate with
providers of centralized infrastructures (e.g., to enable users to
communicate with their peers on platforms with vendor lock-in).

Trust provided by global Certificate Authorities (e.g., commercial
X.509 CAs) SHALL NOT be signaled as trustworthy (cf.
[E=D.birk-pep-trust-rating]) to users of pEp (e.g., when
interoperating with peers using S/MIME) by default.
3.4. User Experience (UX)

Implementers of pEp MUST take special care not to confuse users with technical terms, especially those of cryptography (e.g., "keys", "certificates" or "fingerprints"), unless users explicitly ask for such terms; i.e., advanced settings MAY be available, in some cases further options may even be required. However, those SHALL NOT be unnecessarily exposed to users of pEp implementations at the first sight.

The authors believe widespread adoption of end-to-end cryptography is much less of an issue, if the users are not hassled and visibly forced in any way to use cryptography, i.e., a goal of pEp is that users can just rely on the principles of Privacy by Default.

By consequence, this means that users must not wait for cryptographic tasks (e.g., key generation or public key retrieval) to finish, before being able to have their respective message clients ready to communicate. This finally means, pEp implementers MUST ensure that the ability to draft, send and receive messages is always preserved - even if that means a message is sent out unencrypted, thus being in accordance with the Opportunistic Security approach outlined in [RFC7435].

In turn, pEp implementers MUST ensure a Privacy Status is clearly visible to the user - on contact as well as on message level - so that users easily understand, which level of privacy messages are about to be sent or were received with, respectively.

[[Note: We are aware of the fact that usually UX requirements are not part of RFCs. However, to have massively more people engaged in secure end-to-end encryption and at the same time to avoid putting users at risk, we believe requiring certain straightforward signaling for the users to be a good idea - in a similar way as this happens to be the case for already popular Instant Messaging services.]]

4. pEp identity system

In pEp, users MUST have the possibility to have different identities.

pEp users MUST have the option to choose different identities. This allows an Internet user to decide how to reveal oneself to the world and is an important element to achieve privacy.

The different identities MUST NOT correlate with other by default. On the other hand, combining different identities MUST be supported (to support aliases).
4.1. Terms

4.1.1. Key

A key is an OpenPGP compatible asymmetric key pair. Other formats and temporary symmetrical keys can be generated by Key Mapping.

Keys in pEp are identified by the full fingerprint (fpr) of its public key.

4.1.2. User

A user is a real world human being or a group of human beings. If it is a single human being, it can be called person.

A user is identified by a user ID (user_id). The user_id SHOULD be a UUID, it MAY be an arbitrary unique string.

The own user can have a user_id like all other users. If it doesn’t, then it has PEP_OWN_USERID "pEpOwnUserId" as user_id.

A user can have a default key.

4.1.3. Address

An address is a network address, e.g., an SMTP address or another URI.

[[ Note: It might be necessary to introduce further addressing schemes through IETF contributions or IANA registrations, e.g., implementing pEp to bridge to popular messaging services with no URIs defined. ]]

4.1.4. Identity

An identity is a (possibly pseudonymous) representation of a user encapsulating how this user appears in the network.

An identity is defined by the mapping of user_id to address. If no user_id is known, it is guessed by mapping of username and address.

An identity can have a temporary user_id as a placeholder until a real user_id is known.

An identity can have a default key.
In current pEp implementations for each email account a different key pair is created, which allows a user to retain different identities.

In Appendix A.1 you can find how a pEp identity is defined in the reference implementation of the pEp Engine.

### 4.2. Example: Difference between pEp and OpenPGP

<table>
<thead>
<tr>
<th>pEp</th>
<th>OpenPGP</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>(no concept)</td>
<td>ID for a person, i.e. a contact</td>
</tr>
<tr>
<td>username + address</td>
<td>uid</td>
<td>comparable only for email</td>
</tr>
<tr>
<td>fpr</td>
<td>fingerprint</td>
<td>used as key ID in pEp</td>
</tr>
<tr>
<td>(no concept)</td>
<td>Key ID</td>
<td>does not exist in pEp</td>
</tr>
</tbody>
</table>

### 5. Key Management

In order to achieve the goal of widespread adoption of secure communications, key management in pEp MUST be automatized.

A pEp implementation MUST ensure cryptographic keys for end-to-end cryptography are generated for every identity configured (or instantly upon its configuration), if no secure cryptographic setup can be found. Users SHALL NOT be stopped from communicating - this also applies for initial situations where cryptographic keys are not generated fast enough. This process MUST be carried out in the background so the user is not stopped from communicating.

pEp includes a Trust Rating system defining what kind of encryption is considered reliable and is thus secure enough for usage in pEp implementations. This also applies for keys already available for the given identity. If the available keys are considered insecure (e.g., insufficient key length), pEp implementers are REQUIRED to generate new keys for use with the respective identity.

An example for the rating of communication types, the definition of the data structure by the pEp Engine reference implementation is provided in Appendix A.2.
5.1. Private Keys

Private keys in pEp implementations MUST always be held on the end user’s device(s): pEp implementers MUST NOT rely on private keys stored in centralized remote locations. This applies even for key storages where the private keys are protected with sufficiently long passphrases. It MUST be considered a violation of pEp’s P2P design principle to rely on centralized infrastructures. This also applies for pEp implementations created for applications not residing on a user’s device (e.g., web-based MUAs). In such cases, pEp implementations MUST be done in a way the locally-held private key can neither be directly accessed nor leaked to the outside world.

[[ Note: It is particularly important that browser add-ons implementing pEp functionality do not obtain their cryptographic code from a centralized (cloud) service, as this must be considered a centralized attack vector allowing for backdoors, negatively impacting privacy. ]]

A decentralized proposition - the pEp Key Synchronization protocol. [E-D.birk-pep-keysync] - defines how pEp users can distribute their private keys among different devices in a secure and trusted manner: this allows Internet users to read their messages across their different devices, when sharing a common address (e.g., the same email account).

5.2. Public Key Distribution

Implementers of pEp are REQUIRED to ensure that the identity’s public key is attached to every outgoing message. However, this MAY be omitted if the peer previously received a message encrypted with the public key of the sender.

The sender’s public key SHOULD be sent encrypted whenever possible, i.e. when a public key of the receiving peer is available. If no encryption key of the recipient is available, the sender’s public key MAY be sent unencrypted. In either case, this approach ensures that messaging clients (e.g., MUAs that at least implement OpenPGP) do not need to have pEp implemented to see a user’s public key. Such peers thus have the chance to (automatically) import the sender’s public key.

If there is already a known public key from the sender of a message and it is still valid and not expired, new keys MUST not be used for future communication, unless they are signed by the previous key (to avoid a MITM attack). Messages MUST always be encrypted with the receiving peer’s oldest public key, as long as it is valid and not expired.
Implementers of pEp SHALL prevent that public keys attached to messages (e.g., in email) are displayed to the user, in order to avoid users getting confused by a file they cannot potentially deal with.

Metadata (e.g., email headers) MUST NOT be added to announce a user’s public key. This is considered unnecessary information leakage as it may affect users’ privacy, which depends also on a country’s data retention laws. Furthermore, this affects interoperability to existing users (e.g., in the OpenPGP field) that have no notion of such header fields and thus lose the ability to import any such keys distributed this way. It SHOULD, though, be supported to obtain other users’ public keys by extracting them from respective header fields of received messages (in case such approaches get widespread).

Keyservers or generally intermediate approaches to obtain a peer’s public key SHALL NOT be used by default. On the other hand, the user MAY be provided with the option to opt-in for remote locations to obtain keys, considering the widespread adoption of such approaches for key distribution.

Keys generated or obtained by pEp clients SHALL NOT be uploaded to any (intermediate) keystore locations without the user’s explicit consent.

5.3. Passphrases

Passphrases to protect a user’s private key MUST be supported by pEp implementations, but SHALL NOT be enforced by default. That is, if a pEp implementation finds a suitable (i.e., secure enough) cryptographic setup, which uses passphrases, pEp implementations MUST provide a way to unlock the key. However, if a new key pair is generated for a given identity no passphrase SHALL be put in place. The authors assume that the enforcement of secure (i.e., unique and long enough) passphrases would massively reduce the number of pEp users (by hassling them), while providing little to no additional privacy for the common cases of passive monitoring being carried out by corporations or state-level actors.

6. Privacy Status

For end-users, the most important component of pEp, which MUST be made visible on a per-recipient and per-message level, is the Privacy Status.

By colors, symbols and texts a user SHALL immediately understand how private

o a communication channel with a given peer was or ought to be and
o a given message was or ought to be.

The Privacy Status in its most general form MUST be expressed with traffic lights semantics (and respective symbols and texts), whereas the three colors yellow, green and red can be applied for any peer or message - like this immediately indicating how secure and trustworthy (and thus private) a communication was or ought to be considered. In cases no (special) Privacy Status can be inferred for peers or messages, no color (or the gray color) MUST be shown and respective texts - being "unknown" or "unreliable" - MUST be shown.

The detailed Privacy Status as an end-user element of the pEp Trust Rating system with all its states and respective representations to be followed is outlined in [E-D.birk-pep-trust-rating].

7. Options in pEp

In this section a non-exhaustive selection of options is provided.

7.1. Option "Passive Mode"

By default the sender attaches its public key to any outgoing message (cf. Section 5.2). For situations where a sender wants to ensure that it only attaches a public key to an Internet user which has a pEp implementation, a Passive Mode MUST be available.

7.2. Option "Disable Protection"

This option SHALL not affect the user’s ability to decipher already received or sent messages. [[ TODO: Public key added in these cases? ]]

Protection can be disabled generally or selectively.

7.2.1. For all communications

Implementers of pEp MUST provide an option "Disable Protection" for the user’s choice to disable any outgoing encryption and signing.

7.2.2. For some communications

Implementers of pEp MUST provide an option to allow users to disable protection (encryption and signing) for specific contacts or messages.
7.3. Option "Extra Keys"

For internal environments there may be a need to centrally decrypt persons’ messages for archiving or other legal purposes (e.g., in the contexts of public offices and enterprises) by authorized personnel. Therefore, pEp implementers MAY provide an "Extra Keys" option where a message gets encrypted with at least one additional public key. The corresponding (shared) secret(s) to decrypt are intended to be held - safely - by CISO staff and/or other authorized personnel for such an organization. [[ TODO: Shared secret? no private key? ]]

The Extra Keys feature MUST NOT be activated by default for any network address and is intended to be an option only for organizational identities and their corresponding network addresses and accounts - not for addresses used for private purposes. That is, the Extra Keys feature is a feature which SHOULD NOT apply to all identities a user might possess, even if activated.

7.4. Option "Blacklist Keys"

An option "Blacklist Keys" MUST be provided for an advanced user to be able to disable keys which the user does not want to be used anymore for any new communications. However, the keys SHALL NOT be deleted. It MUST still be possible to verify and decipher past communications.

7.5. Establishing trust between peers

In pEp, Trustwords [I-D.birk-pep-trustwords] are used for users to compare the authenticity of peers in order to mitigate MITM attacks.

By default, Trustwords MUST be used to represent two peers’ fingerprints of their public keys in pEp implementations.

In order to retain compatibility with peers not using pEp implementations (e.g., Mail User Agents (MUAs) with OpenPGP implementations without Trustwords), it is REQUIRED that pEp implementers give the user the choice to show both peers’ fingerprints instead of just their common Trustwords.

8. Security Considerations

By attaching the sender’s public key to outgoing messages, Trust on First Use (TOFU) is established. However, this is prone to MITM attacks. Cryptographic key subversion is considered Pervasive Monitoring (PM) according to [RFC7258]. Those attacks can be mitigated, if the involved users compare their common Trustwords. This possibility MUST be made easily accessible to pEp users in the
user interface implementation. If for compatibility reasons (e.g., with OpenPGP users) no Trustwords can be used, then an comparatively easy way to verify the respective public key fingerprints MUST be implemented.

As the use of passphrases for private keys is not advised, devices themselves SHOULD use encryption.

9. Implementation Status

9.1. Introduction

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "[...] this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit."

9.2. Reference implementation of pEp’s core

The pEp Foundation provides a reference implementation of pEp’s core principles and functionalities, which go beyond the documentation status of this Internet-Draft. [SRC.pepcore]

pEp’s reference implementation is composed of pEp Engine and pEp Adapters (or bindings), alongside with some libraries which pEp Engine relies on to function on certain platforms (like a NetPGP fork we maintain for the iOS platform).

The pEp engine is a Free Software library encapsulating implementations of:

- Key Management
Key Management in pEp engine is based on GnuPG key chains (NetPGP on iOS). Keys are stored in an OpenPGP compatible format and can be used for different crypto implementations.

- **Trust Rating**

  pEp engine is sporting a two phase trust rating system. In phase one there is a rating based on channel, crypto and key security named "comm_types". In phase 2 these are mapped to user representable values which have attached colors to present them in traffic light semantics.

- **Abstract Crypto API**

  The Abstract Crypto API is providing functions to encrypt and decrypt data or full messages without requiring an application programmer to understand the different formats and standards.

- **Message Transports**

  pEp engine will support a growing list of Message Transports to support any widespread text messaging system including email, SMS, XMPP and many more.

  pEp engine is written in C99 programming language. It is not meant to be used in application code directly. Instead, pEp engine is coming together with a list of software adapters for a variety of programming languages and development environments, which are:

  - pEp COM Server Adapter
  - pEp JNI Adapter
  - pEp JSON Adapter
  - pEp ObjC (and Swift) Adapter
  - pEp Python Adapter
  - pEp Qt Adapter

  **9.3. Abstract Crypto API examples**

  A selection of code excerpts from the pEp Engine reference implementation (encrypt message, decrypt message, and obtain trustwords) can be found in Appendix A.3.
9.4. Current software implementing pEp

The following software implementing the pEp protocols (to varying degrees) already exists; it does not yet go beyond implementing pEp for email, which is described nearer in [E-D.birk-pep-email]:

- pEp for Outlook as add-on for Microsoft Outlook, release [SRC.pepforoutlook]
- pEp for Android (based on a fork of the K9 MUA), release [SRC.pepforandroid]
- Enigmail/pEp as add-on for Mozilla Thunderbird, release [SRC.enigmailpep]
- pEp for iOS (implemented in a new MUA), beta [SRC.pepforios]

pEp for Android, iOS and Outlook are provided by pEp Security, a commercial entity specializing in end-user software implementing pEp while Enigmail/pEp is pursued as community project, supported by the pEp Foundation.

10. Notes

The pEp logo and "pretty Easy privacy" are registered trademarks owned by pEp Foundation in Switzerland, a tax-free, non-commercial entity.

Primarily, we want to ensure the following:

- Software using the trademarks MUST be backdoor-free.
- Software using the trademarks MUST be accompanied by a serious (detailed) code audit carried out by a reputable third-party, for any proper release.

The pEp Foundation will help to support any community-run (non-commercial) project with the latter, be it organizationally or financially.

Through this, the foundation wants to make sure that software using the pEp trademarks is as safe as possible from a security and privacy point of view.
11.  Acknowledgments

The authors would like to thank the following people who have
provided feedback or significant contributions to the development of
this document: Bernie Hoeneisen, Brian Trammell, Enrico Tomae, Eric
Rescorla Neal Walfield, and Stephen Farrel.

[[ TODO: Add those who commented on mailing list (on this draft) as
well as those who provided feedback in person or during BarBoFs. ]]

This work was initially created by pEp Foundation, and then reviewed
and extended with funding by the Internet Society’s Beyond the Net
Programme on standardizing pEp. [ISOC.bnet]

12.  References

12.1.  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,

FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007,

Most of the Time", RFC 7435, DOI 10.17487/RFC7435,

12.2.  Informative References

[E-D.birk-pep-email]
Birk, V. and H. Marques, "pretty Easy privacy (pEp):
Secure and Trusted Email Communication", June 2018,
<https://pep.foundation/dev/repos/internet-
drafts/file/tip/pep-email/draft-birk-pep-email-NN.txt>.

Early draft

[E-D.birk-pep-handshake]
Marques, H., "pretty Easy privacy (pEp): Contact
Authentication through Handshake", June 2018,
<https://pep.foundation/dev/repos/internet-
drafts/file/tip/pep-handshake/
draft-marques-pep-handshake-00.txt>.
Appendix A. Excerpts from the pEp Reference Implementation

This section provides excerpts of the running code from the pEp reference implementation pEp engine (C99 programming language). [[TODO: Maybe rewrite sentence a bit]]

A.1. pEp Identity

How the pEp identity is defined in the data structure (cf. src/pEpEngine.h):

```c
typedef struct _pEp_identity {
    char *address;       // C string with address UTF-8 encoded
    char *fpr;           // C string with fingerprint UTF-8 encoded
    char *user_id;       // C string with user ID UTF-8 encoded
    char *username;      // C string with user name UTF-8 encoded
    PEP_comm_type comm_type; // type of communication with this ID
    char lang[3];        // ISO 639-1 ALPHA-2, last byte is 0
    bool me;             // if this is the local user
    identity_flags_t flags; // identity_flag1 | identity_flag2
                           // | ...
} pEp_identity;
```
A.1.1. Corresponding SQL

Relational table scheme excerpts (in SQL) used in current pEp implementations, held locally for every pEp installation in a SQLite database:

```
CREATE TABLE person (  
    id text primary key,  
    username text not null,  
    main_key_id text  
    references pgp_keypair (fpr)  
    on delete set null,  
    lang text,  
    comment text,  
    device_group text,  
    is_pep_user integer default 0
);

CREATE TABLE identity (  
    address text,  
    user_id text  
    references person (id)  
    on delete cascade on update cascade,  
    main_key_id text  
    references pgp_keypair (fpr)  
    on delete set null,  
    comment text,  
    flags integer default 0,  
    is_own integer default 0,  
    timestamp integer default (datetime('now')),  
    primary key (address, user_id)
);

CREATE TABLE pgp_keypair (  
    fpr text primary key,  
    created integer,  
    expires integer,  
    comment text,  
    flags integer default 0
);
CREATE INDEX pgp_keypair_expires on pgp_keypair (  
    expires
);  
```
A.2. pEp Communication Type

In the following, an example for the rating of communication types, the definition of the data structure (cf. src/pEpEngine.h [SRC.pepcore]):

```c
typedef enum _PEP_comm_type {
    PEP_ct_unknown = 0,
    PEP_ct_no_encryption = 0x01, // generic
    PEP_ct_no_encrypted_channel = 0x02,
    PEP_ct_key_not_found = 0x03,
    PEP_ct_key_expired = 0x04,
    PEP_ct_key_revoked = 0x05,
    PEP_ct_key_b0rken = 0x06,
    PEP_ct_my_key_not_included = 0x09,
    PEP_ct_security_by_obscurity = 0x0a,
    PEP_ct_b0rken_crypto = 0x0b,
    PEP_ct_key_too_short = 0x0c,
    PEP_ct_compromised = 0x0e, // known compromised connection
    PEP_ct_mistrusted = 0x0f, // known mistrusted key
    PEP_ct_unconfirmed_encryption = 0x10, // generic
    PEP_ct_OpenPGP_weak_unconfirmed = 0x11, // RSA 1024 is weak
    PEP_ct_to_be_checked = 0x20, // generic
    PEP_ct_SMIME_unconfirmed = 0x21,
    PEP_ct_CMS_unconfirmed = 0x22,
    PEP_ct_strong_but_unconfirmed = 0x30, // generic
    PEP_ct_OpenPGP_unconfirmed = 0x38, // key at least 2048 bit
    PEP_ct_OTR_unconfirmed = 0x3a,
    PEP_ct_unconfirmed_enc_anon = 0x7f,
    PEP_ct_pEp_unconfirmed = 0x7f,
    PEP_ct_confirmed = 0x80, // this bit decides if trust
} PEP_comm_type;
```
// range 0x81 to 0x8f: reserved
// range 0x90 to 0xbf: confirmed encryption

PEP_ct_confirmed_encryption = 0x90, // generic
PEP_ct_OpenPGP_weak = 0x91, // RSA 1024 is weak (unused)

PEP_ct_to_be_checked_confirmed = 0xa0, // generic
PEP_ct_SMIME = 0xa1,
PEP_ct_CMS = 0xa2,

PEP_ct_strong_encryption = 0xb0, // generic
PEP_ct_OpenPGP = 0xb8, // key at least 2048 bit RSA or EC
PEP_ct_OTR = 0xba,

// range 0xc0 to 0xff: confirmed encryption and anonymization

PEP_ct_confirmed_enc_anon = 0xc0, // generic
PEP_ct_pEp = 0xff
} PEP_comm_type;

A.3. Abstract Crypto API examples

The following code excerpts are from the pEp Engine reference implementation, to be found in src/message_api.h.

