Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 17, 2012

R. Barnes
BBN Technologies

June 15, 2012

JavaScript Message Security Format
draft-barnes-jose-jsms-00.txt

Abstract

Many applications require the ability to send cryptographically secured messages. While the IETF has defined a number of formats for such messages (e.g. CMS) those formats use encodings which are not easy to use in modern applications. This document describes the JavaScript Message Security format (JSMS), a new cryptographic message format which is based on JavaScript Object Notation (JSON) and thus is easy for many applications to generate and parse.

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 http://datatracker.ietf.org/drafts/current/.

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

This Internet-Draft will expire on December 17, 2012.

Table of Contents

1. Introduction 3
2. Conventions Used In This Document 3
3. Overview 3
3.1. Operational Modes 4
3.2. Design Principles 4
3.3. Certificate Processing 5
3.4. Certificate Discovery 5
4. Message Format 5
4.1. Data types 6
4.2. Basic Types 6
4.3. SignedData 7
4.3.1. Signature 7
4.3.2. Generating a SignedData Object 8
4.3.3. Verifying a SignedData Object 8
4.4. AuthenticatedData 9
4.4.1. Generating an AuthenticatedData Object 9
4.4.2. Verifying an AuthenticatedData Object 9
4.5. EncryptedData 10
4.5.1. Generating an EncryptedData Object 10
4.5.2. Decrypting a EncryptedData Object 11
4.6. Useful Objects 11
4.6.1. AlgorithmIdentifier 11
4.6.2. PublicKey 14
4.6.3. WrappedKey 16
5. Compact Format 17
6. Examples 18
6.1. Parameters 18
6.2. SignedData 18
6.3. AuthenticatedData 19
6.4. EncryptedData 20
7. Mapping to CMS 20
8. Comparison to JWS/JWE/JWK 21
9. IANA Considerations 22
10. Security Considerations 22
11. Acknowledgements 23
12. References 23
12.1. Normative References 23
12.2. Informative References 24
Appendix A. Acknowledgments 25
Author's Address 25

1. Introduction

Many applications require the ability to send cryptographically secured (encrypted, digitally signed, etc.) messages. While the IETF has defined a number of formats for such messages, those formats are widely viewed as being excessively complicated for the demands of Web applications, which typically only need the ability to secure simple messages. In addition, existing formats use encoding mechanisms (e.g., ASN.1 DER) which are not congenial for many classes of applications (e.g., Web applications). This presents an obstacle to the deployment of strong security by such applications.

This document describes a new cryptographic message format, JavaScript Message Security (JSMS). This format is intended to meet the need of modern applications, including JavaScript-based Web applications. While JSMS is modeled on existing formats -- principally CMS [RFC5652] -- it uses JavaScript Object Notation (JSON) rather than ASN.1, making it far easier for applications to handle. In the interest of simplicity, JSMS also omits many of less commonly used CMS modes (such as password-based encryption).

2. Conventions Used In This Document

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 [RFC2119].

In order to enable JSON to carry binary data, JSMS makes extensive use of Base64 encoding [RFC4648]. Whenever this document refers to Base64 encoding, we mean the URL-safe variant "base64url" encoding. As stated in section 3.1 of [RFC4648], Base64 does not allow linefeeds. Since linefeeds are not valid characters in a JSON string, whenever a field is specified to be Base64-encoded in this document, it MUST NOT include any line breaks. Base64-encoded fields also MUST NOT include JSON-encoded linefeeds such as "\n". Any linebreaks in the middle of Base64-encoded sections of the examples in this document have been inserted in order to make the examples fit on the page. Any trailing "=" characters SHOULD be removed. They are not needed, because JSON strings have defined lengths (namely the number of characters between unescaped '"' characters).

3. Overview

The JSMS message format is simply a JSON [RFC4627] object with an appropriate collection of fields. Each operating mode will have a separate set of fields, with a common field to distinguish between

the modes.

3.1. Operational Modes

JSMS supports three operational modes:

Signed Data
: A block of data signed by a single signer using his asymmetric key and optionally carrying his certificate.

Authenticated Data
: A block of data with authentication and integrity protection provided using a symmetric-key Message Authentication Code (MAC). The MAC key may be provided in encrypted form (as with Encryped Data) or identified by name.

Encrypted Data
: A block of data encrypted under a random message encryption key (MEK). The MEK is then separately encrypted for each recipient, either via symmetric or asymmetric encryption. The data is always integrity protected, through the use of an Authenticated Encryption with Associated Data (AEAD) algorithm such as AES-GCM or AES-CCM.

Any other desired security functions are provided by composition of these modes. For instance, a signed and encrypted message is produced by first creating a Signed message and then encrypting that data.

3.2. Design Principles

In general, JSMS follows the following design principles.

