Workgroup: TODO Working Group
Internet-Draft: draft-ssmith-said-03
Published: 27 July 2023
Intended Status: Informational
Expires: 28 January 2024
Authors: S. Smith
ProSapien LLC

Self-Addressing IDentifier (SAID)

Abstract

A SAID (Self-Addressing IDentifier) is a special type of content-addressable identifier based on encoded cryptographic digest that is self-referential. The SAID derivation protocol defined herein enables verification that a given SAID is uniquely cryptographically bound to a serialization that includes the SAID as a field in that serialization. Embedding a SAID as a field in the associated serialization indicates a preferred content-addressable identifier for that serialization that facilitates greater interoperability, reduced ambiguity, and enhanced security when reasoning about the serialization. Moreover, given sufficient cryptographic strength, a cryptographic commitment such as a signature, digest, or another SAID, to a given SAID is essentially equivalent to a commitment to its associated serialization. Any change to the serialization invalidates its SAID thereby ensuring secure immutability evident reasoning with SAIDs about serializations or equivalently their SAIDs. Thus SAIDs better facilitate immutably referenced data serializations for applications such as Verifiable Credentials or Ricardian Contracts.

SAIDs are encoded with CESR (Composable Event Streaming Representation) [CESR] which includes a pre-pended derivation code that encodes the cryptographic suite or algorithm used to generate the digest. A CESR primitive's primary expression (alone or in combination) is textual using Base64 URL-safe characters. CESR primitives may be round-tripped (alone or in combination) to a compact binary representation without loss. The CESR derivation code enables cryptographic digest algorithm agility in systems that use SAIDs as content addresses. Each serialization may use a different cryptographic digest algorithm as indicated by its derivation code. This provides interoperable future proofing. CESR was developed for the [KERI] protocol.

Discussion Venues

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

Source for this draft and an issue tracker can be found at https://github.com/trustoverip/tswg-said-specification.

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 28 January 2024.

Copyright Notice

Table of Contents

1. Introduction

The primary advantage of a content-addressable identifier is that it is cryptographically bound to the content (expressed as a serialization), thus providing a secure root-of-trust for reasoning about that content. Any sufficiently strong cryptographic commitment to a content-addressable identifier is functionally equivalent to a cryptographic commitment to the content itself.

A self-addressing identifier (SAID) is a special class of content-addressable identifier that is also self-referential. This requires a special derivation protocol that generates the SAID and embeds it in the serialized content. The reason for a special derivation protocol is that a naive cryptographic content-addressable identifier must not be self-referential, i.e. the identifier must not appear within the content that it is identifying. This is because the naive cryptographic derivation process of a content-addressable identifier is a cryptographic digest of the serialized content. Changing one bit of the serialization content will result in a different digest. Therefore, self-referential content-addressable identifiers require a special derivation protocol.

To elaborate, this approach of deriving self-referential identifiers from the contents they identify, is called self-addressing. It allows any validator to verify or re-derive the self-referential, self-addressing identifier given the contents it identifies. To clarify, a SAID is different from a standard content address or content-addressable identifier in that a standard content-addressable identifier may not be included inside the contents it addresses. Moreover, a standard content-addressable identifier is computed on the finished immutable contents, and therefore is not self-referential. In addition, a self-addressing identifier (SAID) includes a pre-pended derivation code that specifies the cryptographic algorithm used to generate the digest.

An authenticatable data serialization is defined to be a serialization that is digitally signed with a non-repudiable asymmetric key-pair based signing scheme. A verifier, given the public key, may verify the signature on the serialization and thereby securely attribute the serialization to the signer. Many use cases of authenticatable data serializations or statements include a self-referential identifier embedded in the authenticatable serialization. These serializations may also embed references to

other self-referential identifiers to other serializations. The purpose of a self-referential identifier is to enable reasoning in software or otherwise about that serialization. Typically, these self-referential identifiers are not cryptographically bound to their encompassing serializations such as would be the case for a content-addressable identifier of that serialization. This poses a security problem because there now may be more than one identifier for the same content. The first is self-referential, included in the serialization, but not cryptographically bound to its encompassing serialization and the second is cryptographically bound but not self-referential, not included in the serialization.

When reasoning about a given content serialization, the existence of a non-cryptographically bound but self-referential identifier is a security vulnerability. Certainly, this identifier cannot be used by itself to securely reason about the content because it is not bound to the content. Anyone can place such an identifier inside some other serialization and claim that the other serialization is the correct serialization for that self-referential identifier. Unfortunately, a standard content-addressable identifier for a serialization which is bound to the serialization can not be included in the serialization itself, i.e. can be neither self-referential nor self-contained; it must be tracked independently. In contrast, a self-addressing identifier is included in the serialization to which it is cryptographically bound making it self-referential and self-contained. Reasoning about self-addressing identifiers (SAIDs) is secure because a SAID will verify if and only if its encompassing serialization has not been mutated, which makes the content immutable. SAIDs used as references to serializations in other serializations enable tamper-evident reasoning about the referenced serializations. This enables a more compact representation of an authenticatable data serialization that includes other serializations by reference to their SAIDs instead of by embedded containment.