[[ Note: Just a selection; more functionality is available. ]]

A.3.1. Encrypting a Message

Cf. src/message_api.h [SRC.pepcore]:

Birk, et al. Expires December 29, 2018
// encrypt_message() - encrypt message in memory
//
// parameters:
//  session (in)     session handle
//  src (in)         message to encrypt
//  extra (in)       extra keys for encryption
//  dst (out)        pointer to new encrypted message or NULL if no
//                   encryption could take place
//  enc_format (in)  encrypted format
//  flags (in)       flags to set special encryption features
//
// return value:
//      PEP_STATUS_OK           on success
//      PEP_KEY_HAS_AMBIG_NAME   at least one of the recipient
//                              keys has an ambiguous name
//      PEP_UNENCRYPTED         no recipients with usable key,
//                              message is left unencrypted,
//                              and key is attached to it
//
// caveat:
//  the ownership of src remains with the caller
//  the ownership of dst goes to the caller
DYNAMIC_API PEP_STATUS encrypt_message(
    PEP_SESSION session,
    message *src,
    stringlist_t *extra,
    message **dst,
    PEP_enc_format enc_format,
    PEP_encrypt_flags_t flags
);
DYNAMIC_API PEP_STATUS decrypt_message(
    PEP_SESSION session,
    message *src,
    message **dst,
    stringlist_t **keylist,
    PEP_rating *rating,
    PEP_decrypt_flags_t *flags
)
A.3.3. Obtain Common Trustwords

Cf. src/message_api.h [SRC.pepcore]:

```c
// get_trustwords() - get full trustwords string
// for a *pair* of identities
//
// parameters:
//    session (in)  session handle
//    id1 (in)      identity of first party in communication
//                   - fpr can’t be NULL
//    id2 (in)      identity of second party in communication
//                   - fpr can’t be NULL
//    lang (in)     C string with ISO 639-1 language code
//    words (out)   pointer to C string with all trustwords
//                   UTF-8 encoded, separated by a blank each
//                   NULL if language is not supported or
//                   trustword wordlist is damaged or unavailable
//    wsize (out)   length of full trustwords string
//    full (in)     if true, generate ALL trustwords for these
//                   identities.
//                   else, generate a fixed-size subset.
//                   (TODO: fixed-minimum-entropy subset
//                   in next version)
//
// return value:
//    PEP_STATUS_OK            trustwords retrieved
//    PEP_OUT_OF_MEMORY        out of memory
//    PEP_TRUSTWORD_NOT_FOUND  at least one trustword not found
//
// caveat:
//    the word pointer goes to the ownership of the caller
//    the caller is responsible to free() it
//    (on Windoze use pEp_free())
```

```c
DYNAMIC_API PEP_STATUS get_trustwords(
    PEP_SESSION session, const pEp_identity* id1,
    const pEp_identity* id2, const char* lang,
    char **words, size_t *wsize, bool full
);
```
* Move (updated) code to Appendix
* Add Changelog to Appendix
* Add Open Issue section to Appendix
* Fix description of what Extra Keys are
* Fix Passive Mode description
* Better explain pEp’s identity system
  o draft-birk-pep-01:
    * Mostly editorial
  o draft-birk-pep-00:
    * Initial version

Appendix C. Open Issues

[[ RFC Editor: This section should be empty and is to be removed before publication ]]

  o Better explain Passive Mode
  o Better explain / illustrate pEp’s identity system
  o Explain Key Mapping

Authors’ Addresses

Volker Birk
pEp Foundation
Oberer Graben 4
CH-8400 Winterthur
Switzerland

Email: volker.birk@pep.foundation
URI: https://pep.foundation/
Abstract

In public-key cryptography comparing the public keys’ fingerprints of the communication partners involved is vital to ensure that there is no man-in-the-middle (MITM) attack on the communication channel. Fingerprints normally consist of a chain of hexadecimal chars. However, comparing hexadecimal strings is often impractical for regular users and prone to misunderstandings.

To mitigate these challenges, this memo proposes the comparision of trustwords as opposed to hexadecimal strings. Trustwords are common words in a natural language (e.g., English) to which the hexadecimal strings are mapped to. This makes the verification process more natural.
1. Introduction

In public-key cryptography comparing the public keys’ fingerprints of the communication partners involved is vital to ensure that there is no man-in-the-middle (MITM) attack on the communication channel. Fingerprints normally consist of a chain of hexadecimal chars. However, comparing hexadecimal strings is often impractical for regular users and prone to misunderstandings.

To mitigate these challenges, this memo proposes the comparison of trustwords as opposed to hexadecimal strings. Trustwords are common words in a natural language (e.g., English) to which the hexadecimal strings are mapped to. This makes the verification process more natural.

Trustwords are used to achieve easy contact verification in pEp’s proposition of Privacy by Default [pEp] for end-to-end encryption situations after the peers have exchanged public keys opportunistically.

Trustwords may also be used for purposes other than contact verification.
2. Terms

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].

3. The Concept of Trustword Mapping

4. Example

A fingerprint typically looks like:

F482 E952 2F48 618B 01BC 31DC 5428 D7FA ACDC 3F13

Its mapping to trustwords looks like:

dog house brother town fat bath school banana kite task

[[Actual mapping for English should be used here and perhaps an example for another language.]]

Instead of the former hexadecimal string, users can compare ten common words of their language.

5. Previous work

The basic concept of trustword mapping has been already documented in the past, e.g. for use in One-Time Passwords (OTP) [RFC2289] or the PGP Word List ("Pretty Good Privacy word list" [PGPwordlist], also called a biometric word list, to compare fingerprints.

6. Number of Trustwords for a language

Previous proposals have the shortcoming of a limited number of trustwords and they are usually only available in English. If the number of trustwords is low, a lot of trustwords need to be compared, which make a comparison somewhat cumbersome for users, i.e. leads to degraded usability. To reduce the number of trustwords to compare, 16-bit scalars are mapped to natural language words. Therefore, the size (by number of key--value pairs) of any key--value pair structure MUST be 65536, the keys being the enumeration of the Trustwords (starting at 0) and the values being individual natural language words in the respective language.

However, the number of unique values to be used in a language may be less than 65536. This can be addressed e.g. by using the same value (trustword) for more than one key. However, the entropy of the representation is slightly reduced.
Example. A Trustwords list of just 42000 words still allows for an entropy of $\log_2(42000) \approx 15.36$ bits in 16-bit mappings.

It is for further study, what minimal number of words (or entropy) should be required.

7. The nature of the words

Every Trustwords list SHOULD be cleared from swearwords in order to not offense users. This is a task to be carried out by speakers of the respective natural language.

8. IANA Considerations

Each natural language requires a different set of trustwords. To allow implementors for identical trustword lists, a IANA registry is to be established. The IANA registration policy according to [RFC8126] will likely be "Expert Review" and "Specification Required".

An IANA registration will contain:

- language code according to ISO 639-3
- version number
- list of up to 65536 trustwords

The details of the IANA registry and requirements for the expert to assess the specification are for further study.

9. Security Considerations

There are no special security considerations.

10. Acknowledgements

This work was initially created by pEp Foundation, and then reviewed and extended with funding by the Internet Society’s Beyond the Net Programme on standardizing pEp. [bnet]

11. References


Volker Birk
pEp Foundation
Email: vb@pep-project.org

Hernani Marques
pEp Foundation
Email: hernani.marques@pep.foundation
IANA Registration of Trustword Lists: Guide, Template and IANA Considerations
draft-birk-pep-trustwords-02

Abstract

This document specifies the IANA Registration Guidelines for Trustwords, describes corresponding registration procedures, and provides a guideline for creating Trustword list specifications.

Trustwords are common words in a natural language (e.g., English) to which the hexadecimal strings are mapped to. This makes verification processes (e.g., comparison of fingerprints), more practical and less prone to misunderstandings.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 28, 2018.

Copyright Notice

Copyright (c) 2018 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
1. Introduction

In public-key cryptography comparing the public keys’ fingerprints of the communication partners involved is vital to ensure that there is no man-in-the-middle (MITM) attack on the communication channel. Fingerprints normally consist of a chain of hexadecimal chars. However, comparing hexadecimal strings is often impractical for regular human users and prone to misunderstandings.
To mitigate these challenges, several systems offer the comparison of Trustwords as an alternative to hexadecimal strings. Trustwords are common words in a natural language (e.g., English) to which the hexadecimal strings are mapped to. This makes the verification process more natural for human users.

For example, in pEp's proposition of Privacy by Default [I-D.birk-pep] Trustwords are used to achieve easy contact verification for end-to-end encryption. Trustword comparison is offered after the peers have exchanged public keys opportunistically. Examples for Trustword lists used by current pEp implementations can be found in CSV format, here: https://pep.foundation/dev/repos/pEpEngine/file/tip/db.

In addition to contact verification, Trustwords are also used for other purposes, such as Human-Readable 128-bit Keys [RFC1751], One Time Passwords (OTP) [RFC1760] [RFC2289], SSH host-key verification, VPN Server certificate verification, and to import or synchronize secret key across different devices of the same user [E-D.birk-pep-keysync]. Further ideas include to use Trustwords for contact verification in Extensible Messaging and Presence Protocol (XMPP) [RFC6120], for X.509 [RFC3647] certificate verification in browsers or in block chain applications for crypto currencies.

2. Terms

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].

- Handshake: The process when Alice - e.g. in-person or via phone - contacts Bob to verify Trustwords (or by fallback: fingerprints) is called handshake. [E-D.birk-pep-handshake]

- Man-in-the-middle attack (MITM): cf. [RFC4949]

3. The Concept of Trustword Mapping

3.1. Example

A fingerprint typically looks like:

F482 E952 2F48 618B 01BC 31DC 5428 D7FA ACDC 3F13

Its mapping to English Trustwords could look like:

dog house brother town fat bath school banana kite task
Or its mapping to German Trustwords could like like:

klima gelb lappen weg trinken alles kaputt rasen rucksack durch

Instead of the former hexadecimal string, users can compare ten common words of their language.

Note: This examples are for illustration purposes only and do not make use any any published Trustword list.

3.2. Previous work

The basic concept of Trustwords mapping has been already documented in the past, e.g. for use in One-Time Passwords (OTP) [RFC1751] [RFC1760] [RFC2289] or the PGP Word List ("Pretty Good Privacy word list" [PGP.wl], also called a biometric word list, to compare fingerprints.

Regarding today’s needs, previous proposals have the following shortcomings:

- Limited number of Trustwords (small Trustword dictionaries), which generally results in more Trustwords to be compared
- Usually only available in English language, which does not normally allow its usage by non-English speakers in a secure manner

Furthermore, there are differences in the basic concept:

- This work allows for better tailoring the target audience to ordinary human users, i.e. not technical stuff (or IT geeks) only.
- As in many usage scenarios the Trustwords are only read (and compared), but not written down nor typed in by humans, there is a less strong need to keep the Trustwords themselves short. One such scenario is to use a side channel (e.g. phone) to compare the Trustwords. In fact longer Trustwords increases increase the entropy, as the dictionary is larger and the likelihood for phonetic collision can be decreased.

3.3. Number of Trustwords for a language

If the number of Trustwords is low, a lot of Trustwords need to be compared, which make a comparison somewhat cumbersome for users. This may lead to degraded usability.
To reduce the number of Trustwords to compare, in pEp’s proposition of Privacy by Default [I-D.birk-pep] 16-bit scalars are mapped to natural language words. Therefore, the size (by number of key-value pairs) of any key-value pair structure is 65536. However, the number of unique values to be used in a language may be less than 65536. This can be addressed e.g. by using the same value (Trustword) for more than one key. In these cases, the entropy of the representation is slightly reduced. (Example: A Trustwords list of just 42000 words still allows for an entropy of $\log_2(42000) \approx 15.36$ bits in 16-bit mappings.)

On the other hand, small sized Trustword lists allow for Trustwords with shorter strings, which are easier to use in scenarios where Trustwords have to be typed in e.g. OTP applications.

The specification allows for different dictionary sizes.

3.4. Language

Although English is rather widespread around the world, the vast majority of the world’s population does not speak English. For an application to be useful for ordinary people, localization is a must. Thus, Trustword lists in different languages can be registered.

For applications where two humans establish communication it is very likely that they share a common language. So far no real use case for translations between Trustword lists in different languages has been identified. As translations also drastically increase the complexity for IANA registrations, translations of Trustwords beyond the scope of this document.

3.5. The nature of the words

Every Trustwords list SHOULD be cleared from swearwords in order to not offense users. This is a task to be carried out by speakers of the respective natural language (i.e., by native language speakers).

4. IANA Considerations

Each natural language requires a different set of Trustwords. To allow implementers for identical Trustword lists, a IANA registry is to be established. The IANA registration policy according to [RFC8126] is "Expert Review" and "Specification Required".

[[ Note: Further details of the IANA registry and requirements for the expert to assess the specification are for further study. A similar approach as used in [RFC6117] is likely followed. ]]}
4.1. Registration Template (XML chunk)

<record>
  <languagecode>
    <!-- ISO 639-3 (e.g. eng, deu, ...) -->
  </languagecode>
  <bitsize>
    <!-- How many bits can be mapped with this list (e.g. 8, 16, ...) -->
  </bitsize>
  <numberofuniquewords>
    <!-- number of unique words registered (e.g. 256, 65536, ...) -->
  </numberofuniquewords>
  <bijective>
    <!-- whether or not the list allows for a two-way-mapping (e.g. yes, no) -->
  </bijective>
  <version>
    <!-- version number within language (e.g. b.1.2, n.0.1, ...) -->
  </version>
  <registrationdocs>
    <!-- Change accordingly -->
    <xref type="rfc" data="rfc2551"/>
  </registrationdocs>
  <requesters>
    <!-- Change accordingly -->
    <xref type="person" data="John_Doe"/>
    <xref type="person" data="Jane_Dale"/>
  </requesters>
  <additionalinfo>
    <paragraph>
      <!-- Text with additional information about the Wordlist to be registered -->
    </paragraph>
    <artwork>
      <!-- There can be artwork sections, too -->
    </artwork>
  </additionalinfo>
  <wordlist>
    <!-- Change accordingly -->
    <w0>first</w0>
    <w1>second</w1>
    [...]
    <w65535>last</w65535>
  </wordlist>
</record>
Authors of a Wordlist are encouraged to use these XML chunks as a template to create the IANA Registration Template.

4.2. IANA Registration

An IANA registration will contain the following elements:

4.2.1. Language Code (<languagecode>)

The language code follows the ISO 639-3 specification [ISO693], e.g., eng, deu.

[[ Note: It is for further study, which of the ISO 639 Specifications is most suitable to address the Trustwords’ challenge. ]]

Example usage for German:

e.g. <languagecode>deu</languagecode>

4.2.2. Bit Size (<bitsize>)

The bit size is the number of bits that can be mapped with the Wordlist. The number of registered words in a word list MUST be \(2^{(<\text{bitsize})}\).

Example usage for 16-bit Wordlist:

e.g. <bitsize>16</bitsize>

4.2.3. Number Of Unique Words (<numberofuniquewords>)

The number of unique words that are registered.

e.g. <numberofuniquewords>65536</numberofuniquewords>
4.2.4. Bijectivity (<bijective>)

Whether the registered Wordlist has a one-to-one mapping, meaning the number of unique words registered equals $2^{(<\text{bitsize>})}$.

Valid content: ( yes | no )

e.g.  <bijective>yes</bijective>

4.2.5. Version (<version>)

The version of the Wordlist MUST be unique within a language code.

[[ Note: Requirements to a "smart" composition of the version number are for further study ]]

e.g.  <version>b.1.2</version>

4.2.6. Registration Document(s) (<registrationdocs>)

Reference(s) to the Document(s) containing the Wordlist

e.g.  <registrationdocs>
    <xref type="rfc" data="rfc4979"/>
  </registrationdocs>

e.g.  <registrationdocs>
    <xref type="rfc" data="rfc8888"/> (obsoleted by RFC 9999)
    <xref type="rfc" data="rfc9999"/>
  </registrationdocs>

e.g.  <registrationdocs>
  </registrationdocs>

4.2.7. Requesters (<requesters>)

The persons requesting the registration of the Wordlist. Usually these are the authors of the Wordlist.
e.g.  <requesters>
    <xref type="person" data="John_Doe"/>
  </requesters>

  <people>
    <person id="John_Doe">
      <name>John Doe</name>
      <org>Example Inc.</org>
      <uri>mailto:john.doe@example.com</uri>
      <updated>2018-06-20</updated>
    </person>
  </people>

Note: If there is more than one requester, there must be one <xref>
element per requester in the <requesters> element, and one <person>
chunk per requester in the <people> element.

4.2.8. Further Information (<additionalinfo>)

Any other information the authors deem interesting.

e.g.  <additionalinfo>
    <paragraph>more info goes here</paragraph>
  </additionalinfo>

Note: If there is no such additional information, then the
<additionalinfo> element is omitted.

