uuidrev
Internet-Draft
Updates: 9562 (if approved)
Intended status: Standards Track
Expires: 12 October 2026

K. R. Davis
Cisco Systems
10 April 2026

Longer Universally Unique IDentifiers (UUIDs)
draft-davis-uuidrev-uuid-long-00

Abstract

This document extends Universally Unique Identifiers (UUIDs) beyond 128 bits to facilitate enhanced collision resistance and proper room for embedding additional data within a given UUID algorithm.

These longer variable-length UUIDs ("UUID Long") leverage a previously unused variant bit "F" and feature a new sub-typing mechanism created to ensure there is enough space to define many future UUID algorithms within this new variant of UUIDs.

This document updates RFC9562.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://github.com/kyzer-davis/uuid-long/blob/main/draft-davis-uuidrev-uuid-long.md. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-davis-uuidrev-uuid-long/.

Discussion of this document takes place on the Revise Universally Unique Identifier Definitions (uuidrev) Working Group mailing list (mailto:uuidrev@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/uuidrev/. Subscribe at https://www.ietf.org/mailman/listinfo/uuidrev/.

Source for this draft and an issue tracker can be found at https://github.com/kyzer-davis/uuid-long.

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 12 October 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction

There are a few main driving factors behind extending UUID beyond 128 bits covered by the next sections.

1.1. The Need for Increased Entropy

While existing UUID formats provide sufficient entropy for most use cases; there exist scenarios where even more entropy is required to further reduce collision probabilities or guessability.

Further, while creating UUIDv7 during the draft phases of RFC9562, a common discussion point surrounded the number of bits allocated to entropy vs the number of bits allocated to the embedded timestamp. The 128 bit limits on UUID created a situation where the community had to balance timestamp granularity vs entropy. This resulted in "sliding" bits one way or other trying to find a happy medium. While in the end a fine balance was achieved; the entire problem could have been avoided if there were more bits available to the UUID format.

With the additional length added by UUID Long; an application can generate a UUID with certainty that it is truly "unique across space and time".

1.2. Requirements for Additional Embedded Data

Some implementations require more than 128 bits to properly embed all of the application specific data they require for a given UUID algorithm. Some examples include database metadata like entity types, checksum values, shard/partition identifiers (see [OrderlyID]), and even node identifiers for distributed UUID generation.

UUID Long provides ample bit space for an algorithm to properly embed all of the items required for the application logic to function.

1.3. A better UUID sub-typing system

128 bit UUIDs within the "OSF DCE / IETF" variant space are limited to 16 versions. This version limit artificially inhibits innovation of new UUID algorithms (a problem partly solved by UUIDv8).

This drawback of the "OSF DCE / IETF" variant space was observed while working on [RFC9562], in particular to future name-based UUID layouts that replace "UUIDv3" and "UUIDv5". With the number of hashing algorithms available and the possibility that at any point one may be deprecated; there was little chance of getting consensus on leveraging one of the few remaining versions for such an algorithm.

With UUID Long, as per section Section 3.2, there is ample room for future UUID Long Algorithms.

1.4. Beyond Fixed-Length

With a fixed-length UUID, there is no room to grow as future protocols change and severely limits the ability to innovate in the space.

A point of contention that has come up many times across the community is that 128 bits is not enough for modern protocols and applications. Common alternate unique identifier lengths are 160 bits, 192 bits and 256 bits.

UUID Long provides these fixed length values and the flexibility to accommodate future needs via the variable-length design.

Some example items that may change over time are, but not limited to, hashing algorithms, signature algorithms, and post-quantum computing related algorithms. Any of these could exceed a limit if UUID Long does not select a large enough maximum value.

See Section 7 for more considerations around generating and parsing UUID Long values.

2. Conventions and Definitions

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

2.1. Notational Conventions

Throughout this document "UUID Long" generally references any variable length UUID longer than 128 bits while "UUID Short" references fixed-length 128-bit UUIDs in the prose of this document.

Field and Bit Layout in this document use a custom format borrowed from [RFC9000] rather than those featured in [RFC9562]. The purpose of this format is to summarize, not define, protocol elements. Prose defines the complete semantics and details of structures.

Layout items are named and then followed by a list of fields surrounded by a pair of matching braces. Each field in this list is separated by commas.

Individual fields include length information, plus indications about fixed value, optionality, or repetitions. Individual fields use the following notational conventions, with all lengths in bits:

x (A):
: Indicates that x is A bits long

x (L) = C:
: Indicates that x has a fixed value of C; the length of x is described by L, which can use any of the length forms above

x (..):
: Indicates that x has a variable length with no fixed upper limit

x (A..B):
: Indicates that x can be any length from A to B; A can be omitted to indicate a minimum of zero bits, and B can be omitted to indicate no set upper limit

This document uses network byte order (that is, big endian) values. Fields are placed starting from the high-order bits of each byte.

By convention, individual fields reference a complex field by using the name of the complex field.