2. Generation and Verification Protocols

The *self-addressing identifier* (SAID) verification protocol is as follows:

* Make a copy of the embedded CESR [CESR] encoded SAID string included in the serialization.

* replace the SAID field value in the serialization with a dummy string of the same length. The dummy character is #, that is, ASCII 35 decimal (23 hex).

*Compute the digest of the serialization that includes the dummy value for the SAID field. Use the digest algorithm specified by the CESR [CESR] derivation code of the copied SAID.

*Encode the computed digest with CESR [CESR] to create the final derived and encoded SAID of the same total length as the dummy string and the copied embedded SAID.

*Compare the copied SAID with the recomputed SAID. If they are identical then the verification is successful; otherwise unsuccessful.

2.1. Example Computation

The CESR [CESR] encoding of a Blake3-256 (32 byte) binary digest has 44 Base-64 URL-safe characters. The first character is E which represents Blake3-256. Therefore, a serialization of a fixed field data structure with a SAID generated by a Blake3-256 digest must reserve a field of length 44 characters. Suppose the initial value of the fixed field serialization is the following string:

```
field0______field1_____________________________________field2______
```

In the string, field1 is of length 44 characters. The first step to generating the SAID for this serialization is to replace the contents of field1 with a dummy string of # characters of length 44 as follows:

```
field0______############################################field2______
```

The Blake3-256 digest is then computed on the above string and encoded in CESR format. This gives the following SAID:

```
E8wYuBjhslETYaLZcxMkWrhVbMcA8RS1pKYl7nJ77ntA
```

The dummy string is then replaced with the SAID above to produce the final serialization with embedded SAID as follows:

```
field0______E8wYuBjhslETYaLZcxMkWrhVbMcA8RS1pKYl7nJ77ntA______
```

To verify the embedded SAID with respect to its encompassing serialization above, just reverse the generation steps.

2.2. Serialization Generation

2.2.1. Order-Preserving Data Structures

The crucial consideration in SAID generation is reproducibility. This requires the ordering and sizing of fields in the serialization

to be fixed. Data structures in most computer languages have fixed fields. The example above is such an example.

A very useful type of serialization especially in some languages like Python or JavaScript is of self-describing data structures that are mappings of (key, value) or (label, value) pairs. These are often also called dictionaries or hash tables. The essential feature needed for reproducible serialization of such mappings is that mapping preserve the ordering of its fields on any round trip to/ from a serialization. In other words the mapping is ordered with respect to serialization. Another way to describe a predefined order preserving serialization is canonicalization or canonical ordering. This is often referred to as the mapping canonicalization problem.

The *natural* canonical ordering for such mappings is *insertion order* or sometimes called *field creation order*. Natural order allows the fields to appear in a preset order independent of the lexicographic ordering of the labels. This enables functional or logical ordering of the fields. Logical ordering also allows the absence or presence of a field to have meaning. Fields may have a priority given by their relative order of appearance. Fields may be grouped in logical sequence for better usability and labels may use words that best reflect their function independent of their relative lexicographic ordering. The most popular serialization format for mappings is *JSON*. Other popular serializations for mappings are CBOR and MsgPack.

In contrast, from a functional perspective, lexicographic ordering appears un-natural. In lexicographic ordering the fields are sorted by label prior to serialization. The problem with lexicographic ordering is that the relative order of appearance of the fields is determined by the labels themselves not some logical or functional purpose of the fields themselves. This often results in oddly-labeled fields that are so named merely to ensure that the lexicographic ordering matches a given logical ordering.

Originally mappings in most if not all computer languages were not insertion order preserving. The reason is that most mappings used hash tables under the hood. Early hash tables were highly efficient but by nature did not include any mechanism for preserving field creation or field insertion order for serialization. Fortunately, this is no longer true in general. Most if not all computer languages that support dictionaries or mappings as first-class data structures now support variations that are insertion order preserving.

For example, since version 3.6 the default dict object in Python is insertion order preserving. Before that, Python 3.1 introduced the OrderedDict class which is insertion order preserving, and before

that, custom classes existed in the wild for order preserving variants of a Python dict. Since version 1.9 the Ruby version of a dict, the Hash class, is insertion order preserving. Javascript is a relative latecomer but since ECMAScript ES6 the insertion ordering of JavaScript objects was preserved in Reflect.ownPropertyKeys(). Using custom replacer and reviver functions in .stringify and .parse allows one to serialize and de-serialize JavaScript objects in insertion order. Moreover, since ES11 the native .stringify uses insertion order all text string labeled fields in Javascript objects. It is an uncommon use case to have non-text string labels in a mapping serialization. A list is usually a better structure in those cases. Nonetheless, since ES6 the new Javascript Map object preserves insertion order for all fields for all label types. Custom replacer and reviver functions for .stringify and .parse allows one to serialize and de-serialize Map objects to/from JSON in natural order preserving fashion. Consequently, there is no need for any canonical serialization but natural insertion order preserving because one can always use lexicographic ordering to create the insertion order.