4.2.9. Wordlist (<wordlist>)

The full Wordlist to be registered. The number of words MUST be a
power of 2 as specified above. The element names serve as key used
for enumeration of the Trustwords (starting at 0) and the elements
contains the values being individual natural language words in the
respective language.

e.g.  <wordlist>
  <w0>first</w0>
  <w1>second</w1>
  [...]  
  <w65535>last<w65535>
</wordlist>

} ]>

[[ Note: The exact representation of the Wordlist is for further
study. ]]
5. Security Considerations

There are no special security considerations.

6. Acknowledgements

The authors would like to thank the following people who have provided feedback or significant contributions to the development of this document: Andrew Sullivan, Claudio Luck, Daniel Kahn Gilmore, Michael Richardson, Rich Salz, and Yoav Nir.

This work was initially created by pEp Foundation, and then reviewed and extended with funding by the Internet Society’s Beyond the Net Programme on standardizing pEp. [ISOC.bnet]

7. References

7.1. Normative References


7.2. Informative References


Early draft

Early draft


Appendix A. IANA XML Template Example

This section contains a non-normative example of the IANA Registration Template XML chunk.

```xml
<record>
  <languagecode>lat</languagecode>
  <bitsize>16</bitsize>
  <numberofuniquewords>57337</numberofuniquewords>
  <bijective>no</bijective>
  <version>n.0.1</version>
  <registrationdocs>
    <xref type="rfc" data="rfc2551"/>
  </registrationdocs>
  <requesters>
    <xref type="person" data="Julius_Caesar"/>
  </requesters>
  <additionalinfo>
    <paragraph>
      This Wordlist has been optimized for the Roman Standards Process.
    </paragraph>
  </additionalinfo>
  <wordlist>
    <w0>errare</w0>
    <w1>humanum</w1>
    [...]
    <w65535>est</w65535>
  </wordlist>
</record>

<people>
  <person id="Julius_Caesar">
    <name>Julius Caesar</name>
    <org>Curia Romana</org>
    <uri>mailto: julius.cesar@example.com</uri>
    <updated>1999-12-31</updated>
  </person>
</people>
```
Appendix B. Document Changelog

[[ RFC Editor: This section is to be removed before publication ]]

- draft-birk-pep-trustwords-02:
  * Minor editorial changes and bug fixes
  * Added more items to Open Issues
  * Add usage example

- draft-birk-pep-trustwords-01:
  * Included feedback from mailing list and IETF-101 SECDISPATCH
    WG, e.g.
    + Added more explanatory text / less focused on the main use case
    + Bit size as parameter
  * Explicitly stated translations are out-of-scope for this document
  * Added draft IANA XML Registration template, considerations, explanation and examples
  * Added Changelog to Appendix
  * Added Open Issue section to Appendix

Appendix C. Open Issues

[[ RFC Editor: This section should be empty and is to be removed before publication ]]

- More explanatory text for Trustword use cases, properties and requirements
- Further details of the IANA registry and requirements for the expert to assess the specification
- Decide which ISO language code either 639-1 or 639-3 to use, i.e., ISO-639-1 (e.g., ca, de, en, ...) as currently used in pEp implementations (running code) or ISO-693-3 (eng, deu, ita, ...)
- Adjust exact representation of wordlists
* e.g. XML, CSV, ...

* Syntax for non-ASCII letters or language symbols (UTF-8) in Wordlists

  o Need for optional entropy value assigned to words, to account for similar phonetics among words in the same wordlist?

  o Need for an additional field, to define what a wordlist is optimized for, e.g., "entropy", "minimize word lengths", ...?

  o Work out (requirements for) "smart" composition of the version number

  o Decide whether in non-bijective Wordlists the redundant words need to be repeated in the IANA Registration

  o Register only a hash over the wordlist with IANA?

  o Does it make sense to open registrations for other patterns than just words, e.g., images?

Authors' Addresses

Volker Birk
pEp Foundation
Oberer Graben 4
CH-8400 Winterthur
Switzerland

  Email: volker.birk@pep.foundation
  URI:   https://pep.foundation/

Hernani Marques
pEp Foundation
Oberer Graben 4
CH-8400 Winterthur
Switzerland

  Email: hernani.marques@pep.foundation
  URI:   https://pep.foundation/
New Media Stack  
draft-jennings-dispatch-new-media-00

Abstract

A sketch of a proposal for a new media stack for interactive communications.

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 6, 2018.

Copyright Notice

Copyright (c) 2018 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.
1. Introduction

This draft proposes a new media stack to replace the existing stack RTP, DTLS-SRT, and SDP Offer Answer. The key parts of this stack are connectivity layer, the transport layer, the media layer, a control API, and the signaling layer.
The connectivity layer uses a simplified version of ICE, called snowflake [I-D.jennings-dispatch-snowflake], to find connectivity between endpoints and change the connectivity from one address to another as different networks become available or disappear. It is based on ideas from [I-D.jennings-mmusic-ice-fix].

The transport layer uses QUIC to provide a hop by hop encrypted, congestion controlled transport of media. Although QUIC does not currently have all of the partial reliability mechanisms to make this work, this draft assumes that they will be added to QUIC.

The media layer uses existing codecs and packages them along with extra header information to provide information about, when the sequence needs to be played back, which camera it came from, and media streams to be synchronized.

The control API is an abstract API that provides a way for the media stack to report it capabilities and features and a way for the an application tell the media stack how it should be configured. Configuration includes what codec to use, size and frame rate of video, and where to send the media.

The signaling layer is based on an advertisement and proposal model. Each endpoint can create an advertisement that describes what it supports including things like supported codecs and maximum bitrates. A proposal can be sent to an endpoint that tells the endpoint exactly what media to send and receive and where to send it. The endpoint can accept or reject this proposal in total but cannot change any part of it.

2. Terminology

- **media stream**: Stream of information from a single sensor. For example, a video stream from a single camera. A stream may have multiple encodings for example video at different resolutions.

- **encoding**: A encoded version of a stream. A given stream may have several encodings at different resolutions. One encoding may depend on other encodings such as forward error corrections or in the case of scalable video codecs.

- **flow**: A logical transport between two computers. Many media streams can be transported over a single flow. The actually IP address and ports used to transport data in the flow may change over time as connectivity changes.

- **message**: Some data or media that to be sent across the network along with metadata about it. Similar to an RTP packet.
o media source: a camera, microphone or other source of data on an endpoint

o media sink: a speaker, screen, or other destination for data on an endpoint

o TLV: Tag Length Value. When used in the draft, the Tag, Length, and any integer values are coded as variable length integers similar to how this is done in CBOR.

3. Connectivity Layer

3.1. Snowflake – New ICE

All that is needed to discover the connectivity is way to:

o Gather some IP/ports that may work using TURN2 relay, STUN2, and local addresses.

o A controller, which might be running in the cloud, to inform a client to send a STUN2 packet from a given source IP/port to a given destination IP/port.

o The receiver notifies the controller about information on received STUN2 packets.

o The controller can tell the sender the secret that was in the packet to prove consent of the receiver to receive data then the sending client can allow media to flow over that connection.

The actually algorithm used to decide on what pairs of addresses are tested and in what order does not need to be agreed on by both the sides of the call - only the controller needs to know it. This allows the controller to use machine learning, past history, and heuristics to find an optimal connection much faster than something like ICE.

The details of this approach are described in [I-D.jennings-dispatch-snowflake].

3.2. STUN2

The speed of setting up a new media flow is often determined by how many STUN2 checks need to be done. If the STUN2 packets are smaller, then the stun checks can be done faster without risk of causing congestion. The STUN2 server and client share a secret that they use for authentication and encryption. When talking to a public STUN2 server this secret is the empty string.
3.2.1. STUN2 Request

A STUN2 request consists of the following TLVs:

- a magic number that uniquely identifies this as a STUN2 request packet with minimal risk of collision when multiplexing.
- a transaction ID that uniquely identifies this request and does not change in retransmissions of the same request.
- an optional sender secret that can be used by the receiver to prove that it received the request. In WebRTC the browser would create the secret but the JavaScript on the sending side would know the value.

The packet is encrypted by using the secret and an AEAD crypto to create a STUN2 packet where the first two fields are the magic number and transaction ID which are only authenticated followed by the rest of the fields that are authenticated and encrypted followed by the AEAD authentication data.

The STUN2 requests are transmitted with the same retransmission and congestion algorithms as STUN2 in WebRTC 1.0

3.2.2. STUN2 Response

A STUN2 response consists of the following TLVs:

- a magic number that uniquely identifies this as a STUN2 response packet with minimal risk of collision when multiplexing.
- the transaction ID from the request.
- the IP address and port the request was received from.

The packet is encrypted where the first two fields are the magic number and transaction ID which are only authenticated followed by the rest of the fields that are authenticated and encrypted followed by the AEAD authentication data.

3.3. TURN2

Out of band, the client tells the TURN2 server the fingerprint of the cert it uses to authenticate with and the TURN2 server gives the client two public IP:port address pairs. One is called inbound and other called outbound. The client connects to the outbound port and authenticates to TURN2 server using the TLS domain name of server. The TURN2 server authenticates the client using mutual TLS with
fingerprint of cert provided by the client. Any time a message or stun packet is received on the matched inbound port, the TURN2 server forwards it to the client(s) connected to the outbound port.

A single TURN2 connection can be used for multiple different calls or session at the same time and a client could choose to allocate the TURN2 connection at the time that it started up. It does not need to be done on a per session basis.

The client can not send from the TURN2 server.

<table>
<thead>
<tr>
<th>Client A (Media Receiver)</th>
<th>Turn Server</th>
<th>Client B (Media Sender)</th>
</tr>
</thead>
</table>
| (1) OnInit Register (A’s fingerprint)----------->
| (2) Register Response (Port Pair (L,R))<--------
| L(left of Server), R(Right of Server)
| (3) Setup TLS Connection (L port) ...............B send’s media to A
| (4) Media Tx (Received on Port R)<-------------
| (5) Media Tx (Sent from Port L)<--------------

Jennings
Expire September 6, 2018
4. Transport Layer

The responsibility of the transport layer is to provide an end to end crypto layer equivalent to DTLS and they must ensure adequate congestion control. The transport layer brings up a flow between two computers. This flow can be used by multiple media streams.

The MTI transport layer is QUIC with packets sent in an unreliable mode.

This is secured by checking the fingerprints of the DTLS connection match the fingerprints provided at the control layer or by checking the names of the certificates match what was provided at control layer.

The transport layer needs to be able to set the DSCP values in transmitting packets as specified by the control layer.

The transport MAY provide a compression mode to remove the redundancy of the non-encrypted portion of the media messages such as GlobalEncodingID. For example, a GlobalEncodingID could be mapped to a QUIC channel and then it could be removed before sending the message and added back on the receiving side.

The transport need to be able to ensure that it has a very small chance of being confused with the STUN2 traffic it will be multiplexed with.

5. Media Layer - RTP3

Each message consist of a set of TLV headers with metadata about the packet, followed by payload data such as the output of audio or video codec.

There are several message headers that help the receiver understand what to do with the media. The TLV header are the follow:

- Conference ID: Integer that will be globally unique identifier for the for all applications using a common call signaling system. This is set by the proposal.

- Endpoint ID: Integer to uniquely identify the endpoint with within scope of conference ID. This is set by the proposal.

- Source ID: Integer to uniquely identify the input source within the scope a endpoint ID. A source could be a specific camera or a microphone. This is set by the endpoint and included in the advertisement.
o Sink ID: integer to uniquely identify the sink within the scope a
  endpoint ID. A sink could be a speaker or screen. This is set by
  the endpoint and included in the advertisement.

o Encoding ID: integer to uniquely identify the encoding of the
  stream within the scope of the stream ID. Note there may be
  multiple encodings of data from the same source. This is set by
  the proposal.

o Salt: salt to use for forming the initialization vector for AEAD.
  The salt shall be sent as part of the packet and need not be sent
  in all the packets. This is created by the endpoint sending the
  message.

o GlobalEncodingID: 64 bit hash of concatenation of conference ID,
  endpoint ID, stream ID, encoding ID

o Capture time: Time when the first sample in the message was
  captured. It is a NTP time in ms with the high order bits
  discarded. The number of bits in the capture time needs to be
  large enough that it does not wrap in for the lifetime of this
  stream. This is set by the endpoint sending the message.

o Sequence ID: When the data captured for a single point in time is
  too large to fit in a single message, it can be split into
  multiple chunks which are sequentially numbered starting at 0
  corresponding to the first chunk of the message. This is set by
  the endpoint sending the message.

o GlobalMessageID: 64 bit hash of concatenation of conference ID,
  endpoint ID, encoding ID, sequence ID

o Active level: this is a number from 0 to 100 indicates the level
  that the sender of this media wishes it to be considered active
  media. For example if it was voice, it would be 100 if the person
  was clearly speaking, and 0 if not, and perhaps a value in the
  middle if it was uncertain. This allows an media switch to select
  the active speaker in the in a conference call.

o Location in room: relative location in room enumerated starting at
  front left and moving around clockwise. This helps get the
  correct content on left and right screens for video and helps with
  for spatial audio

o Reference Frame: bool to indicate if this message is part of a
  reference frame
o DSCP : DSCP to use on transmissions of this message and future messages on this GlobalEncodingID

o Layer ID : Integer indicating which layer is for scalable video codecs. SFU may use this to selectively drop a frame.

The keys used for the AEAD are unique to a given conference ID and endpoint ID.

If the message has any of the following headers, they must occur in the following order followed by all other headers:

1. GlobalEncodingID,
2. GlobalMessageID,
3. conference ID,
4. endpoint ID,
5. encoding ID,
6. sequence ID,
7. active level,
8. DSCP

Every second there must be at least one message in each encoding that contains:

o conference ID,

o endpoint ID,

o encoding ID,

o salt,

o and sequence ID headers

but they are not needed in every packet.

The sequence ID or GlobalMessageID is required in every message and periodically there should be a message with the capture time.
5.1. Securing the messages

The whole message is end to end secured with AEAD. The headers are authenticated while the payload data is authenticated and encrypted. Similar to how the IV for AES-GCM is calculated in SRTP, in this case the IV is computed by xor'ing the salt with the concatenation of the GlobalEncodingID and low 64 bits of sequence ID. The message consists of the authenticated data, followed by the encrypted data, then the authentication tag.

5.2. Sender requests

The control layer supports requesting retransmission of a particular media message identified by IDs and capture time it would contain.

The control layer supports requesting a maximum rate for each given encoding ID.

5.3. Data Codecs

Data messages including raw bytes, xml, senml can all be sent just like media by selecting an appropriate codec and a software based source or sink. An additional parameter to the codec can indicate if reliably delivery is needed and if in order delivery is needed.

5.4. Forward Error Correction

A new Reed-Solomon based FEC scheme based on [I-D.ietf-payload-flexible-fec-scheme] that provides FEC over messages needs to be defined.

5.5. MTI Codecs

Implementation MUST support at least G711, Opus, H.264 and AV1

Video codecs use square pixels.

Video codecs MUST support any aspect ratio within the limits of their max width and height.

Video codecs MUST support a min width and min height of 1.

All video on the wire is oriented such that the first scan line in the frame is up and first pixel in the scan line is on the left.

T.38 fax and DTMF are not supported. Fax can be sent as a TIFF imager over a data channel and DTMF can be done as an application specific information over a data channel.
5.6. Message Key Agreement

The secret for encrypting messages can be provided in the proposal by value or by a reference. The reference approach allows the client to get it from a messaging system where the server creating the proposal may not have access to the secret. For example, it might come from a system like [I-D.barnes-mls-protocol].

6. Control Layer

The control layer needs an API to find out what the capabilities of the device are, and then a way to set up sending and receiving streams. All media flow are only in one direction. The control is broken into control of connectivity and transports, and control of media streams.

6.1. Transport Capabilities API

An API to get information for remote connectivity including:

- set the IP, port, and credential for each TURN2 server
- can return the IP, port tuple for the remote side to send to TURN2 server
- gather local IP, port, protocol tuples for receiving media
- report SHA256 fingerprint of local TLS certificate
- encryption algorithms supported
- report an error for a bad TURN2 credential

6.2. Media Capabilities API

Send and receive codecs are consider separate codecs and can have separate capabilities though the default to the same if not specified separately.

For each send or receive audio codec, an API to learn:

- codec name
- the max sample rate
- the max sample size
- the max bitrate
For each send or receive video codec, an API to learn:

- codec name
- the max width
- the max height
- the max frame rate
- the max pixel depth
- the max bitrate
- the max pixel rate (pixels / second)

6.3. Transport Configuration API

To create a new flow, the information that can be configured is:

- turn server to use
- list of IP, Port, Protocol tuples to try connecting to
- encryption algorithm to use
- TLS fingerprint of far side

An api to allow modification of the follow attributes of a flow:

- total max bandwidth for flow
- forward error correction scheme for flow
- FEC time window
- retransmission scheme for flow
- addition IP, Port, Protocol pairs to send to that may improve connectivity

6.4. Media Configuration API

For all streams:

- set conference ID
- set endpoint ID
- set encoding ID
- salt and secret for AEAD
- flag to pause transition

For each transmitted audio stream, a way to set the:
- audio codec to use
- media source to connect
- max encoded bitrate
- sample rate
- sample size
- number of channels to encode
- packetization time
- process as one of: automatically set, raw, speech, music
- DSCP value to use
- flag to indicating to use constant bit rate

For each transmitted video stream, a way to set
- video codec to use
- media source to connect to
- max width and max height
- max encoded bitrate
- max pixel rate
- sample rate
- sample size
- process as one of: automatically set, rapidly changing video, fine detail video
- DSCP value to use
For each transmitted video stream, a way to tell it to:
  - encode the next frame as an intra frame

For each transmitted data stream:
  - a way to send a data message and indicate reliable or unreliable transmission

For each received audio stream:
  - audio codec to use
  - media sink to connect to
  - lip sync flag

For each received video stream:
  - video codec to use
  - media sink to connect to
  - lip sync flag

For each received data stream:
  - notification of received data messages

Note on lip sync: For any streams that have the lip sync flag set to true, the render attempts to synchronize their play back.

6.5. Transport Metrics
  - report gathering state and completion

6.6. Flow Metrics API
  For each flow, report:
    - report connectivity state
    - report bits sent
    - report packets lost
6.7. Stream Metrics API

For sending streams:
- Bits sent
- packets lost

For receiving streams:
- capture time of most recently receives packet
- endpoint ID of more recently received packet
- bits received
- packets lost

For video streams (send & receive):
- current encoded width and height
- current encoded frame rate

7. Call Signaling

Call signaling is out of scope for usages like WebRTC but other usages may want a common REST API they can use.

Call signaling works be having the client connect to a server when it starts up and send its current advertisement and open a web socket or to receive proposals from the server. A client can make a rest call indicating the parties(s) it wishes to connect to and the server will then send proposals to all clients that connect them. The proposal tell each client exactly how to configure it’s media stack and MUST be either completely accepted, or completely rejected.

