Abstract

This document identifies a minimum set of standards and requirements that must be supported by a Video Relay Service (VRS) Video Access Technology Reference Platform (VATRP)-compliant client and United States Telecommunications Relay Service providers required to be VATRP compliant. This Relay User Equipment specification only specifies a minimum set of requirements. It does not prohibit VRS providers or endpoint developers from developing or deploying additional capabilities, provided that doing so will not prevent compliance with the requirements specified here.

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 August 5, 2019.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document.
Table of Contents

1. Introduction ....................................... 3
2. Scope .................................................. 3
3. Terminology ............................................. 3
4. Requirements Language .................................. 6
5. General Requirements .................................... 6
6. SIP Signaling ............................................ 6
   6.1. Registration ......................................... 7
   6.2. Session Establishment ............................... 8
   6.2.1. Normal Call Origination ......................... 8
   6.2.2. One-Stage Dial-Around Origination ............... 9
   6.2.3. RUE Contact Information ......................... 10
   6.2.4. Incoming Calls .................................. 10
   6.2.5. Emergency Calls ................................ 11
   6.3. Mid Call Signaling .................................. 11
   6.4. URI Representation of Phone Numbers .............. 11
   6.5. NAT Traversal ...................................... 12
7. Media .................................................... 12
   7.1. Text-Based Communication .......................... 12
   7.2. Video Codecs ....................................... 12
   7.3. Audio Codecs ....................................... 13
   7.4. DTMF Digits ........................................ 13
   7.5. Session Description Protocol ...................... 13
   7.6. Privacy ............................................. 13
8. SRTP and SRTCP ........................................... 13
   8.1. Bandwidth Negotiation Flow Control and Media Performance 13
   8.2. SRTP / SAVPF Profile for SRTCP Feedback .......... 14
   8.3. Negative Acknowledgment, Packet Loss Indicator, and Full Intraframe Request Features .......... 14
9. Contacts ................................................ 14
   9.1. CardDAV Login and Synchronization ................ 14
   9.2. Contacts Import/Export Service .................... 15
10. Mail Waiting Indicator (MWI) .......................... 15
11. Provisioning and Provider Selection .................... 15
   11.1. RUE Provider Selection ............................ 15
   11.2. RUE Configuration Service ........................ 17
   11.3. Schemas .......................................... 20
12. Acknowledgements ...................................... 23
13. IANA Considerations ................................... 23
14. Security Considerations ................................ 23
1. Introduction

Video Relay Service (VRS) is a form of Telecommunications Relay Service (TRS) that enables persons with hearing disabilities who use sign language, such as American Sign Language (ASL), to communicate with voice telephone users through video equipment. These services also enable communication between such individuals directly in suitable modalities, including any combination of sign language via video, real-time text (RTT), and speech.

This Interoperability Profile for Relay User Equipment (RUE) is a profile of the Session Initiation Protocol (SIP) and related media protocols that enables end-user equipment registration and calling for VRS calls. It specifies the minimal set of call flows, Internet Engineering Task Force (IETF) and ITU-T standards that must be supported, provides guidance where the standards leave multiple implementation options, and specifies minimal and extended capabilities for RUE calls.

This RUE profile supports the requirements of relay services in the United States, as described in 47 CFR 64.601 et seq., but may be applicable to similar uses elsewhere.

2. Scope

This RUE Specification documents the standards and controls associated with the Video Access Technology Reference Platform (VATRP). This RUE specification identifies the minimum set of standards for the interface between the VATRP and Providers’ networks to which the VATRP adheres. This RUE Specification does not prohibit the implementation of additional features or functionality by any Provider. It also contains some Provider-optional features. If a Provider offers the feature described by the optional specification on at least one endpoint, the Provider MUST supply the standardized interface described in this document for that feature. This edition of the RUE specification does not address Provider-to-Provider communication (covered in the US VRS Provider Interface Profile) or the user interface to the RUE.

3. Terminology

Communication Assistant (CA): The ASL interpreter stationed in a TRS-registered call center working for a VRS Provider, acting as part of the wire of a call to provide functionally equivalent phone service.
Communication modality (modality): A specific form of communication that may be employed by two users, e.g., English voice, Spanish voice, American Sign Language, English lip-reading, or French real-time-text. Here, one communication modality is assumed to encompass both the language and the way that language is exchanged. For example, English voice and French voice are two different communication modalities.

Default video relay service: The video relay service operated by a subscriber’s default VRS provider.

Default video relay service Provider (default Provider): The VRS provider that registers, and assigns a telephone number to, a specific subscriber. A subscriber’s default Provider provides the VRS that handles incoming relay calls to the user. The default Provider also handles outgoing relay calls by default.

Dial-around call: A relay call where the subscriber specifies the use of a VRS provider other than one of the Providers with whom the subscriber is registered. This can be accomplished by the user dialing a "front-door" number for a VRS provider and signing or texting a phone number to call ("two-stage"). Alternatively, this can be accomplished by the user's RUE software instructing the server of its default VRS provider to automatically route the call through the alternate Provider to the desired public switched telephone network (PSTN) directory number ("one-stage").

Full Intra Request (FIR): A request to a media sender, requiring that media sender to send a Decoder Refresh Point at the earliest opportunity. FIR is sometimes known as "instantaneous decoder refresh request", "video fast update request", or "fast update request".

NANP: North America Numbering Plan (please refer to: http://nationalnanpa.org).

Point-to-Point Call (P2P Call): A call between two RUEs, without including a CA.

Relay call: A call that allows persons with hearing or speech disabilities to use a RUE to talk to users of traditional voice services with the aid of a communication assistant (CA) to relay the communication. Please refer to FCC-VRS-GUIDE.

Relay number database (RND): The iTRS Relay Number Database (RND) functions as a 10-digit NANP phone number lookup for SIP and H.323 URLs for TRS subscribers.
Relay-to-relay call: A call between two subscribers each using different forms of relay (video relay, IP relay, TTY), each with a separate CA to assist in relaying the conversation.