Minimize implementation complexity
: Wherever possible, protocol choices have been made such that the time and effort required to implement the protocol in many different programming languages will be minimized. This means that optimizations for bandwidth, CPU, and memory utilization have been explicitly avoided.

Base64 as the only encoding
: Any data that does not have a straightforward string representation (binary values, large integers, etc.) is base64-encoded (see: [RFC4648]). In some cases, hexadecimal encodings might be more convenient, but consistency is even more important to reduce implementation complexity.

No canonicalization
: In many cryptographic message formats, canonical encodings are used to allow the same value to be computed at both sender and recipient (e.g., for digital signatures). This is inconvenient in JSON, which just views messages as a bundle of key/value pairs. Instead, whenever canonicalization would be required, the relevant

data is serialized and base64-encoded for transport, allowing both sides to run computations over the same original set of octets.

In-memory processing

We assume that the entire message can fit in main memory and make no effort to design a wire representation which can be handled in small chunks in a single pass. This means, for instance, that there is no need to have a message digest indicator at the beginning of the message and then the signature at the end, as is done in CMS. Fields are simply serialized in whatever order is most convenient for the JSON implementation. The examples in this document are generally shown in whatever order seems most readable and are not normative.

Consistency with CMS

To simplify the adaptation of existing cryptographic modules and the validation of JSMS implementations, changes from the CMS cryptographic operations are minimized. JSMS is semantically equivalent to a profile of CMS, as described in Section 7.

3.3. Certificate Processing

Experience has shown that certificate handling (path construction) is one of the trickier parts of building a cryptographic system. While JSMS supports PKIX certificates, its certificate processing is far simpler than that of CMS. (It also supports the use of bare public keys in order to avoid the use of X.509 altogether.) When a JSMS agent provides its certificate, it must provide an ordered chain (as in TLS [RFC5246]) terminating in its own certificate, thus removing the need to construct certificate paths. The certificates MUST be ordered with the end-entity certificate first and each certificate that follows signing the certificate immediately preceding it.

3.4. Certificate Discovery

JSMS will often be used in an online messaging environments with users that have an address of the form user@domain, such as email, XMPP, or SIP. As such, protocols such as WebFinger [I-D.hammer-webfinger] or an end-to-end protocol can be used to retrieve appropriate certificates. Downstream uses of JSMS SHOULD define a discovery mechanism suitable for the intended use.

4. Message Format

A JSMS object is a JSON object that encodes cryptographic informaton related to a content byte string. This document specifies the set of keys that must be present in a JSMS object, what the associated values are, and how these values are generated and processed in order to realize security features. In processing JSMS objects, unknown

keys MUST be ignored.

JSMS defines three top-level types of secure object, each of which provides a specific cryptographic protection to a byte string.

SignedData: Signature using a public-key digital signature algorithm
AuthenticatedData: Authentication using a Message Authentication Code (MAC)
EncryptedData: Encryption and authentication using an Authenticated Encryption with Associated Data (AEAD) algorithm

4.1. Data types

For each field in a JSON object, we define the type of information that must be included in that field. At base are the object, array, string, number types defined by JSON. We also use two special sub-classes of strings: Fields with type "Token" contain a string drawn from a defined list of strings (e.g., an IANA registry for algorithm names). Fields with type "ByteString" contain a Base64-encoded byte string (note the considerations related to Base64 encoding in Section 2 above).

In addition to the primitive data types, Section 4.6 defines a collection of useful object types that are used by the top-level JSMS objects. These are simply referred to by name when they appear as a field value in another object.

4.2. Basic Types

The following elements are common to all JSMS messages:

"version": REQUIRED Number. The version of JSMS used by this objec.t. This field MUST be set to 1.
"type": REQUIRED Token. The type of this JSMS object. This field MUST be set to one of the following values
  "signed": SignedData object
  "authenticated": AuthenticatedData object
  "encrypted": EncryptedData object
"content": OPTIONAL ByteString. The content byte string, Base64-encoded.

If the "content" key is not present in a given JSMS object, then the JSMS object is "detached". In this case, the content must be associated with the JSMS object through some out-of-band mechanism before the JSMS object can be processed. Note that there is a risk that detached JSMS object might become invalid if the content is transformed, even if this transformation preserves the semantics of the content. For example, if the content is a JSON object, and the

object passes through an intermediate process that adds whitespace or re-orders the fields in the object (neither of which changes the meaning of the object), then the recipient will not be able to verify the signature. For this reason, detached JSMS objects SHOULD NOT be used unless there is a canonical form for the content being processed.