The signaling is based on the the advertisement proposal ideas from [I-D.peterson-sipcore-advprop].

We define one round trip of signaling to be a message going from a client up to a server in the cloud, then down to another client which
returns a response along the reverse path. With this definition SIP is takes 1.5 round trips or more if TURN is needed to set up a call while this takes 0.5 round trips.

8. Signaling Examples

8.1. Simple Audio Example

8.1.1. Simple audio advertisement

```json
{
   "receivet":{
      "relay":"2001:db8::10:443",
      "stunSecret":"s8i739dk8",
      "tlsFingerprintSHA256":"1283938"
   },
   "stun":203.0.113.10:43210",
   "stunSecret":"s8i739dk8",
   "tlsFingerprintSHA256":"1283938"
   "local":192.168.0.2:443",
   "stunSecret":"s8i739dk8",
   "tlsFingerprintSHA256":"1283938"
   
   "sources":{
      "sourceID":1,
      "sourceType":"audio",
      "codecs":{
         "codecName":"opus",
         "maxBitrate":128000
      }
   }
   "sinkID":1,
   "sourceType":"audio",
   "codecs":{
      "sourceID":1,
      "sourceType":"audio",
      "codecs":{
```
8.1.2. simple audio proposal

{  
    "receiveAt": [  
        {  
            "relay": "2001:db8::10:443",  
            "stunSecret": "s8i739dk8"  
        },  
        {  
            "stun": "203.0.113.10:43210",  
            "stunSecret": "s8i739dk8"  
        },  
        {  
            "local": "192.168.0.10:443",  
            "stunSecret": "s8i739dk8"  
        }  
    ],  
    "sendTo": [  
        {  
            "relay": "2001:db8::20:443",  
            "stunSecret": "20kdiu83kd8",  
            "tlsFingerprintSHA256": "9389739"  
        },  
        {  
            "stun": "203.0.113.20:43210",  
            "stunSecret": "20kdiu83kd8",  
            "tlsFingerprintSHA256": "9389739"  
        },  
        {  
            "local": "192.168.0.20:443",  
            "stunSecret": "20kdiu83kd8",  
            "tlsFingerprintSHA256": "9389739"  
        }  
    ],  
    "sendStreams": [  
        {  
            "codecName": "opus",  
            "maxBitrate": 256000  
        },  
        {  
            "codecName": "g711"  
        }  
    ]  
}
"conferenceID": 4638572387,
"endpointID": 23,
"sourceID": 1,
"encodingID": 1,
"codecName": "opus",
"AEAD": "AES128-GCM",
"secret": "xy34",
"maxBitrate": 24000,
"packetTime": 20
],
"receiveStreams": [
    {
        "conferenceID": 4638572387,
        "endpointID": 23,
        "sinkID": 1,
        "encodingID": 1,
        "codecName": "opus",
        "AEAD": "AES128-GCM",
        "secret": "xy34"
    }
]
}

8.2. Simple Video Example

Advertisement for simple send only camera with no audio

{
    "sources": [
        {
            "sourceID": 1,
            "sourceType": "video",
            "codecs": [
                {
                    "codecName": "av1",
                    "maxBitrate": 20000000,
                    "maxWidth": 3840,
                    "maxHeight": 2160,
                    "maxFrameRate": 120,
                    "maxPixelRate": 248832000,
                    "maxPixelDepth": 8
                }
            ]
        }
    ]
}
8.2.1. Proposal sent to camera

```json
{
    "sendTo": [
        {
            "relay": "2001:db8::20:443",
            "stunSecret": "20kdiu83kd8",
            "tlsFingerprintSHA256": "9389739"
        }
    ],
    "sendStreams": [
        {
            "conferenceID": 0,
            "endpointID": 0,
            "sourceID": 0,
            "encodingID": 0,
            "codecName": "av1",
            "AEAD": "NULL",
            "width": 640,
            "height": 480,
            "frameRate": 30
        }
    ]
}
```

8.3. Simulcast Video Example

Advertisement same as simple camera above but proposal has two streams with different encodingID.
```json
{
  "sendTo": [
    {
      "relay": "2001:db8::20:443",
      "stunSecret": "20kdiu83kd8",
      "tlsFingerprintSHA256": "9389739"
    }
  ],
  "sendStreams": [
    {
      "conferenceID": 0,
      "endpointID": 0,
      "sourceID": 0,
      "encodingID": 1,
      "codecName": "av1",
      "AEAD": "NULL",
      "width": 1920,
      "height": 1080,
      "frameRate": 30
    },
    {
      "conferenceID": 0,
      "endpointID": 0,
      "sourceID": 0,
      "encodingID": 2,
      "codecName": "av1",
      "AEAD": "NULL",
      "width": 240,
      "height": 240,
      "frameRate": 15
    }
  ]
}
```

8.4. FEC Example

8.4.1. Advertisement includes a FEC codec.
{  
  "sources": [  
    {  
      "sourceID": 1,  
      "sourceType": "video",  
      "codecs": [  
        {  
          "codecName": "av1",  
          "maxBitrate": 20000000,  
          "maxWidth": 3840,  
          "maxHeight": 2160,  
          "maxFrameRate": 120,  
          "maxPixelRate": 248832000,  
          "maxPixelDepth": 8  
        },  
        {  
          "codecName": "flex-fec-rs"  
        }  
      ]  
    }  
  ]  
}

8.4.2. Proposal sent to camera
{  
  "sendTo": [
    {
      "relay": "2001:db8::20:443",
      "stunSecret": "20kdiu83kd8",
      "tlsFingerprintSHA256": "9389739"
    }
  ],
  "sendStreams": [
    {
      "conferenceID": 0,
      "endpointID": 0,
      "sourceID": 0,
      "encodingID": 1,
      "codecName": "av1",
      "AEAD": "NULL",
      "width": 640,
      "height": 480,
      "frameRate": 30
    },
    {
      "conferenceID": 0,
      "endpointID": 0,
      "sourceID": 0,
      "encodingID": 2,
      "codecName": "flex-fec-rs",
      "fecRepairWindow": 200,
      "fecRepairEncodingIDs": [
        1
      ]
    }
  ]
}

9. Switched Forwarding Unit (SFU)

When several clients are in conference call, the SFU can forward packets based on looking at which clients needs a given GlobalEncodingID. By looking at the "active level", the SFU can figure out which endpoints are the active speaker and forward only those. The SFU never changes anything in the message.

10. Informative References
Internet-Draft                  new-media                     March 2018

[I-D.barnes-mls-protocol]
Barnes, R., Millican, J., Omara, E., Cohn-Gordon, K., and
draft-barnes-mls-protocol-00 (work in progress), February
2018.

[I-D.ietf-payload-flexible-fec-scheme]
Singh, V., Begen, A., Zanaty, M., and G. Mandyam, "RTP
Payload Format for Flexible Forward Error Correction
(FEC)", draft-ietf-payload-flexible-fec-scheme-05 (work in
progress), July 2017.

[I-D.jennings-dispatch-snowflake]
Jennings, C. and S. Nandakumar, "Snowflake — A Lighweight,
Asymmetric, Flexible, Receiver Driven Connectivity
Establishment", draft-jennings-dispatch-snowflake-01 (work
in progress), March 2018.

[I-D.jennings-mmusic-ice-fix]
Jennings, C., "Proposal for Fixing ICE", draft-jennings-
music-ice-fix-00 (work in progress), July 2015.

[I-D.peterson-sipcore-advprop]
Peterson, J. and C. Jennings, "The Advertisement/Proposal
Model of Session Description", draft-peterson-sipcore-
advprop-01 (work in progress), March 2011.

Author’s Address

Cullen Jennings
Cisco

Email: fluffy@iii.ca
Abstract

A sketch of a proposal for a modular media stack for interactive communications.

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 http://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 19, 2018.

Copyright Notice

Copyright (c) 2018 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 (http://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. Goals ............................................................ 3
3. Overview .......................................................... 4
4. Terminology ..................................................... 4
5. Architecture .................................................... 5
6. Connectivity Layer ............................................... 5
   6.1. Snowflake - New ICE ........................................ 6
   6.2. STUN2 .......................................................... 6
   6.2.1. STUN2 Request ............................................ 6
   6.2.2. STUN2 Response .......................................... 6
   6.3. TURN2 .......................................................... 7
7. Transport Layer .................................................. 8
8. Media Layer - RTP3 ............................................... 9
   8.1. RTP Meta Data ................................................ 12
   8.2. Securing the messages ...................................... 12
   8.3. Sender requests .............................................. 12
   8.4. Data Codecs .................................................. 13
   8.5. Media Keep Alive ............................................ 13
   8.6. Forward Error Correction ................................... 13
   8.7. MTI Codecs ................................................... 13
   8.7.1. Audio ....................................................... 13
   8.7.2. Video ....................................................... 13
   8.7.3. Annotation ................................................ 14
   8.7.4. Application Data Channels ................................ 14
   8.7.5. Reverse Requests & Stats ................................ 14
   8.8. Message Key Agreement ..................................... 15
9. Control Layer .................................................... 15
   9.1. Transport Capabilities API ................................ 15
   9.2. Media Capabilities API ..................................... 15
   9.3. Transport Configuration API ................................ 16
   9.4. Media Configuration API .................................... 16
   9.5. Transport Metrics ........................................... 18
   9.6. Flow Metrics API ............................................ 18
   9.7. Stream Metrics API ......................................... 19
10. Call Signalling - JABBER2 ...................................... 19
11. Signalling Examples ............................................ 20
   11.1. Simple Audio Example ..................................... 20
   11.1.1. simple audio advertisement ............................ 20
   11.1.2. simple audio proposal .................................. 21
   11.2. Simple Video Example ..................................... 22
   11.2.1. Proposal sent to camera ................................ 23
   11.3. Simulcast Video Example ................................... 24
   11.4. FEC Example ................................................ 24
   11.4.1. Advertisement includes a FEC codec. ................... 24
   11.4.2. Proposal sent to camera ................................ 25
12. Switched Forwarding Unit (SFU) ................................. 26
1. Introduction

This draft is an accumulation of various ideas some people are thinking about. Most of them are fairly separable and could be morphed into existing protocols though this draft takes a blank sheet of paper approach to considering what would be the best think if we were starting from scratch. With that in place, it is possible to ask which of these ideas makes sense to back patch into existing protocols.

2. Goals

- Better connectivity by enabling situations where asymmetric media is possible.
- Design for SFU (Switch Forwarding Units). Design for multiparty calls first then consider two-party calls as a specialized subcase of that.
- Designed for client servers with server-based control of clients
- Faster setup
- Pluggable congestion control
- Much much simpler
- End to end security
- Remove ability to use STUN / TURN in DDOS reflection attacks
- Ability for receiver of video to tell the sender about size changes of display window such that the sender can match
- Eliminate the problems with ROC in SRTP
- Address reasons people have not used from SDES to DTLS-SRTP
- Separation of call setup and ongoing call / conference control
3. Overview

This draft proposes a new media stack to replace the existing stack RTP, DTLS-SRTP, and SDP Offer Answer. The key parts of this stack are connectivity layer, the transport layer, the media layer, a control API, and the singling layer.

The connectivity layer uses a simplified version of ICE, called snowflake [I-D.jennings-dispatch-snowflake], to find connectivity between endpoints and change the connectivity from one address to another as different networks become available or disappear. It is based on ideas from [I-D.jennings-mmusic-ice-fix].

The transport layer uses QUIC to provide a hop by hop encrypted, congestion controlled transport of media. Although QUIC does not currently have all of the partial reliability mechanisms to make this work, this draft assumes that they will be added to QUIC.

The media layer uses existing codecs and packages them along with extra header information to provide information about, when the sequence needs to be played back, which camera it came from, and media streams to be synchronized.

The control API is an abstract API that provides a way for the media stack to report it capabilities and features and a way for the an application tell the media stack how it should be configured. Configuration includes what codec to use, size and frame rate of video, and where to send the media.

The singling layer is based on an advertisement and proposal model. Each endpoint can create an advertisement that describes what it supports including things like supported codecs and maximum bitrates. A proposal can be sent to an endpoint that tells the endpoint exactly what media to send and receive and where to send it. The endpoint can accept or reject this proposal in total but cannot change any part of it.

4. Terminology

- media stream: Stream of information from a single sensor. For example, a video stream from a single camera. A stream may have multiple encodings for example video at different resolutions.
encoding: A encoded version of a stream. A given stream may have several encodings at different resolutions. One encoding may depend on other encodings such as forward error corrections or in the case of scalable video codecs.

flow: A logical transport between two computers. Many media streams can be transported over a single flow. The actually IP address and ports used to transport data in the flow may change over time as connectivity changes.

message: some data or media that to be sent across the network along with metadata about it. Similar to an RTP packet.

media source: a camera, microphone or other source of data on an endpoint

media sink: a speaker, screen, or other destination for data on an endpoint

TLV: Tag Length Value. When used in the draft, the Tag, Length, and any integer values are coded as variable length integers similar to how this is done in CBOR.

5. Architecture

Much of the deployments architecture of IETF media designs are based on a distributed controller for the media stack that is running peer to peer in each client. Nearly all deployments, by they a cloud based conferencing systems or an enterprise PBX, use a central controller that acts as an SBC to try and controll each client. The goal here would be an deployment architecture that

- support a single controller that controlled all the device in a given conference or call. The controller could be in the cloud or running on one of the endpoints.

- design for multi party conference calls first and treat 2 party calls as a specialed sub case of that

- design with the assumption that an light weight SFU (Switched Forwarding Unit) was used to distribute media for conference calls.

6. Connectivity Layer
6.1. Snowflake - New ICE

All that is needed to discover the connectivity is way to:

- Gather some IP/ports that may work using TURN2 relay, STUN2, and local addresses.
- A controller, which might be running in the cloud, to inform a client to send a STUN2 packet from a given source IP/port to a given destination IP/port.
- The receiver notifies the controller about information on received STUN2 packets.
- The controller can tell the sender the secret that was in the packet to prove consent of the receiver to receive data then the sending client can allow media to flow over that connection.

The actually algorithm used to decide on what pairs of addresses are tested and in what order does not need to be agreed on by both the sides of the call - only the controller needs to know it. This allows the controller to use machine learning, past history, and heuristics to find an optimal connection much faster than something like ICE.

The details of this approach are described in [I-D.jennings-dispatch-snowflake]. Many of ideas in this can be traced back to [I-D.kaufman-rtcweb-traversal].

6.2. STUN2

The speed of setting up a new media flow is often determined by how many STUN2 checks need to be done. If the STUN2 packets are smaller, then the stun checks can be done faster without risk of causing congestion.

6.2.1. STUN2 Request

A STUN2 request consists of, well, really nothing. The STUN client just opens a QUIC connection to the STUN server.

6.2.2. STUN2 Response

When the STUN2 server receives a new QUIC connection, it responds with the IP address and port that the connection came from.

The client can check it is talking to the correct STUN server by checking the fingerprint of the certificate. Protocols like ICE
would need to exchange theses fingerprints instead of all the crazy stun attributes.

Thanks to Peter Thatcher for proposing STUN over QUIC.

6.3.  TURN2

TODO: make TURN2 run over QUIC

Out of band, the client tells the TURN2 server the fingerprint of the cert it uses to authenticate with. The TURN2 server gives the client two public IP:port address pairs. One is called inbound and other called outbound. The client connects to the outbound port and authenticates to TURN2 server using the TLS domain name of server. The TURN2 server authenticates the client using mutual TLS with fingerprint of cert provided by the client. Any time a message or stun packet is received on the matched inbound port, the TURN2 server forwards it to the client(s) connected to the outbound port.

A single TURN2 connection can be used for multiple different calls or session at the same time and a client could choose to allocate the TURN2 connection at the time that it started up. It does not need to be done on a per session basis.

The client can not send from the TURN2 server.
7. Transport Layer

The responsibility of the transport layer is to provide an end to end crypto layer equivalent to DTLS and they must ensure adequate congestion control. The transport layer brings up a flow between two computers. This flow can be used by multiple media streams.

The MTI transport layer is QUIC with packets. It assumes that QUIC has a way to delivers the packets in an efficient unreliable mode as well as an optional way to deliver important metadata packets in a reliable mode. It assumes that QUIC can report up to the rate adaptation layer a current max target bandwidth that QUIC can transmit at. It’s possible these are all unrealistic characteristics.
of QUIC in which case a new transport protocol should be developed that provides these and is layered on top of DTLS for security.

This is secured by checking the fingerprints of the DTLS connection match the fingerprints provided at the control layer or by checking the names of the certificates match what was provided at control layer.

The transport layer needs to be able to set the DSCP values in transmitting packets as specified by the control layer.

The transport MAY provide a compression mode to remove the redundancy of the non-encrypted portion of the media messages such as GlobalEncodingID. For example, a GlobalEncodingID could be mapped to a QUIC channel and then it could be removed before sending the message and added back on the receiving side.

The transport need to be able to ensure that it has a very small chance of being confused with the STUN2 traffic it will be multiplexed with. (Open issue – if the STUN2 runs on top of same transport, this becomes less of issue)

The transport crypto needs to be able to export server state that can be passed out of band to the client to enable the client to make a zero RTT connection to the server.

8. Media Layer – RTP3

Each message consist of a set of TLV headers with metadata about the packet, followed by payload data such as the output of audio or video codec.

There are several message headers that help the receiver understand what to do with the media. The TLV header are the follow:

- Conference ID: Integer that will be globally unique identifier for the for all applications using a common call singling system. This is set by the proposal.

- Endpoint ID: Integer to uniquely identify the endpoint with within scope of conference ID. This is set by the proposal.

- Source ID: integer to uniquely identify the input source within the scope a endpoint ID. A source could be a specific camera or a microphone. This is set by the endpoint and included in the advertisement.
o Sink ID: integer to uniquely identify the sink within the scope of an endpoint ID. A sink could be a speaker or screen. This is set by the endpoint and included in the advertisement. An endpoint sending media can have this set. If it is set it should transmit it for 3 frames any time it changes and once every 5 seconds. An SFU can add, modify, or delete this from any media packet. TODO - How to use this for SFU controlled layout - for example, if have 100 users in conference and want to put the 10 most recent speakers in thumbnails. Do we need this at all?

o Encoding ID: integer to uniquely identify the encoding of the stream within the scope of the source ID. Note there may be multiple encodings of data from the same source. This is set by the proposal.

o Salt: salt to use for forming the initialization vector for AEAD. The salt shall be sent as part of the packet and need not be sent in all the packets. This is created by the endpoint sending the message.

o GlobalEncodingID: 64 bit hash of concatenation of conference ID, endpoint ID, source ID, encoding ID

o Capture time: Time when the first sample in the message was captured. It is a NTP time in ms with the high order bits discarded. The number of bits in the capture time needs to be large enough that it does not wrap in for the lifetime of this stream. This is set by the endpoint sending the message.

o Sequence ID: When the data captured for a single point in time is too large to fit in a single message, it can be split into multiple chunks which are sequentially numbered starting at 0 corresponding to the first chunk of the message. This is set by the endpoint sending the message.

o GlobalMessageID: 64 bit hash of concatenation of conference ID, endpoint ID, encoding ID, sequence ID

o Active level: this is a number from 0 to 100 indicates the probability that the sender of this media wishes it to be considered active media. For example if it was voice, it would be 100 if the person was clearly speaking, and 0 if not, and perhaps a value in the middle if it was uncertain. This allows a media switch to select the active speaker in the a conference call.

o Location: relative or absolute location, direction of view, and field view. With video coming from drones, 360 cameras, VR light field cameras, and complex video conferencing rooms, this provides
the information about the camera or microphone that the receiver can use to render the correct view. This is end to end encrypted.