Figure 1 provides an example:

```
Example Structure {
  One-bit Field (1),
  7-bit Field with Fixed Value (7) = 61,
  Arbitrary-Length Field (..),
  Variable-Length Field (8..24),
  Field With Minimum Length (16..),
  Field With Maximum Length (..128),
}
```

Figure 1: Example Format

3. UUID Long Format

At the core UUID Long features the same base characteristics as [RFC9562], Section 4 featured in UUID Short. UUID Long may be represented in all of the same ways as you would expect with a UUID (e.g text, integer, binary, UUID URN, etc.)

The UUID Long Data Block starts at bit 129 separated by the dash character "-" in the textual representation of UUID Long. This separation allows at-a-glance readability around the variable-length UUID Long Data.

The generalized layout of UUID Long is Figure 2.

```
UUID Long Structure {
  UUID Short Part A (64),
  UUID Variant (4) = 0xF,
  UUID Short Part B (60),
  UUID Long Data (32..3968),
}
```

Figure 2: Example UUID Long Bit and Field Layout

Further, the base UUID Short string format with hex and dashes is also found in the string format of UUID Long. Including this in the base syntax ensures backwards compatibility as per Section 6. The UUID Long string representation is defined by Figure 3 and Table 1.

```
xxxxxxxx-xxxx-xxxx-Fxxx-xxxxxxxxxxxx-yy..zz
```

Figure 3: UUID Long Field Layout in Hex

ID	Description	Bits
x	Short UUID Bits	124
F	Frozen Variant byte (for backwards compatibility). See Section 3.1	4
yy..zz	Variable length UUID Long Data with length described by LLLL. Minimum one byte, maximum 496 bytes	32..3968

Table 1: UUID Long String Layout Descriptors

A properly constructed UUID Long value will be, at a minimum, 160 bits or 20 octets. The maximum value for a UUID Long is computed as UUID Short Length (128) + Maximum UUID Long Data Length (3,968) which is 4,096 bits (512 octets). Note that these calculations do not take into account the UUID Long Encoding Block (4 bytes or 32 bits) described in Section 3.2 which is placed at the start of the variable-length UUID Long data.

While it is theoretically possible to extend UUID Long beyond the maximum total length of 4,096 bits; this value was chosen to be sufficiently large to allow for any type of data that needs to be implemented. If the time comes when UUID Long needs to be extended beyond 4,096 bits; this specification can easily be updated to allow for larger values due to the nature of the variable-length design of UUID Long.

This would be accomplished by updating the UUID Encoding Block definition (which already supports describing UUID Longs beyond 4096 bits via the two byte field) and updating the maximum length of the UUID Long Data field in the layout definitions above.

3.1. Variant Field

This section updates [RFC9562], Section 4.1 to split the unused final variant of "111x" into two variants as described by the Table 2 table.

Splitting the final variant space ensures that the "E" variant may be used by future UUID Short definitions while the "F" variant is used to signal a UUID Long Variant and maximum value for UUID Short as per [RFC9562], Section 5.10.

These "F"rozen variant bits are set to all 1’s (b1111).

Msb0	Msb1	Msb2	Msb3	Variant	Description
1	1	1	0	E	Reserved for future definition.
1	1	1	1	F	The variant used by UUID Long in this document. Also includes Max UUID as per [RFC9562], Section 5.10.

Table 2: UUID Variant Updates

UUID Long algorithms featuring the frozen Variant F MUST use the sub-typing logic and encoding block described in Section 3.2.

3.2. Sub-Typing Logic and Encoding Block

UUID Long does not re-use the "version" nomenclature (or bit positions unless otherwise noted) from [RFC9562]. This serves to help implementations easily distinguish 128 bit or 128+ bit UUIDs in text and provide an opportunity for defining a better sub-typing system within this new variant space.

UUID Long instead moves the sub-typing logic to a new 4 byte UUID Long Encoding Block placed at the start of the encoded UUID Long algorithm and prefixed after the UUID sub-variant algorithm is computed and the underlying UUID Long value is created.

The first 2 bytes of the UUID Long Encoding Block feature a sub-typing system with two levels of hierarchy. The first is a "Sub-Variant" abbreviated "sv" which indicates the grouping of UUID Long algorithm types. The second level of UUID Long sub-typing is defined as simply the "algorithm" which can be abbreviated "a". The Sub-Variant plus Algorithm (SV+A) serve as the identity behind a particular UUID Long value.

With this in mind "Sub-Variant 1, Algorithm 4" can be expressed as "sv1a4" or "UUIDsv1a4" throughout this document. Note that "UUIDv4" or "UUID Version 4" is usually used to reference a UUID algorithm as specified by [RFC9562], Section 5.4 and does not represent UUID Long algorithms in this document.

The final 2 bytes of the UUID Long Encoding Block include a length descriptor, expressed in octets (bytes), for the variable-length UUID Long data which can be used by applications in order to understand

where a UUID Long value ends. The value stored in this field represents the number of octets of UUID Long data following the UUID Short portion, not a bit count or character count. The length descriptor MUST NOT take into account the UUID Long Encoding Block itself and only describe the octet length of the variable-length UUID Long data.

