Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: 24 May 2021

D. DeVault
SourceHut

20 November 2020

Binary Application Record Encoding (BARE) draft-devault-bare-01

Abstract

The Binary Application Record Encoding (BARE) is a data format used to represent application records for storage or transmission between programs. BARE messages are concise and have a well-defined schema, and implementations may be simple and broadly compatible. A schema language is also provided to express message schemas out-of-band.

Comments

Comments are solicited and should be addressed to the mailing list at ~sircmpwn/public-inbox@lists.sr.ht and/or the author(s).

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 24 May 2021.

Copyright Notice

Table of Contents

1. Introduction

The purpose of the BARE message encoding, like hundreds of others, is to encode application messages. The goals of such encodings vary (leading to their proliferation); BARE's goals are the following:

* Concise messages

* A well-defined message schema

* Broad compatibility with programming environments

* Simplicity of implementation

This document specifies the BARE message encoding, as well as a schema language which may be used to describe the layout of a BARE message. The schema of a message must be agreed upon in advance by each party exchanging a BARE message; message structure is not encoded into the representation. The schema language is useful for this purpose, but not required.

1.1. Terminology

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].

1.2. Use-cases

The goals of a concise, binary, strongly-typed, and broadly-compatible structured message encoding format support a broad number of use-cases. Examples include:

* Self-describing authentication tokens for web services
* Opaque messages for transmitting arbitrary state between unrelated internet services
* A representation for packets in an internet protocol
* A structured data format for encrypted or signed application messages
* A structured data format for storing data in persistent storage

The conciseness of a BARE-encoded message enables representing structured data under strict limitations on message length in a large variety of contexts. The simple binary format may also be easily paired with additional tools, such as plain-text encodings, compression, or cryptography algorithms, as demanded by the application's needs, without increasing the complexity of the message encoding. A BARE message has a comparible size and entropy to the underlying state it represents.

The BARE schema language also provides a means of describing the format of BARE messages without implementation-specific details. This encourages applications which utilize BARE to describe their state in a manner which other programmers can easily utilize for application inter-operation. The conservative set of primitives offered by BARE aids in making such new implementations easy to write.

2. Specification of the BARE Message Encoding

A BARE message is a single value of a pre-defined type, which may be of an aggregate type enclosing multiple values. Unless otherwise specified there is no additional container or structure around the value; it is encoded plainly.

A BARE message does not necessarily have a fixed length, but the schema author may make a deliberate choice to constrain themselves to types of well-defined lengths if this is desired.

The names for each type are provided to establish a vocabulary for describing a BARE message schema out-of-band, by parties who plan to exchange BARE messages. The type names used here are provided for this informative purpose, but are more rigourously specified by the schema language specification in Section 3.

2.1. Primitive Types

Primitive types represent exactly one value.

uint
: An unsigned integer with a variable-length encoding. Each octet of the encoded value has the most-significant bit set, except for the last octet. The remaining bits are the integer value in 7-bit groups, least-significant first.

  The maximum precision of such a number is 64-bits. The maximum length of an encoded uint is therefore 10 octets.

  Numbers which require all ten octets will have 6 bits in the final octet which do not have meaning, between the least- and most-significant bits. The implementation MUST set these to zero.

int
: A signed integer with a variable-length encoding. Signed integers are represented as uint using a "zig-zag" encoding: positive values x are written as 2x + 0, negative values are written as 2(^x) + 1. In other words, negative numbers are complemented and whether to complement is encoded in bit 0.

  The maximum precision of such a number is 64-bits. The maximum length of an encoded int is therefore 10 octets.

  Numbers which require all ten octets will have 6 bits in the final octet which do not have meaning, between the least- and most-significant bits. The implementation MUST set these to zero.

u8, u16, u32, u64
: Unsigned integers of a fixed precision, respectively 8, 16, 32, and 64 bits. They are encoded in little-endian (least significant octet first).

i8, i16, i32, i64
: Signed integers of a fixed precision, respectively 8, 16, 32, and 64 bits. They are encoded in little-endian (least significant octet first), with two's compliment notation.

f32, f64
: Floating-point numbers represented with the IEEE 754 [IEEE.754.1985] binary32 and binary64 floating point number formats.