4.3. SignedData

A SignedData object MUST have a "type" field set to "signed". In addition, a SignedData object contains the following keys:

"digestAlgorithm": REQUIRED AlgorithmIdentifier. The digest algorithm used in signing the content.
"signatures": REQUIRED Array of Signature. One or more digital signatures over the content.
"certificates": OPTIONAL Array of String. A certificate chain associating the signer's public key with an identifier. Each element in the array is a string containing the Base64-encoded representation of a DER-formatted certificate. The certificates MUST be ordered with the end-entity certificate first and each certificate that follows signing the certificate immediately preceding it.
"certificatesURI": OPTIONAL String. An HTTP or HTTPS URI referring to a certificate chain. The referenced resource MUST have type "application/json" and contain an array of certificates in the same format as the "certificates" element above, including the ordering constraint.

4.3.1. Signature

A Signature object represents the signature over the content in the SignedData object by a specific key pair. A Signature objec can contain the following keys:

"signatureAlgorithm": REQUIRED AlgorithmIdentifier. The signature algorithm used in signing the content.
"key" REQUIRED PublicKey. The public key identifier for the signer, represented as a PublicKey object (see Section 4.6.2)
"signature" REQUIRED ByteString. The Base64-encoded signature value

If the "key" value represents the public key as an identifier, then a certificate for the signer MUST be provided by setting either the "certificates" or "certificatesURL" fields. The subject key in the end-entity certificate MUST match the identifier in the "key" value; the certificate SHOULD contain the subjectKeyIdentifier field, with a value matching the "key" value. (Note that this implies that when there are multiple signers, only one key can be represented by ID.)

4.3.2. Generating a SignedData Object

The inputs to the process of generating a SignedData object are:

- The content, as a byte string
- A digest algorithm
- One or more signature algorithms and asymmetric key pairs

To generate the signature for SignedData object, the originator takes the following steps:

1. Compute the message digest by applying the digest algorithm to the content.
2. For each signing key pair, compute the signature by using the signature algorithm to sign the message digest with the private key from the asymmetric key pair.

The originator then encodes the SignedData object by including the appropriate AlgorithmIdentifiers for the digest algorithms, a Signature object for each signature, and (optionally) the content.

4.3.3. Verifying a SignedData Object

To verify a SignedData object, the recipient takes the following steps:

1. Verify that the digest and signature algorithms are supported. Otherwise, report an error and fail.
2. Compute the content byte string by decoding the "content" value of the JSMS object. If the JSMS object does not contain a "content" field, retrieve the content by other means.
3. Compute the message digest by applying the digest algorithm to the content.
4. Compute the signature by decoding the "signature" value of the JSMS object.
5. Compute the public key:
   * If the key is represented directly, then decode it according to the rules specified by the algorithm name.
   * If the key is represented by an ID, then retrieve the corresponding subject public key from the end-entity certificate . If no "certificates" or "certificatesURI" value is present, then report an error and fail.
   * If the key is represented by a URI, retrieve the public key from the URI.
6. Verify the signature by using the signature algorithm to verify the message digest with the public key.

4.4. AuthenticatedData

An AuthenticatedData object MUST have a "type" field set to "authenticated". In addition, an AuthenticatedData object contains the following keys:

"algorithm": REQUIRED AlgorithmIdentifier. The MAC algorithm used to authenticate the content.
"mac": REQUIRED ByteString. The MAC value
"keys": OPTIONAL Array of WrappedKey. Wrapped versions of the symmetric key used for this MAC. Each element in the array MUST be a WrappedKey object (see below)
"keyId": OPTIONAL ByteString. An opaque identifier for a pre-shared MAC key

An AuthenticatedData object MUST contain either the "key" field or the "keyId" field, so that the recipient knows which key to use to verify the MAC.

4.4.1. Generating an AuthenticatedData Object

The inputs to the process of generating a AuthenticatedData object are:

- The content, as a byte string
- A MAC algorithm
- A MAC key and key identifier, or
- One or more recipient keys and key encipherment algorithms

If the recipient key is specified rather than the MAC key directly, then a random MAC key is generated and encoded in a WrappedKey objects for each recipient (see Section 4.6.3). Once the MAC key has been determined, the originator uses the MAC algorithm and MAC key to compute the MAC over the content byte string.

The originator then encodes the AuthenticatedData object by including the appropriate AlgorithmIdentifier for the MAC algorithm and the Base64 representations of the MAC value and (optionally) the content. If the MAC key was specified directly, then the Base64 representation of the key identifier is set as the "keyId" value; otherwise, the WrappedKey objects are collected in an array and set as the "keys" value.

4.4.2. Verifying an AuthenticatedData Object

To verify a AuthenticatedData object, the recipient takes the following steps:

1. Verify that the MAC algorithm is supported. If not, report an error and fail.
2. Compute the content byte string by decoding the "content" value of the JSMS object. If the JSMS object does not contain a "content" field, retrieve the content by other means.
3. Compute the MAC key:
   * If the "keyId" value is present and represents a known key, use the identified key.
   * If the "keys" value is present, check each WrappedKey object to determine if it matches a known key for this recipient. If any of the wrapped keys matches, unwrap the key from the first one and use it (see Section Section 4.6.3). Otherwise, report an error and fail.
4. Use the MAC algorithm and MAC key to compute the MAC over the content byte string
5. Decode the MAC value from the "mac" field.
6. Verify that the computed MAC matches the MAC from the object.