Relay service (RS): A service that allow a registered subscriber to use a RUE to make and receive relay calls, point-to-point calls, and relay-to-relay calls. The functions provided by the relay service include the provision of media links supporting the communication modalities used by the caller and callee, and user registration and validation, authentication, authorization, automatic call distributor (ACD) platform functions, routing (including emergency call routing), call setup, mapping, call features (such as call forwarding and video mail), and assignment of CAs to relay calls.

Relay service Provider (Provider): An organization that operates a relay service. A subscriber selects a relay service Provider to assign and register a telephone number for their use, to register with for receipt of incoming calls, and to provide the default service for outgoing calls.

Relay user: Please refer to "subscriber".

Relay user E.164 Number (user E.164): The telephone number assigned to the RUE in ITU-T E.164 format.

Relay user equipment (RUE): A SIP user agent (UA) enhanced with extra features to support a subscriber in requesting and using relay calls. A RUE may take many forms, including a stand-alone device; an application running on a general-purpose computing device such as a laptop, tablet or smart phone; or proprietary equipment connected to a server that provides the RUE interface.

Sign language: A language that uses hand gestures and body language to convey meaning including, but not limited to, American Sign Language (ASL).

Subscriber: An individual who has registered with a Provider and who obtains service by using relay user equipment. This is the traditional telecom term for an end-user customer, which in our case is a relay user.

Telecommunications relay services (TRS): Telephone transmission services that provide the ability for an individual who has a hearing impairment or speech impairment to engage in communication by wire or radio with a hearing individual in a manner that is functionally equivalent to the ability of an individual who does not have a hearing impairment or speech impairment to communicate using voice communication services by wire or radio. TRS includes services that
enable two-way communication between an individual who uses a
Telecommunications Device for the Deaf (TDD) or other non-voice
terminal device and an individual who does not use such a device.

Video relay service (VRS): A relay service for people with hearing or
speech disabilities who use sign language to communicate using video
equipment (video RUE) with other people in real time. The video link
allows the CA to view and interpret the subscriber’s signed
conversation and relay the conversation back and forth with the other
party.

4. Requirements Language

The keywords "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]

5. General Requirements

All HTTP/HTTPS connections specified throughout this document MUST
use HTTPS. Both HTTPS and all SIP Transactions MUST use TLS 1.2
[RFC5246] or later, using a server-side certificate issued by a well-
known Certificate Agency. The minimum cipher suite to be supported
is TLS_RSA_WITH_AES_256_CBC_SHA256.
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 support is RECOMMENDED.

During the establishment of secure connections with a provider, the
RUE MAY be asked by the server for a client certificate. In that
case it SHOULD provide a client certificate. Providers MAY reject
requests that fail to provide a recognized certificate.

All text data payloads not otherwise constrained by a specification
in another standards document MUST be encoded as Unicode UTF/8.

6. SIP Signaling

The RUE and Providers MUST conform to the following core SIP
standards [RFC3261] (Base SIP) [RFC3263] (Locating SIP Servers),
[RFC3264] (Offer/Answer), [RFC3840] (User Agent Capabilities),
[RFC5626] (Outbound), [RFC4566] (Session Description Protocol),
[RFC3323] (Privacy), [RFC3605] (RTCP Attribute in SDP), [RFC6665]
(SIP Events), [RFC3311] (UPDATE Method), [RFC5393] (Loop-Fix),
[RFC5658] (Record Route fix), [RFC5954] (ABNF fix), [RFC3960] (Early
Media), and [RFC6442] (Geolocation Header).

In addition, the RUE MUST, and Providers MAY, conform to [RFC3327]
(Path), [RFC5245] (ICE), [RFC3326] (Reason header), [RFC3515] (REFER
Method), [RFC3891] (Replaces Header), [RFC3892] (Referred-By).
RUEs MUST include a "User-Agent" header field uniquely identifying the RUE application, platform, and version in all SIP requests, and MUST include a "Server" header field with the same content in SIP responses.

6.1. Registration

The RUE MUST register with a SIP registrar, following [RFC3261] and [RFC5626]. If the configuration (please refer to Section 11) contains multiple "outbound-proxies", then the RUE MUST use them as specified in [RFC5626] to establish multiple flows.

The request-URI for the REGISTER request MUST contain the "provider-domain" from the configuration. The To-URI and From-URI MUST be identical URIs, formatted as specified in Section 13, using the "phone-number" and "provider-domain" from the configuration.

The RUE determines the URI to resolve by initially determining if an outbound proxy is configured. If it is, the URI will be that of the outbound proxy. If no outbound proxy is configured, the URI will be the Request-URI from the REGISTER request. The RUE extracts the domain from that URI and consults the DNS record for that domain. The DNS entry MUST contain NAPTR records conforming to RFC3263. One of those NAPTR records MUST specify TLS as the preferred transport for SIP. For example, a DNS NAPTR query for "sip:p1.red.example.netv" could return:

```
IN NAPTR 50  50 "s" "SIPS+D2T" " _sips._tcp.p1.red.example.net
IN NAPTR 90  50 "s" "SIP+D2T"  " _sip._tcp.p1.red.example.net
```

If the RUE receives a 439 (First Hop Lacks Outbound Support) response to a REGISTER request, it MUST re-attempt registration without using the outbound mechanism.

The registrar MAY authenticate using SIP MD5 digest authentication. The credentials to be used (username and password) MUST be supplied within the credentials section of the configuration and identified by the realm the registrar uses in a digest challenge. This username/password combination SHOULD NOT be the same as that used for other purposes, such as retrieving the RUE configuration or logging into the Provider’s customer service portal. Because MD5 is considered insecure, [I-D.yusef-sipcore-digest-scheme] SHOULD be implemented by both the RUE and Providers and SHA-based digest algorithms SHOULD be used for digest authentication.

If the registration request fails with an indication that credentials from the configuration are invalid, then the RUE SHOULD retrieve a fresh version of the configuration. If credentials from a freshly
retrieved configuration are found to be invalid, then the RUE MUST cease attempts to register and SHOULD inform the RUE User of the problem.

Support for multiple simultaneous registrations by Providers is OPTIONAL, as described in Section 2.

