Network Working Group E. Rescorla
Internet-Draft RTFM, Inc.
Intended status: Standards Track J. Hildebrand
Expires: September 8, 2011 Cisco Systems, Inc.
March 7, 2011

JavaScript Message Security Format
draft-rescorla-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 congenial for Web applications. This document describes a new cryptographic message format which is based on JavaScript Object Notation (JSON) and thus is easy for Web applications to generate and parse.

Status of this Memo

Copyright Notice

Table of Contents

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 BER/DER) which are not congenial for 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) intended to meet the need of the Web environment. While JSMS is modeled on existing formats -- principally CMS [RFC5652] -- it uses JavaScript Object Notation (JSON) rather than ASN.1/BER/DER, making it far easier for Web applications to handle. In the interest of simplicity, JSMS also omits as many as possible of the CMS modes (multiple signatures, 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].

3. Overview

The JSMS message format is simply a JSON [RFC4627] dictionary 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 two operational modes:

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, either via a separate Message Authentication Code (MAC) or an Authenticated Encryption with Associated Data (AEAD) algorithm such as AES-GCM or AES-CCM.

Signed Data
: A block of data signed by a single signer using his asymmetric key and optionally carrying his certificate. Multiple signatures are not permitted in order to keep things simple.

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. (See Section 4.6 for more on composition.

3.2. Conventions

In general, JSMS follows the following structural conventions:

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.

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. 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. In addition, because many implementations will not want to do any ASN.1/BER processing at all, we will define a Web Service which applications can use for chain validation and translation to an easy-to-parse format. (See [TODO]).

3.4. Certificate Discovery

JSMS will often be used in an online messaging environment 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

All of the field definitions in this section make use of JSON Schema [I-D.zyp-json-schema]. For each of the fields that is designed to hold an enumerated value, a registry will be created allowing other values to be used in addition to the values enumerated in the schema.

4.1. Base64 Handling

As stated in section 3.1 of [RFC4648], Base64 does not require linefeeds after a specific number of characters. 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 are unintended side-effects of the production process.

Implementation Note: Much existing Base64-encoding code will generate linefeeds every 64 or 76 characters of output. Ensure that these linefeeds are removed before inserting the output into a JSON structure.

4.2. Content Object

JSMS operates by providing transformations on "Content" objects, which are just mime-typed JSON objects. These objects are then wrapped in a signed/encrypted wrapper with the following fields:

ContentType: A MIME [RFC2045] media type that MUST be included indicating the type of the "Data" field.

Type: The constant string "content", to facilitate easy determination that this is the target content. This is useful (for example) in certain operating conditions where you must continue to unwrap layers of signatures until you get to the content. This field MUST be included.

Data: The data value MUST be included as a text encoded as Base64 (See: [RFC4648]).

ID: An OPTIONAL universally unique ID that identifies this message, for use in detecting replay attacks.

Created: An OPTIONAL field describing the UTC date/time that the content was encoded into JSON, formatted according to the "date-time" production of [RFC3339].

Signing and encryption transform a "Content" object into "Signed" and "Encrypted" objects respectively. Verification and decryption transform "Signed" and "Encrypted" objects back into "Content" objects. For example:

```
{
    "ContentType":"text/plain; charset=UTF-8",
    "Type":"content",
    "Data":"SGVsbG8sIFdvcmxkCg==",
    "ID":"746a4c9f-8e84-4313-b669-81590ee2949e",
    "Created":"2011-03-07T16:17Z"
}
```

Figure 1: Content Example

4.3. Common Elements

A JSMS message is a JSON dictionary object containing a set of specific values.

The following fields MUST be present in all messages:

Version: The version number. For this specification this value MUST be set to the string "1.0". See Section 5 for details on version handling.

Type: The type of the message. MUST be either "signed" or "encrypted", to indicate a signed message (Section 4.4) or an encrypted message (Section 4.5) respectively.

4.4. Signed Data

A "signed" message contains a signed data block plus a digital signature over that data. To simplify implementation, only one signer is allowed. In addition to the required fields from Section 4.3, the fields in a signature message are:

SignedData: This field MUST consist of a Base64-encoded "Content" structure (see Section 4.2), which MUST have been encoded into octets as UTF-8 prior to Base64-encoding. The signature is computed over the UTF-8 octet stream before Base64-encoding to ensure that the sender and receiver have the exact same representation.