The full 4 byte UUID Encoding block can be observed in Figure 4 or Figure 5 and described succinctly in Table 3.

```
UUID Long Encoding Block {
  Sub-Variant Encoding (8),
  Algorithm Encoding (8),
  UUID Long Data Length Descriptor (16)
}
```

Figure 4: Example UUID Long Encoding Block Bit and Field Layout

```
SVAALLLL-xxxxxxxx-xxxx-xxxx-Fxxx-xxxxxxxxxxxx-xx..zz
```

Figure 5: UUID Long Encoding Block and UUID Long Field Layout in Hex

ID	Description	Bits
SV	One Byte Sub-Variant with 256 possible values	8
AA	One Byte Algorithm with 256 possible values	8
LLLL	Two Byte length descriptor in octets with a max value of 496 describing the UUID Long data	16

Table 3: UUID Long String Layout Descriptors

3.3. Sub-Variants

UUID Long defines four starting sub-variant groupings as defined by Table 4.

Sub-Variant ID	Description
sv0	Experimental/Custom Algorithms
sv1	Random Based Algorithms
sv2	Time Based Algorithms
sv3	Hash-based Algorithms
sv4-sv255	Reserved for future algorithm groupings as required

Table 4: UUID Long Sub-Variants

Future sub-variants in the space (sv4-sv255) can be allocated where a grouping of algorithms is required; but if a current sub-variant is applicable for a new algorithm, the new algorithm should be grouped under a given sub-variant.

The four starting sub-variant groupings mirror the four generic types of UUID algorithms observed in [RFC9562].

3.4. Encoding

The default, widely implemented, "hex and dash" text presentation format of 128 bit UUID short values is already inefficient at conveying the underlying bits of UUID. This problem is only exacerbated by creating 128+ bit UUIDs.

Implementations generating or parsing UUID Long values MUST utilize at least one method defined in [ALT_UUID_ENCODING] to create a more efficient UUID Long value. The "extended hex and dash" format MAY be utilized for UUID Long though it is discouraged. The usage of this format throughout this document is for illustrative purposes only.

For example the minimum and maximum UUID Long values using those found in [ALT_UUID_ENCODING] are found in the table Table 5 found by computing the maximum character length of an all 1s value at each bit length. At Base32 and above the number of characters used by the minimum length of UUID Long is still less than the 128 bit hex and dash format of UUID Short without the UUID Long Encoding Block.

Applications MUST ensure that UUID Long values leverage natural boundaries and pad the least significant, right-most bits where required to achieve a proper value for encoding as various BaseXX Alphabets in [ALT_UUID_ENCODING].

Encoding	Variant	Min UUID Long (160 bits)	Max UUID Long (4096 bits)
Base16	THIS DRAFT	8 + 45 (53)	8 + 1029 (1037)
Base16	[RFC4648], Section 8	8 + 40 (48)	8 + 1024 (1032)
Base32	[RFC4648], Section 6	7 + 32 (39)	7 + 820 (827)
Base32	[RFC4648], Section 7	7 + 32 (39)	7 + 820 (827)
Base32	[Base32human]	7 + 32 (39)	7 + 820 (827)
Base36	---	7 + 31 (38)	7 + 793 (800)
Base52	---	6 + 29 (35)	6 + 719 (725)
Base58	[Base58btc]	6 + 28 (34)	6 + 700 (706)
Base62	[Base62ieee]	6 + 27 (33)	6 + 688 (694)
Base62	[Base62sort]	6 + 27 (33)	6 + 688 (694)
Base64	[RFC4648], Section 4	6 + 27 (33)	6 + 683 (689)
Base64	[RFC4648], Section 5	6 + 27 (33)	6 + 683 (689)
Base64	[Base64sort]	6 + 27 (33)	6 + 683 (689)
Base85	[Z85]	5 + 25 (30)	5 + 640 (645)

Table 5: Alt UUID Encoding Length Comparison

4. Fixed-Length 160/192/256 bit UUID Long

Although UUID Long is variable length and features a very large top end; implementations may end up generating fixed-length UUID Long Values as described in this section. See Section 7 for security discussion about this topic.

A common UUID length requested by the community is 160, 192 and 256 bit UUID values. With UUID Long generating these values is a trivial task.

We can easily calculate the new bits by using the following logic (for completeness up to 2048 has been illustrated.) Once calculated these can be filled with the appropriate application specific data. Apply the Variant bits as per Section 3.1 and the appropriate sub-variant algorithm encoding as per Section 3.2 and then build the UUID Long Encoding Block and you now have a fixed-length UUID Long value of the required length.

160 - UUID Short Length (128) = 32 bits of additional UUID Long data
192 - UUID Short Length (128) = 64 bits of additional UUID Long data
256 - UUID Short Length (128) = 128 bits of additional UUID Long data
512 - UUID Short Length (128) = 384 bits of additional UUID Long data
1024 - UUID Short Length (128) = 896 bits of additional UUID Long data
2048 - UUID Short Length (128) = 1920 bits of additional UUID Long data

5. UUID Long Algorithms

As mentioned in Section 3.2, UUID Long Algorithms are grouped at the Sub-Variant level.

UUID Long first maps the [RFC9562] versions to algorithms in the appropriate sub-variant algorithm space. The sub-variant algorithm identifier has been ’smeared’ for ease of understanding when referencing the old values. For example: "UUIDv4 == UUIDsv1a4" and "UUIDv7 == UUIDsv2a7" where the final number in each abbreviation matches.

