INTERNET-DRAFT
Internet Engineering Task Force (IETF)
Intended Status: Proposed Standard
Expires: 6 February 2020

R. Housley
Vigil Security

6 August 2019

Using Pre-Shared Key (PSK) in the Cryptographic Message Syntax (CMS) <draft-ietf-lamps-cms-mix-with-psk-06.txt>

Abstract

The invention of a large-scale quantum computer would pose a serious challenge for the cryptographic algorithms that are widely deployed today. The Cryptographic Message Syntax (CMS) supports key transport and key agreement algorithms that could be broken by the invention of such a quantum computer. By storing communications that are protected with the CMS today, someone could decrypt them in the future when a large-scale quantum computer becomes available. Once quantum-secure key management algorithms are available, the CMS will be extended to support the new algorithms, if the existing syntax does not accommodate them. In the near-term, this document describes a mechanism to protect today's communication from the future invention of a large-scale quantum computer by mixing the output of key transport and key agreement algorithms with a pre-shared key.

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."

Table of Contents

1. Introduction

The invention of a large-scale quantum computer would pose a serious challenge for the cryptographic algorithms that are widely deployed today [S1994]. It is an open question whether or not it is feasible to build a large-scale quantum computer, and if so, when that might happen [NAS2019]. However, if such a quantum computer is invented, many of the cryptographic algorithms and the security protocols that use them would become vulnerable.

The Cryptographic Message Syntax (CMS) [RFC5652][RFC5083] supports key transport and key agreement algorithms that could be broken by the invention of a large-scale quantum computer [C2PQ]. These algorithms include RSA [RFC8017], Diffie-Hellman [RFC2631], and Elliptic Curve Diffie-Hellman [RFC5753]. As a result, an adversary that stores CMS-protected communications today, could decrypt those communications in the future when a large-scale quantum computer becomes available.

Once quantum-secure key management algorithms are available, the CMS will be extended to support them, if the existing syntax does not already accommodate the new algorithms.

In the near-term, this document describes a mechanism to protect today's communication from the future invention of a large-scale quantum computer by mixing the output of existing key transport and key agreement algorithms with a pre-shared key (PSK). Secure communication can be achieved today by mixing a strong PSK with the output of an existing key transport algorithm, like RSA [RFC8017], or an existing key agreement algorithm, like Diffie-Hellman [RFC2631] or Elliptic Curve Diffie-Hellman [RFC5753]. A security solution that is believed to be quantum resistant can be achieved by using a PSK with sufficient entropy along with a quantum resistant key derivation function (KDF), like HKDF [RFC5869], and a quantum resistant encryption algorithm, like 256-bit AES [AES]. In this way, today's CMS-protected communication can be resistant to an attacker with a large-scale quantum computer.

In addition, there may be other reasons for including a strong PSK besides protection against the future invention of a large-scale quantum computer. For example, there is always the possibility of a cryptoanalytic breakthrough on one or more of the classic public-key algorithm, and there are longstanding concerns about undisclosed trapdoors in Diffie-Hellman parameters [FGHT2016]. Inclusion of a strong PSK as part of the overall key management offer additional protection against these concerns.

Note that the CMS also supports key management techniques based on symmetric key-encryption keys and passwords, but they are not discussed in this document because they are already quantum resistant. The symmetric key-encryption key technique is quantum resistant when used with an adequate key size. The password technique is quantum resistant when used with a quantum-resistant key derivation function and a sufficiently large password.

1.1. Terminology

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.

1.2. ASN.1

CMS values are generated using ASN.1 [X680], which uses the Basic Encoding Rules (BER) and the Distinguished Encoding Rules (DER) [X690].

1.3. Version Numbers

The major data structures include a version number as the first item in the data structure. The version number is intended to avoid ASN.1 decode errors. Some implementations do not check the version number prior to attempting a decode, and then if a decode error occurs, the version number is checked as part of the error handling routine. This is a reasonable approach; it places error processing outside of the fast path. This approach is also forgiving when an incorrect version number is used by the sender.

Whenever the structure is updated, a higher version number will be assigned. However, to ensure maximum interoperability, the higher version number is only used when the new syntax feature is employed. That is, the lowest version number that supports the generated syntax is used.

2. Overview

The CMS enveloped-data content type [RFC5652] and the CMS authenticated-enveloped-data content type [RFC5083] support both key transport and key agreement public-key algorithms to establish the key used to encrypt the content. No restrictions are imposed on the key transport or key agreement public-key algorithms, which means that any key transport or key agreement algorithm can be used, including algorithms that are specified in the future. In both cases, the sender randomly generates the content-encryption key, and then all recipients obtain that key. All recipients use the sender-generated symmetric content-encryption key for decryption.

This specification defines two quantum-resistant ways to establish a symmetric key-encryption key, which is used to encrypt the sender-generated content-encryption key. In both cases, the PSK is used as one of the inputs to a key-derivation function to create a quantum-

resistant key-encryption key. The PSK MUST be distributed to the sender and all of the recipients by some out-of-band means that does not make it vulnerable to the future invention of a large-scale quantum computer, and an identifier MUST be assigned to the PSK.