DigestAlgorithm: The message digest used to compute the signature. This field MUST be present for RSA-based signatures but MAY be omitted for future signatures which do not allow flexible digests. For now, this field MUST have the value "SHA-256", meaning the digest algorithm was SHA-256 [FIPS-180-3].

SignatureAlgorithm: The signature algorithm used to compute the signature. This field MUST be present. For now, this field MUST have the value "RSA-PKCS1-1.5", meaning the signature algorithm was RSASSA-PKCS1-v1_5 as specified in [RFC3447].

Signer: The signer’s identity, expressed as a URI [RFC3986]. This field MUST be present.

CertChain: The signer’s certificate chain, if any (see Section 4.4.2.1).

Signature: The Base64-encoded signature, which MUST be included (see Section 4.4.1).

```
{
    "SignedData":"ewogICAgIkNvbnRlbnRUeXBlIjoidGV4dC9wbGFpbjsgY2hhcn
                  NldD1VVEYtOCIsCiAgICAiVHlwZSI6ImNvbnRlbnQiLAogICAg
                  IkRhdGEiOiJTR1ZzYkc4c0lGZHZjbXhrQ2c9PSIsCiAgICAiSU
                  QiOiI3NDZhNGM5Zi04ZTg0LTQzMTMtYjY2OS04MTU5MGVlMjk0
                  OWUiLAogICAgIkNyZWF0ZWQiOiIyMDExLTAzLTA3VDE2OjE3Wi
                  IKfQ==",
    "DigestAlgorithm":"SHA-256",
    "SignatureAlgorithm":"RSA-PKCS1-1.5",
    "Signer":"xmpp:romeo@example.net",
    "Signature":"sNsxJltUaz4pSzAtJiPZagUMV4SwWugWexGbffK/WJRDi2uq7TxN
                 /V9SwG/kvQ7CaTABbeUuc6cKGO5YxnH5hME3bHB5L9PKPWSjxzxo
                 68RPxQyPli2YJDDHKVPbofEa86CLqYcwTF5qrcL7fQFvlRSOVxpS
                 SJfIdiAJNA+nEnk="
}
```

Figure 2: Signed Message Example

4.4.1. Signature Computation

The signature is computed over the string prior to base64 encoding. I.e., the processing order for encoding is:

1. Serialize the inner "Content" value into a UTF8-encoded octet series X.
2. Compute the signature value over X, and call the result Y. (In the case of signatures which use digests, this means feed the literal octets of the signature into the digest function.)
3. Compute the Base64 representation of X and insert it into the "SignedData" field of the message.
4. Compute the Base64 representation of Y, and insert the result into the "Signature" field.

This procedure removes dependencies on the exact serialization algorithm; variation in spacing, field order, etc. do not affect signature validity since the Base64 representation preserves them on the wire and protects them from modification by intermediaries.

Note: An alternative algorithm would be to compute the signature on the base64 representation itself, but this has two disadvantages: (1) any intermediaries which change spacing/line breaks would break the signature. (2) it is inconsistent with the algorithm for encryption (Section 4.5), which is designed to avoid multiple base64 encoding.

This procedure only specifies the input to the signature computation. The details of the computation depend on the signature algorithm itself. The mapping from code points to algorithms is found in

Section 6.

4.4.2. Signature Verification

In order to verify the signature, the steps of the previous section are reversed.

1. Process the provided "Signer" and "CertChain" fields as described in Section 4.4.2.1 in order to determine the sender’s public key.
2. Base64 decode the "SignedData" field in order to recover a string X.
3. Verify the "Signature" field against X using the sender’s public key and the "SignatureAlgorithm" and "DigestAlgorithm" fields. If the signature fails, return an error.
4. Deserialize X to recover the inner "Content" value.
5. Check any "ID" or "Created" fields for replay.
6. Using the value of the "ContentType" field to give MIME type context, Base64-decode the "Data" field to retrieve the intended message.

4.4.2.1. Certificate Processing

JSMS uses the "CertChain" element to carry certificate chains. For the moment, each certificate in the chain is expected to be a PKIX certificate BER-encoded then Base64-encoded. Future versions of this document will likely specify other valid certificate formats, since one of the goals of this format is to avoid . The meaning of the fields is described below:

Type: The type of the certificate chain. The only defined value is "PKIX", referring to PKIX [RFC5280] certificates.