The first 16 sub-variant algorithm values (a0-a15) in each sub-variant space are reserved for matching the appropriate [RFC9562] versions. This ensures that a future IETF spec can define both a UUID Short Version and UUID Long sub-variant algorithm that line up nicely to each other. With 256 possible sub-variant algorithms in each of the 256 sub-variant spaces; 16 reserved sub-variant algorithm identifiers should be no problem.

When the time comes that all 16 [RFC9562] versions have been allocated to their appropriate UUID Long SV+A IDs, or are no longer in need of the mapping space; outstanding sub-variant algorithm identifiers MAY be used by future UUID Long specifications.

Other UUID sub-types that exist in other variant spaces MAY leverage unused sub-variant algorithm identifiers, starting at a16, for UUID Long versions of the existing algorithms.

Generally speaking for sub-variant algorithms based on the RFC9562 versions; there are two main areas that need to be described:

1. How to leverage the new UUID Long bits.
2. Define the RFC9562 version bit handling.

The following sections illustrate the current sub-variant algorithm mappings for UUID Long along with the methods for generating a UUID Long value for a given sub-variant algorithm.

For all algorithms the following two statements apply, even if they are based on an RFC9562-based algorithm.

* The Variant bits are always overwritten to "F" as per Section 3.1.
* The UUID Long Encoding Block is encoded with the sub-variant id, algorithm id and long data length descriptor as per Figure 4.

TODO: where to slot UUIDv2

5.1. Sub-Variant 0 (Experimental/Custom)

Algorithm Identifiers in this sub-variant space SHOULD be used for custom, experimental or vendor-specific use cases. UUIDv8 has been mapped to UUIDsv0a8 in this document and is the only current algorithm in this space defined by Table 6.

Vendors are encouraged to use this space for testing and experimental algorithms before finalization into another sub-variant algorithm identifier. At which point the Algorithm Identifier in this sub-variant can be released for continued use.

SV ID	Algorithm ID	Name	9562 Version (if applicable)	Algorithm Definition Link
sv0	a8	Custom	UUIDv8	Section 5.1.1

Table 6: Sub-Variant 0 Algorithms

5.1.1. sv0a8

sv0a8 is based on UUIDv8 from [RFC9562], Section 5.8 with the following deltas:

* UUID Long Data can be leveraged as a new "custom_d" field of arbitrary size within the UUID Long data as shown in Figure 6. The length of this new data is calculated and inserted into the UUID Long Encoding Block.
* The version behavior does not need to remain the same as [RFC9562], Section 4.2 and can be set to whatever an implementation desires.

```
UUIDsv0a8 Structure {
  custom_a (48),
  9562 Version (4),
  custom_b (12),
  UUID Variant (4) = 0xF,
  custom_c (60),
  custom_d (8..3936),
}
```

Figure 6: Example sv0a8 Bit and Field Layout

Note that where possible, for experimental use cases, implementations are encouraged to apply for a sub-variant algorithm for their UUID Long Algorithm.

TODO: Link to process section if this is finalized.

5.2. Sub-Variant 1 (Random)

Algorithm Identifiers in this sub-variant space MUST be related to random, pseudorandom, or other similar methods of generating UUID Long values.

UUIDv4 has been mapped to UUIDsv1a4 in this document and is the only current algorithm in this space defined by Table 7.

SV ID	Algorithm ID	Name	9562 Version (if applicable)	Algorithm Definition Link
sv1	a4	Random	UUIDv4	Section 5.2.1

Table 7: Sub-Variant 1 Algorithms

5.2.1. sv1a4

sv1a4 is based on UUIDv4 from [RFC9562], Section 5.4 with the following deltas:

* UUID Long Data can be leveraged as a new "random_d" field of arbitrary size within the UUID Long data as shown in Figure 7. The length of this new data is calculated and inserted into the UUID Long Encoding Block.
* The version behavior does not need to remain the same as [RFC9562], Section 4.2 and these 4 version bits MAY also be randomized.

```
UUIDsv1a4 Structure {
  random_a (48),
  9562 Version (4),
  random_b (12),
  UUID Variant (4) = 0xF,
  random_c (60),
  random_d (8..3936),
}
```

Figure 7: Example sv1a4 Bit and Field Layout

Examples of UUIDsv1a4 can be seen in Appendix B.1.

5.3. Sub-Variant 2 (Time)

Algorithm Identifiers in this sub-variant space MUST be related to UUIDs which feature timestamps.

UUIDv1, UUIDv6 and UUIDv7 have been mapped to UUIDsv2a1, UUIDsv2a6, UUIDsv2a7 where required as per Table 8.

SV ID	Algorithm ID	Name	9562 Version (if applicable)	Algorithm Definition Link
sv2	a1	Gregorian Time-based	UUIDv1	Section 5.3.1
sv2	a6	Reordered Gregorian Time-based	UUIDv6	Section 5.3.2
sv2	a7	Unix Time-based (MS)	UUIDv7	Section 5.3.3

Table 8: Sub-Variant 2 Algorithms

TODO: Discuss if we want sv2a16 as Unix Time-based (Nanosecond time resolution)... this timestamp resolution was a big ask from the community.