Multiple simultaneous RUE SIP registrations from different RUE devices with the same SIP URI SHOULD be permitted by the Provider. The Provider MAY limit the total number of simultaneous registrations. When a new registration request is received that results in exceeding the limit on simultaneous registrations, the Provider MAY then prematurely terminate another registration; however, it SHOULD NOT do this if it would disconnect an active call.

If a Provider prematurely terminates a registration to reduce the total number of concurrent registrations with the same URI, it SHOULD take some action to prevent the affected RUE from automatically re-registering and re-triggering the condition.

6.2. Session Establishment

6.2.1. Normal Call Origination

After initial SIP registration, the RUE adheres to SIP [RFC3261] basic call flows, as documented in [RFC3665].

The RUE MUST route all calls through the outbound proxy of the default Provider.

INVITE requests used to initiate calls SHOULD NOT contain Route headers. Route headers MAY be included in one-stage dial-around calls and emergency calls. The SIP URIs in the To field and the Request-URI MUST be formatted as specified in subsection 6.4 using the destination phone number. The domain field of the URIs SHOULD be the "provider-domain" from the configuration (e.g., sip:+13115552368@red.example.com;user=phone). The same exceptions apply, including anonymous calls.

Anonymous calls MUST be supported by both the RUE and Providers. An anonymous call is signaled per [RFC3323].

The From-URI MUST be formatted as specified in Section 6.4, using the phone-number and "provider-domain" from the configuration. It SHOULD also contain the display-name from the configuration when present. (Please refer to Section 11.2.)
Negotiated media MUST follow the guidelines specified in Section 7 of this document.

To allow time to timeout an unanswered call and direct it to a videomail server, the User Agent Client MUST NOT impose a time limit less than the default SIP Invite transaction timeout of 3 minutes.

6.2.2. One-Stage Dial-Around Origination

Outbound dial-around calls allow a RUE user to select any Provider to provide interpreting services for any call. "Two-stage" dial-around calls involve the RUE calling a telephone number that reaches the dial-around Provider and using signing or DTMF to provide the called party telephone number. In two-stage dial-around, the To URI is the URI of the dial-around Provider and the domain of the URI is the Provider domain from the configuration.

One-stage dial-around is a method where the called party telephone number is provided in the To URI and the Request-URI, using the domain of the dial-around Provider.

For one-stage dial-around, the RUE MUST follow the procedures in Section 6.2.1 with the following exception: the domain part of the SIP URIs in the To field and the Request-URI MUST be the domain of the dial-around Provider, discovered according to Section 11.1.

The following is a partial example of a one-stage dial-around call from VRS user +1-555-222-0001 hosted by red.example.com to a hearing user +1-555-123-4567 using dial-around to green.example.com for the relay service. Only important details of the messages are shown and many header fields have been omitted:
6.2.3. RUE Contact Information

To identify the owner of a RUE, the initial INVITE for a call from a RUE, or the 200 OK accepting a call by a RUE, identifies the owner by sending a Call-Info header with a purpose parameter of "rue-owner". The URI MAY be an HTTPS URI or Content-Indirect URL. The latter is defined by [RFC2392] to locate message body parts. This URI type is present in a SIP message to convey the RUE ownership information as a MIME body. The form of the RUE ownership information is an xCard. Please refer to [RFC6442] for an example of using Content-Indirect URLs in SIP messages. Note that use of the Content-Indirect URL usually implies multiple message bodies ("mime/multipart").

6.2.4. Incoming Calls

The RUE MUST accept inbound calls sent to it by the proxy mentioned in the configuration.

If multiple simultaneous RUE SIP registrations from different RUE devices with the same SIP URI exist, the Provider MUST parallel fork
the call to all registered RUEs so that they ring at the same time. The first RUE to reply with a 200 OK answers the call and the Provider MUST CANCEL other call branches.

6.2.5. Emergency Calls

The RUE MUST comply with [RFC6881] for handling of emergency calls.

Providers MAY comply with RFC6881 for handling of emergency calls. In addition, they MUST:

- Accept RUE emergency calls complying with the specifications in this document;
- Recognize such calls as emergency calls and properly handle them as such;
- Address other behavior not specified by RFC6881 as specified in Section 6.2.

Specifically, if the emergency call is to be handled using E9-1-1 (VPC) procedures, the Provider is responsible for modifying the INVITE to conform to the VPC requirements. In this case, location MAY be extracted from the RFC6881 conformant INVITE and used to propagate it to the VPC where possible with the emergency call. Because the RUE may have a more accurate and timely location of the device than the typical manual entry location for nomadic RUE devices, the RUE MUST send a Geolocation header containing its location in the REGISTER request if the configuration specifies it. The Provider MAY use that information to populate the location of the device in the VPC before any emergency call.

6.3. Mid Call Signaling

The RUE and Providers MUST support re-INVITE to renegotiate media session parameters (among other uses). Per Section 8, the RUE MUST, and providers SHOULD, be able to support an INFO request for full frame refresh for devices in a call with the RUE that do not support RTCP mechanisms (please refer to Section 8.3). The RUE MUST support REFER to enable call transfer.

6.4. URI Representation of Phone Numbers

SIP URIs constructed from non-URI sources (dial strings) and sent to SIP proxies by the RUE MUST be represented as follows, depending on whether they can be represented as an E.164 number.
A dial string that can be written as an E.164 formatted phone number MUST be represented as a SIP URI with a URI ";user=phone" tag. The user part of the URI MUST be in conformance with ‘global-number’ defined in [RFC3966]. The user part MUST NOT contain any ‘visual-separator’ characters.

Dial strings that cannot be written as E.164 numbers MUST be represented as dialstring URIs, as specified by [RFC4967], e.g., sip:411@red.example.net;user=dialstring.

Relay Service URIs and User Address of Records (AoR) MUST resolve (in accord with [RFC3263]) to globally routable IPv4 addresses. The AoRs MAY also resolve to IPv6 addresses.

6.5. NAT Traversal

The RUE MUST support ICE [RFC5245] and MUST be able to use STUN [RFC5389] and TURN [RFC5766] servers, when specified in the configuration. If a STUN or TURN server issues a challenge for digest credentials, the RUE MUST attempt to continue using matching credentials from the configuration.