Chain: An array of certificate values. In the case of "PKIX" certificates this is a list of base64-encoded DER/BER PKIX certificate values. PKIX certificates MUST be represented in order with each certificate certifying the next and the final certificate representing the end-entity.

```
   {
       "Type":"PKIX",
       "Chain":[
           "MIICPjCCAaegAwIBAgIBETANBgkqhkiG9w0BAQUFADBDMRMwEQ
            YKCZImiZPyLGQBGRYDY29tMRcwFQYKCZImiZPyLGQBGRYHZXhh
            bXBsZTETMBEGA1UEAxMKRXhhbXBsZSBDQTAeFw0wNDA0MzAxND
            I1MzRaFw0wNTA0MzAxNDI1MzRaMEMxEzARBgoJkiaJk/IsZAEZ
            FgNjb20xFzAVBgoJkiaJk/IsZAEZFgdleGFtcGxlMRMwEQYDVQ
            QDEwpFeGFtcGxlIENBMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCB
            iQKBgQDC15dtKHCqW88jLoBwOe7bb9Ut1WpPejQt+SJyR3Ad74
            DpyjCMAMSabltFtG6l5myUDfqR6UD8JZ3Ht2gZVo8RcGrX8ckR
            Tzp+P5mNbnaldF9epFVT5cdoNlPHHTsSpoX+vW6hyt81UKwI17
            m0flz+4qMs0SOEqpjAm2YYmmhH6QIDAQABo0IwQDAdBgNVHQ4E
            FgQUCGivhTPIOUp6+IKTjnBqSiCELDIwDgYDVR0PAQH/BAQDAg
            EGMA8GA1UdEwEB/wQFMAMBAf8wDQYJKoZIhvcNAQEFBQADgYEA
            bPgCdKZh4mQEplQMbHITrTxH+/ZlE6mFkDPqdqMm2fzRDhVfKL
            fvk7888+I+fLlS/BZuKarh9Hpv1X/vs5XK82aIg06hNUWEy7yb
            uMitxV5G2QsOjYDhMyvcviuSfkpDqWrvimNhs25HOL7oDaNnXf
            P6kYE8krvFXyUl63zn2KE=",
           "MIICcTCCAdqgAwIBAgIBEjANBgkqhkiG9w0BAQUFADBDMRMwEQ
            YKCZImiZPyLGQBGRYDY29tMRcwFQYKCZImiZPyLGQBGRYHZXhh
            bXBsZTETMBEGA1UEAxMKRXhhbXBsZSBDQTAeFw0wNDA5MTUxMT
            Q4MjFaFw0wNTAzMTUxMTQ4MjFaMEMxEzARBgoJkiaJk/IsZAEZ
            FgNjb20xFzAVBgoJkiaJk/IsZAEZFgdleGFtcGxlMRMwEQYDVQ
            QDEwpFbmQgRW50aXR5MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCB
            iQKBgQDhauQDMJcCPPQQ87UeTX8Ue/b10HjppIrwo3Xs7bZWln
            +ImYWa8j5od4frntGfwLQX3KuJI6QdfhYjTE+oTfUxuHyq4xpJ
            CfRLJtsnZzCCEgFK6Rq2wQxTi2z8L3pD7DM2fjKye9WqzwEUxh
            LsE/ItFHqLIVgUE0xGo5ryFpX/IwIDAQABo3UwczAhBgNVHREE
            GjAYgRZlbmQuZW50aXR5QGV4YW1wbGUuY29tMB0GA1UdDgQWBB
            QXe5Iw/0TWZuGQECJsFk/AjkHdbTAfBgNVHSMEGDAWgBQIaK+F
            M8g5Snr4gpOOcGpKIIQsMjAOBgNVHQ8BAf8EBAMCBsAwDQYJKo
            ZIhvcNAQEFBQADgYEAACAoNFtoMgG7CjYOrXHFlRrhBM+urcdi
            FKQbNjHA4gw92R7AANwQoLqFb0HLYnq3TGOBJl7SgEVeM+dwRT
            s5OyZKnDvyJjZpCHm7+5ZDd0thi6GrkWTg8zdhPBqjpMmKsr9z
            1E3kWORi6rwgdJKGDs6EYHbpc7vHhdORRepiXc0="
       ]
   }
```

Figure 3: PKIX CertChain Example

The recipient MUST verify the certificate chain (in the case of PKIX certificates according to [RFC5280]). If any validation failure occurs, the implementation MUST abort processing and return an error.