TODO: Reserve an sv2a17 for custom epoch time, also a big item that came from the community.

5.3.1. sv2a1

sv2a1 is based on UUIDv1 from [RFC9562], Section 5.1 with the following deltas:

* UUID Long Data can be leveraged as an "extended_node" field within the UUID Long data as shown in Figure 8. The length of this new data is calculated and inserted into the UUID Long Encoding Block.
* The node value MAY feature IEEE 802 MAC address and random data of arbitrary size or be fully randomized using portions of the original node bits and variable-length UUID Long data.
* The version bits MAY also be randomized since this does not affect the sortability of this algorithm.
* The clock_seq value is reduced by 2 bits to accommodate the new variant bits as per Section 3.1.

```
UUIDsv2a1 Structure {
  time_low (32),
  time_mid (16),
  9562 Version (4),
  time_high (12),
  UUID Variant (4) = 0xF,
  clock_seq (12),
  node (48),
  extended_node (8..3936),
}
```

Figure 8: Example sv2a1 Bit and Field Layout

5.3.2. sv2a6

sv2a6 is based on UUIDv6 from [RFC9562], Section 5.6 with the following deltas:

* UUID Long Data can be leveraged as an "extended_node" field within the UUID Long data as shown in Figure 9. The length of this new data is calculated and inserted into the UUID Long Encoding Block.
* The node value MAY feature IEEE 802 MAC address and random data of arbitrary size or be fully randomized using portions of the original node bits and variable-length UUID Long data.
* The version behavior MUST remain the same as [RFC9562], Section 4.2 to ensure proper sortability, which is a key feature of this UUID’s algorithm.
* The clock_seq value is reduced by 2 bits to accommodate the new variant bits as per Section 3.1.

```
UUIDsv2a6 Structure {
  time_high (32),
  time_mid (16),
  9562 Version (4) = 0x6,
  time_low (12),
  UUID Variant (4) = 0xF,
  clock_seq (12),
  node (48),
  extended_node (8..3936),
}
```

Figure 9: Example sv2a6 Bit and Field Layout

5.3.3. sv2a7

sv2a7 is based on UUIDv7 [RFC9562], Section 5.7 with the following deltas:

* UUID Long Data can be leveraged as a "rand_c" field within the UUID Long data as shown in Figure 10. The length of this new data is calculated and inserted into the UUID Long Encoding Block.
* The version behavior MUST remain the same as [RFC9562], Section 4.2 to ensure proper sortability, which is a key feature of this UUID’s algorithm.

```
UUIDsv2a7 Structure {
  unix_ts_ms (48),
  9562 Version (4) = 0x7,
  rand_a (12),
  UUID Variant (4) = 0xF,
  rand_b (60),
  rand_c (8..3936),
}
```

Figure 10: Example sv2a7 Bit and Field Layout

An Example of UUIDsv2a7 can be seen in Appendix B.2.

5.4. Sub-Variant 3 (Hashing)

Algorithm Identifiers in this sub-variant space MUST be related to hash-based UUIDs computed using "names" and "namespaces" as defined by [RFC9562], Section 6.5. UUIDv5 has been mapped to UUIDsv3a5 while new hashing protocols utilize algorithms a16 through a27.

SV ID	Algorithm ID	Name	9562 Version (if applicable)	Algorithm Definition Link	Reference
sv3	a5	SHA-1	UUIDv5	Section 5.4.1	[FIPS180-4]
sv3	a16	SHA-224		Section 5.4.2	[FIPS180-4]
sv3	a17	SHA-256		Section 5.4.2	[FIPS180-4]
sv3	a18	SHA-384		Section 5.4.2	[FIPS180-4]
sv3	a19	SHA-512		Section 5.4.2	[FIPS180-4]
sv3	a20	SHA-512/224		Section 5.4.2	[FIPS180-4]
sv3	a21	SHA-512/256		Section 5.4.2	[FIPS180-4]
sv3	a22	SHA3-224		Section 5.4.2	[FIPS202]
sv3	a23	SHA3-256		Section 5.4.2	[FIPS202]
sv3	a24	SHA3-384		Section 5.4.2	[FIPS202]
sv3	a25	SHA3-512		Section 5.4.2	[FIPS202]
sv3	a26	SHAKE128		Section 5.4.2	[FIPS202]
sv3	a27	SHAKE256		Section 5.4.2	[FIPS202]

Table 9: Sub-Variant 3 Algorithms

Note that UUIDv3 has not been mapped to UUIDsv3a3 because the current MD5-based algorithm from [RFC9562], Section 5.3 does not have any requirements for bits past 128. Thus there is no need for a UUID Long equivalent of this algorithm.

5.4.1. sv3a5

sv3a5 is based on UUIDv5 from [RFC9562], Section 5.5 with the following deltas:

* The original algorithm requires that parts of the SHA-1 hash be truncated to fit the 128 bit layout; however, with UUID Long these extra bits can be embedded into the UUID Long Data as "sha1_discard" seen in Figure 11. The length of this discarded data is calculated and inserted into the UUID Long Encoding Block.
* The version MUST NOT remain the same as [RFC9562], Section 4.2. As a result, the bits that would have been overwritten to a hard coded "5" are now left as the original portions of the hash.

```
UUIDsv3a5 Structure {
  sha1_high (48),
  9562 Version (4),
  sha1_mid (12),
  UUID Variant (4) = 0xF,
  sha1_low (60),
  sha1_discard (8..3936),
}
```

Figure 11: Example sv3a5 Bit and Field Layout

An Example of UUIDsv3a5 can be seen in Appendix B.3.