- Reference Frame: bool to indicate if this message is part of a reference frame. Typically, a SFU will switch to the new video stream at the start of a reference frame.

- DSCP: DSCP to use on transmissions of this message and future messages on this GlobalEncodingID

- Layer ID: Integer indicating which layer is for scalable video codecs. SFU may use this to selectively drop a frame.

The keys used for the AEAD are unique to a given conference ID and endpoint ID.

If the message has any of the following headers, they must occur in the following order followed by all other headers:

1. GlobalEncodingID,
2. GlobalMessageID,
3. conference ID,
4. endpoint ID,
5. encoding ID,
6. sequence ID,
7. active level,
8. DSCP

Every second there must be at least one message in each encoding that contains:

- conference ID,
- endpoint ID,
- encoding ID,
- salt,
- and sequence ID headers
but they are not needed in every packet.

The sequence ID or GlobalMessageID is required in every message and periodically there should be message with the capture time.

8.1. RTP Meta Data

We tend to end up with a few categories of data associated with the media:

- Stuff you need at the same time you get the media. For example, this is a reference frame.

- Stuff you need soon but not instantly. For example the name of the speaker in a given rectangle of a video stream.

And it tends to change at different rates:

- Stuff that you need to process the media and may change but does not change quickly and you don’t need it with every frame. For example, salt for encryption.

- Stuff that you need to join the media but may never change. For example, resolution of the video is.

TODO - think about how to optimize design for each type of meta data.

8.2. Securing the messages

The whole message is end to end secured with AEAD. The headers are authenticated while the payload data is authenticated and encrypted. Similar to how the IV for AES-GCM is calculated in SRTP, in this case the IV is computed by xor’ing the salt with the concatenation of the GlobalEncodingID and low 64 bits of sequence ID. The message consists of the authenticated data, followed by the encrypted data, then the authentication tag.

8.3. Sender requests

The control layer supports requesting retransmission of a particular media message identified by IDs and capture time it would contain.

The control layer supports requesting a maximum rate for each given encoding ID.
8.4. Data Codecs

Data messages including raw bytes, xml, senml can all be sent just like media by selecting an appropriate codec and a software based source or sink. An additional parameter to the codec can indicate if reliably delivery is needed and if in order delivery is needed.

8.5. Media Keep Alive

Provided by transport.

8.6. Forward Error Correction

A new Reed-Solomon based FEC scheme based on [I-D.ietf-payload-flexible-fec-scheme] that provides FEC over messages needs to be defined.

8.7. MTI Codecs

8.7.1. Audio

Implementation MUST support at least G711 and Opus

8.7.2. Video

Implementation MUST support at least H.264 and AV1

Video codecs use square pixels.

Video codecs MUST support any aspect ratio within the limits of their max width and height.

Video codecs can specify a maximum pixel rate, maximum frame rate, maximum images size. The can also specify a list of binary flags of supported features which are defined by the codec and may be supported by the codec for encode, decode, or neither where each feature can be independently controlled. They can not impose constraints beyond that. Some existing codecs like vp8 may easily fit into that while some codec like H264 may need some suspects defined as new codecs to meet the requirements for this. It is not expected that all the nuances that could be negotiated with SDP for 264 would be supported in this new media.

Video codecs MUST support a min width and min height of 1.

All video on the wire is oriented such that the first scan line in the frame is up and first pixel in the scan line is on the left.
T.38 fax and DTMF are not supported. Fax can be sent as a TIFF imager over a data channel and DTMF can be done as an application specific information over a data channel.

TODO: Capture the list of what metadata video encoders produce * if it is a reference frame or not * resolution * frame-rate ? * capture time of frame

TODO: Capture the list of what metadata video encoders needs. * capture timestamp * source and target resolution * source and target frame-rate * target bitrate * max bitrate * max pixel rate

8.7.3. Annotation

Optional support for annotation based overlay using vector graphics such as a subset of SVG.

8.7.4. Application Data Channels

Need support for application defined data in both a reliable and unreliable datagram mode.

8.7.5. Reverse Requests & Stats

The hope is that this is not needed.

Much of what goes in the reverse direction of the media in RTCP is either used for congestion control, diagnostics, or control of the codec such as requesting to resend a frame or sending a new intra codec frame for video. The design reduces the need for this.

The congestion control information which is needed quickly is all handled at QUIC layer.

The diagnostic type information can be reported from the endpoint to the controller and does not need to flow at the media level.

Information that needs to be delivered reliably can be sent that way at the QUIC level remove the need for retransmit type request. System that use selective retransmission to recover from packet loss of media do not tend to work as well for interactive medias as forward error correction schemes because of the large latency they introduce.

Information like requesting a new intra codec frame for video often needs to come from the controller and can be sent over the signalling and control layer.
8.8. Message Key Agreement

The secret for encrypting messages can be provided in the proposal by value or by a reference. The reference approach allows the client to get it from a messaging system where the server creating the proposal may not have access to the secret. For example, it might come from a system like [I-D.barnes-mls-protocol].

9. Control Layer

The control layer needs an API to find out what the capabilities of the device are, and then a way to set up sending and receiving streams. All media flow are only in one direction. The control is broken into control of connectivity and transports, and control of media streams.

9.1. Transport Capabilities API

An API to get information for remote connectivity including:

- set the IP, port, and credential for each TURN2 server
- can return the IP, port tuple for the remote side to send to TURN2 server
- gather local IP, port, protocol tuples for receiving media
- report SHA256 fingerprint of local TLS certificate
- encryption algorithms supported
- report an error for a bad TURN2 credential

9.2. Media Capabilities API

Send and receive codecs are consider separate codecs and can have separate capabilities though the default to the same if not specified separately.

For each send or receive audio codec, an API to learn:

- codec name
- the max sample rate
- the max sample size
- the max bitrate
For each send or receive video codec, an API to learn:

- codec name
- the max width
- the max height
- the max frame rate
- the max pixel depth
- the max bitrate
- the max pixel rate (pixels/second)

9.3. Transport Configuration API

To create a new flow, the information that can be configured is:

- turn server to use
- list of IP, Port, Protocol tuples to try connecting to
- encryption algorithm to use
- TLS fingerprint of far side

An api to allow modification of the follow attributes of a flow:

- total max bandwidth for flow
- forward error correction scheme for flow
- FEC time window
- retransmission scheme for flow
- addition IP, Port, Protocol pairs to send to that may improve connectivity

9.4. Media Configuration API

For all streams:

- set conference ID
- set endpoint ID
For each transmitted audio stream, a way to set the:

- audio codec to use
- media source to connect
- max encoded bitrate
- sample rate
- sample size
- number of channels to encode
- packetization time
- process as one of: automatically set, raw, speech, music
- DSCP value to use
- flag to indicating to use constant bit rate
- optionally set a sinkID to periodically include in the media

For each transmitted video stream, a way to set

- video codec to use
- media source to connect to
- max width and max height
- max encoded bitrate
- max pixel rate
- sample rate
- sample size
- process as one of: automatically set, rapidly changing video, fine detail video
For each transmitted video stream, a way to tell it to:

- encode the next frame as an intra frame

For each transmitted data stream:

- a way to send a data message and indicate reliable or unreliable transmission

For each received audio stream:

- audio codec to use
- media sink to connect to
- lip sync flag

For each received video stream:

- video codec to use
- media sink to connect to
- lip sync flag

For each received data stream:

- notification of received data messages

Note on lip sync: For any streams that have the lip sync flag set to true, the render attempts to synchronize their playback.

9.5. Transport Metrics

- report gathering state and completion

9.6. Flow Metrics API

For each flow, report:

- report connectivity state
9.7. Stream Metrics API

For sending streams:
- Bits sent
- Packets lost

For receiving streams:
- Capture time of most recently receives packet
- Endpoint ID of more recently received packet
- Bits received
- Packets lost

For video streams (send & receive):
- Current encoded width and height
- Current encoded frame rate

10. Call Signalling - JABBER2

Call signalling is out of scope for usages like WebRTC but other uses may want a common REST API they can use.

Call signalling works by having the client connect to a server when it starts up and send its current advertisement and open a web socket or to receive proposals from the server. A client can make a REST call indicating the parties(s) it wishes to connect to and the server will then send proposals to all clients that connect them. The proposal tell each client exactly how to configure its media stack and MUST be either completely accepted, or completely rejected.
The signalling is based on the advertisement proposal ideas from [I-D.peterson-sipcore-advprop].

We define one round trip of signalling to be a message going from a client up to a server in the cloud, then down to another client which returns a response along the reverse path. With this definition SIP takes 1.5 round trips or more if TURN is needed to set up a call while this takes 0.5 round trips.

11. Signalling Examples

11.1. Simple Audio Example

11.1.1. simple audio advertisement

```
{  
    "receiveAt": [  
        {  
            "relay": "2001:db8::10:443",  
            "stunSecret": "s8i739dk8",  
            "tlsFingerprintSHA256": "1283938"  
        },  
        {  
            "stun": "203.0.113.10:43210",  
            "stunSecret": "s8i739dk8",  
            "tlsFingerprintSHA256": "1283938"  
        },  
        {  
            "local": "192.168.0.2:443",  
            "stunSecret": "s8i739dk8",  
            "tlsFingerprintSHA256": "1283938"  
        }  
    ],  
    "sources": [  
        {  
            "sourceID": 1,  
            "sourceType": "audio",  
            "codecs": [  
                {  
                    "codecName": "opus",  
                    "maxBitrate": 128000  
                },  
                {  
                    "codecName": "g711"  
                }  
            ]  
        }  
    ]
}
```
"sinks": [
  {
    "sinkID": 1,
    "sourceType": "audio",
    "codecs": [
      {
        "codecName": "opus",
        "maxBitrate": 256000
      },
      {
        "codecName": "g711"
      }
    ]
  }
]

11.1.2. simple audio proposal

{ "receiveAt": [
  {
    "relay": "2001:db8::10:443",
    "stunSecret": "s8i739dk8"
  },
  {
    "stun": "203.0.113.10:43210",
    "stunSecret": "s8i739dk8"
  },
  {
    "local": "192.168.0.10:443",
    "stunSecret": "s8i739dk8"
  }
], "sendTo": [
  {
    "relay": "2001:db8::20:443",
    "stunSecret": "20kdiu83kd8",
    "tlsFingerprintSHA256": "9389739"
  },
  {
    "stun": "203.0.113.20:43210",
    "stunSecret": "20kdiu83kd8",
    "tlsFingerprintSHA256": "9389739"
  },
  {
    "local": "192.168.0.20:443",
    "stunSecret": "20kdiu83kd8",
    "tlsFingerprintSHA256": "9389739"
  }
]
"tlsFingerprintSHA256":"9389739"
],
"sendStreams": [
{
"conferenceID":4638572387,
"endpointID":23,
"sourceID":1,
"encodingID":1,
"codecName":"opus",
"AEAD":"AES128-GCM",
"secret":"xy34",
"maxBitrate":24000,
"packetTime":20
}
],
"receiveStreams": [
{
"conferenceID":4638572387,
"endpointID":23,
"sinkID":1,
"encodingID":1,
"codecName":"opus",
"AEAD":"AES128-GCM",
"secret":"xy34"
}
]

11.2. Simple Video Example

Advertisement for simple send only camera with no audio
11.2.1. Proposal sent to camera

```json
{
    "sendTo": {
        "relay": "2001:db8::20:443",
        "stunSecret": "20kdiu83kd8",
        "tlsFingerprintSHA256": "9389739"
    },
    "sendStreams": {
        "conferenceID": 0,
        "endpointID": 0,
        "sourceID": 0,
        "encodingID": 0,
        "codecName": "av1",
        "AEAD": "NULL",
        "width": 640,
        "height": 480,
        "frameRate": 30
    }
}
```
11.3. Simulcast Video Example

Advertisement same as simple camera above but proposal has two streams with different encodingID.

```json
{
    "sendTo":{
        "relay":"2001:db8::20:443",
        "stunSecret":"20kdiu83kd8",
        "tlsFingerprintSHA256":"9389739"
    },
    "sendStreams":[
        {
            "conferenceID":0,
            "endpointID":0,
            "sourceID":0,
            "encodingID":1,
            "codecName":"av1",
            "AEAD":"NULL",
            "width":1920,
            "height":1080,
            "frameRate":30
        },
        {
            "conferenceID":0,
            "endpointID":0,
            "sourceID":0,
            "encodingID":2,
            "codecName":"av1",
            "AEAD":"NULL",
            "width":240,
            "height":240,
            "frameRate":15
        }
    ]
}
```

11.4. FEC Example

11.4.1. Advertisement includes a FEC codec.
{  
  "sources": [
    {
      "sourceID": 1,
      "sourceType": "video",
      "codecs": [
        {
          "codecName": "av1",
          "maxBitrate": 20000000,
          "maxWidth": 3840,
          "maxHeight": 2160,
          "maxFrameRate": 120,
          "maxPixelRate": 248832000,
          "maxPixelDepth": 8
        },
        {
          "codecName": "flex-fec-rs"
        }
      ]
    }
  ]
}

11.4.2. Proposal sent to camera
12. Switched Forwarding Unit (SFU)

When several clients are in conference call, the SFU can forward packets based on looking at which clients needs a given GlobalEncodingID. By looking at the "active level", the SFU can figure out which endpoints are the active speaker and forward only those. The SFU never changes anything in the message.

12.1. Software Defined Networking

Is it possible to use the packet recycling concepts in SDN to forward a single packet to multiple endpoints? Can the way SDN forwarding would work be adapted to use a SDN router as a SFU?
12.2. Vector Packet Processors

Can we use fast VPP systems like fd.io to create a SFU?

12.3. Information Centric Networking

What changes would be needed to map RTP2 into the prefix and suffix of hICN?

13. Acknowledgements

Thank you for input from: Harald Alvestrand, Espen Berger, Matthew Kaufman, Patrick Linskey, Eric Rescorla, Peter Thatcher, Malcolm Walters Martin Thomson

14. Other Work

rfc7016

draft-kaufman-rtcweb-traversal

Consider using terminology from rfc7656

docs.google.com/presentation/d/1Sg_1TVCCkJvZ8Egz5oa0CP01TC2rNdv9HVu7W38Y4zA/edit#slide=id.g29a8672e18_22_120

docs.google.com/presentation/d/1o-o5jZBLw3Py1OuenzWDkxDG6NiSMlHvGw5KemKWLw/edit#slide=id.g2f8f4acff1_1_249

cs.chromium.org/chromium/src/third_party/webrtc/common_video/include/video_frame.h

15. Style of specification

Fundamental driven by experiments. The proposal is to have a high level overview document where we document some of the design - this document could be a start of that. Then write a a spec for each on of the separable protocol parts such as STUN2, TURN2, etc.

The protocol specs would contain a high level overview like you might find on a wikipedia page and the details of the protocol encoding would be provided in an open source reference implementation. The test code for the references implementation helps test the spec. The implementation is not optimized for performance but instead is simply trying to clearly illustrate the protocol. Particular version of the draft would be bound to a tagged version of the source code. All the
source code would be under normal IETF IPR rules just like it was included directly in the draft.

16. Informative References

[I-D.barnes-mls-protocol]

[I-D.ietf-payload-flexible-fec-scheme]

[I-D.jennings-dispatch-snowflake]

[I-D.jennings-mmusic-ice-fix]

[I-D.kaufman-rtcweb-traversal]

[I-D.peterson-sipcore-advprop]

Author’s Address

Cullen Jennings
Cisco

Email: fluffy@iii.ca
A Language for Rules Describing JSON Content  
draft-newton-json-content-rules-09

Abstract

This document describes a language for specifying and testing the  
expected content of JSON structures found in JSON-using protocols,  
software, and processes.

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 April 1, 2018.

Copyright Notice

Copyright (c) 2017 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.  A First Example: Specifying Content ................. 3
   1.2.  A Second Example: Testing Content .................. 3
2.  Overview of the Language ............................... 5
3.  Lines and Comments ...................................... 7
4.  Rules .................................................. 8
   4.1.  Rule Names and Assignments ......................... 8
   4.2.  Annotations ......................................... 9
   4.3.  Starting Points and Root Rules ..................... 10
   4.4.  Type Specifications ................................. 10
   4.5.  Primitive Specifications ............................ 12
      4.5.1.  Numbers, Booleans and Null .................... 12
      4.5.2.  Strings ...................................... 13
   4.6.  Any Type ........................................... 15
   4.7.  Member Specifications ............................... 16
   4.8.  Object Specifications ............................... 16
   4.9.  Array Specifications ................................ 19
      4.9.1.  Unordered Array Specifications .................. 21
   4.10. Group Specifications ................................ 21
   4.11. Ordered and Unordered Groups in Arrays .............. 22
   4.12. Sequence and Choice Combinations in Array, Object, and Group Specifications .................. 22
   4.13. Repetition in Array, Object, and Group Specifications .. 23
5.  Directives .............................................. 26
   5.1.  jcr-version ....................................... 26
   5.2.  ruleset-id ....................................... 27
   5.3.  import ........................................... 27
6.  Tips and Tricks .......................................... 28
   6.1.  Any Member with Any Value .......................... 28
   6.2.  Lists of Values .................................... 29
   6.3.  Groups in Arrays ................................... 29
   6.4.  Groups in Objects .................................. 30
   6.5.  Group Rules as Macros ............................... 31
   6.6.  Object Mixins ...................................... 31
   6.7.  Subordinate Dependencies ............................. 31
7.  Implementation Status .................................... 32
   7.1.  JCR Validator ...................................... 32
   7.2.  Codalogic JCR Parser ................................ 33
   7.3.  JCR Java .......................................... 33
8.  ABNF Syntax ............................................. 33
9.  Acknowledgements ........................................ 39
10. References .............................................. 39
   10.1.  Normative References ............................... 39
   10.2.  Informative References ............................. 40
   10.3.  URIs ............................................. 40
1. Introduction

This document describes JSON Content Rules (JCR), a language for specifying and testing the interchange of data in JSON [RFC7159] format used by computer protocols and processes. The syntax of JCR is not JSON but is "JSON-like", possessing the conciseness and utility that has made JSON popular.