4.5. EncryptedData

An EncryptedData object MUST have a "type" field set to "encrypted". Note also that in an EncryptedData object, the "content" field contains the encrypted form of the content, not the content itself (as plaintext). An EncryptedData object contains the following keys in addition to any common fields:

"algorithm": REQUIRED AlgorithmIdentifier. The encryption algorithm used to encrypt the content
"keys": REQUIRED Array of WrappedKey. Wrapped versions of the symmetric key used to encrypt the content. Each element in the array MUST be a WrappedKey object (see Section 4.6.3).
"mac": OPTIONAL ByteString. The MAC value, if required by the algorithm

Note that although the "mac" field is optional, an EncryptedData object always has an integrity check. All of the encryption algorithms used in JSMS are "Authenticated Encryption with Associated Data" algorithms, which include an authentication / integrity fuction by definition. The MAC field is optional because some AEAD algorithms have a separate MAC value (e.g., GCM), while others incorporate the MAC value into the ciphertext (e.g., CCM).

4.5.1. Generating an EncryptedData Object

The inputs to the process of generating a SignedData object are:

- The content, as a byte string
- An encryption algorithm
- One or more recipient keys and key encipherment algorithms

The originator generates a random encryption key of a length suitable for the encryption algorithm, then encodes it in a WrappedKey object for each recipient (see Section 4.6.3). The content is then encrypted using the generated encryption key and the specified encryption algorithm.

The originator then encodes the EncryptedData object by including the appropriate AlgorithmIdentifier for the encryption algorithm, an array containing the WrappedKey objects, and (optionally) the Base64 representation of the content.

4.5.2. Decrypting a EncryptedData Object

To decrypt an EncryptedData object, the recipient takes the following steps:

1. Verify that the encryption algorithm is supported. If not, report an error and fail.
2. Compute the content byte string by decoding the "content" value of the JSMS object. If the JSMS object does not contain a "content" field, retrieve the content by other means.
3. Locate the encryption key: Check each WrappedKey object to determine if it matches a known key for this recipient. If any of the wrapped keys matches, unwrap the key from the first one and use it (see Section 4.6.3). Otherwise, report an error and fail.
4. Decrypt the content using the encryption key and the specified encryption algorithm.
5. Verify that the integrity check in the AEAD decryption was successful. If not, report an error and fail.
6. Return the decrypted content.

4.6. Useful Objects

In this section we define some common object types that are used across the top-level objects above.

4.6.1. AlgorithmIdentifier

An AlgorithmIdentifier object names a cryptographic algorithm and specifies any associated parameters such as nonces or initialization vectors (IVs). If the algorithm has no parameters, then the AlgorithmIdentifier object is simply a token representing the name of the algorithm, drawn from an IANA registry of algorithm names.

If the algorithm specifies parameters, the AlgorithmIdentifier object is a JSON object. There is only one required field, "name". Any other fields are specified in the algorithm definition.

"name": REQUIRED Token. The name of the algorithm, chosen from one of the IANA registries defined by this document.

The following table summarizes the algorithms to be used with JSMS.

[[More detail to be added later, in a separate document]]

Name	Parameters	Reference	Example
SIGNING			
rsa	no	[RFC3447]	"rsa"
dsa	yes (p,q,g)	[FIPS186]	{name:"dsa", p:1, q:2, g:3}
ecdsa	yes (curve)	[RFC6090]	{name:"ecdsa", curve:"P-256"}
DIGEST			
sha1	no	[FIPS180-1]	"sha1"
sha256	no	[FIPS180-3]	"sha256"
sha384	no	[FIPS180-3]	"sha384"
sha512	no	[FIPS180-3]	"sha512"
MAC			
hs1	no	[FIPS180-1]	"hs1"
hs256	no	[FIPS180-3]	"hs256"
hs384	no	[FIPS180-3]	"hs384"
hs512	no	[FIPS180-3]	"hs512"
ENCRYPTION			
aes128-ccm	yes (n,M)	[RFC3610]	{name:"aes128-ccm", n:"ZONce...lU-g", m:8}
aes128-gcm	yes (iv)	[McGrew & Viega]	{name:"aes128-gcm", iv:"ZONce...lU-g"}
KEY ENCIPHERMENT			
aes	no	[RFC3394]	"aes"
rsaes-oaep	no	[RFC3447]	"rsaes-oaep"
KEY AGREEMENT			
dh-es	yes (group)	[RFC2631]	{name:"dh-es", group: 14}
ecdh-es	yes (curve)	[RFC6090]	{name:"ecdh-es", curve:"P-256"}