  The encoder MUST NOT encode NaN into a BARE message, and the decoder SHOULD raise an error if it encounters such a value.

bool
: A boolean value, either true or false, encoded as a u8 type with a value of one or zero, respectively representing true or false.

  If a value other than one or zero is found in the u8 representation of the bool, the message is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

enum
: An unsigned integer value from a set of possible values agreed upon in advance, encoded with the uint type.

  An enum whose uint value is not a member of the values agreed upon in advance is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

  Note that this makes the enum type unsuitable for representing a several enum values which have been combined with a bitwise OR operation.

string
: A string of text. The length of the text in octets is encoded first as a uint, followed by the text data represented with the UTF-8 encoding [RFC3629].

  If the data is found to contain invalid UTF-8 sequences, it is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

data<length>
: Arbitrary data with a fixed "length" in octets, e.g. data<16>. The data is encoded literally in the message, and MUST NOT be greater than 18,446,744,073,709,551,615 octets in length (the maximum value of a u64).

data
: Arbitrary data of a variable length in octets. The length is encoded first as a uint, followed by the data itself encoded literally.

void
: A type with zero length. It is not encoded into BARE messages.

2.2. Aggregate Types

Aggregate types may store zero or more primitive or aggregate values.

optional<type>
: A value of "type" which may or may not be present, e.g. optional<u32>. Represented as either a u8 with a value of zero, indicating that the optional value is unset; or a u8 with a value of one, followed by the encoded data of the optional type.

  An optional value whose initial u8 is set to a number other than zero or one is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

[length]type
: A list of "length" values of "type", e.g. [10]uint. The length is not encoded into the message. The encoded values of each member of the list are concatenated to form the encoded list.

[]type
: A variable-length list of values of "type", e.g. []string. The length of the list (in values) is encoded as a uint, followed by the encoded values of each member of the list concatenated.

map[type A]type B

An mapping of values of type B keyed by values of type A, e.g. map[u32]string. The encoded representation of a map begins with the number of key/value pairs as a uint, followed by the encoded key/value pairs concatenated. Each key/value pair is encoded as the encoded key concatenated with the encoded value.

A message with repeated keys is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

(type | type | ...)
: A tagged union whose value may be one of any type from a set of types, e.g. (int | uint | string). Each type in the set is assigned a numeric identifier. The value is encoded as the selected type's identifier represented with the uint encoding, followed by the encoded value of that type.

  A union with a tag value that does not have a corresponding type assigned is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

struct
: A set of values of arbitrary types, concatenated in an order agreed upon in advance. Each value is referred to as a "field", and field has a name and type.

2.3. User-Defined Types

A user-defined type gives a name to another type. This creates a distinct type whose representation is equivalent to the named type. An arbitrary number of user-defined types may be used for the same underlying type; each is distinct from the other.

2.4. Invariants

The following invariants are specified:

* Any type which is ultimately a void type (either directly or via a user-defined type) MUST NOT be used as an optional type, struct member, list member, map key, or map value. Void types may only be used as members of the set of types in a tagged union.

* The lengths of fixed-length arrays and data types MUST be at least one.

* Structs MUST have at least one field.

* Unions MUST have at least one type, and each type MUST NOT be repeated.

* Map keys MUST be of a primitive type which is not data or data<length>.

* Each named value of an enum type MUST have a unique value.

3. BARE Schema Language Specification

The use of the schema language is optional. Implementations SHOULD support decoding arbitrary BARE messages without a schema document, by defining the schema in a manner which utilizes more native tools available from the programming environment.

However, it may be useful to have a schema document for use with code generation, documentation, or interoperability. A domain-specific language is provided for this purpose.

3.1. Lexical Analysis

During lexical analysis, "#" is used for comments; if encountered, the "#" character and any subsequent characters are discarded until a line feed (%x0A) is found.