5.4.2. sv3a16 - sv3a27

sv3a16 - sv3a27 describe Name-Based UUID generation using new hashing algorithms. From an operational standpoint the same fields are described for all of these algorithms. This is shown in Figure 12.

The algorithm and creation of these UUID Long values is the same as [RFC9562], Section 5.5 with the following deltas:

* The desired hash algorithm is used in place of SHA-1.
* The 9562 Version is not used and those 4 bits retain their value from the hash.
* The bits beyond 128 are placed in "hash_low" with the length calculated and inserted into the UUID Long Encoding Block.

```
UUID Long Hash-Based Structure {
  hash_high (64),
  UUID Variant (4) = 0xF,
  hash_middle (60),
  hash_low (8..3936),
}
```

Figure 12: Example UUID Long Hash-Based Bit and Field Layout

Example of UUIDsv3a17, using SHA-256, can be seen in Appendix B.4.

6. Compatibility with 128 Bit UUIDs

UUID Long values are prefixed with a 32-bit UUID Long Encoding Block followed by the UUID value itself. The UUID portion (bits 33 through 160 at minimum) is structured identically to a UUID Short in the first 128 bits. However, because UUID Long uses the "F" variant (b1111), many existing UUID parsers and database UUID types will reject the UUID portion during variant validation.

To derive a compatible 128-bit UUID from a UUID Long value, an implementation MUST first strip the 32-bit encoding block prefix to isolate the 128-bit UUID Short portion. Stripping the prefix alone does NOT produce a value that is generally accepted as a valid [RFC9562] UUID because the F variant may not be recognized by existing parsers.

Implementations that need a valid 128-bit UUID SHOULD then actively transform the isolated value by overwriting the variant bits to produce a valid "OSF DCE / IETF" variant (b10xx) UUID. For example, an implementation could clear the two most significant variant bits to produce a valid RFC 9562 variant before passing the value to downstream systems. Implementations MUST document any such transformation and be aware that the resulting 128-bit value will differ from the original UUID Long’s UUID Short portion.

Note that the version bit-space is not a requirement in UUID Long thus some UUID long algorithms may have varying data at this position. The bits still exist, so for systems that do not read the variant bit first, they may see inconsistent results if trying to read only the version or version and then variant.

7. Security Considerations

UUID Long shares many of the same security considerations as [RFC9562]. The main security consideration with UUID Long is the maximum length of data and possible buffer overflows which lead to other vulnerabilities. Implementations that only expect 128 bit UUIDs MUST NOT read beyond 128 bits.

7.1. Parsing and Length Validation

Implementations that parse UUID Long values MUST validate the UUID Long Data Length Descriptor field before allocating memory or reading data. Specifically, a parser MUST:

* Verify that the length descriptor does not exceed an implementation-defined maximum.

* Verify that the length descriptor does not exceed the number of remaining input bytes.

* Reject any UUID Long value where either check fails.

General-purpose UUID libraries that do not have application-specific requirements SHOULD default to a maximum UUID Long Data length of 128 bytes (1024 bits). This default SHOULD be configurable to allow applications with different requirements to adjust the limit as needed.

7.2. Generation Limits

An implementation may choose to put limits on the length of UUID Long values that are generated to protect from using UUID Long as a conveyance mechanism to retrieve buffer overflowed data exploited by other means. For example, an implementation may choose to generate UUID Long values of a maximum length of 1024 bits and no more. Thus limiting the potential for side-channel exploits that may try to take advantage of the variable-length properties of UUID Long.

7.3. Data Integrity

By default the UUID Long value (and UUID Short) do not feature any hash/signature method. An attacker could modify the UUID Long Data Length Descriptor bits and include new data in an attempt to force some buffer overflow condition or append data that was not part of the original algorithm. An algorithm MAY choose to create a hash/ digital signature on the final UUID Long value and provide this hash to a peer in order to provide some level of data integrity.

Further, where possible introspection into the UUID is discouraged as per [RFC9562], Section 6.12.

8. IANA Considerations

TODO: IANA when things are finalized. Things like add sub-variant algorithms to sub-types section of UUID registry. https://www.iana.org/assignments/uuid/uuid.xhtml#uuid-subtypes

9. References

9.1. Normative References

[ALT_UUID_ENCODING]
"Alternate UUID Encoding Methods", n.d., <https://datatracker.ietf.org/doc/draft-davis-uuidrev-alt-uuid-encoding-methods/>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <https://www.rfc-editor.org/rfc/rfc9562>.

9.2. Informative References

[Base32human]
Crockford, D. and K. Davis, "Base32 for Humans", April 2026, <https://datatracker.ietf.org/doc/draft-crockford-davis-base32-for-humans/>.