Obviously, there will be more detail needed beyond the above, and some IANA considerations to create the necessary registries. For some algorithms, there will be specific notes about how they are to

be used with JSMS, for example:

- The signature value produced by DSA is comprised of two integers. The byte string to be filled in the "signature" field is the two-element JSON array containing two integers, "[r,s]"
- RSAES-OAEP is always used with SHA-256 and the default MGF1 masking generation function
- Elliptic curves may only be specified by name, not by directly specifying curve parameters. [[We may define our own registry, or re-use the ones from TLS/IKE.]]
- AEAD algorithms are only used for authenticated encryption; there is never associated data. Further AEAD algorithms may be defined using [draft-mcgrew-aead-aes-cbc-hmac-sha1]

4.6.2. PublicKey

A PublicKey object describes the public key used by a signer. The key may be specified as a JSON structure directly, as a URI, or as an identifier. A PublicKey object has the following fields:

"type" OPTIONAL Token. The name of the algorithm with which this key is to be used
"id" OPTIONAL ByteString. An identifier for the key
"uri" OPTIONAL String. A URI pointing to a direct form of the key

If the key is specified directly, then the "type" key MUST be present; the "id" and "uri" fields MAY be present. Subsequent entries in the array specify the elements of the key, in a manner determined by the algorithm. Formats for RSA and ECDH/ECDSA public keys are specified below.

If the key is provided as a URI, then the "uri" field MUST be present, containing a URI where the key can be retrieved, in the JSON format described above. The method that the recipient of a JSMS object uses to retrieve the key will depend on the URI scheme. For HTTP URIs, the relying party MUST issue an HTTP request with the GET method and an Accept header including the MIME type for JSMS PublicKey object, "[[MIMETYPE-TBD]]". For MAILTO, SIP, and XMPP URIs, the recipient MAY use the WebFinger protocol [I-D.hammer-webfinger] to retrieve a public key for the user.

If the key is referenced by an opaque identifier or "fingerprint", then the "id" field MUST be present, and contain the Base64-encoded SHA-1 hash of the public key, represented as a DER-encoded subjectPublicKeyInfo data structure. (This fingerprint value is the same as the one commonly included in the subjectKeyIdentifier field in an X.509 certificate.)

The recipient of a JSMS object can determine which of the above cases

a given key falls into by seeking the three fields in sequence. If a "type" field is present, then the key is represented directly. If a "uri" field is present, then the key is represented directly, but must be retreived from the URI. Finally, if the "id" field is the only one of the three present, then the key is represented by ID only, and must be retrieved from somewhere else (e.g., from a certificate in the JSMS object).

Example: {"id": "i1LbR8FCEw-aiFcAAfUvpp75wdY="}

Example: {"uri": "xmpp:juliet@example.com"}

4.6.2.1. RSA Public Key

An RSA public key comprises two additional parameters in addition to the algorithm identifier "rsa".

"n": REQUIRED ByteString. The modulus, represented as an integer in network byte order (big-endian)
"e": REQUIRED Integer. The public exponent, represented as an integer in network byte order (big-endian)

Example: {"type":"rsa", "n":98739...04251, "e": 3}

4.6.2.2. Elliptic-Curve Public Key

Public keys for several types of elliptic curve algorithms, including ECDSA and ECDH, have the same format, namely an point on a specified elliptic curve. In an elliptic curve PublicKey object, the curve parameters are specified in the algorithm identifier, and there are two additional fields that specify the point on the curve:

"x": REQUIRED ByteString. The x coordinate of the point
"y": REQUIRED Integer. The y coordinate of the point. MUST be equal to 0 or 1.

These coordinates correspond to the compressed form of an elliptic curve point, as specified in [[SEC01]]. In terms of the calculation specified in section 2.3.3 of [[SEC01]], the "x" coordinate is the byte string X and the "y" coordinate is the reduced y coordinate (or, equivalently, Y mod 2).

Example: {"type":"ecdh",
"x":"IIIs_x1m6Na6xKN37vOwvy7AvFeG9HhBN2EN3u5EZQ4", "y": 1}

4.6.3. WrappedKey

In JSMS objects that use symmetric keys (for MAC or encryption), it is necessary for the originator to convey the symmetric key used for in JSMS computations to the recipient. The WrappedKey object is a JSON object that allows these keys to be provided either using key transport or key agreement. The following fields may be present in a WrappedKey object:

"type": REQUIRED TOKEN The type of wrapping being done. This document defines the following values for this field:
- "encryption": Symmetric key transport. The "KEKIdentifier" field MUST be present. Any other non-required fields MUST be ignored.
- "transport": Asymmetric key transport. The "recipientKey" field MUST be present. Any other non-required fields MUST be ignored.
- "agreement": Key agreement. The "originatorKey" and "recipientKey" MUST be present, and the "userKeyMaterial" field MAY be present. Any other non-required fields MUST be ignored.