Providers MAY operate STUN and TURN servers for RUEs to use (please refer to Section 11.2 for configuration). Alternatively, providers MAY use Back to Back User Agents to modify media end points to achieve NAT traversal. This may involve media relays.

The RUE and providers MUST support SIP outbound [RFC5626] (please also refer to Section 6.1).

7. Media

7.1. Text-Based Communication

Real-time text support by Providers is OPTIONAL as described in Section 2.

The RUE MUST and Providers SHOULD support real-time text ([RFC4102] and [RFC4103]) via T.140 media. One original and two redundant generations MUST be transmitted and supported, with a 300 ms transmission interval.

7.2. Video Codecs

The RUE and Providers MUST implement the video/H.264 [RFC6184] Constrained Baseline Profile, Level 1.3, packetization mode 1. H.265 Main Profile, Level 2.1 or better MAY be supported.
7.3. Audio Codecs

The RUE and Providers MUST support G.711 mu-law. Providers SHOULD, and the RUE MUST, support G.722. In addition, G.722.1 and/or G.722.2 MAY also be supported.

7.4. DTMF Digits

The RUE and Providers MUST support the "audio/telephone-event" [RFC4733] media type. They MUST support conveying event codes 0 through 11 (DTMF digits "0"-"9", "+", ") defined in Table 7 of [RFC4733]. Handling of other tones is OPTIONAL.

7.5. Session Description Protocol

The SDP offers and answers MUST conform to the Offer/Answer Section of the VRS US Provider Profile [pip] document.

7.6. Privacy

The RUE MUST be able to control privacy of the user by implementing a one-way mute of audio and or video, without signaling, locally, but MUST maintain any NAT bindings by periodically sending media packets on all active media sessions containing silence/comfort noise/black screen/etc. per [RFC6263].

8. SRTP and SRTCP

All media streams between the RUE and another endpoint or relay Provider MUST be exchanged using the secure real-time transport protocol (SRTP) [RFC3550] using DTLS [RFC5763], [RFC5764], except that for backwards compatibility with older video endpoints, RTP MAY be negotiated if SRTP negotiation fails. All SRTP/RTP and SRTCP/RTCP traffic over UDP MUST use symmetric RTP [RFC4961]. Receivers of SRTP/RTP traffic MUST be capable of processing SRTP/RTP packets with a different packetization rate than the rate used for sending.

8.1. Bandwidth Negotiation Flow Control and Media Performance

The RUE and Providers MUST support codec control messages as described in [RFC5104]. During a call, codec control messages SHOULD be used to negotiate maximum bitrate. Specifically, Temporary Maximum Media Stream Bit Rate Request (TMMBR) SHOULD be used where endpoints have detected the need to decrease or increase the bit rate. Where either side of a session does not support CCM TMMBR, re-INVITE messages MAY be used during a call to renegotiate the use of bandwidth.
8.2. SRTP / SAVPF Profile for SRTCP Feedback

The RUE and Providers MUST support the SRTP/SAVPF profile per [RFC4585] for SRTCP feedback support. Supporting the SRTP/SAVPF profile allows implementations to use advanced RTCP mechanisms, like indicating packet loss, requesting intra frame, and temporary bitrate change indication, which are essential for video streams.

Supported SAVPF messages MUST be declared by RTCP Feedback attributes. Because implementations convey media streams from RUEs of varying background, there may be situations when no SAVPF attributes are supported in a session.

For calls to devices that do not support SAVPF, it is recommended to make an initial INVITE with "SAVPF". If some media are not accepted with that profile, then a re-INVITE SHOULD be made with AVP declaration on the non-accepted media and the other media unchanged. The only combinations required are SAVPF and AVP.

8.3. Negative Acknowledgment, Packet Loss Indicator, and Full Intraframe Request Features

NACK SHOULD be used when negotiated and conditions warrant its use. Signaling picture losses as Packet Loss Indicator (PLI) SHOULD be preferred, as described in [RFC5104].

FIR SHOULD be used only in situations where not sending a decoder refresh point would render the video unusable for the users, as per RFC5104 subsection 4.3.1.2.

For backwards compatibility with calling devices that do not support the foregoing methods, the RUE MUST implement and use SIP INFO messages to send and receive XML encoded Picture Fast Update messages according to [RFC5168].

9. Contacts

9.1. CardDAV Login and Synchronization

Support of CardDAV by Providers is OPTIONAL, as described in Section 2.

The RUE MUST and Providers MAY be able to synchronize the user’s contact directory between the RUE endpoint and one maintained by the user’s VRS provider using CardDAV ([RFC6352] and [RFC6764]).

The configuration MAY supply a username and domain identifying a CardDAV server and address book for this account.
To access the CardDAV server and address book, the RUE MUST follow Section 6 of RFC6764, using the chosen username and domain in place of an email address. If the request triggers a challenge for digest authentication credentials, the RUE MUST attempt to continue using matching "credentials" from the configuration. If no matching credentials are configured, the RUE MUST use the SIP credentials from the configuration. If the SIP credentials fail, the RUE MUST query the user.

Synchronization using CardDAV MUST be a two-way synchronization service, with proper handling of asynchronous adds, changes, and deletes at either end of the transport channel.

9.2. Contacts Import/Export Service

Each Provider MUST supply a standard xCard import/export interface and the RUE MUST be able to export/import the list of contacts in xCard [RFC6351] XML format.

The RUE accesses this service via the "contacts" URI in the configuration. The URL MUST resolve to identify a web server resource that imports/exports contact lists for authorized users.

The RUE stores/retrieves the contact list (address book) by issuing an HTTPS POST or GET request. If the request triggers a challenge for digest authentication credentials, the RUE MUST attempt to continue using matching "credentials" from the configuration. If no credentials are configured, the RUE MUST query the user.

10. Mail Waiting Indicator (MWI)

Support of MWI by Providers is OPTIONAL, as described in Section 2

The RUE MUST and Providers SHOULD support subscriptions to "message-summary" events [RFC3842] to the URI specified in the configuration if the Provider supports message waiting indicator on any endpoint.