[Base58btc]
Bitcoin, "Bitcoin Base58 Implementation", commit fae71d3, November 2008, <https://github.com/bitcoin/bitcoin/blob/master/src/base58.cpp>.

[Base62ieee]
IEEE, "A secure, lossless, and compressed Base62 encoding", November 2008, <https://ieeexplore.ieee.org/document/4737287>.

[Base62sort]
Wu, P.-C., "A base62 transformation format of ISO 10646 for multilingual identifiers", August 2001, <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.408>.

[Base64sort]
Davis, K., "A Sortable Base64 Alphabet", December 2025, <https://datatracker.ietf.org/doc/draft-brown-davis-base64-sort/>.

[FIPS180-4]
National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[FIPS202] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

[OrderlyID]
piljoong, "Distributed ID Formats Are Architectural Commitments, Not Just Data Types", December 2025, <https://piljoong.dev/posts/distributed-id-generation-complicated/>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[Z85] iMatix Corporation, "32/Z85", 2013, <https://rfc.zeromq.org/spec/32/>.

Appendix A. Changelog

draft-00:

* Initial Release

Appendix B. Test Vectors

Due to the variable length nature of the UUID Long Data field there could be an infinite number of test vectors. The sections below attempt to summarize the key points of the sub-variant algorithms as described by the body of this document.

TODO: Add other test vectors as things are finalized.

B.1. Example sv1a4 values

The table, Table 10, details varying levels of random bits, as well as commonly requested UUID Long lengths (160/192/256) in an attempt to illustrate the difference between UUID length and Embedded Data Length. This is all compared to UUIDv4 as seen in the first row of the table.

For example, one can generate a fixed 256 bit UUID Long value with random data and this UUID Long value will contain 252 bits of random after applying the 0xF variant and 0x010400 encoding block for sv1a4 with 32 bits of long data bringing the total length of the UUID Long to 288 bits.

256 bit length with 252 bits of random data is far larger than UUIDv4’s 122 bits of random data.

However, if further guarantees are required around randomness and size of the outputs are not a problem, then generating a 512 bit UUID which features 508 bits of random data can also solve an applications needs.

DOC	Type	UUID Length (with encoding block)	Variant	Sub-Type	Total Random	Long Data	Example
RFC9562	UUIDv4	128	2	4	122	n/a	73e94fe0-e951-4153-aaf3-50e4e6089d9d
DRAFT	sv1a4	160 (192)	4	32	156	32 (x0020)	01040020-3ed8afd7-4a31-e2c5-f9c2-63e65cee20ee-0bb665e0
DRAFT	sv1a4	192 (224)	4	32	188	64 (x0040)	01040040-2f70cb74-91e5-f901-fe27-9d9d9704a625-aea06e67af31c3ef
DRAFT	sv1a4	256 (288)	4	32	252	128 (x0080)	01040080-35225f5c-78a0-50ba-f1c0-4ea2d181b096-7560113a765de7610e33d2aa69142289
DRAFT	sv1a4	512 (544)	4	32	508	384 (x0180)	01040180-2e675f90-fc04-b5ae-f687-4eb094c7c24e-649756dcee4b980e674f9ff0bed1c0a996b1b9fae89ea0107bc703e8cb64ccb58d7e8ad573747beb32f6c73d91b4d2ca

Table 10: UUID Random Example

B.2. Example sv2a7 Value

This example UUIDsv2a7 test vector utilizes a well-known Unix epoch timestamp with millisecond precision to fill the first 48 bits.

rand_a, rand_b, rand_c are filled with 64 bits of random data.

The timestamp is Tuesday, February 22, 2022 2:22:22.00 PM GMT-05:00 represented as 0x017F22E279B0 or 1645557742000

UUIDsv2a7 Test Vector {
  UUID Long Encoding Block (32) = 0x02070040,
  unix_ts_ms (48) = 0x017F22E279B0,
  9562 Version (4) = 0x7,
  rand_a (12) = 0xFE6,
  UUID Variant (4) = 0xF,
  rand_b (60) = 0x76E2B86F151FB04,

```
  rand_c (64) = 0xE6B4400B21E888CD,
}

02070040-017F22E2-79B0-7FE6-F76E-2B86F151FB04-E6B4400B21E888CD
```

B.3. Example sv3a5 Value

```
   Namespace (DNS):  6ba7b810-9dad-11d1-80b4-00c04fd430c8
   Name:             www.example.com
   ----------------------------------------------------------
   SHA-1: 2ed6657de927468b55e12665a8aea6a22dee3e35

   A:          2ed6657d-e927-468b-55e1-2665a8aea6a2-2dee3e35
   B:          xxxxxxxx-xxxx-xxxx-Fxxx-xxxxxxxxxxxx
   C:          2ed6657d-e927-468b-f5e1-2665a8aea6a2
   D:                                                -2dee3e35
   E:          2ed6657d-e927-468b-f5e1-2665a8aea6a2-2dee3e35
   F: 03050020-2ed6657d-e927-468b-f5e1-2665a8aea6a2-2dee3e35
```

   * Line A details the full SHA-1 as a hexadecimal value with the dashes inserted.
   * Line B details the F variant hexadecimal positions, which must be overwritten.
   * Line C details the final value after the variant has been overwritten.
   * Line D details the leftover values from the original SHA-1 computation (Note that these have a length of 32 bits)
   * Line E details the leftover values appended to form the full UUID Long of form sv3a5 without the encoding block.
   * Line F details the full UUID Long of form sv3a5 with the encoding block prefixed.