Once the certificate chain is validated, the end-entity certificate must contain an identity which matches the "Signer" field. In the case of PKIX certificates, the certificate MUST contain a subjectAltName field of type "uniformResourceIdentifier". This field MUST be equivalent to the URI in the "Signer" field. If not, an error MUST be returned.

4.5. Encrypted Data

An "encrypted" message contains an encrypted "Content" block. All "encrypted" messages contain a symmetric integrity check, either via a MAC or via an AEAD [RFC5116] algorithm such as Galois/Counter Mode (GCM: [GCM]). A message may be encrypted to an arbitrary number of recipients. Each recipient is represented by a "Recipient" block, which contains a copy of the keying material encrypted for that recipient. Both symmetric and asymmetric key establishment is supported. In order to support both integrity and encryption, what is carried in the Recipient block is a Content Master Key (CMK) which is then used with a Key Derivation Function (KDF) to generate the Content Encryption Key (CEK) used to encrypt the message and the Content Integrity Key (CIK) used with the MAC. In addition to the required fields from Section 4.3 the fields in an encrypted message are:

Recipients: The list of recipients. This is an array of Recipient objects, each of which establishes the CMK for that recipient.

KDF: Specifies the key derivation function used to generate the CEK and the CIK from the CMK. This field MAY be absent if an AEAD algorithm is used, in which case the CEK is derived by copying the CMK.

Encryption: Specifies the properties of the encryption. The Algorithm field MUST contain the encryption algorithm and the IV field specifies the initialization vector (if required for the algorithm). This field MUST be present.

Integrity: Specifies the properties of the integrity check. The Algorithm field MUST contain the MAC algorithm and the Value field MUST contain the MAC. This field MAY be absent if no integrity check is used.

Data: Contains the ciphertext.

Each Recipient object provides an encrypted copy of the CMK for a single recipient. The meaning of the fields is described below:

KEKidentifier Describes the key encrypting key (KEK) used to encrypt the CMK. Either a "RecipientName" or a "KeyIdentifier" MUST be provided. If the "RecipientName" is provided, then a "CertificateDigest" SHOULD be provided.

RecipientName: Provides the recipient’s name in URI form.

CertificateDigest: For now, the SHA-1 fingerprint of the PKIX certificate associated with the recipient.

KeyIdentifier The name of a shared symmetric key known to both sender and recipient. This need not be globally unique as long as it is unique within the recipient’s context.

Algorithm: The algorithm used to encrypt the CMK. For now, one of "RSA-PKCS1-1.5" (meaning RSASSA-PKCS1-v1_5 as specified in [RFC3447]) or "AES-256-CBC" (meaning [FIPS-180-3]). Note the JSMS only supports key transport and not key agreement (since key agreement can always be turned into key transport).

Value: The CMK encrypted under the specified algorithm and key.

4.5.1. Message Encryption

The message encryption process is as follows.

1. Generate a random CMK. The CMK MUST have a length at least equal to that of the larger of the required integrity or encryption keys and MUST be generated randomly. See [RFC4086] for considerations on generating random values. [[TODO - we need a section on generating randomness in browsers - it’s easy to screw up]]
2. Encrypt the CMK for each recipient (see Section 4.5.4)
3. Generate a random IV (if required for the algorithm).
4. Run the key derivation algorithm (see Section 4.5.3) to generate the CEK and CIK (if not using an AEAD algorithm).
5. Serialize the content into a bitstring M.
6. Encrypt M using the CEK and IV to form the bitstring C.
7. Set the Value element equal to the base64-encoded representation of C.
8. If not using an AEAD algorithm, compute the function I = MAC(CIK, C) using the chosen integrity algorithm. Note that this is EtA encryption which is considered the best cryptographic choice (See: [krawczyk-ate]). Set the Integrity.Value element equal to the base64-encoded representation of I.

4.5.2. Message Decryption

The message decryption process is the reverse of the encryption process.

1. Identify a Recipient block which appears to reference a key known to the recipient.
2. Decrypt the CMK. If this fails and another Recipient block appears plausible, that MAY be tried.