The content-encryption key or content-authenticated-encryption key is quantum-resistant, and the sender establishes it using these steps:

When using a key transport algorithm:

1. The content-encryption key or the content-authenticated-encryption key, called CEK, is generated at random.

2. The key-derivation key, called KDK, is generated at random.

3. For each recipient, the KDK is encrypted in the recipient's public key, then the key derivation function (KDF) is used to mix the pre-shared key (PSK) and the KDK to produce the key-encryption key, called KEK.

4. The KEK is used to encrypt the CEK.

When using a key agreement algorithm:

1. The content-encryption key or the content-authenticated-encryption key, called CEK, is generated at random.

2. For each recipient, a pairwise key-encryption key, called KEK1, is established using the recipient's public key and the sender's private key. Note that KEK1 will be used as a key-derivation key.

3. For each recipient, the key derivation function (KDF) is used to mix the pre-shared key (PSK) and the pairwise KEK1, and the result is called KEK2.

4. For each recipient, the pairwise KEK2 is used to encrypt the CEK.

As specified in Section 6.2.5 of [RFC5652], recipient information for additional key management techniques are represented in the OtherRecipientInfo type. Two key management techniques are specified in this document, and they are each identified by a unique ASN.1 object identifier.

The first key management technique, called keyTransPSK, see Section 3, uses a key transport algorithm to transfer the key-derivation key from the sender to the recipient, and then the key-

derivation key is mixed with the PSK using a KDF. The output of the KDF is the key-encryption key, which is used for the encryption of the content-encryption key or content-authenticated-encryption key.

The second key management technique, called keyAgreePSK, see Section 4, uses a key agreement algorithm to establish a pairwise key-encryption key, which is then mixed with the PSK using a KDF to produce a second pairwise key-encryption key, which is then used to encrypt the content-encryption key or content-authenticated-encryption key.

3. keyTransPSK

Per-recipient information using keyTransPSK is represented in the KeyTransPSKRecipientInfo type, which is indicated by the id-ori-keyTransPSK object identifier. Each instance of KeyTransPSKRecipientInfo establishes the content-encryption key or content-authenticated-encryption key for one or more recipients that have access to the same PSK.

The id-ori-keyTransPSK object identifier is:

```
id-ori OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) TBD1 }

id-ori-keyTransPSK OBJECT IDENTIFIER ::= { id-ori 1 }
```

The KeyTransPSKRecipientInfo type is:

```
KeyTransPSKRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 0
  pskid PreSharedKeyIdentifier,
  kdfAlgorithm KeyDerivationAlgorithmIdentifier,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  ktris KeyTransRecipientInfos,
  encryptedKey EncryptedKey }

PreSharedKeyIdentifier ::= OCTET STRING

KeyTransRecipientInfos ::= SEQUENCE OF KeyTransRecipientInfo
```

The fields of the KeyTransPSKRecipientInfo type have the following meanings:

version is the syntax version number. The version MUST be 0. The CMSVersion type is described in Section 10.2.5 of [RFC5652].

pskid is the identifier of the PSK used by the sender. The identifier is an OCTET STRING, and it need not be human readable.

kdfAlgorithm identifies the key-derivation algorithm, and any associated parameters, used by the sender to mix the key-derivation key and the PSK to generate the key-encryption key. The KeyDerivationAlgorithmIdentifier is described in Section 10.1.6 of [RFC5652].

keyEncryptionAlgorithm identifies a key-encryption algorithm used to encrypt the content-encryption key. The KeyEncryptionAlgorithmIdentifier is described in Section 10.1.3 of [RFC5652].

ktris contains one KeyTransRecipientInfo type for each recipient; it uses a key transport algorithm to establish the key-derivation key. KeyTransRecipientInfo is described in Section 6.2.1 of [RFC5652].

encryptedKey is the result of encrypting the content-encryption key or the content-authenticated-encryption key with the key-encryption key. EncryptedKey is an OCTET STRING.

4. keyAgreePSK

Per-recipient information using keyAgreePSK is represented in the KeyAgreePSKRecipientInfo type, which is indicated by the id-ori-keyAgreePSK object identifier. Each instance of KeyAgreePSKRecipientInfo establishes the content-encryption key or content-authenticated-encryption key for one or more recipients that have access to the same PSK.

The id-ori-keyAgreePSK object identifier is:

```
id-ori-keyAgreePSK OBJECT IDENTIFIER ::= { id-ori 2 }
```

The KeyAgreePSKRecipientInfo type is:

```
KeyAgreePSKRecipientInfo ::= SEQUENCE {
  version CMSVersion,  -- always set to 0
  pskid PreSharedKeyIdentifier,
  originator [0] EXPLICIT OriginatorIdentifierOrKey,
  ukm [1] EXPLICIT UserKeyingMaterial OPTIONAL,
  kdfAlgorithm KeyDerivationAlgorithmIdentifier,
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  recipientEncryptedKeys RecipientEncryptedKeys }
```

The fields of the KeyAgreePSKRecipientInfo type have the following meanings:

version is the syntax version number. The version MUST be 0. The CMSVersion type is described in Section 10.2.5 of [RFC5652].