B.4. Example sv3a17 Value

```
Namespace (DNS): 6ba7b810-9dad-11d1-80b4-00c04fd430c8
Name:            www.example.com
------------------------------------------------------------------
SHA-256: 5c146b143c524afd938a375d0df1fbf6fe12a66b645f72f6158759387e51f3c8

A:          5c146b14-3c52-4afd-938a-375d0df1fbf6-fe12a66b645f72f6158759387e51f3c8
B:          xxxxxxxx-xxxx-xxxx-Fxxx-xxxxxxxxxxxx
C:          5c146b14-3c52-4afd-f38a-375d0df1fbf6
D:                                                -fe12a66b645f72f6158759387e51f3c8
E:          5c146b14-3c52-4afd-f38a-375d0df1fbf6-fe12a66b645f72f6158759387e51f3c8
F: 03110080-5c146b14-3c52-4afd-f38a-375d0df1fbf6-fe12a66b645f72f6158759387e51f3c8
```

   * Line A details the full SHA-256 as a hexadecimal value with the dashes inserted.
   * Line B details the F variant hexadecimal positions, which must be overwritten.
   * Line C details the final value after the variant has been overwritten.
   * Line D details the leftover values from the original SHA-256 computation (Note that these have a length of 128 bits)

* Line E details the leftover values appended to form the full UUID Long of form sv3a17 without the encoding block.
* Line F details the full UUID Long of form sv3a17 with the encoding block prefixed.

B.5. Further Encoding Examples

The following test vectors illustrate the minimum (160-bit) UUID Long values encoded as all 0s and all 1s across the various BaseXX alphabets from Table 5.

The encoding block for these examples uses sv0a0 (SV=0x00, AA=0x00) with LLLL value 0x0004 (4 bytes of long data).

All 0s values have the F variant set at the appropriate position; all 1s values are entirely 0xFF.

B.5.1. Minimum UUID Long (160 bits) All 0s

Encoding	Variant	Value
Base16	THIS DRAFT	00000004-00000000-0000-0000-f000-000000000000-00000000
Base16	[RFC4648], Section 8	0000000400000000000000f000000000000000000000000
Base32	[RFC4648], Section 6	AAAAAAEAAAAAAAAAAAB4AAAAAAAAAAAAAAAAA
Base32	[RFC4648], Section 7	0000004000000000001S000000000000000000
Base32	[Base32human]	0000004000000000001W000000000000000000
Base36	---	00000040000000000778rxuo7mqegxp0fcao
Base52	---	AAAAAEAAAAAAAAAAAZzXatxaqveqUnXJFs
Base58	[Base58btc]	111115111111111115XgaZChk8x2RpiFTD
Base62	[Base62ieee]	00000400000000001YbB1W92hKBNBJ9iy
Base62	[Base62sort]	00000400000000001YbB1W92hKBNBJ9iy
Base64	[RFC4648], Section 4	AAAAAEAAAAAAAAAAA8AAAAAAAAAAAAAA

Base64	[RFC4648], Section 5	AAAAAEAAAAAAAAAAA8AAAAAAAAAAAAAA
Base64	[Base64sort]	-----3----------w---------------
Base85	[Z85]	000040000000000&ntjHu1d=1an-h*

Table 11: 160-bit All 0s Boundary Encoding

B.5.2. Minimum UUID Long (160 bits) All 1s

Encoding	Variant	Value
Base16	THIS DRAFT	00000004-ffffffff-ffff-ffff-ffff-fffffffffff-ffffffff
Base16	[RFC4648], Section 8	00000004ffffffffffffffffffffffffffffffffffff
Base32	[RFC4648], Section 6	AAAAAAE7777777777777777777777777777
Base32	[RFC4648], Section 7	0000004VVVVVVVVVVVVVVVVVVVVVVVVVV
Base32	[Base32human]	0000004ZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
Base36	---	0000004twj4yidkw7a8pn4g709kzmfoaol3x8f
Base52	---	AAAAAEBQBgmlPXkFMhxLjXnmwANGzikNDAP
Base58	[Base58btc]	1111154ZrjxJnU1LA5xSyrWMNuXTvSYKwt
Base62	[Base62ieee]	000004aWgEPTl1tmebfsQzFP4bxwgy80V
Base62	[Base62sort]	000004aWgEPTl1tmebfsQzFP4bxwgy80V
Base64	[RFC4648], Section 4	AAAAAEP//////////////////////////
Base64	[RFC4648], Section 5	AAAAAEP__*__**__**__**__*
Base64	[Base64sort]	-----3Ezzzzzzzzzzzzzzzzzzzzzzzz
Base85	[Z85]	00004&j{+1Vmjq.eU!hqMq17MmuaY0

+--------+-------------+---+

Table 12: 160-bit All 1s Boundary Encoding

Author’s Address

Kyzer R. Davis
Cisco Systems
Email: kydavis@cisco.com, kyzer.davis@outlook.com