1.1. A First Example: Specifying Content

The following JSON data describes a JSON object with two members, "line-count" and "word-count", each containing an integer.

{ "line-count" : 3426, "word-count" : 27886 }

Figure 1

This is also JCR that describes a JSON object with a member named "line-count" that is an integer that is exactly 3426 and a member named "word-count" that is an integer that is exactly 27886.

For a protocol specification, it is probably more useful to specify that each member is any integer and not specific, exact integers:

{ "line-count" : integer, "word-count" : integer }

Figure 2

Since line counts and word counts should be either zero or a positive integer, the specification may be further narrowed:

{ "line-count" : 0.. , "word-count" : 0.. }

Figure 3

1.2. A Second Example: Testing Content

Building on the first example, this second example describes the same object but with the addition of another member, "file-name".
The following JCR describes objects like it.

```
{
  "file-name"  : "rfc7159.txt",
  "line-count" : 3426,
  "word-count" : 27886
}
```

**Figure 4**

For the purposes of writing a protocol specification, JCR may be broken down into named rules to reduce complexity and to enable reuse. The following example takes the JCR from above and rewrites the members as named rules.

```
{
  $fn,  
  $lc,  
  $wc
}
```

$fn = "file-name"  : string
$lc = "line-count" : 0..
$wc = "word-count" : 0..

**Figure 6**

With each member specified as a named rule, software testers can override them locally for specific test cases. In the following example, the named rules are locally overridden for the test case where the file name is "rfc4627.txt".

```
$fn = "file-name"  : "rfc4627.txt"
$lc = "line-count" : 2102
$wc = "word-count" : 16714
```

**Figure 7**
In this example, the protocol specification describes the JSON object in general and an implementation overrides the rules for testing specific cases.

All figures used in this specification are available here [1].

2. Overview of the Language

JCR is composed of rules (as the name suggests). A collection of rules that is processed together is a ruleset. Rulesets may also contain comments, blank lines, and directives that apply to the processing of a ruleset.

Rules are composed of two parts, an optional rule name and a rule specification. A rule specification can be either a type specification or a member specification. A member specification consists of a member name specification and a type specification.

A type specification is used to specify constraints on a superset of a JSON value (e.g. number / string / object / array etc.). In addition to defining primitive types (such as string or integer), array types, and object types, type specifications may define the JCR specific concept of group types.

Type specifications corresponding to arrays, objects and groups may be composed of other rule specifications.

A member specification is used to specify constraints on a JSON member (i.e. members of a JSON object).

Rules with rule name assignments may be referenced in place of type specifications and member specifications.

Rules may be defined across line boundaries and there is no line continuation syntax.

Any rule consisting only of a type specification is considered a root rule. Unless otherwise specified, all the root rules of a ruleset are evaluated against a JSON instance or document.

Putting it all together, Figure 9 describes the JSON in Figure 8.
Example JSON shamelessly lifted from RFC 4627

```
{
   "Image": {
      "Width": 800,
      "Height": 600,
      "Title": "View from 15th Floor",
      "Thumbnail": {
         "Url": "http://www.example.com/image/481989943",
         "Height": 125,
         "Width": 100
      },
      "IDs": [116, 943, 234, 38793]
   }
}
```

Figure 8
Rules describing Figure 8

; the root of the JSON instance is an object
; this root rule describes that object
{

 ; the object specification contains
 ; one member specification
 "Image" : {

 ; $width and $height are defined below
 $width,
 $height,

 ; "Title" member specification
 "Title" : string,

 ; "Thumbnail" member specification, which
 ; defines an object
 "Thumbnail": {

 ; $width and $height are re-used again
 $width, $height,

 "Url" : uri
 },

 ; "IDs" member that is an array of
 ; one ore more integers
 "IDs" : [ integer * ]

}

; The definitions of the rules $width and $height
$width  = "Width" : 0..1280
$height = "Height" : 0..1024

Figure 9

3. Lines and Comments

There is no statement terminator and therefore no need for a line
continuation syntax. Rules may be defined across line boundaries.
Blank lines are allowed.
Comments are the same as comments in ABNF [RFC4234]. They start with a semi-colon (’;’) and continue to the end of the line.

4. Rules

Rules have two main components, an optional rule name assignment and a type or member specification.

Type specifications define arrays, objects, etc... of JSON and may reference other rules using rule names. Most type specifications can be defined with repetitions for specifying the frequency of the type being defined. In addition to the type specifications describing JSON types, there is an additional group specification for grouping types.

Member specifications define members of JSON objects, and are composed of a member name specification and either a type specification or a rule name referencing a type specification.

Rules may also contain annotations which may affect the evaluation of all or part of a rule. Rules without a rule name assignment are considered root rules, though rules with a rule name assignment can be considered a root rule with the appropriate annotation.

Type specifications, depending on their type, can contain zero or more other specifications or rule names. For example, an object specification might contain multiple member specifications or rule names that resolve to member specifications or a mixture of member specifications and rule names. For the purposes of this document, specifications and rule names composing other specifications are called subordinate components.

4.1. Rule Names and Assignments

Rule names are signified with the dollar character (’$’), which is not part of the rule name itself. Rule names have two components, an optional ruleset identifier alias and a local rule name.

Local rule names must start with an alphabetic character (a-z,A-Z) and must contain only alphabetic characters, numeric characters, the hyphen character (’-’) and the underscore character (’_’). Local rule names are case sensitive, and must be unique within a ruleset (that is, no two rule name assignments may use the same local rule name).

Ruleset identifier aliases enable referencing rules from another ruleset. They are not allowed in rule name assignments, and only found in rule names referencing other rules. Ruleset identifiers
must start with an alphabetic character and contain no whitespace. Ruleset identifiers are case sensitive. Simple use cases of JCR will most likely not use ruleset identifiers.

In Figure 10 below, "http://ietf.org/rfcYYYY.JCR" and "http://ietf.org/rfcXXXX.JCR" are ruleset identifiers and "rfcXXXX" is a ruleset identifier alias.

```
# ruleset-id http://ietf.org/rfcYYYY.JCR
# import http://ietf.org/rfcXXXX.JCR as rfcXXXX
$my_encodings = ( "mythic" | "magic" )
$all_encodings = ( $rfcXXXX.encodings | $my_encodings )
```

Figure 10

There are two forms of rule name assignments: assignments of primitive types and assignments of all other types. Rule name assignments to primitive type specifications separate the rule name from the type specification with the character sequence ‘=:', whereas rule name assignments for all other type specifications only require the separation using the ‘=' character.

```
;rule name assignments for primitive types
$foo        =: "foo"
$some_string =: string

;rule name assignments for arrays
$bar = [ integer, integer, integer ]

;rule name assignment for objects
$bob = { "bar" : $bar, "foo" : $foo }
```

Figure 11

This is the one little "gotcha" in JCR. This syntax is necessary so that JCR parsers may readily distinguish between rule name assignments involving string and regular expressions primitive types and member names of member specifications.

4.2. Annotations

Annotations may appear before a rule name assignment, before a type or member specification, or before a rule name contained within a type specification. In each place, there may be zero or more annotations. Each annotation begins with the character sequence "@[" and ends with "]". The following is an example of a type specification with the not annotation (explained in Section 4.14):
This specification defines the annotations "root", "not", and "unordered", but other annotations may be defined for other purposes.

4.3. Starting Points and Root Rules

Evaluation of a JSON instance or document against a ruleset begins with the evaluation of a root rule or set of root rules. If no root rule (or rules) is specified locally at runtime, the set of root rules specified in the ruleset are evaluated. The order of evaluation is undefined.

The set of root rules specified in a ruleset is composed of all rules without a rule name assignment and all rules annotated with the "@{root}" annotation.

The "@{root}" annotation may either appear before a rule name assignment or before a type definition. It is ignored if present before referenced rule name inside of a type specification.

4.4. Type Specifications

The syntax of each type of type specifications varies depending on the type:
; primitive types can be string
; or number literals
; or number ranges
"foo"
2
1..10

; primitive types can also be more generalized types
string
integer

; primitive type rules may be named
$my_int =: 12

; member specifications consist of a member name
; followed by a colon and then followed by another
; type specification or a rule name
; (example shown with a rule name assignment)
$mem1 = "bar" : "baz"
$mem2 = "fizz" : $my_int

; member names may either be quoted strings
; or regular expressions
; (example shown with a rule name assignment)
$mem3 = /^[^dev][0-9]$/ : 0..4096

; object specifications start and end with "curly braces"
; object specifications contain zero
; or more member specifications
; or rule names which reference a member specification
{ $mem1, "foo" : "fuzz", "fizz" : $my_int }

; array specifications start and end with square brackets
; array specifications contain zero
; or more non-member type specifications
[ 1, 2, 3, $my_int ]

; finally, group specifications start and end with parenthesis
; groups contain other type specifications
( [ integer, integer], $rule1 )
$rule1 = [ string, string ]
4.5. Primitive Specifications

Primitive type specifications define content for JSON numbers, booleans, strings, and null.

4.5.1. Numbers, Booleans and Null

The rules for booleans and null are the simplest and take the following forms:

```
true
false
boolean
null
```

Figure 14

Rules for numbers can specify the number be either an integer or floating point number:

```
integer
float
double
```

Figure 15

The keyword ‘float’ represents a single precision IEEE-754 floating point number represented in decimal. The keyword ‘double’ represents a double precision IEEE-754 floating point number represented in decimal format.

Numbers may also be specified as an absolute value or a range of possible values, where a range may be specified using a minimum, maximum, or both:

```
n
n..m
..m
n..
n.f
n.f..m.f
..m.f
n.f..
```

Figure 16

When specifying a minimum and a maximum, both must either be an integer or a floating point number. Thus to specify a floating point
number between zero and ten a definition of the following form is used:

0.0..10.0

Figure 17

Integers may also be specified as ranges using bit lengths preceded by the ‘int’ or ‘uint’ words (i.e. ‘int8’, ‘uint16’). The ‘int’ prefix specifies the integer as being signed whereas the ‘uint’ prefix specifies the integer as being unsigned.

; 0..255
uint8

; -32768..32767
int16

; 0..65535
uint16

; -9223372036854775808..9223372036854775807
int64

; 0..18446744073709551615
uint64

Figure 18

4.5.2. Strings

JCR provides a large number of data types to define the contents of JSON strings. Generically, a string may be specified using the word ‘string’. String literals may be specified using a double quote character followed by the literal content followed by another double quote. And regular expressions may be specified by enclosing a regular expression within the forward slash (‘/’) character.
; any string
string

; a string literal
"she sells sea shells"

; a regular expression
/"she sells .*/

Figure 19

Regular expressions are not implicitly anchored and therefore must be explicitly anchored if necessary.

A string can be specified as a URI [RFC3986] using the word ‘uri’, but also may be more narrowly scoped to a URI of a specific scheme. Specific URI schemes are specified with the word ‘uri’ followed by two period characters (‘..' ) followed by the URI scheme.

; any URI
uri

;a URI narrowed for an HTTPS uri
uri..https

Figure 20

IP addresses may be specified with either the word ‘ipv4’ for IPv4 addresses [RFC1166] or the word ‘ipv6’ for IPv6 addresses [RFC5952]. Fully qualified A-label and U-label domain names may be specified with the words ‘fqdn’ and ‘idn’.

Dates and time can be specified as formats found in RFC 3339 [RFC3339]. The word ‘date’ corresponds to the full-date ABNF rule, the word ‘time’ corresponds to the full-time ABNF rule, and the word ‘datetime’ corresponds to the ‘date-time’ ABNF rule.

Email addresses formatted according to RFC 5322 [RFC5322] may be specified using the ‘email’ word, and E.123 phone numbers may be specified using the word ‘phone’.
Binary data can be specified in string form using the encodings specified in RFC 4648 [RFC4648]. The word ‘hex’ corresponds to base16, while ‘base32’, ‘base32hex’, ‘base64’, and ‘base64url’ correspond with their RFC 4648 counterparts accordingly.

Figure 21

Figure 22
4.6. Any Type

It is possible to specify that a value can be of any type allowable by JSON using the word ‘any’. The ‘any’ type specifies any primitive type, array, or object.

4.7. Member Specifications

Member specifications define members of JSON objects. Unlike other type specifications, member specifications cannot be root rules and must be part of an object specification or preceded by a rule name assignment.

Member specifications consist of a member name specification followed by a colon character (‘:’) followed by either a subordinate component, which is either a rule name or a primitive, object, array, or group specification. Member name specifications can be given either as a quoted string using double quotes or as a regular expression using forward slash (‘/’ ) characters. Regular expressions are not implicitly anchored and therefore must have explicit anchors if needed.

;member name will exactly match "locationURI"
$location_uri = "locationURI" : uri

;member name will match "eth0", "eth1", ... "eth9"
$iface_mappings = /\eth[0-9]$ : ipv4

Figure 23

4.8. Object Specifications

Object specifications define JSON objects and are composed of zero or more subordinate components, each of which can be either a rule name, member specification, or group specification. The subordinate components are enclosed at the start with a left curly brace character (‘{‘) and at the end with a right curly brace character (‘}’).

Evaluation of the subordinate components of object specifications is as follows:

- No order is implied for the members of the object being evaluated.
- Subordinate components of the object specification are evaluated in the order they appear.
Each member of the object being evaluated can only match one subordinate component.

Any members not matched against a subordinate component are ignored.

The following examples illustrate matching of JSON objects to JCR object specifications.

As order is not implied for the members of objects under evaluation, the following rule will match the JSON in Figure 25 and Figure 26.

```json
{ "locationUri" : uri, "statusCode" : integer }
```

Figure 24

```
{ "locationUri" : "http://example.com", "statusCode" : 200 }
```

Figure 25

```
{ "statusCode" : 200, "locationUri" : "http://example.com" }
```

Figure 26
Because subordinate components of an object specification are evaluated in the order in which they are specified (i.e. left to right, top to bottom) and object members can only match one subordinate component of an object specification, the rule o1 below will not match against the JSON in Figure 28 but the rule o2 below will match it.

```plaintext
; zero or more members that match "p0", "p1", etc
; and a member that matches "p1"
$o1 = { /^p\d+$/ : integer *, "p1" : integer }

; a member that matches "p1" and
; zero or more members that match "p0", "p1", etc
$o2 = { "p1" : integer, /^p\d+$/ : integer * }
```

The first subordinate of rule o1 specifies that an object can have zero or more members (that is the meaning of "*", see Section 4.13) where the member name is the letter ‘p’ followed by a number (e.g. "p0", "p1", "p2"), and the second rule specifies a member with the exact member name of "p1". Rule o2 has the exact same member specifications but in the opposite order. Figure 28 does not match rule o1 because all of the members match the first subordinate rule leaving none to match the second subordinate rule. However, rule o2 does match because the first subordinate rule matches only one member of the JSON object allowing the second subordinate rule to match the other member of the JSON object.

Figure 27

```plaintext
{ "p0" : 1, "p1" : 2 }
```

Figure 28

As stated above, members of objects which do not match a rule are ignored. The reason for this validation model is due to the nature of the typical access model to JSON objects in many programming languages, where members of the object are obtained by referencing the member name. Therefore extra members may exist without harm.

However, some specifications may need to restrict the members of a JSON object to a known set. To construct a rule specifying that no extra members are expected, the `{not}` annotation (see Section 4.14) may be used with a "match-all" regular expression as the last subordinate component of the object specification.
The following rule will match the JSON object in Figure 30 but will not match the JSON object in Figure 31.

```json
{ "foo" : 1, "bar" : 2, @not // : any + }
```

Figure 29

```json
{ "foo" : 1, "bar" : 2 }
```

Figure 30

```json
{ "foo" : 1, "bar" : 2, "baz" : 3 }
```

Figure 31

This works because subordinate components are evaluated in the order they appear in the object rule, and the last component accepts any member with any type but fails to validate if one or more of those components are found due to the @not annotation.

4.9. Array Specifications

Array specifications define JSON arrays and are composed of zero or more subordinate components, each of which can either be a rule name or a primitive, array, object or group specification. The subordinate components are enclosed at the start with a left square brace character ('[') and at the end with a right square brace character (']').

Evaluation of the subordinate components of array specifications is as follows:

- The order of array items is implied unless the @unordered annotation is present.
- Subordinate components of the array specification are evaluated in the order they appear.
- Each item of the array being evaluated can only match one subordinate component of the array specification.
- If any items of the array are not matched, then the array does not match the array specification.

These rules are further explained in the examples below.
Unlike object specifications, order is implied in array specifications by default. That is, the first subordinate component will match the first element of the array, the second subordinate component will match the second element of the array, and so on.

Take for example the following ruleset:

```plaintext
; the first element of the array is to be a string
; the second element of the array is to be an integer
$a1 = [ string, integer ]
```

```plaintext
; the first element of the array is to be an integer
; the second element of the array is to be a string
$a2 = [ integer, string ]
```

It defines two rules, a1 and a2. The array in the following JSON will not match a1, but will match a2.

```json
[ 24, "Bob Smurd" ]
```

If an array has more elements than can be matched from the array specification, the array does not match the array specification. Or stated differently, an array with unmatched elements does not validate. Using the example array rule a2 from above, the following array does not match because the last element of the array does not match any subordinate component:

```json
[ 24, "Bob Smurd", "http://example.com/bob_smurd" ]
```

To allow an array to contain any value after guaranteeing that it contains the necessary items, the last subordinate component of the array specification should accept any item:
; the first element of the array is to be an integer
; the second element of the array is to be a string
; anything else can follow
$a3 = [ integer, string, any * ]

The JSON array in Figure 35 will validate against the a3 rule in this example.

Figure 36

4.9.1. Unordered Array Specifications

Array specifications can be made to behave in a similar fashion to object specifications with regard to the order of matching with the @unordered annotation.

In the ruleset below, a1 and a2 have the same subordinate components given in the same order. a2 is annotated with the @unordered annotation.

```
$a1 = [ string, integer ]
$a2 = @unordered [ string, integer ]
```

Figure 37

The JSON array below does not match a1 but does match a2.

```
[ 24, "Bob Smurd" ]
```

Figure 38

Like ordered array specifications, the subordinate components in an unordered array specification are evaluated in the order they are specified. The difference is that they need not match an element of the array in the same position as given in the array specification.

Finally, like ordered array specifications, unordered array specifications also require that all elements of the array be matched by a subordinate component. If the array has more elements than can be matched, the array does not match the array specification.

4.10. Group Specifications

Unlike the other type specifications, group specifications have no direct tie with JSON syntax. Group specifications simply group together their subordinate components. Group specifications enclose one or more subordinate components with the parenthesis characters.
Group specifications and any nesting of group specifications, must conform to the allowable set of type specifications of the type specifications in which they are contained. For example, a group specification inside of an array specification may not contain a member specification since member specifications are not allowed as direct subordinates of array specifications (arrays contain values, not object members in JSON). Likewise, a group specification referenced inside an object specification must only contain member specifications (JSON objects may only contain object members).