2.3. Example Python dict to JSON Serialization with SAID

Suppose the initial value of a Python dict is as follows:

```
{
    "said": "",
    "first": "Sue",
    "last": "Smith",
    "role": "Founder"
}
```

As before the SAID will be a 44 character CESR encoded Blake3-256 digest. The serialization will be *JSON*. The said field value in the dict is to be populated with the resulting SAID. First the value of the said field is replaced with a 44 character dummy string as follows:

```
{
    "said": "############################################",
    "first": "Sue",
    "last": "Smith",
    "role": "Founder"
}
```

The dict is then serialized into JSON with no extra whitespace. The serialization is the following string:

```
{"said":"############################################","first":"Sue","la
```

The Blake3-256 digest is then computed on that serialization above and encoded in CESR to provide the SAID as follows:

```
EnKa0ALimLL8eQdZGzglJG_SxvncxkmvwFDhIyLFchUk
```

The value of the said field is now replaced with the computed and encoded SAID to produce the final serialization with embedded SAID as follows:

```
{"said":"EnKa0ALimLL8eQdZGzglJG_SxvncxkmvwFDhIyLFchUk","first":"Sue","la
```

The final serialization may be converted to a python dict by deserializing the JSON to produce:

```
{
    "said": "EnKa0ALimLL8eQdZGzglJG_SxvncxkmvwFDhIyLFchUk",
    "first": "Sue",
    "last": "Smith",
    "role": "Founder"
}
```

The generation steps may be reversed to verify the embedded SAID. The SAID generation and verification protocol for mappings assumes that the fields in a mapping serialization such as JSON are ordered in stable, round-trippable, reproducible order, i.e., canonical. The natural canonical ordering is called field insertion order.

2.4. Example Schema Immutability using JSON Schema with SAIDs

SAIDs make JSON Schema fully self-contained with self-referential, unambiguously cryptographically bound, and verifiable content-addressable identifiers. We apply the SAID derivation protocol defined above to generate the $id field.

First, replace the value of the $id field with a string filled with dummy characters of the same length as the eventual derived value for $id.

```
 {
     "$id": "############################################",
     "$schema": "http://json-schema.org/draft-07/schema#",
     "type": "object",
     "properties": {
         "full_name": {
         "type": "string"
     }
 }
```

Second, make a digest of the serialized schema contents that include the dummy value for the $id.

EZT9Idj7zLA0Ek6o8oevixdX20607CljNg4zrf_NQINY

Third, replace the dummy identifier value with the derived identifier value in the schema contents.

```
{
    "$id": "EZT9Idj7zLA0Ek6o8oevixdX20607CljNg4zrf_NQINY",
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
        "full_name": {
        "type": "string"
    }
}
```

Usages of SAIDs within authentic data containers as demonstrated here are referred to as self-addressing data (SAD).

2.5. Discussion

As long as any verifier recognizes the derivation code of a SAID, the SAID is a cryptographically secure commitment to the contents in which it is embedded; it is a cryptographically verifiable, self-referential, content-addressable identifier. Because a SAID is both self-referential and cryptographically bound to the contents it identifies, anyone can validate this binding if they follow the *derivation protocol* outlined above.

To elaborate, this approach of deriving self-referential identifiers from the contents they identify, is called self-addressing. It allows any validator to verify or re-derive the self-referential, self-addressing identifier given the contents it identifies. To clarify, a SAID is different from a standard content address or content-addressable identifier in that a standard content-addressable identifier may not be included inside the contents it addresses. Moreover, a standard content-addressable identifier is computed on the finished immutable contents, and therefore is not self-referential.

3. Appendix: Embedding SAIDs in URLs

ToDo. Provide normative protocol for embedding a SAID in a URL to replace a bare SAID in a data structure (field map). The purpose is to ease the transition from web 2.0 URL centric infrastructure to zero-trust infrastructure. This is a caveated adoption vector because it mixes discovery (URL) with integrity (SAID) layers. The OOBI protocol is an example of using embedded SAIDs inside URLs merely for verifiable discovery while using the bare SAID in the discovered data item.

4. Appendix: JSON Schema with SAIDs

ToDo. Provide normative rules for using SAIDs to lock-down JSON Schema (immutable) to prevent schema malleability attacks.

5. 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.

6. Security Considerations

TODO Security

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[CESR] Smith, S., "Composable Event Streaming Representation (CESR)", 2021, <https://datatracker.ietf.org/doc/draft-ssmith-cesr/>.

[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>.

8.2. Informative References

[KERI] Smith, S., "Key Event Receipt Infrastructure (KERI)", 2021, <https://arxiv.org/abs/1907.02143>.

Acknowledgments

Members of the keripy development team and the ToIP ACDC WG.

Author's Address

S. Smith

ProSapien LLC

Email: sam@prosapien.com