"algorithm": REQUIRED AlgorithmIdentifier The algorithm used to encrypt the symmetric key

"encryptedKey": REQUIRED BYTES The symmetric key, encrypted according to the algorithm indicated by the "algorithm" value

"KEKIdentifier": OPTIONAL BYTES An opaque identifier for the symmtric key encryption key

"originatorKey": OPTIONAL PublicKey The public key of the originator

"recipientKey": OPTIONAL PublicKey The public key of the recipient

"userKeyMaterial": OPTIONAL BYTES User key material

The techniques used for wrapping and unwrapping the encrypted key is determined by "type" and "algorithm" fields. In general, the options are the same as for CMS [RFC5280], without the option for password-based key wrapping.

"encryption": The key is encrypted under a pre-shared symmetric key encryption key identified by the "KEKIdentifier" field

"transport": The key is encrypted under the recipient's public key, identified in the "recipientKey" field.

"agreement": The key is encrypted under a shared secret derived using a key agreement algorithm combining the originator's private key and the recipient's public key, corresponding to the "originatorKey" and "recipientKey", respectively. The value provided in the "userKeyMaterial" field may be used to provide additional entropy.

[[More detail to be added.]]

5. Compact Format

The compact JSON format of a JSMS object is identical to the normal JSMS format, except that field names are replaced with shorter equivalent field names. Translations for the field names above are given in the table below. In a given JSMS object, field names MUST either all be in long form or all be in short form. An implementation MUST reject a JSMS object with mixed long and short names as improperly formatted.

Common	
version	v
type	t
signed	s
authenticated	au
encrypted	en
content	c

SignedData	
digestAlgorithm	da
signatures	ss
certificates	ce
certificatesURI	cu

AuthenticatedData	
algorithm	a
mac	mac
keys	ks
keyId	ki

EncryptedData	
algorithm	a
keys	ks
mac	mac

Signature	
signatureAlgorithm	sa
key	k
signature	sg

AlgorithmIdentifier	
name	nm

PublicKey	
type	t
id	i
uri	u

WrappedKey	
type	t
encryption	ec
transport	tr
agreement	ag
algorithm	a
encryptedKey	ek
KEKIdentifier	i
originatorKey	o
recipientKey	r
userKeyMaterial	uk

In applications where a JSMS object is required to be URL-safe, it is RECOMMENDED that it be rendered in the compact serialization, then Base64-encoded.

[[If there is a desire to avoid double-base64url-encoding things, then we could define a mechanism for moving some fields out of the object.]]

6. Examples

This section contains complete examples of all three JSMS types. All white space is for readability only, and must be removed before the examples can be considered valid JSMS objects.

6.1. Parameters

RSA key:

```
{
  "type": "rsa",
  "n": "AfWGinFrdktMCi4LkD_vcIsqc0m4JSS0rNDk_5Zdi8fwja_qH0M7d3
        U4tPUw7L0gP1iSMakdTKX0S7uTV_v9FeY8_WrxDgbphrH9Zaz0PvTL
        OuiKfRkMWK5A6nzl_PdP7_ujDWkvHKhWcJtM7irdn9K059X21EDtuq
        GJyq7_v_c_",
  "e": "AQAB",
  "d": "EMwfyOqzfJQgZyhl_W40k8SpNdfgDpmqjBiPYubhLqIk7LZns6XDO3
        7ZuLiZxT_WP04uMZ7UmV5URwUJVlxEpmfozhtLooCTP1oWtRQQjhTa
        Pz1f5nRKoHsO8e3PZY7O44ut2prRWNNxYxDk52rH9GTECqGAmDNb1f
        he6zX4KJk="
}
```

```
Key Tag:               HK1RA8AQwcI=
Symmetric key:         rQS8Dx6WQ_xDWTER8mAHnw==
```

```
Content:
"Attack at dawn!"
```

6.2. SignedData

In this object the content is signed under the specified RSA key pair, using SHA256 as the digest.

```
{
    "version": 1,
    "type": "signed",
    "digestAlgorithm": "sha256",
    "content": "QXR0YWNrIGF0IGRhd24h",
    "signatures": [{
        "signatureAlgorithm": "rsa",
        "key": {
            "type": "rsa",
            "n": "AfWGinFrdktMCi4LkD_vcIsqc0m4JSS0rNDk_5Zdi8
                  fwja_qH0M7d3U4tPUw7L0gP1iSMakdTKX0S7uTV_v9
                  FeY8_WrxDgbphrH9Zaz0PvTLOuiKfRkMWK5A6nzl_P
                  dP7_ujDWkvHKhWcJtM7irdn9K059X21EDtuqGJyq7_
                  v_c_",
            "e": "AQAB",
        },
        "signature": "AJll1tVYsRtGeHaJenAU-U3x4LxXklNoGrFwyu
                      xJWnYIeLZL16Ib7ZPvD79peMiSQAHAdLKcI8e-
                      CpU6HNQ-MxeE-tEXvaXOxuNZfVG9LBP9hq_ZwX
                      SguffHHzS9lLtVB0OzrXeszXtqD5igmeco1A0E
                      8eabzujA4bdN6Umyc7rA"
    }]
}
```