The following is an example of a group specification:

```json
$the_brady = [ $parents, $children ]

$children = ( "Greg", "Marsha", "Bobby", "Jan" )
$parents = ( "Mike", "Carol" )
```

Figure 39

Like the subordinate components of array and object specifications, the subordinate components of a group specification are evaluated in the order they appear.

4.11. Ordered and Unordered Groups in Arrays

Section 4.9.1 specifies that arrays can be evaluated by the order of the items in the array or can be evaluated without order. Section 4.10 specifies that arrays may have group rules as subordinate components.

The evaluation of a group specification inside an array specification inherits the ordering property of the array specification. If the array specification is unordered, then the items of the group specification are also considered to be unordered. And if the array specification is ordered, then the items of the group specification are also considered to be ordered.

4.12. Sequence and Choice Combinations in Array, Object, and Group Specifications

Combinations of subordinate components in array, object, and group specifications can be specified as either a sequence ("and") or a choice ("or"). A sequence is a subordinate component followed by the comma character (',') followed by another subordinate component. A choice is a subordinate component followed by a pipe character ('|') followed by another subordinate component.
; sequence ("and")
[ "this", "that" ]

; choice ("or")
[ "this" | "that" ]

Figure 40

Sequence and choice combinations cannot be mixed, and group specifications must be used to explicitly declare precedence between a sequence and a choice. Therefore, the following is illegal:

[ "this", "that" | "the_other" ]

Figure 41

The example above should be expressed as:

[ "this", ( "that" | "the_other" ) ]

Figure 42

NOTE: A future specification will clarify the choice ('|') operation as inclusive or, exclusive or ("xor") or otherwise. At present readers should assume the choice ('|') operator is an inclusive or. However, for objects and unordered arrays that is not ideal, nor is xor. We are in the process of defining an algorithm to "rewrite" choices of rules for use with inclusive or which is more suitable for the data model of JSON.

4.13. Repetition in Array, Object, and Group Specifications

Evaluation of subordinate components in array, object, and group specifications may be succeeded by a repetition expression denoting how many times the subordinate component should be evaluated. Repetition expressions are specified using a Kleene symbol ('?', '+', or '*') or with the '*' symbol succeeded by specific minimum and/or maximum values, each being non-negative integers. Repetition expressions may also be appended with a step expression, which is the '%' symbol followed by a positive integer.

When no repetition expression is present, both the minimum and maximum are 1.

A minimum and maximum can be expressed by giving the minimum followed by two period characters ('..') followed by the maximum, with either the minimum or maximum being optional. When the minimum is not
explicitly specified, it is assumed to be zero. When the maximum is
not explicitly specified, it is assumed to be positive infinity.

```plaintext
; exactly 2 octets
$word = [ $octet *2 ]
$octet =: int8

; 1 to 13 name servers
[ $name_servers *1..13 ]
$name_servers =: fqdn

; 0 to 99 ethernet addresses
{ /^eth.*/ : $mac_addr *..99 }
$mac_addr =: hex

; four or more bytes
[ $octet *4.. ]
```

Figure 43

The allowable Kleene operators are the question mark character ('?')
which specifies zero or one (i.e. optional), the plus character ('+')
which specifies one or more, and the asterisk character ('*') which
specifies zero or more.

```plaintext
; age is optional
{ "name" : string, "age" : integer ? }

; zero or more errors
$error_set = ( string * )

; 1 or more integer values
[ integer + ]
```

Figure 44

A repetition step expression may follow a minimum to maximum
expression or the zero or more Kleene operator or the one or more
Kleene operator.

- When the repetition step follows a minimum to maximum expression
  or the zero or more Kleene operator ('*'), it specifies that the
total number of repetitions present in the JSON instance being
validated minus the minimum repetition value must be a multiple of
the repetition step (e.g. the total repetitions minus the minimum
repetition value must be divisible by the step value with a
remainder of zero).
When the repetition step follows a one or more Kleene operator ('+'), the minimum repetition value is set equal to the repetition step value and the total number of repetitions minus the step value must be a multiple of the repetition step value.

The following is an example for repetition steps in repetition expressions.

```json
; there must be at least 2 name servers
; there may be no more than 12 name servers
; there must be an even number of name servers
; e.g. 2, 4, 6, 8, 10, 12
[ $name_servers *2..12%2 ]
$name_servers =: fqdn

; minimum is zero
; maximum is 100
; must be an even number
{ /^eth.*/ : $mac_addr *..100%2 }
$mac_addr =: hex

; at least 32 octets
; must be be in groups of 16
; e.g. 32, 48, 64 etc
[ $octet *32..%16 ]
$octet =: int8

; if there are to be error sets,
; their number must be divisible by 4
; e.g. 0, 4, 8, 12 etc
$error_set = ( string *%4 )

; Throws of a pair of dice must be divisible by 2
; e.g. 2, 4, 6 etc
$dice_throws = ( 1..6 +%2 )
```

Figure 45


The evaluation of a rule can be changed with the `{not}` annotation. With this annotation, a rule that would otherwise match does not, and a rule that would not have matched does.
5. Directives

Directives modify the processing of a ruleset. There are two forms of the directive, the single line directive and the multi-line directive.

Single line directives appear on their own line in a ruleset, begin with a hash character ("#") and are terminated by the end of the line. They take the following form:

```
# directive_name parameter_1 parameter_2 ...
```

Figure 47

Multi-line directives also appear on their own lines, but may span multiple lines. They begin with the character sequence "#{" and end with "}". The take the following form:

```
#{ directive_name
    parameter_1 parameter_2
    parameter_3
    ...
}
```

Figure 48

This specification defines the directives "jcr-version", "ruleset-id", and "import", but other directives may be defined.

5.1. jcr-version

This directive declares that the ruleset complies with a specific version of this standard. The version is expressed as a major integer followed by a period followed by a minor integer.

```
# jcr-version 0.7
```

Figure 49
The major.minor number signifying compliance with this document is "0.7". Upon publication of this specification as an IETF proposed standard, it will be "1.0".

```
# jcr-version 1.0

Figure 50
```

Ruleset authors are advised to place this directive as the first line of a ruleset.

This directive may have optional extension identifiers following the version number. Each extension identifiers is preceded by the plus ('+') character and separated by white space. The format of extension identifiers is specific to the extension, but it is recommended that they are terminated by a version number.

```
# jcr-version 1.0 +co-constraints-1.2 +jcr-doc-1.0

Figure 51
```

5.2. ruleset-id

This directive identifies a ruleset to rule processors. It takes the form:

```
# ruleset-id identifier

Figure 52
```

An identifier can be a URL (e.g. http://example.com/foo), an inverted domain name (e.g. com.example.foo) or any other form that conforms to the JCR ABNF syntax that a ruleset author deems appropriate. To a JCR processor the identifier is treated as an opaque, case-sensitive string.

5.3. import

The import directive specifies that another ruleset is to have its rules evaluated in addition to the ruleset where the directive appears.

The following is an example:

```
# import http://example.com/rfc9999 as rfc9999

Figure 53
```
The rule names of the ruleset to be imported may be referenced by prepending the alias followed by a period character (‘.’) followed by the rule name (i.e. "alias.name"). To continue the example above, if the ruleset at http://example.com/rfc9999 were to have a rule named ‘encoding’, rules in the ruleset importing it can refer to that rule as ‘rfc9999.encoding’.

6. Tips and Tricks

6.1. Any Member with Any Value

Because member names may be specified with regular expressions, it is possible to construct a member rule that matches any member name. As an example, the following defines an object with a member with any name that has a value that is a string:

```json
{  // : string }
```

Figure 54

The JSON below matches the above rule.

```json
{  "foo" : "bar" }
```

Figure 55

Likewise, the JSON below also matches the same rule.

```json
{  "fuzz" : "bazz" }
```

Figure 56

Constructing an object with a member of any name with any type would therefore take the form:

```json
{  // : any }
```

Figure 57

The above rule matches not only the two JSON objects above, but the JSON object below.

```json
{  "fuzz" : 1234 }
```

Figure 58
6.2. Lists of Values

Group specifications may be used to create enumerated lists of primitive data types, because primitive specifications may contain a group specification, which may have multiple primitive specifications. Because a primitive specification must resolve to a single data type, the group specification must only contain choice combinations.

Consider the following examples:

```json
; either an IPv4 or IPv6 address
$address =: ( ipv4 | ipv6 )

; allowable fruits
$fruits =: ( "apple" | "banana" | "pear" )
```

Figure 59

6.3. Groups in Arrays

Groups may be a subordinate component of array specifications:

```json
[ ( ipv4 | ipv6 ), integer ]
```

Figure 60

Unlike primitive specifications, subordinate group specifications in array specifications may have sequence combinations and contain any type specification.

```json
; a group in an array
[ ( $first_name, $middle_name ?, $last_name ), $age ]

; a group referenced from an array
[ $name, $age ]
$name = ( $first_name, $middle_name ?, $last_name )

$first_name =: string
$middle_name =: string
$last_name =: string
$age =: 0..
```

Figure 61
6.4. Groups in Objects

Groups may be a subordinate component of object specifications: Subordinate group specifications in object specifications may have sequence combinations but must only contain member specifications.

; a group in an object
{ ( $title, $date, $author ), $paragraph + }

; a group referenced from an object
{ $front_matter, $paragraph + }
$front_matter = ( $title, $date, $author )

$title = "title" : string
$date = "date" : date
$author = "author" : [ string * ]
$paragraph = /p[0-9]*/ : string

Figure 62

NOTE: A future specification will clarify the choice ('|') operation as inclusive or, exclusive or ("xor") or otherwise. At present readers should assume the choice ('|') operator is an inclusive or. We are in the process of defining an algorithm to "rewrite" choices of rules for use with inclusive or which is more suitable for the data model of JSON. Such a change will impact the guidance given below.

When using groups to use both sequences and choices of member specifications, consideration must be given to the processing of object specifications where unmatch member specifications are ignored (see Figure 23).

A casual reading of this rule might lead a reader to believe that the JSON object in Figure 64 would not match, however it does because the extra member (either "foo" or "baz") is not matched but is ignored.

( "bar" : string, { "foo" : integer | "baz" : string } )

Figure 63

( "bar" : "thing", "foo" : 2, "baz" : "thingy" )

Figure 64

The rule in Figure 63 must be modified to either match all extra rules, as in Figure 65, or the logic of the rules must be rewritten
to explicitly negate the presence of the unwanted members, as in Figure 66.

```json
{ "bar":string, ( "foo":integer | "baz":string ), @{not} //:any + }
```

Figure 65

```json
{ "bar":string,
  ( ( "foo":integer , @{not} "baz":string ) |
    ( "baz":string , @{not} "foo":integer )
  )
}
```

Figure 66

6.5. Group Rules as Macros

The syntax for group specifications accommodates one or more subordinate components and a repetition expression for each. Other than grouping multiple rules, a group specification can be used as a macro definition for a single rule.

```json
$paragraphs = ( /p[0-9]*/ : string + )
```

Figure 67

6.6. Object Mixins

Group rules can be used to create object mixins, a pattern for writing data models similar in style to object derivation in some programming languages. In the example in below, both obj1 and obj2 have a members "foo" and "fob" with obj1 having the additional member "bar" and obj2 having the additional member "baz".

```json
$mixin_group = ( "foo" : integer, "fob" : uri )

$obj1 = { $mixin_group, "bar" : string }

$obj2 = { $mixin_group, "baz" : string }
```

Figure 68

6.7. Subordinate Dependencies

In object and array specifications, there may be situations in which it is necessary to condition the existence of a subordinate component on the existence of a sibling subordinate component. In other words, example_two should only be evaluated if example_one evaluates positively. Or put another way, a member of an object or an item of
an array may be present only on the condition that another member or item is present.

In the following example, the referrer_uri member can only be present if the location_uri member is present.

```json
; $referrer_uri can only be present if
; $location_uri is present
{ ( $location_uri, $referrer_uri? )? }

$location_uri = "locationURI" : uri
$referrer_uri = "referrerURI" : uri
```

Figure 69

7. Implementation Status

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7492]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7492], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

7.1. JCR Validator

The JCR Validator, written in Ruby, currently implements all portions of this specification, and has been used extensively to prototype various aspects of JCR under consideration. It’s development has gone hand-in-hand with this specification.

This software is primarily produced by the American Registry for Internet Numbers (ARIN) and freely distributable under the ISC license.
Source code for this software is available on GitHub at <https://github.com/arineng/jcrvalidator>. This software is also easily obtained as a Ruby Gem through the Ruby Gem system.

7.2. Codalogic JCR Parser

The Codalogic JCR Parser is a C++ implementation of a JCR parsing engine, and is a work in progress. It is targeted for the Windows platform.

This software is produced by Codalogic Ltd and freely distributable under the Gnu LGPL v3 license.

Source code is available on GitHub at <https://github.com/codalogic/cl-jcr-parser>.

7.3. JCR Java

JCR Java is a work in progress and currently only implements the parsing of JCR rulesets according to the ABNF using a custom parsing framework.

This software is produced by the American Registry for Internet Numbers (ARIN) and freely distributable under the MIT license.

Source code is available on BitBucket at <https://bitbucket.org/anewton_1998/jcr_java>.

8. ABNF Syntax

The following ABNF describes the syntax for JSON Content Rules. A text file containing these ABNF rules can be downloaded from [JCR_ABNF].

```
jcr          = *( sp-cmt / directive / root-rule / rule )
sp-cmt       = spaces / comment
spaces       = 1*( WSP / CR / LF )
DSPs         = ; Directive spaces
              1*WSP / ; When in one-line directive
              1*sp-cmt ; When in multi-line directive
comment      = ";" *comment-char comment-end-char
comment-char = HTAB / %x20-10FFFF
              ; Any char other than CR / LF
comment-end-char = CR / LF
directive    = ";" (one-line-directive / multi-line-directive)
one-line-directive = [ DSPs ]
```
(directive-def / one-line-tbd-directive-d)

*WSP eol
multi-line-directive = "{" *sp-cmt
    ( directive-def /
    multi-line-tbd-directive-d )
*sp-cmt "}

directive-def = jcr-version-d / ruleset-id-d / import-d
jcr-version-d = jcr-version-fw DSPs major-version
    "." minor-version
    *( DSPs "+" [ DSPs ] extension-id )
major-version = non-neg-integer
minor-version = non-neg-integer
extension-id = ALPHA *not-space
ruleset-id-d = ruleset-id-fw DSPs ruleset-id
import-d = import-fw DSPs ruleset-id
    [ DSPs as-fw DSPs ruleset-id-alias ]
ruleset-id = ALPHA *not-space
not-space = \x21-10FFFF
ruleset-id-alias = name
one-line-tbd-directive-d = directive-name
    [ WSP one-line-directive-parameters ]
directive-name = name
one-line-directive-parameters = *not-eol
not-eol = HTAB / \x20-10FFFF
eol = CR / LF
multi-line-tbd-directive-d = directive-name
    [ 1*sp-cmt multi-line-directive-parameters ]
multi-line-directive-parameters = multi-line-parameters
multi-line-parameters = *(comment / q-string / regex / not-multi-line-special)
not-multi-line-special = spaces / \x21 / \x23-2E / \x30-3A / \x3C-7C / \x7E-10FFFF ; not ", /, ; or }

root-rule = value-rule / group-rule
rule = annotations "$" rule-name *sp-cmt
      "+" *sp-cmt rule-def
rule-name = name
target-rule-name = annotations "$"
    [ ruleset-id-alias "." ]
rule-name = ALPHA *( ALPHA / DIGIT / "-" / "-" )

rule-def = member-rule / type-designator rule-def-type-rule /
        array-rule / object-rule / group-rule /
        target-rule-name

rule-name = name
rule-def-type-rule = type-fw 1*sp-cmt / ":" *sp-cmt
rule-def-type-rule = value-rule / type-choice
value-rule = primitive-rule / array-rule / object-rule
member-rule = annotations
  member-name-spec *sp-cmt ":" *sp-cmt type-rule
member-name-spec = regex / q-string
type-rule = value-rule / type-choice / target-rule-name
type-choice = annotations "(" type-choice-items *( choice-combiner type-choice-items ) ")"
explicit-type-choice = type-designator type-choice
type-choice-items = *sp-cmt { type-choice / type-rule } *sp-cmt
annotations = *{( "@{" *sp-cmt annotation-set *sp-cmt "}"
  *sp-cmt }
annotation-set = not-annotation / unordered-annotation /
  root-annotation / tbd-annotation
not-annotation = not-kw
unordered-annotation = unordered-kw
root-annotation = root-kw
tbd-annotation = annotation-name [ spaces annotation-parameters ]
anotation-name = name
annotation-parameters = multi-line-parameters
primitive-rule = annotations primitive-def
primitive-def = string-type / string-range / string-value /
  null-type / boolean-type / true-value /
  false-value / double-type / float-type /
  float-range / float-value /
  integer-type / integer-range / integer-value /
  sized-int-type / sized-uint-type / ipv4-type /
  ipv6-type / ipaddr-type / fqdn-type / idn-type /
  uri-type / phone-type / email-type /
  datetime-type / date-type / time-type /
  hex-type / base32hex-type / base32-type /
  base64url-type / base64-type / any
null-type = null-kw
boolean-type = boolean-kw
true-value = true-kw
false-value = false-kw
string-type = string-kw
string-value = q-string
string-range = regex
double-type = double-kw
float-type = float-kw
float-range = float-min "." [ float-max ] / "." float-max
float-min = float
float-max = float
float-value = float
integer-type = integer-kw
integer-range = integer-min ".." [ integer-max ] / ".." integer-max
integer-min = integer
integer-max = integer
integer-value = integer
sized-int-type = int-kw pos-integer
sized-uint-type = uint-kw pos-integer
ipv4-type = ipv4-kw
ipv6-type = ipv6-kw
ipaddr-type = ipaddr-kw
fqdn-type = fqdn-kw
idn-type = idn-kw
uri-type = uri-kw [ ".." uri-scheme ]
phone-type = phone-kw
email-type = email-kw
datetime-type = datetime-kw
date-type = date-kw
time-type = time-kw
hex-type = hex-kw
base32hex-type = base32hex-kw
base32-type = base32-kw
base64url-type = base64url-kw
base64-type = base64-kw
any = any-kw

object-rule = annotations "{" *sp-cmt
  [ object-items *sp-cmt ] "}"
object-items = object-item [ 1*( sequence-combiner object-item ) /
  1*( choice-combiner object-item ) ]
object-item = object-item-types *sp-cmt [ repetition *sp-cmt ]
object-item-types = object-group / member-rule / target-rule-name
object-group = annotations "(" *sp-cmt [ object-items *sp-cmt ] ")"

array-rule = annotations "[" *sp-cmt
  [ array-items *sp-cmt ] "]"
array-items = array-item [ 1*( sequence-combiner array-item ) /
  1*( choice-combiner array-item ) ]
array-item = array-item-types *sp-cmt [ repetition *sp-cmt ]
array-item-types = array-group / type-rule / explicit-type-choice
array-group = annotations "(" *sp-cmt [ array-items *sp-cmt ] ")"

group-rule = annotations "(" *sp-cmt
  [ group-items *sp-cmt ] ")"
group-items = group-item [ 1*( sequence-combiner group-item ) /
  1*( choice-combiner group-item ) ]
group-item = group-item-types *sp-cmt [ repetition *sp-cmt ]
group-item-types = group-group / member-rule /
  type-rule / explicit-type-choice
group-group = group-rule
sequence-combiner = "," *sp-cmt
choice-combiner  = "|" *sp-cmt

repetition       = optional / one-or-more / 
                  repetition-range / zero-or-more
optional         = "?"
one-or-more      = "+" [ repetition-step ]
zero-or-more     = "*" [ repetition-step ]
repetition-range = "*" *sp-cmt ( 
                  min-max-repetition / min-repetition / 
                  max-repetition / specific-repetition )
min-max-repetition = min-repeat ".." max-repeat 
                    [ repetition-step ]
min-repetition   = min-repeat ".." [ repetition-step ]
max-repetition   = ".." max-repeat [ repetition-step ]
max-repeat       = non-neg-integer
specific-repetition = non-neg-integer
repetition-step  = "%" step-size
step-size        = non-neg-integer

integer         = "0" / ["-" ] pos-integer
non-neg-integer = "0" / pos-integer
pos-integer     = digit1-9 *DIGIT

float           = [ minus ] int frac [ exp ]
                   ; From RFC 7159 except 'frac' required
minus           = %x2D ; -
plus            = %x2B ; +
int             = zero / ( digit1-9 *DIGIT )
digit1-9        = %x31-39 ; 1-9
frac            = decimal-point 1*DIGIT
decimal-point   = %x2E ; .
exp             = e [ minus / plus ] 1*DIGIT
e              = %x65 / %x45 ; e E
zero            = %x30 ; 0

q-string        = quotation-mark *char quotation-mark
                   ; From RFC 7159
char            = unescaped /
                   escape ( 
%22 / ; " quotation mark U+0022
%5C / \ reverse solidus U+005C
%2F / / solidus U+002F
%62 / ; b backspace U+0008
%66 / ; f form feed U+000C
%6E / ; n line feed U+000A
%72 / ; r carriage return U+000D
escape = %x5C ; \n
quotation-mark = %x22 ; "

unescaped = %x20-21 / %x23-5B / %x5D-10FFFF

regex = /^(/( escape "/" / not-slash ) "/")