In notification bodies, videomail messages SHOULD be reported using "message-context-class multimedia-message" defined in [RFC3458].

11. Provisioning and Provider Selection

11.1. RUE Provider Selection

To allow the user to select a relay service, the RUE MAY obtain, on startup, a list of Providers from a configured accessible URL.

The provider list, formatted as JSON, contains:
o Version: Specifies the version number of the Provider list format. A new version number SHOULD only be used if the new version is not backwards-compatible with the older version. A new version number is not needed if new elements are optional and can be ignored by older implementations.

o Providers: An array where each entry describes one Provider. Each entry consists of the following items:

* name: This parameter contains the text label identifying the Provider and is meant to be displayed to the human VRS user.

* domain: The domain parameter is used for configuration purposes by the RUE (as discussed in Section 11.2) and as the domain to use when targeting one-stage dial-around calls to this Provider (as discussed in Section 6.2.2).

* operator: (OPTIONAL) The operator parameter is a SIP URL that identifies the operator "front-door" that VRS users may contact for manual (two-stage) dial-around calls.

The VRS user interacts with the RUE to select from the Provider list one or more Providers with whom the user has already established an account.

```json
{
    "version": 1,
    "providers": [
        {
            "name": "Red",
            "domain": "red.example.net",
            "operator": "sip:operator@red.example.net"
        },
        {
            "name": "Green",
            "domain": "green.example.net",
            "operator": "sip:+18885550123@green.example.net;user=phone"
        },
        {
            "name": "Blue",
            "domain": "blue.example.net"
        }
    ]
}
```

Example of a Provider list JSON object
11.2. RUE Configuration Service

The RUE is provisioned with one or more URIs that may be queried for configuration with HTTPS.

The data returned will include a set of key/value configuration parameters to be used by the RUE, formatted as a JSON object and identified by the associated [RFC7159] "application/json" MIME type, to allow for other formats in the future.

The configuration data payload includes the following data items. Items not noted as (OPTIONAL) are REQUIRED. If other unexpected items are found, they MUST be ignored.

- **version**: Identifies the version of the configuration data format. A new version number SHOULD only be used if the new version is not backwards-compatible with the older version. A new version number is not needed if new elements are optional and can be ignored by older implementations.

- **lifetime**: Specifies how long (in seconds) the RUE MAY cache the configuration values.

- **display-name**: (OPTIONAL) A user-friendly name to identify the subscriber when originating calls.

- **phone-number**: The telephone number (in E.164 format) assigned to this subscriber. This becomes the user portion of the SIP URI identifying the subscriber.

- **provider-domain**: The DNS domain name of the default Provider servicing this subscriber.

- **outbound-proxies**: (OPTIONAL) A list of URIs of SIP proxies to be used when sending requests to the Provider. Multiple URIs identify alternative (redundant) paths to the Provider.

- **mwi**: (OPTIONAL) A URI identifying a SIP event server that generates "message-summary" events for this subscriber.

- **videomail**: (OPTIONAL) A SIP URI that can be called to retrieve videomail messages.

- **contacts**: An HTTPS URI that may be used to export (retrieve) the subscriber’s complete contact list managed by the Provider.

- **carddav**: (OPTIONAL) A username and domain name (separated by "@") identifying a "CardDAV" server and user name that can be
used to synchronize the RUE’s contact list with the contact list
managed by the Provider.

- sendLocationWithRegistration: True if the RUE should send a
  Geolocation Header with REGISTER, false if it should not.
  Defaults to false if not present.

- ice-servers: (OPTIONAL) An array of URLs identifying STUN and TURN
  servers available for use by the RUE for establishing media
  streams in calls via the Provider.

- credentials: (OPTIONAL) An array of sets of credentials available
  for use in responding to SIP, HTTP, contacts, cardDAV, STUN, and
  TURN digest authentication challenges from specified realms. Each
  set consists of the following items:

  * realm: The realm to which this set of credentials applies.

  * username: The username field to be used in responding to a
    challenge.

  * password: The password to use in generating the response to the
    challenge.

```json
{
    "version": 1,
    "lifetime": 86400,
    "display-name": "Bob Smith",
    "phone-number": "+18135551212",
    "provider-domain": "red.example.net",
    "outbound-proxies": [
        "sip:p1.red.example.net",
        "sip:p2.red.example.net"
    ],
    "mwi": "sip:+18135551212@red.example.net",
    "videomail": "sip:+18135551212@vm.red.example.net",
    "contacts": "https://red.example.net:443/contacts/1dess45awd"
    "carddav": "bob@red.example.com",
    "sendLocationWithRegistration": false,
    "ice-servers": [
        "stun:stun.l.google.com:19302",
        "turn:turn.red.example.net:3478"
    ],
    "credentials": [
        {
            "realm": "red.example.net",
            "username": "bob",
            "password": "reg-pw"
        }
    ]
}
```
Example JSON configuration payload

The wire format of the data is in keeping with the standard JSON description in RFC7159.

The "lifetime" parameter in the configuration indicates how long the RUE MAY cache the configuration values. If the RUE caches configuration values, it MUST cryptographically protect them. The RUE SHOULD retrieve a fresh copy of the configuration before the lifetime expires or as soon as possible after it expires. The lifetime is not guaranteed: the configuration may change before the lifetime value expires. In that case, the Provider MAY indicate this by generating authorization challenges to requests and/or prematurely terminating a registration.