6.3. AuthenticatedData

In this object the content is authenticated with a MAC under a randomly-generated key (AuthenticatedData Key above), wrapped using the key encryption key above, identified by the above key tag.

```
{
    "version": 1,
    "type": "authenticated",
    "algorithm": "hs256",
    "content": "QXR0YWNrIGF0IGRhd24h",
    "mac": "990xwhrsX-COXUN0uF09HUHLU2CjdneeMqTtM4sGVDY=",
    "keys": [{
        "type": "encryption",
        "algorithm": "aes",
        "encryptedKey": "Dbf2O_ZIX0_Zfj-0aU6zQjn3xixj6vm7LVX
                         XFDdX4xqie5bZUS1nnstIPYOyzxNx9Udt-J
                         LZZh-zM8A_FbsZ8zAibdJ3EPyd",
        "KEKIdentifier": "HK1RA8AQwcI="
    }]
}
```

As another example, the following object is a detached MAC (over the same content string) in the compact encoding. Here we use the key as the MAC key directly (instead of as a key encryption key). The

object is shown both in raw JSON form and in the Base64 encoding.

```
{"v": 1,"t": "au","a": "hs256","ki": "HK1RA8AQwcI=",
  "mac": "PMVmhmrgbj-KNybfMqHu4ySJ0GnVrwe11MKpiuuGlIQ="}

eyJ2IjogMSwidCI6ICJhdSIsImEiOiAiaHMyNTYiLCJraSI6ICJISzFSQThBUXdj
ST0iLA0KICAibWFjIjogIlBNVm1obXJnYmotS055YmZNcUh1NHlTSjBHblZyd2Ux
MU1LcGl1dUdsSVE9In0=
```

6.4. EncryptedData

In this object, the content is encrypted under the general AEAD algorithm using AES-128-CBC for encryption and HMAC-SHA1 for authentication. The keys are described above as "EncryptedData Key (E)" and "EncryptedData Key (A)", respectively. The temporary keys are wrapped using the PKCS#1 wrapping, under the RSA key pair above.

```
{
    "version": 1,
    "type": "encrypted",
    "algorithm": {
        "name": "aes128-ccm",
        "n": "LTR8s7KKbd1QlQ==",
        "m": 8
    },
    "content": "0nkXCLOVxM2oNJOsDCwASLTODIMVZQE=",
    "keys": [{
        "type": "transport",
        "algorithm": "rsaes-oaep",
        "encryptedKey": "AbAxRnd_u7lICJlBskq3kgQVs54RLMgOjNmALXF
                         JjKqsQ4kLNL60VAoEswGOd2arGfcxoMCw9wMeSP
                         FOIvOXGvSt2wJXR_6kwzOJv_YyTC_eZUJHpcLNr
                         jKxB7Zf2_ap24W6JqcOYYVy2DhECcPgyvVRA_Ql
                         ZNHFYdqaImgOKJv-",
        "recipientKey": {
            "type": "rsa",
            "n": "AfWGinFrdktMCi4LkD_vcIsqc0m4JSS0rNDk_5Zdi8fwja
                  _qH0M7d3U4tPUw7L0gP1iSMakdTKX0S7uTV_v9FeY8_Wrx
                  DgbphrH9Zaz0PvTLOuiKfRkMWK5A6nzl_PdP7_ujDWkvHK
                  hWcJtM7irdn9K059X21EDtuqGJyq7_v_c_",
            "e": "AQAB",
        }
    }]
}
```

7. Mapping to CMS

The JSMS message format is semantically equivalent to a profile of

the Cryptographic Message Syntax (CMS), and mirrors a fair bit of its syntactical structure as well. The top-level message types each map to top-level CMS types: SignedData to SignedData, AuthenticatedData to AuthenticatedData, and EncryptedData to AuthEnvelopedData [RFC5083]. The main difference other than encoding is that many optional fields have been removed, for example the protected and unprotected attributes.

This similarity also applies to the secondary objects. Just as in CMS, AlgorithmIdentifier objects carry an identifier for the algorithm (here a name instead of an OID) and any related parameters. The PublicKey object format is an amalgam of the SubjectKeyIdentifier from CMS and the SubjectPublicKeyInfo from X.509. PublicKey objects can be mapped to CMS constructs by converting them to SubjectKeyIdentifier objects (using the appropriate hash) and including a certificate containing the public key. The WrappedKey object format maps directly to the CMS RecipientInfo structure, with the above considerations related to public keys, and without the option for password-based wrapping.