[ regex-modifiers ]

not-slash = HTAB / CR / LF / %x20-2E / %x30-10FFFF
;

Any char except "/"

regex-modifiers = *( "i" / "s" / "x" )

uri-scheme = 1*ALPHA


;; Keywords

any-kw = %x61.6E.79 ; "any"
as-kw = %x61.73 ; "as"
base32-kw = %x62.61.73.65.33.32 ; "base32"
base32hex-kw = %x62.61.73.65.33.32.68.65.78 ; "base32hex"
base64-kw = %x62.61.73.65.36.34 ; "base64"
base64url-kw = %x62.61.73.65.36.34.75.72.6C ; "base64url"
boolean-kw = %x62.6F.6F.6C.65.61.6E ; "boolean"
date-kw = %x64.61.74.65 ; "date"
datetime-kw = %x64.61.74.65.74.69.6D.65 ; "datetime"
double-kw = %x64.6F.75.62.6C.65 ; "double"
email-kw = %x65.6D.61.69.6C ; "email"
false-kw = %x66.61.6C.73.65 ; "false"
float-kw = %x66.6C.6F.61.74 ; "float"
fqdn-kw = %x66.71.64.6E ; "fqdn"
hex-kw = %x68.65.78 ; "hex"
import-kw = %x69.6D.70.6F.72.74 ; "import"
int-kw = %x69.6E.74 ; "int"
integer-kw = %x69.6E.74.65.67.65.72 ; "integer"
ipaddr-kw = %x69.70.61.64.64.72 ; "ipaddr"
ipv4-kw = %x69.70.76.34 ; "ipv4"
ipv6-kw = %x69.70.76.36 ; "ipv6"
jcr-version-kw = %x6A.63.72.2D.76.65.72.73.69.6F.6E ; "jcr-version"
not-kw = %x6E.6F.74 ; "not"
null-kw = %x6E.75.6C.6C ; "null"
phone-kw = %x70.68.6F.66.65 ; "phone"
root-kw = %x72.6F.66.74 ; "root"
ruleset-id-kw = %x72.75.6C.65.73.65.74.2D.69.64 ; "ruleset-id"
string-kw = %x73.74.72.69.6E.67 ; "string"
time-kw = %x74.69.6D.65 ; "time"
true-kw = %x74.72.75.65 ; "true"
type-kw = %x74.79.70.65 ; "type"
uint-kw = %x75.69.6E.74 ; "uint"
unordered-kw = %x75.6E.6F.72.64.65.72.65.64 ; "unordered"
uri-kw = %x75.72.69 ; "uri"

;;; Referenced RFC 5234 Core Rules
ALPHA = %x41-5A / %x61-7A ; A-Z / a-z
CR = %x0D ; carriage return
DIGIT = %x30-39 ; 0-9
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
HTAB = %x09 ; horizontal tab
LF = %x0A ; linefeed
SP = %x20 ; space
WSP = SP / HTAB ; white space

Figure 70: ABNF for JSON Content Rules

9. Acknowledgements

John Cowan, Andrew Biggs, Paul Kyzivat and Paul Jones provided feedback and suggestions which led to many changes in the syntax.

10. References

10.1. Normative References

[JCR_ABNF]


10.2. Informative References

[I-D.cordell-jcr-co-constraints]
Cordell, P. and A. Newton, "Co-Constraints for JSON Content Rules", draft-cordell-jcr-co-constraints-00 (work in progress), March 2016.


10.3. URIs


Appendix A. Co-Constraints

This specification defines a small set of annotations and directives for JCR, yet the syntax is extensible allowing for other annotations and directives. [I-D.cordell-jcr-co-constraints] ("Co-Constraints for JCR") defines further annotations and directives which define more detailed constraints on JSON messages, including co-constraints (constraining parts of JSON message based on another part of a JSON message).
Appendix B. Testing Against JSON Content Rules

One aspect of JCR that differentiates it from other format schema languages are the mechanisms helpful to developers for taking a formal specification, such as that found in an RFC, and evolving it into unit tests, which are essential to producing quality protocol implementations.

B.1. Locally Overriding Rules

As mentioned in the introduction, one tool for testing would be the ability to locally override named rules. As an example, consider the following rule which defines an array of strings.

\[
\$statuses = [ \text{string } * ]
\]

Figure 71

Consider the specification where this rule is found does not define the values but references an extensible list of possible values updated independently of the specification, such as in an IANA registry.

If a software developer desired to test a specific situation in which the array must at least contain the status "accepted", the rules from the specification could be used and the statuses rule could be explicitly overridden locally as:

This rule will evaluate positively with the JSON in Figure 73

\[
\$statuses = @\{unordered\} [ \"accepted\", \text{string } * ]
\]

Figure 72

\[
[ \"submitted\", \"validated\", \"accepted\" ]
\]

Figure 73

Alternatively, the developer may need to ensure that the status "denied" should not be present in the array:

This rule will fail to evaluate the JSON in Figure 75 thus signaling a problem.

\[
\$statuses = @\{unordered\} @\{not\} [ \"denied\" + , \text{string } * ]
\]

Figure 74
[ "submitted", "validated", "denied" ]

Figure 75

B.2. Rule Callbacks

In many testing scenarios, the evaluation of rules may become more complex than that which can be expressed in JCR, sometimes involving variables and interdependencies which can only be expressed in a programming language.

A JCR processor may provide a mechanism for the execution of local functions or methods based on the name of a rule being evaluated. Such a mechanism could pass to the function the data to be evaluated, and that function could return to the processor the result of evaluating the data in the function.

Appendix C. Changes from -07 and -08

This revision of the document makes no substantive changes to any parts of the specification. Some of the ABNF has been updated to more correctly allow group rules, and other small change have been made to the ABNF to make it simpler.

Authors’ Addresses

Andrew Lee Newton
American Registry for Internet Numbers
PO Box 232290
Centreville, VA  20120
US
Email: andy@arin.net
URI:   http://www.arin.net

Pete Cordell
Codalogic
PO Box 30
Ipswich  IP5 2WY
UK
Email: pete.cordell@codalogic.com
URI:   http://www.codalogic.com
Well-Known Uniform Resource Identifiers (URIs)
draft-nottingham-rfc5785bis-08

Abstract

This memo defines a path prefix for "well-known locations", "/.well-known/", in selected Uniform Resource Identifier (URI) schemes.

Note to Readers

_RFC EDITOR: please remove this section before publication_

This draft is a proposed revision of RFC5875.

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/rfc5785bis [1].

The most recent (often, unpublished) draft is at https://mnot.github.io/I-D/rfc5785bis/ [2].

Recent changes are listed at https://github.com/mnot/I-D/commits/gh-pages/rfc5785bis [3].

See also the draft’s current status in the IETF datatracker, at https://datatracker.ietf.org/doc/draft-nottingham-rfc5785bis/ [4].

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."
1. Introduction

Some applications on the Web require the discovery of information about an origin [RFC6454] (sometimes called "site-wide metadata") before making a request. For example, the Robots Exclusion Protocol (http://www.robotstxt.org/ [5]) specifies a way for automated processes to obtain permission to access resources; likewise, the Platform for Privacy Preferences [P3P] tells user-agents how to discover privacy policy before interacting with an origin server.
While there are several ways to access per-resource metadata (e.g., HTTP headers, WebDAV’s PROPFIND [RFC4918]), the perceived overhead (either in terms of client-perceived latency and/or deployment difficulties) associated with them often precludes their use in these scenarios.

At the same time, it has become more popular to use HTTP as a substrate for non-Web protocols. Sometimes, such protocols need a way to locate one or more resources on a given host.

When this happens, one solution is to designate a "well-known location" for data or services related to the origin overall, so that it can be easily located. However, this approach has the drawback of risking collisions, both with other such designated "well-known locations" and with resources that the origin has created (or wishes to create). Furthermore, defining well-known locations usurps the origin’s control over its own URI space [RFC7320].

To address these uses, this memo defines a path prefix in HTTP(S) URIs for these "well-known locations", "/.well-known/". Future specifications that need to define a resource for such metadata can register their use to avoid collisions and minimise impingement upon origins’ URI space.

Well-known URIs can also be used with other URI schemes, but only when those schemes’ definitions explicitly allow it.

2. Notational Conventions

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. Well-Known URIs

A well-known URI is a URI [RFC3986] whose path component begins with the characters "/.well-known/", and whose scheme is "http" [RFC7230], "https" [RFC7230], "ws" [RFC6455], "wss" [RFC6455], or another scheme that has explicitly been specified to use well-known URIs.

Applications that wish to mint new well-known URIs MUST register them, following the procedures in Section 5.1.

For example, if an application registers the name ‘example’, the corresponding well-known URI on ‘http://www.example.com/’ would be ‘http://www.example.com/.well-known/example’.
Registered names MUST conform to the segment-nz production in [RFC3986]. This means they cannot contain the "/" character.

Registered names for a specific application SHOULD be correspondingly precise; "squatting" on generic terms is not encouraged. For example, if the Example application wants a well-known location for metadata, an appropriate registered name might be "example-metadata" or even "example.com-metadata", not "metadata".

At a minimum, a registration will reference a specification that defines the format and associated media type(s) to be obtained by dereferencing the well-known URI, along with the URI scheme(s) that the well-known URI can be used with. If no URI schemes are explicitly specified, "http" and "https" are assumed.

Typically, applications will use the default port for the given scheme; if an alternative port is used, it MUST be explicitly specified by the application in question.

It MAY also contain additional information, such as the syntax of additional path components, query strings and/or fragment identifiers to be appended to the well-known URI, or protocol-specific details (e.g., HTTP [RFC7231] method handling).

Note that this specification defines neither how to determine the hostname to use to find the well-known URI for a particular application, nor the scope of the metadata discovered by dereferencing the well-known URI; both should be defined by the application itself.

Also, this specification does not define a format or media-type for the resource located at "/.well-known/" and clients should not expect a resource to exist at that location.

Well-known URIs are rooted in the top of the path’s hierarchy; they are not well-known by definition in other parts of the path. For example, "/.well-known/example" is a well-known URI, whereas "/foo/.well-known/example" is not.

See also Section 4 for Security Considerations regarding well-known locations.

3.1. Registering Well-Known URIs

The "Well-Known URIs" registry is located at "https://www.iana.org/assignments/well-known-uris/". Registration requests can be made by following the instructions located there or
by sending an email to the "wellknown-uri-review@ietf.org" mailing list.

Registration requests consist of at least the following information:

URI suffix: The name requested for the well-known URI, relative to "/.well-known/"; e.g., "example".

Change controller: For Standards-Track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s): Reference to the document that specifies the field, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

Status: One of "permanent" or "provisional". See guidance below.

Related information: Optionally, citations to additional documents containing further relevant information.

General requirements for registered relation types are described in Section 3.

Standards-defined values have a status of "permanent". Other values can also be registered as permanent, if the Experts find that they are in use, in consultation with the community. Other values should be registered as "provisional".

Provisional entries can be removed by the Experts if - in consultation with the community - the Experts find that they are not in use. The Experts can change a provisional entry’s status to permanent at any time.

Note that well-known URIs can be registered by third parties (including the expert(s)), if the expert(s) determines that an unregistered well-known URI is widely deployed and not likely to be registered in a timely manner otherwise. Such registrations still are subject to the requirements defined, including the need to reference a specification.

4. Security Considerations

Applications minting new well-known URIs, as well as administrators deploying them, will need to consider several security-related issues, including (but not limited to) exposure of sensitive data,
denial-of-service attacks (in addition to normal load issues), server
and client authentication, vulnerability to DNS rebinding attacks,
and attacks where limited access to a server grants the ability to
affect how well-known URIs are served.

4.1. Interaction with Web Browsing

Applications using well-known URIs for "http" or "https" URLs need to
be aware that well-known resources will be accessible to Web
browsers, and therefore are able to be manipulated by content
obtained from other parts of that origin. If an attacker is able to
inject content (e.g., through a Cross-Site Scripting vulnerability),
they will be able to make potentially arbitrary requests to the well-
known resource.

HTTP and HTTPS also use origins as a security boundary for many other
mechanisms, including (but not limited to) Cookies [RFC6265], Web
Storage [WEBSTORAGE] and many capabilities.

Applications defining well-known locations should not assume that
they have sole access to these mechanisms, or that they are the only
application using the origin. Depending on the nature of the
application, mitigations can include:

- Encrypting sensitive information
- Allowing flexibility in the use of identifiers (e.g., Cookie
  names) to avoid collisions with other applications
- Using the ‘HttpOnly’ flag on Cookies to assure that cookies are
  not exposed to browser scripting languages [RFC6265]
- Using the ‘Path’ parameter on Cookies to assure that they are not
  available to other parts of the origin [RFC6265]
- Using X-Content-Type-Options: nosniff [FETCH] to assure that
  content under attacker control can’t be coaxed into a form that is
  interpreted as active content by a Web browser

Other good practices include:

- Using an application-specific media type in the Content-Type
  header, and requiring clients to fail if it is not used
- Using Content-Security-Policy [CSP] to constrain the capabilities
  of active content (such as HTML [HTML5]), thereby mitigating
  Cross-Site Scripting attacks
o Using Referrer-Policy [REFERRER-POLICY] to prevent sensitive data in URLs from being leaked in the Referer request header

o Avoiding use of compression on any sensitive information (e.g., authentication tokens, passwords), as the scripting environment offered by Web browsers allows an attacker to repeatedly probe the compression space; if the attacker has access to the path of the communication, they can use this capability to recover that information.

4.2. Scoping Applications

This memo does not specify the scope of applicability for the information obtained from a well-known URI, and does not specify how to discover a well-known URI for a particular application.

Individual applications using this mechanism must define both aspects; if this is not specified, security issues can arise from implementation deviations and confusion about boundaries between applications.

Applying metadata discovered in a well-known URI to resources other than those co-located on the same origin risks administrative as well as security issues. For example, allowing "https://example.com/.well-known/example" to apply policy to "https://department.example.com", "https://www.example.com" or even "https://www.example.com:8000" assumes a relationship between hosts where there might be none, giving control to a potential attacker.

Likewise, specifying that a well-known URI on a particular hostname is to be used to bootstrap a protocol can cause a large number of undesired requests. For example, if a well-known HTTPS URI is used to find policy about a separate service such as e-mail, it can result in a flood of requests to Web servers, even if they don’t implement the well-known URI. Such undesired requests can resemble a denial-of-services attack.

4.3. Hidden Capabilities

Applications using well-known locations should consider that some server administrators might be unaware of its existence (especially on operating systems that hide directories whose names begin with "."). This means that if an attacker has write access to the .well-known directory, they would be able to control its contents, possibly without the administrator realising it.
5. IANA Considerations

5.1. The Well-Known URI Registry

This specification updates the registration procedures for the "Well-Known URI" registry, first defined in [RFC5785]; see Section 3.1.

Well-known URIs are registered on the advice of one or more experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from [RFC8126]).

The Experts’ primary considerations in evaluating registration requests are:

- Conformance to the requirements in Section 3
- The availability and stability of the specifying document
- The considerations outlined in Section 4

IANA will direct any incoming requests regarding the registry to this document and, if defined, the processes established by the expert(s); typically, this will mean referring them to the registry Web page.

Upon publication, IANA should:

- Replace all references to RFC 5988 in that registry have been replaced with references to this document.
- Update the status of all existing registrations to "permanent".

6. References

6.1. Normative References


6.2. Informative References


6.3. URIs


Appendix A. Frequently Asked Questions

Aren’t well-known locations bad for the Web? They are, but for various reasons — both technical and social — they are sometimes necessary. This memo defines a "sandbox" for them, to reduce the risks of collision and to minimise the impact upon pre-existing URIs on sites.

Why /.well-known? It’s short, descriptive, and according to search indices, not widely used.

What impact does this have on existing mechanisms, such as P3P and robots.txt? None, until they choose to use this mechanism.

Why aren’t per-directory well-known locations defined? Allowing every URI path segment to have a well-known location (e.g.,
"/images/.well-known") would increase the risks of colliding with a pre-existing URI on a site, and generally these solutions are found not to scale well, because they’re too "chatty".

Appendix B. Changes from RFC5785

- Allow non-Web well-known locations
- Adjust IANA instructions
- Update references
- Various other clarifications
- Add "ws" and "wss" schemes

Author’s Address

Mark Nottingham

Email: mnot@mnot.net
URI: https://www.mnot.net/