Note: In some cases, the RUE may successfully retrieve a fresh copy of the configuration using digest credentials cached from the prior retrieval. If this is not successful, then the RUE will need to ask the user for the username and password. Unfortunately, this authentication step might occur when the user is not present, preventing SIP registration and thus incoming calls. To avoid this situation, the RUE MAY retrieve a new copy of the configuration when it knows the user is present, even if there is time before the lifetime expires.
11.3. Schemas

The following JSON schemas are for the Provider List and the RUE Configuration. These are represented using the JSON Content Rules [JCR] schema notation.

```json
{
    "version": 1,
    "providers": [
        {  
            "name": string,
            "domain": fqdn,
            "operator": {  
                "uri": "front-door" access to provider
            },  
            "domain": fqdn,
            "operator": {  
                "uri": "sip uri"
            },
            "operator": {  
                "uri": "allow future extensions"
            }
        }  
        * /^.*$/ : any ; (allow future extensions)
    ]  
    * /^.*$/ : any ; (allow future extensions)
}
```

Provider List JSON Schema
The following illustrates the message flow for retrieving a RUE automatic configuration using HTTPS Digest Authentication:

```
,RUE   DNS    HTTPS Server   Provider
     |       |            | Global Settings |
RUE User       ---------       ------------       ---

[1] Select a VRS Provider name
    ____->

[2] NAPTR "SFUA.CFG" red.example.net
       ------->

    <-----

[4] If NAPTR found, query DNS server.red.example.net
```
If NXDOMAIN, query DNS config.red.example.net

IP Address of Config Server

Establish TLS connection

HTTP: https://config.red.example.net/v1

HTTP: 401 Unauthorized
WWW-Authenticate Digest realm="Y" qop="auth,auth-int" nonce=

Query for userid/pw

Authorization Digest username="bob" realm="Y" qop="auth"
nonce=... response="..." ...

Find subscriber information for username="bob"

Subscriber specific configuration information

Retrieve provider specific settings

Provider configuration information

200 OK + JSON merge subscriber + provider configs

RUE User

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RUE</td>
<td>DNS</td>
<td>HTTPS Server</td>
<td>Provider</td>
</tr>
<tr>
<td>CRM</td>
<td></td>
<td></td>
<td>Global Settings</td>
</tr>
</tbody>
</table>

12. Acknowledgements

13. IANA Considerations

This memo includes no request to IANA.

14. Security Considerations

The RUE is required to communicate with servers on public IP addresses and specific ports to perform its required functions. If it is necessary for the RUE to function on a corporate or other network that operates a default-deny firewall between the RUE and these services, the user must arrange with their network manager for passage of traffic through such a firewall in accordance with the protocols and associated SRV records as exposed by the Provider. Because VRS providers may use different ports for different services, these port numbers may differ from Provider to Provider.

15. Normative References

[I-D.yusef-sipcore-digest-scheme]


Internet-Draft        Relay User Equipment Profile         February 2019


Authors' Addresses

Brian Rosen
Mars, PA
US

Phone: +1 724 382 1051
Email: br@brianrosen.net
Jim Malloy  
The MITRE Corporation  
McLean, VA  
US  

Phone: +1 703 983 2835  
Email: jmalloy@mitre.org  

Brett Henderson  
The MITRE Corporation  
McLean, VA  
US  

Phone: +1 619 758 6071  
Email: brhenderson@mitre.org
Abstract

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

Status of This Memo

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

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

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

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

Copyright Notice

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

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

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

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

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

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

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

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

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

For potential uses outside of cryptography see [JSONCOMP].

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

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

3. Detailed Operation

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

3.1. Creation of Input Data

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

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

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

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

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

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

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

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

3.2.1. Whitespace

Whitespace between JSON elements MUST NOT be emitted.

3.2.2. Serialization of Primitive Data Types

Assume that you parse a JSON object like the following:

```
{
    "numbers": [333333333.33333329, 1E30, 4.50,
                 2e-3, 0.000000000000000000000000001],
    "string": "\u20ac\$\u000F\u000aA’\u0042\u0022\u005c\"\/",
    "literals": [null, true, false]
}
```

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

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

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

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

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

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

3.2.2.2. Serialization of Strings

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

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

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

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

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

3.2.2.3. Serialization of Numbers

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

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

3.2.3. Sorting of Object Properties

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

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

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

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

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

- JSON Object properties MUST be sorted in a recursive manner which means that possible JSON child Objects MUST have their properties sorted as well.
- JSON Array data MUST also be scanned for presence of JSON Objects (and applying associated property sorting), but array element order MUST NOT be changed.

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

- The sorting process is applied to property name strings in their "raw" (unescape) form. That is, a newline character is treated as U+000A.
- Property name strings to be sorted are formatted as arrays of UTF-16 [UNICODE] code units. The sorting is based on pure value comparisons, where code units are treated as unsigned integers, independent of locale settings.
- Property name strings either have different values at some index that is a valid index for both strings, or their lengths are different, or both. If they have different values at one or more index positions, let k be the smallest such index; then the string
whose value at position k has the smaller value, as determined by using the < operator, lexicographically precedes the other string. If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

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

""
"a"
"aa"
"ab"

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

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

3.2.4. UTF-8 Generation

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

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

7b 22 6c 69 74 65 72 61 6c 73 22 3a 5b 6e 75 6c 6c 2c 74 72 75 65 2c 66 61 6c 73 65 5d 22 6e 75 6d 62 65 72 73 22 3a 5b 33 33 33 33 33 33 33 33 2e 33 33 33 33 33 33 33 2c 31 65 2b 33 30 2c 34 2e 35 2c 30 2e 30 30 32 2c 31 65 2d 32 37 5d 22 73 74 72 69 6e 67 22 3a 22 e2 82 ac 24 5c 75 30 30 66 5c 6e 41 27 42 5c 22 5c 5c 5c 5c 5c 5c 5c 5c 5c 22 2f 22 7d

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

This document has no IANA actions.