3. Run the key derivation algorithm (see Section 4.5.3) to generate the CEK and CIK (if not using an AEAD algorithm).
4. If not using an AEAD algorithm, compute the integrity check value I’ on the binary representation of the Value element using the indicated integrity check. If the Integrity.Value does not match I’, then an error MUST be reported and processing MUST be aborted.
5. Decrypt the binary representation of the Value element and output the result

4.5.3. Key Derivation

The key derivation process converts the CMK into a CEK. It assumes as a primitive a Key Derivation Function (KDF) which notionally takes three arguments:

MasterKey:	The master key used to compute the individual use keys
Label:	The use key label, used to differentiate individual use keys
Length:	The length of the desired use key

The only real KDF specified in this document is the TLS PRF, which is invoked as PRF(MasterKey, Label) with an empty seed and produces an arbitrary length output. The appropriate number of bits (Length) is simply extracted from the beginning of the output. The KDF name "P_XXX" in this document refers the the TLS [RFC5246] PRF using P_XXX as the underlying P_hash function.

To compute the CEK from the CMK, the label "Encryption" is used.

To compute the CIK from the CMK, the label "Integrity" is used.

When AEAD algorithms are used the KDF element MUST NOT be present. When they are not used, it MUST be present.

4.5.4. CMK Encryption

JSMS supports two forms of CMK encryption:

- o Asymmetric encryption under the recipient’s public key.
- o Symmetric encryption under a shared key.

4.5.4.1. Asymmetric Encryption

In the asymmetric encryption mode, the CMK is encrypted under the recipient’s public key. The only currently defined asymmetric encryption mode is RSA-PKCS1-1.5, which refers to [RFC3447] RSAES-PKCS1-v1_5.

4.5.4.2. Symmetric Encryption

In the symmetric encryption mode, the CMK is encrypted under a symmetric key shared between the sender and receiver. All such modes MUST provide integrity for the CMK. This document defines four such modes: AES-128-CBC, AES-256-CBC referring to the [RFC5649] AES key wrapping modes and AES-128-GCM, AES-256-GCM, referring to AES encryption with GCM. For GCM the random 64-bit IV is prepended to the ciphertext.

4.6. Composition

This document does not specify a combination signed and encrypted mode. However, because the contents of a message can be arbitrary, and encryption and data origin authentication can be provided by recursively encapsulating multiple JSMS messages. In general, senders SHOULD sign the message and then encrypt the result (thus encrypting the signature). This prevents attacks in which the signature is stripped, leaving just an encrypted message, as well as providing privacy for the signer.

5. Version Processing

For the moment, all version numbers in the protocol MUST be 1.0. Receivers MUST return an error for any other version number. More interesting version processing will be defined in the future.

6. IANA Considerations

[TODO]
- Register MIME types
- Registries for signature, encryption, MAC
- Well known HTTP URLs

7. Security Considerations

Much more to follow here.

8. References

8.1. 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.

[I-D.zyp-json-schema]
Zyp, K. and G. Court, "A JSON Media Type for Describing the Structure and Meaning of JSON Documents", draft-zyp-json-schema-03 (work in progress), November 2010.

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

8.2. 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.

[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. JSON Schema

The following schemas formally define various namespaces used in this document, in conformance with [I-D.zyp-json-schema]. Because validation of JSON documents is optional, these schemas are not normative and are provided for descriptive purposes only.

A.1. Message Contents Schema

```
   {
       "description":"Message Contents",
       "type":"object",
       "properties":{
           "ContentType":{
               "description":"A MIME content type",
               "type":"string",
               "required":true
           },
           "Type":{
               "description":"Dictionary type",
               "type":"string",
               "enum":["content"],
               "required":true
           },
           "Data":{
               "description":"The underlying data",
               "type":"string",
               "required":true
           },
           "ID":{
               "description":"(optional) unique ID for this message",
               "type":"string"
           },
           "Created":{
               "description":"(optional) time the message was created",
               "type":"string",
               "format":"date-time"
           }
       }
   }
```

A.2. Common Elements Schema

```
   {
       "description":"The basic schema for a JSMS message",
       "type":"object",
       "properties":{
           "Type":{
               "description":"Message type",
               "type":"string",
               "enum":["signed", "encrypted"]
           },
           "Version":{
               "description":"Version number for the message",
               "type":"string",
               "enum":["1.0"]
           }
       }
   }
```

