OPEN NETWORKING
FOUNDATION

Negotiable Datapath Model and Table Type Pattern Signing

Version 1.0

ONF TR-537

2016-09-08

ONF Document Type: Technical Recommendation
ONF Document Name: Negotiable Datapath Model and Table Type Pattern Signing

Disclaimer

Table of Contents

1 Overview

1.1 Objectives

The NDM / TTP signing specification is intended to enable NDMs / TTPs to be signed by their authors, in order to enable consumers of the NDMs / TTPs to detect modifications (malicious or inadvertent).

TTPs can be but do not have to be represented in JSON. Non-TTP NDM formats have not been defined yet. As this specification permits any sequence of bytes (whether printable or not) to be signed, it should be able to accommodate any existing or future class of NDM.

The consumers could be individuals or automated systems (e.g. controllers and switches). The materials involved in this process, including NDMs / TTPs, signatures, and any other associated information (e.g. public keys and certificates), could be retrieved over a network or stored locally. The ability to access these materials over a network can therefore not be guaranteed to be present. This specification accordingly covers generation and validation of signatures with the materials being at rest. (Materials are described as being stored in files for convenience, but they may be stored in other containers).

The specification assumes that signatures will be detached from the materials being signed. Furthermore, the materials being signed are permitted to be the original unmodified (human readable) format, or Base64URL encoded files (to tolerate transmission over channels modifying whitespace and line endings). This should enable signatures to be usable without negating any currently supported usage scenarios or affecting the productivity or performance of the consumers.

1.2 Terminology

Term / Acronym	Definition
TTP	Table Type Pattern
NDM	Negotiable Datapath Model
JSON	JavaScript Object Notation
Base64URL encoding	URL safe Base64 encoding: a mechanism to encode an arbitrary string of 8-bit characters into a format which only uses the characters ‘0’-’9’, ‘A’-’Z’, ‘a’-’z’, ‘-’, and ‘_’, as defined in [4] section 5, with all whitespace, line endings, and ‘=’ characters removed.

Term / Acronym	Definition
File	A storage container able to store and retrieve a sequence of bytes. (This term is used for convenience in this document. The information may not actually be stored in a file system managed by the operating system. Any container with the ability to store and retrieve information may be used.)

2 Signature Generation

2.1 Preparation of Material to be Distributed and Signed

The input (material to be signed) is assumed to be in a file. Recall that the materials being signed and distributed are permitted to be the original unmodified (human readable) format (e.g. files containing JSON, in the case of TTPs), or Base64URL encoded files (to tolerate transmission over channels modifying whitespace and line endings). The following procedures can optionally be used to perform the manipulations to achieve this, resulting in an updated file, or modified information in a temporary file. Either the original file or the modified / temporary file are then signed.

2.1.1 Optional: Pretty Printing of JSON Input

TTPs can be represented in JSON. Other NDMs may also be represented or representable in JSON.

When the input is supplied in JSON format, it is recommended (but not required) to pretty print the input, using the following command, or an equivalent mechanism:

```
python -mjson.tool <infile >outfile
```

where “infile” and “outfile” are the names of the input file and the output file respectively. This requires Python 2.6 or later.

2.1.2 Optional: Base64URL Encoding of Input

Optionally the input can be Base64URL encoded. This will ensure that the material to be signed can be transported over a channel which modifies whitespace (e.g. line endings). The following command can be used to achieve this:

```
python -c "import base64, sys; print base64.urlsafe_b64encode(sys.stdin.read()).replace('=','')" <infile >outfile
```

where “infile” and “outfile” are the names of the input file and the output file respectively. This requires Python 2.4 or later.

If this is not done, signature verification will fail if any part of the material to be signed (including whitespace and line endings) is changed, irrespective of whether or not such a change modifies how the material is interpreted (e.g. how a parser will interpret the input).

Due to the ease of use benefits associated with retaining directly human readable format in signed items, and because whitespace / line ending modifications can be prevented in many usage environments, Base64URL encoding is not mandated by this specification.