5. Security Considerations

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

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

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

6. Acknowledgements

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

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

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

7. References

7.1. Normative References


7.2. Informal References


7.3. URIs

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

Appendix A. ES6 Sample Canonicalizer

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

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

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

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

<table>
<thead>
<tr>
<th>IEEE-754</th>
<th>JSON Representation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>0</td>
<td>Zero</td>
</tr>
<tr>
<td>8000000000000000</td>
<td>0</td>
<td>Minus zero</td>
</tr>
<tr>
<td>0000000000000001</td>
<td>5e-324</td>
<td>Min pos number</td>
</tr>
<tr>
<td>8000000000000001</td>
<td>-5e-324</td>
<td>Min neg number</td>
</tr>
<tr>
<td>7fefefffffffffff</td>
<td>1.7976931348623157e+308</td>
<td>Max pos number</td>
</tr>
<tr>
<td>fffefffffffffff</td>
<td>-1.7976931348623157e+308</td>
<td>Max neg number</td>
</tr>
<tr>
<td>4340000000000000</td>
<td>9007199254740992</td>
<td>Max pos integer (1)</td>
</tr>
<tr>
<td>c340000000000000</td>
<td>-9007199254740992</td>
<td>Max neg integer (1)</td>
</tr>
<tr>
<td>4430000000000000</td>
<td>29514790517935283000</td>
<td>2**68</td>
</tr>
<tr>
<td>7fffffff11111111</td>
<td>NaN</td>
<td>NaN (3)</td>
</tr>
<tr>
<td>7ff0000000000000</td>
<td>Infinity</td>
<td>(3)</td>
</tr>
<tr>
<td>44b52d02c7e14af5</td>
<td>9.999999999999997e+22</td>
<td></td>
</tr>
<tr>
<td>44b52d02c7e14af6</td>
<td>1e+23</td>
<td></td>
</tr>
</tbody>
</table>
Notes:

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

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

(3) Invalid. See Section 3.2.2.3.

Appendix C. Canonicalized JSON as "Wire Format"

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

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

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

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

Appendix D. Dealing with Big Numbers

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

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

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

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

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

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

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

### Appendix E. String Subtype Handling

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

E.1. Subtypes in Arrays

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

Appendix F. Implementation Guidelines

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

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

A compatible signature verification scheme would then be as follows:

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

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

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

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

Appendix G. Open Source Implementations

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

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

Appendix H. Other JSON Canonicalization Efforts

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

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

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

The JCS specification is currently developed at: https://github.com/cyberphone/ietf-json-canon [10].

The most recent "editors’ copy" can be found at: https://cyberphone.github.io/ietf-json-canon [11].

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

Authors’ Addresses

Anders Rundgren
Independent
Montpellier
France

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

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

Email: bret_jordan@symantec.com

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

Email: erdtman@spotify.com
JSContact: A JSON representation of addressbook data
draft-stepanek-jscontact-02

Abstract

This specification defines a data model and JSON representation of contact information that can be used for data storage and exchange in address book or directory applications. It aims to be an alternative to the vCard data format and to be unambiguous, extendable and simple to process. In contrast to the JSON-based jCard format, it is not a direct mapping from the vCard data model and expands semantics where appropriate.

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 22, 2019.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

Stepanek & Loffredo Expires December 22, 2019 [Page 1]
1. Introduction

This document defines a data model for contact data normally used in address book or directory applications and services. It aims to be an alternative to the vCard data format [RFC6350] and to provide a JSON-based standard representation of contacts data.

The key design considerations for this data model are as follows:

- Most of the initial set of attributes should be taken from the vCard data format [RFC6350] and extensions ([RFC6473], [RFC6474], [RFC6715], [RFC6869], [RFC8605]). The specification should add new attributes or value types, or not support existing ones, where appropriate. Conversion between the data formats need not fully preserve semantic meaning.

- The attributes of the contacts data represented must be described as a simple key-value pair, reducing complexity of its representation.

- The data model should avoid all ambiguities and make it difficult to make mistakes during implementation.

- Extensions, such as new properties and components, MUST NOT lead to requiring an update to this document.

The representation of this data model is defined in the I-JSON format [RFC7493], which is a strict subset of the JavaScript Object Notation (JSON) Data Interchange Format [RFC8259]. Using JSON is mostly a
pragmatic choice: its widespread use makes JSContact easier to adopt, and the availability of production-ready JSON implementations eliminates a whole category of parser-related interoperability issues.

1.1. Relation to the xCard and jCard formats

The xCard [RFC6351] and jCard [RFC7095] specifications define alternative representations for vCard data, in XML and JSON format respectively. Both explicitly aim to not change the underlying data model. Accordingly, they are regarded as equal to vCard in the context of this document.

1.2. Terminology

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

2. Contact

MIME type: "application/jscontact+json;type=jscontact"

A JSContact object stores contact information about a person, organization or company. It has the following properties:

- **uid**: String (mandatory). An identifier, used to associate the object as the same across different systems, addressbooks and views. [RFC4122] describes a range of established algorithms to generate universally unique identifiers (UUID), and the random or pseudo-random version is recommended. For compatibility with [RFC6350] UIDs, implementations MUST accept both URI and free-form text.

- **prodId**: String (optional). The identifier for the product that created the JSContact object.

- **updated**: String (mandatory). The date and time when the data in this JSContact object was last modified. The timestamp MUST be formatted as specified in [RFC3339].

- **kind**: String (optional). The kind of the entity the Contact represents. The value MUST be either one of the following values, registered in a future RFC, or a vendor-specific value:

  * "individual": a single person
* "org": an organization
* "location": a named location
* "device": a device, such as appliances, computers, or network elements
* "application": a software application

  o fullName: FullName[] (mandatory). The full name(s) of a contact. A FullName object has the following properties:

      * name: String (mandatory) The full name (e.g. the personal name and surname of an individual, the name of an organization).
      * language: String (optional) The [RFC5646] language tag of this name, if any.
      * isPreferred: Boolean (optional, default: "false"). Whether this FullName is the preferred name for this contact.

  o structuredName: StructuredName (optional). The name of this contact, structured by its constituents. A StructuredName object has the following properties:

      * prefix: String[] (optional). The honorific title(s) of the contact (e.g. "Mr", "Ms", "Dr").
      * personalName: String[] (optional). The personal name(s) of a contact (also known as "first name", "give name").
      * surname: String[] (optional). The surname(s) of a contact (also known as "last name", "family name").
      * additionalName: String[] (optional). The additional name(s) of a contact (also known as "middle name").
      * suffix: String[] (optional). The honorific suffix(es) of the contact (e.g. "B.A.", "Esq.").

  o nickname: String[] (optional). The nickname(s) of the contact.

  o birthday: String (optional). The contact’s birth date in the form "YYYY-MM-DD" (any part may be all 0s for unknown) or a [RFC3339] timestamp.

  o birthPlace: String (optional). The contact’s birth place.
o deathDate: String (optional). The contact’s death date in the form "YYYY-MM-DD" (any part may be all 0s for unknown) or a [RFC3339] timestamp.

o deathPlace: String (optional). The contact’s death place.

o anniversary: String (optional). The contact’s anniversary date in the form "YYYY-MM-DD" (any part may be all 0s for unknown).

o organization: String[] (optional). The company or organization name and units associated with this contact. The first entry in the list names the organization, and any following entries name organizational units.

o jobTitle[]: String (optional). The job title(s) or functional position(s) of the contact.

o role[]: String (optional). The role(s), function(s) or part(s) played in a particular situation by the contact. In contrast to a job title, the roles might differ for example in project contexts.

o emails: ContactMethod[] (optional). An array of ContactMethod objects where the values are email addresses. Types are:

* "personal" The address is for emailing the contact in a personal context.