3.2. ABNF Grammar

The syntax of the schema language is provided here in Augmented Backus-Naur form [RFC5234]. However, this grammar differs from [RFC5234] in that strings are case-sensitive (e.g. "type" does not match TypE).

```
schema = [WS] user-types [WS]

user-type =  "type" WS user-type-name WS non-enum-type
user-type =/ "enum" WS user-type-name WS enum-type
user-types = user-type / (user-types WS user-type)

type            = non-enum-type / enum-type
non-enum-type   = primitive-type / aggregate-type / user-type-name

user-type-name  = UPPER *(ALPHA / DIGIT) ; First letter is uppercase

primitive-type  =  "int" / "i8"  / "i16" / "i32" / "i64"
primitive-type  =/ "uint" / "u8"  / "u16" / "u32" / "u64"
primitive-type  =/ "f32" / "f64"
primitive-type  =/ "bool"
primitive-type  =/ "string"
```

```
primitive-type  =/ "data" / ("data<" integer ">")
primitive-type  =/ "void"

enum-type       =  "{" [WS] enum-values [WS] "}"
enum-values     =  enum-value / (enum-values WS enum-value)
enum-value      =  enum-value-name
enum-value      =/ (enum-value-name [WS] "=" [WS] integer)
enum-value-name =  UPPER *(UPPER / DIGIT / "_")

aggregate-type  =  optional-type
aggregate-type  =/ array-type
aggregate-type  =/ map-type
aggregate-type  =/ union-type
aggregate-type  =/ struct-type

optional-type   = "optional<" type ">"

array-type      = "[" [integer] "]" type
integer         = 1*DIGIT

map-type        = "map[" type "]" type

union-type      =  "(" union-members ")"
union-members   =  union-member
union-members   =/ (union-members [WS] "|" [WS] union-member)
union-member    =  type [[WS] "=" [WS] integer]

struct-type     = "{" [WS] fields [WS] "}"
fields          = field / (fields WS field)
field           = 1*ALPHA [WS] ":" [WS] type

UPPER           = %x41-5A ; uppercase ASCII letters
ALPHA           = %x41-5A / %x61-7A ; A-Z / a-z
DIGIT           = %x30-39 ; 0-9

WS              = 1*(%x0A / %x09 / " ") ; whitespace
```

See Appendix A for an example schema written in this language.

3.3. Semantic Elements

The names of fields and user-defined types are informational: they are not represented in BARE messages. They may be used by code generation tools to inform the generation of field and type names in the native programming environment.

Enum values are also informational. Values without an integer token are assigned automatically in the order that they appear, starting from zero and incrementing for each subsequent unassigned value. If a value is explicitly specified, automatic assignment continues from that value plus one for subsequent enum values.

Union type members are assigned a tag in the order that they appear, starting from zero and incrementing for each subsequent type. If a tag value is explicitly specified, automatic assignment continues from that value plus one for subsequent values.

4. Application Considerations

Message authors who wish to design a schema which is backwards- and forwards-compatible with future messages are encouraged to use union types for this purpose. New types may be appended to the members of a union type while retaining backwards compatibility with older message types. The choice to do this must be made from the first message version-- moving a struct into a union _does not_ produce a backwards-compatible message.

The following schema provides an example:

```
type Message (MessageV1 | MessageV2 | MessageV3)

type MessageV1 ...

type MessageV2 ...

type MessageV3 ...
```

An updated schema which adds a MessageV4 type would still be able to decode versions 1, 2, and 3.

If a message version is later deprecated, it may be removed in a manner compatible with future versions 2 and 3 if the initial tag is specified explicitly.

```
type Message (MessageV2 = 1 | MessageV3)
```

5. Future Considerations

To ensure message compatibility between implementations and backwards- and forwards-compatibility of messages, constraints on vendor extensions are required. This specification is final, and new types or extensions will not be added in the future. Implementors MUST NOT define extensions to this specification.