The major way in which JSMS diverges from CMS is that it allows the use of static MAC keys, referenced by an identifier. CMS requires the use of random MAC keys, encrypted in a RecipientInfo (i.e., a WrappedKey) for each recipient. JSMS allows the use of random keys, but also includes the "keyId" field to reference static MAC keys directly. The security implications of this change are discussed in Section 10.

In fact, it should be possible to translate JSMS objects back and forth to CMS without changing any values (simply reformatting), with only a couple of exception cases:

- JSMS objects that use static MAC keys cannot be translated to CMS because CMS does not allow this keying mechanism.
- JSMS objects using general AEAD algorithms (according to [[draft-mcgrew-aead-aes-cbc-hmac-sha1]]) because the required algorithm identifiers have not been defined for CMS.
- CMS objects using features that are not supported in JSMS (e.g., password-based key wrapping) cannot be translated to JSMS.

8. Comparison to JWS/JWE/JWK

The overall JSMS structure covers the integrity, authentication, and encryption use cases as the JSON Web Encryption (JWE) and JSON Web Signature (JWS) specifications. Most of the fields in JWS and JWE map conceptually to JSMS fields, with a couple of exceptions. The major differences are as follows:

- The signature and MAC functions of the JWS object are separated into SignedData and AuthenticatedData JSMS objects.
- JSMS is pure JSON, whereas in JWE and JWS only the header parameters are represented in JSON.
- JSMS parameters are not integrity-protected, as they are in JWE and JWS.
- JSMS allows for full algorithm agility in key agreement, while JWE only allows ECDH-ES.
- JSMS supports multiple recipients for EncryptedData and AuthenticatedData objects via the inclusion of multiple WrappedKey objects. Sending a JWE to multiple recipients requires re-encryption of the entire object for each recipient.
- The "typ" and "zip" parameters are not defined in JSMS, but could be added without significant change.
- JSMS requires that recipients MUST ignore unknown header parameters, in order to facilitate extensibility.

The PublicKey structure is analogous to the JSON Web Key (JWK) (with the public key parameters specified in the JSON Web Algorithms (JWA) document). The JWK "use" and "kid" parameters are not defined in JSMS, but could be added without significant change.

9. IANA Considerations

TODO:
- Register MIME types
- Registries for algorithms (signing, hash, MAC, encrypion, encipherment, agreement)

10. Security Considerations

Much more to follow here.

[[Given the CMS mapping above, import CMS security considerations.]]

[[Notes on identity for SignedData and AuthenticatedData: It is important to note that the above verification process only checks that the JSMS object was signed with a given public key. In order for this information to be useful to an applications, it is usually necessary to bind the public key to an application-layer identifier. If the "certificates" or "certificatesURI" value is present, then the recipient SHOULD verify that the chain is valid, and that the the end-entity certificate chains to a trust anchor. In this case, the recipient can consider the identity asserted in the end-entity certificate to be bound to the public key. Applications using this

specification without certificates will need to specify an alternative mechanism for binding public keys to identifiers.]]

[[Notes on the security of static-key MACs. Need to periodically refresh keys.]]

[[For multiple signatures, the considerations of RFC 4853.]]

11. Acknowledgements

The inspirataion and starting point for this document was draft-rescorla-jsms-00. Thanks to Eric Rescorla and Joe Hildebrand for allowing me to re-use a fair bit of their document, and for some helpful early reviews.

12. References

12.1. Normative References

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security

(TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5649] Housley, R. and M. Dworkin, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm", RFC 5649, September 2009.

[RFC5840] Grewal, K., Montenegro, G., and M. Bhatia, "Wrapped Encapsulating Security Payload (ESP) for Traffic Visibility", RFC 5840, April 2010.

[FIPS-180-3] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)", FIPS PUB 180-3, October 2008.

12.2. Informative References

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[I-D.hammer-webfinger] Hammer-Lahav, E., Fitzpatrick, B., and B. Cook, "The WebFinger Protocol", draft-hammer-webfinger-00 (work in progress), October 2009.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

[RFC5083] Housley, R., "Cryptographic Message Syntax (CMS) Authenticated-Enveloped-Data Content Type", RFC 5083, November 2007.

[I-D.ietf-jose-json-web-signature] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-02 (work in progress), May 2012.

[krawczyk-ate] Krawczyk, H., "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)", Advances in cryptology--CRYPTO 2001 August 2001.

[GCM] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", SP 800-38D, November 2007.

Appendix A. Acknowledgments

[TODO]

Author's Address

Richard Barnes
BBN Technologies
1300 N. 17th St.
Arlington, VA 22209
USA

Email: rbarnes@bbn.com