* "work" The address is for emailing the contact in a professional context.

* "other" The address is for some other purpose. A label property MAY be included to display next to the address to help the user identify its purpose.

o phones: ContactMethod[] (optional). An array of ContactMethod objects where the values are phone numbers. Types are:

* "voice" The number is for calling the contact.

* "fax" The number is for sending faxes to the contact.

* "pager" The number is for a pager or beeper associated with the contact.

* "other" The number is for some other purpose. A label property MAY be included to display next to the number to help the user identify its purpose.

The following labels are pre-defined for phone contact methods:
* "private" The phone number should be used in a private context.

* "work" The phone number should be used in a professional context

- online: ContactMethod[] (optional). An array of ContactMethod objects where the values are URIs or usernames associated with the contact for online services. Types are:
  * "uri" The value is a URI, e.g. a website link.
  * "username" The value is a username associated with the contact (e.g. for social media, or an IM client). A label property SHOULD be included to identify what service this is for. For compatibility between clients, this label SHOULD be the canonical service name, including capitalisation. e.g. "Twitter", "Facebook", "Skype", "GitHub", "XMPP".
  * "other" The value is something else not covered by the above categories. A label property MAY be included to display next to the number to help the user identify its purpose.

- preferredContactMethod: String (optional) Defines the preferred contact method. The value MUST be the property name of one of the ContactMethod lists: "emails", "phones", "online", "other".

- addresses: Address[] (optional). An array of Address objects, containing physical locations associated with the contact.

- personalInfo: PersonalInformation[] (optional). A list of personal information about this contact. A PersonalInformation object has the following properties:
  * type: String (mandatory). Specifies the type for this personal information. Allowed values are:
    + "expertise": a field of expertise or credential
    + "hobby": a hobby of this contact
    + "interest": an interest of this contact
    + "other": an information not covered by the above categories
  * value: String (mandatory). The actual contact information. This generally is free-text, but future specifications MAY restrict allowed values depending on the type of this PersonalInformation.
* level: String (optional) Indicates the level of expertise, or engagement in hobby or interest. Allowed values are: "high", "medium" and "low".

o notes: String (optional). Arbitrary notes about the contact.

o categories: String[] (optional). A list of free-text or URI categories that relate to the contact.

A ContactMethod object has the following properties:

o type: String (mandatory). Specifies the context of the contact method. This MUST be taken from the set of values allowed depending on whether this is part of the phones, emails or online property (see above).

o label: String (optional). A label describing the value in more detail, especially if the type property has value "other" (but MAY be included with any type).

o value: String (mandatory). The actual contact method, e.g. the email address or phone number.

o isPreferred: Boolean (optional, default: "false"). Whether this ContactMethod is the preferred for its type. This SHOULD only be one per type.

An Address object has the following properties:

o type: String (mandatory). Specifies the context of the address information. The value MUST be either one of the following values, registered in a future RFC, or a vendor-specific value:

* "home" An address of a residence associated with the contact.
* "work" An address of a workplace associated with the contact.
* "billing" An address to be used with billing associated with the contact..
* "postal" An address to be used for delivering physical items to the contact.
* "other" An address not covered by the above categories.

o label: String (optional). A label describing the value in more detail.
o fullAddress: String (optional). The complete address, excluding type and label. This property is mainly useful to represent addresses of which the individual address components are unknown.

o street: String (optional). The street address. This MAY be multiple lines; newlines MUST be preserved.

o extension: String (optional) The extended address, such as an apartment or suite number, or care-of address.

o postOfficeBox: String (optional) The post office box.

o locality: String (optional). The city, town, village, post town, or other locality within which the street address may be found.

o region: String (optional). The province, such as a state, county, or canton within which the locality may be found.

o postcode: String (optional). The postal code, post code, ZIP code or other short code associated with the address by the relevant country’s postal system.

o country: String (optional). The country name.

o countryCode: String (optional). The ISO-3166-1 country code.


o timeZone: String (optional) Identifies the time zone this address is located in. This SHOULD be a time zone name registered in the IANA Time Zone Database [1]. Unknown time zone identifiers MAY be ignored by implementations.

o isPreferred: Boolean (optional, default: "false"). Whether this Address is the preferred for its type. This SHOULD only be one per type.

3. Contact Group

MIME type: "application/jscontact+json;type=jscontactgroup"

A JSContactGroup object represents a named set of contacts. It has the following properties:

o uid: String (mandatory). A globally unique identifier. The same requirements as for the JSContact uid property apply.
**name**: String (optional). The user-visible name for the group, e.g. "Friends". This may be any UTF-8 string of at least 1 character in length and maximum 255 octets in size. The same name may be used by two different groups.

**contactIds**: String[] (mandatory). The ids of the contacts in the group. Implementations MUST preserve the order of list entries.

### 4. IANA Considerations

TBD

### 5. Security Considerations

TBD

### 6. References

#### 6.1. Normative References

6.2. Informative References


6.3. URIs

[1] https://www.iana.org/time-zones

Authors' Addresses

Robert Stepanek
FastMail
PO Box 234, Collins St West
Melbourne VIC 8007
Australia

Email: rsto@fastmailteam.com

Mario Loffredo
IIT-CNR
Via Moruzzi,1
Pisa 56124
Italy

Email: mario.loffredo@iit.cnr.it