2.2 Generating the Signature

2.2.1 Prerequisites

The signature generation process requires access to the material for which the signature needs to be generated and the private key of the signing entity. The material will be the original file containing the NDM / TTP, or the file containing the result of the modifications applied by the previous sections.

2.2.2 Base64URL Encoding

Where the material to be signed is not already in Base64URL format, it is encoded using the Base64URL scheme using the following command:

```
python -c "import base64, sys; print base64.urlsafe_b64encode(sys.stdin.read()).replace('=','')" <infile >outfile
```

where “infile” and “outfile” are the names of the input file and the output file respectively. This requires Python 2.4 or later. An equivalent mechanism may be substituted.

2.2.3 Signing Details

The signature is generated according to the specification in Section 5.1 and Appendix F of [4] with the payload being the content of the file being signed. The file here is the result of the previous step, or the original input file. Use the algorithms and parameters documented in Section 4 below.

As the result of the procedure in the previous section is already Base64URL encoded, the result is used as is as the payload for [4], i.e. it is not Base64URL encoded again. Line ending characters, other whitespace characters, and trailing ‘=’ characters are removed, as mandated by [4].

The result is stored as a detached signature, i.e. a signature in a separate file.

3 Signature Verification

3.1 Preparation of Input

The material for which the signature is to be verified and the detached signature are assumed to be in individual files. No steps to prepare input are currently defined.

3.2 Verifying the Signature

3.2.1 Prerequisites

The signature verification process requires access to the material for which the signature needs to be verified, the signature, and the public key of the signing entity. Distribution of the public key as well as verification of the public key (e.g. using a Certification Authority) is not covered by this specification.

3.2.2 Base64URL Encoding

Where the material to be signed is not already Base64URL encoded, the content of the file containing the material to be signed is encoded according to the Base64URL specification.

This can be achieved using the following command (Python 2.4 or later is required):

```
python -c "import base64, sys; print base64.urlsafe_b64encode(sys.stdin.read()).replace('=','')" <infile >outfile
```

where “infile” and “outfile” are the names of the input file and the output file respectively.

3.2.3 Verification Details

The signature is verified by following the specification given in Section 5.2 and Appendix F of [4]. Use the algorithms and mechanisms documented in Section 4 below.

As the result of the procedure in the previous section is already Base64URL encoded, the result is used as is as the payload for [4], i.e. it is not Base64URL encoded again. Line ending characters, other whitespace characters, and trailing ‘=’ characters are removed, as mandated by [4].

4 JSON Web Signature Usage Details

Specification compliant implementations must use the following algorithms and mechanisms to generate and verify signatures.

4.1 Algorithm

The signing algorithm for which support is mandated by this specification is RSASSA-PKCS-v1_5 using SHA-512 (encoded as “RS512” in the “alg” field). Other algorithms listed in [4] can optionally be supported.

4.2 Public Key Distribution

Any of the mechanisms specified in Section 4 of [4] may be used to supply the public key or a reference to it and verify the public key (e.g. with reference to a Certification Authority). When generating signatures, a mechanism appropriate for the usage environment needs to be selected. When verifying signatures, all the mechanisms compatible with the usage environment should be supported. This specification cannot mandate any specific mechanism, as it is not known what the usage environment will be (e.g. whether the party verifying signatures will be able to obtain public keys by retrieving them from a URL, or whether the public key and associated certificates must be distributed with the signature itself).

5 References

1. JSON pretty printing library: https://docs.python.org/2/library/json.html
2. Base64URL encoding library: https://docs.python.org/2/library/base64.html
3. Base-N encoding specification: http://tools.ietf.org/html/rfc4648
4. JSON WS specification - IETF RFC7515: https://tools.ietf.org/html/rfc7515
5. TTP specification 1.0: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlow Table Type Patterns v1.0.pdf

6 Acknowledgements

The following individuals contributed to this document: Ben Mack-Crane, Curt Beckmann, Dacheng Zhang, Joe Tardo, Johann Tönsing.

7 Revision History

Date	Revision	Description	Editor
2016-09-08	1.0	Initial version for publication	J H Tönsing