To support the encoding of novel data structures, the implementor SHOULD make use of user-defined types in combination with the data or data<length> types.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

Message parsers are common vectors for security vulnerabilities. BARE addresses this by making the message format as simple as possible. However, the parser MUST be prepared to handle a number of error cases when decoding untrusted messages, such as a union type with an invalid tag, or an enum with an invalid value. Such errors may also arise by mistake, for example when attempting to decode a message with the wrong schema.

Support for data types of an arbitrary, message-defined length (lists, maps, strings, etc) is commonly exploited to cause the implementation to exhaust its resources while decoding a message. However, legitimate use-cases for extremely large data types (possibly larger than the system has the resources to store all at once) do exist. The decoder MUST manage its resources accordingly, and SHOULD provide the application a means of providing their own decoder implementation for values which are expected to be large.

There is only one valid interpretation of a BARE message for a given schema, and different decoders and encoders should be expected to provide that interpretation. If an implementation has limitations imposed from the programming environment (such as limits on numeric precision), the implementor MUST document these limitations, and prevent conflicting interpretations from causing undesired behavior.

8. Normative References

[IEEE.754.1985]
Institute of Electrical and Electronics Engineers, "Standard for Binary Floating-Point Arithmetic", IEEE Standard 754, August 1985.

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

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

Appendix A. Example message schema

The following is an example of a schema written in the BARE schema language.

```
type PublicKey data<128>
type Time string # ISO 8601

enum Department {
  ACCOUNTING
  ADMINISTRATION
  CUSTOMER_SERVICE
  DEVELOPMENT

  # Reserved for the CEO
  JSMITH = 99
}

type Customer {
  name: string
  email: string
  address: Address
  orders: []{
    orderId: i64
    quantity: i32
  }
  metadata: map[string]data
}

type Employee {
  name: string
  email: string
  address: Address
  department: Department
  hireDate: Time
  publicKey: optional<PublicKey>
  metadata: map[string]data
}

type TerminatedEmployee void

type Person (Customer | Employee | TerminatedEmployee)

type Address {
  address: [4]string
  city: string
  state: string
  country: string
}
```

Appendix B. Example Messages

Some basic example messages in hexadecimal are provided for the schema specified in Appendix A.

A "Person" value of type "Customer" with the following values:

name	James Smith
email	jsmith@example.org
address	123 Main Street; Philadelphia; PA; United States
orders (1)	orderId: 4242424242; quantity: 5
metadata	(unset)

Encoded BARE message:

```
00 0b 4a 61 6d 65 73 20 53 6d 69 74 68 12 6a 73
6d 69 74 68 40 65 78 61 6d 70 6c 65 2e 6f 72 67
0b 31 32 33 20 4d 61 69 6e 20 53 74 00 00 00 0c
50 68 69 6c 61 64 65 6c 70 68 69 61 02 50 41 0d
55 6e 69 74 65 64 20 53 74 61 74 65 73 01 b2 41
de fc 00 00 00 00 05 00 00 00 00
```

A "Person" value of type "Employee" with the following values:

name	Tiffany Doe
email	tiffanyd@acme.corp
address	123 Main Street; Philadelphia; PA; United States
department	ADMINISTRATION
hireDate	2020-06-21T21:18:05Z
publicKey	(unset)
metadata	(unset)

Encoded BARE message:

```
01 0b 54 69 66 66 61 6e 79 20 44 6f 65 12 74 69
66 66 61 6e 79 64 40 61 63 6d 65 2e 63 6f 72 70
0b 31 32 33 20 4d 61 69 6e 20 53 74 00 00 00 0c
50 68 69 6c 61 64 65 6c 70 68 69 61 02 50 41 0d
55 6e 69 74 65 64 20 53 74 61 74 65 73 01 19 32
30 32 30 2d 30 36 2d 32 31 54 32 31 3a 31 38 3a
30 35 2b 30 30 3a 30 30 00 00
```

A "Person" value of type "TerminatedEmployee":

Encoded BARE message:

```
02
```

Author's Address

Drew DeVault
SourceHut
454 E. Girard Ave #2R
Philadelphia, PA 19125
United States of America

Phone: +1 719 213 5473
Email: sir@cmpwn.com