A.3. Signed Message Schema

```
   {
       "description":"A signed message",
       "type":"object",
       "extends":message_schema,
       "properties":{
           "Signature":{
               "description":"The signature over the SignedData",
               "type":"object",
               "properties":{
                   "SignedData":{
                       "description":"content to be signed, Base64",
                       "type":"string",
                       "required":true
                   },
                   "DigestAlgorithm":{
                       "description":"",
                       "type":"string",
                       "enum":["SHA-256"]
                   },
                   "SignatureAlgorithm":{
                       "description":"",
                       "type":"string",
                       "enum":["RSA-PKCS1-1.5"]
                   },
                   "Signer":{
                       "description":"",
                       "type":"string",
                       "format":"uri",
                       "required":true
                   },
                   "CertChain": {
                       "description":"the signer’s cert chain",
                       "type":"PKIXcertchain"
                   },
                   "Signature":{
                       "description":"the signature",
                       "type":"string",
                       "required":true
                   }
               }
           }
       }
   }
```

A.4. PKIX Certificate Chain Schema

```
   {
       "description":"A chain of PKIX certificates",
       "id":"PKIXcertchain",
       "properties":{
           "Type":{
               "description":"The type of certificate chain",
               "type":"string",
               "enum":["PKIX"] },
           "Chain":{
               "description":"PKIX certs ordered from root to end",
               "type":"array",
               "items":{
                   "description":"A base64-encoded BER certificate",
                   "type":"string"
               }
           }
       }
   }
```

A.5. Encrypted Message Schema

```
   {
       "description":"An encrypted object",
       "type":"object",
       "extends":message_schema,
       "properties":{
           "Recipients":{
               "description":"The list of recipient blocks",
               "type":"array",
               "required":true,
               "items":{
                   "description":"A single recipient block",
                   "type":"Recipient"
               }
           },
           "KDF":{
               "description":
               "The KDF used to derive the MAC and encryption keys",
               "type":"string",
               "enum":["P_SHA256"]
           },
           "Encryption":{
               "description":"Encryption control information",
               "type":"object",
               "required":true,
               "properties":{
                   "Algorithm":{
                       "description":"The algorithm used to encrypt",
                       "type":"string",
                       "enum":["AES-256-CBC"]
                   },
                   "IV":{
                       "description":"Initialization vector (base64)",
                       "type":"string"
                   }
               }
           },
           "Integrity":{
               "description":"The integrity control information",
               "type":"object",
               "properties":{
                   "Algorithm":{
                       "description":"The MAC algorithm",
                       "type":"string",
                       "enum":["HMAC-SHA-256"]
                   },
                   "Value":{
                       "description":"The MAC value (base64-encoded)",
                       "type":"string",
                       "required":true
                   }
               }
           },
           "Data":{
               "description":"The ciphertext (Base64-encoded)",
               "type":"string",
               "required":true
           }
       }
   }
```

A.6. Recipient Schema

```
   {
       "description":"The recipient of an encrypted object",
       "type":"object",
       "id":"Recipient",
       "properties":{
           "KEKidentifier":{
               "type":"object",
               "description":"Identifies the key encrypting key",
               "properties":{
                   "RecipientName":{
                       "type":"string",
                       "description":"The recipient’s name",
                       "format":"uri"
                   },
                   "CertificateDigest":{
                       "type":"string",
                       "description":"Recipient’s cert fingerprint"
                   },
                   "KeyIdentifier":{
                       "type":"string",
                       "description": "Shared symmetric key (opaque)"
                   }
               }
           },
           "Algorithm":{
               "description":"The algorithm used to protect the CMK",
               "type":"string",
               "enum":["RSA-PKCS1-1.5", "AES-256-CBC"]
           },
           "Value":{
               "description": "Base64 of the encrypted CMK",
               "type":"string"
           }
       }
   }
```

Appendix B. Acknowledgments

[TODO]

Authors' Addresses

Eric Rescorla
RTFM, Inc.
2064 Edgewood Drive
Palo Alto, CA 94303
USA

Email: ekr@rtfm.com

Joe Hildebrand
Cisco Systems, Inc.
1899 Wyknoop Street, Suite 600
Denver, CO 80202
USA

Email: jhildebr@cisco.com