pskid is the identifier of the PSK used by the sender. The identifier is an OCTET STRING, and it need not be human readable.

originator is a CHOICE with three alternatives specifying the sender's key agreement public key. Implementations MUST support all three alternatives for specifying the sender's public key. The sender uses their own private key and the recipient's public key to generate a pairwise key-encryption key. A key derivation function (KDF) is used to mix the PSK and the pairwise key-encryption key to produce a second key-encryption key. The OriginatorIdentifierOrKey type is described in Section 6.2.2 of [RFC5652].

ukm is optional. With some key agreement algorithms, the sender provides a User Keying Material (UKM) to ensure that a different key is generated each time the same two parties generate a pairwise key. Implementations MUST accept a KeyAgreePSKRecipientInfo SEQUENCE that includes a ukm field. Implementations that do not support key agreement algorithms that make use of UKMs MUST gracefully handle the presence of UKMs. The UserKeyingMaterial type is described in Section 10.2.6 of [RFC5652].

kdfAlgorithm identifies the key-derivation algorithm, and any associated parameters, used by the sender to mix the pairwise key-encryption key and the PSK to produce a second key-encryption key of the same length as the first one. The KeyDerivationAlgorithmIdentifier is described in Section 10.1.6 of [RFC5652].

keyEncryptionAlgorithm identifies a key-encryption algorithm used to encrypt the content-encryption key or the content-authenticated-encryption key. The KeyEncryptionAlgorithmIdentifier type is described in Section 10.1.3 of [RFC5652].

recipientEncryptedKeys includes a recipient identifier and encrypted key for one or more recipients. The KeyAgreeRecipientIdentifier is a CHOICE with two alternatives specifying the recipient's certificate, and thereby the recipient's public key, that was used by the sender to generate a pairwise key-encryption key. The encryptedKey is the result of

encrypting the content-encryption key or the content-authenticated-encryption key with the second pairwise key-encryption key. EncryptedKey is an OCTET STRING. The RecipientEncryptedKeys type is defined in Section 6.2.2 of [RFC5652].

5. Key Derivation

Many key derivation functions (KDFs) internally employ a one-way hash function. When this is the case, the hash function that is used is identified by the KeyDerivationAlgorithmIdentifier. HKDF [RFC5869] is one example of a KDF that makes use of a hash function.

A KDF has several input values. This section describes the conventions for using the KDF to compute the key-encryption key for KeyTransPSKRecipientInfo and KeyAgreePSKRecipientInfo. For simplicity, the terminology used in the HKDF [RFC5869] specification is used here.

The KDF inputs are:

IKM is the input keying material; it is the symmetric secret input to the KDF. For KeyTransPSKRecipientInfo, it is the key-derivation key. For KeyAgreePSKRecipientInfo, it is the pairwise key-encryption key produced by the key agreement algorithm.

salt is an optional non-secret random value. The salt is not used.

L is the length of output keying material in octets; the value depends on the key-encryption algorithm that will be used. The algorithm is identified by the KeyEncryptionAlgorithmIdentifier. In addition, the OBJECT IDENTIFIER portion of the KeyEncryptionAlgorithmIdentifier is included in the next input value, called info.

info is optional context and application specific information. The DER-encoding of CMSORIforPSKOtherInfo is used as the info value, and the PSK is included in this structure. Note that EXPLICIT tagging is used in the ASN.1 module that defines this structure. For KeyTransPSKRecipientInfo, the ENUMERATED value of 5 is used. For KeyAgreePSKRecipientInfo, the ENUMERATED value of 10 is used. CMSORIforPSKOtherInfo is defined by the following

ASN.1 structure:

```
CMSORIforPSKOtherInfo ::= SEQUENCE {
  psk                    OCTET STRING,
  keyMgmtAlgType         ENUMERATED {
    keyTrans               (5),
    keyAgree               (10) },
  keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
  pskLength              INTEGER (1..MAX),
  kdkLength              INTEGER (1..MAX) }
```

The fields of type CMSORIforPSKOtherInfo have the following meanings:

psk is an OCTET STRING; it contains the PSK.

keyMgmtAlgType is either set to 5 or 10. For KeyTransPSKRecipientInfo, the ENUMERATED value of 5 is used. For KeyAgreePSKRecipientInfo, the ENUMERATED value of 10 is used.

keyEncryptionAlgorithm is the KeyEncryptionAlgorithmIdentifier, which identifies the algorithm and provides algorithm parameters, if any.

pskLength is a positive integer; it contains the length of the PSK in octets.

kdkLength is a positive integer; it contains the length of the key-derivation key in octets. For KeyTransPSKRecipientInfo, the key-derivation key is generated by the sender. For KeyAgreePSKRecipientInfo, the key-derivation key is the pairwise key-encryption key produced by the key agreement algorithm.

The KDF output is:

OKM is the output keying material, which is exactly L octets. The OKM is the key-encryption key that is used to encrypt the content-encryption key or the content-authenticated-encryption key.

6. ASN.1 Module

This section contains the ASN.1 module for the two key management techniques defined in this document. This module imports types from other ASN.1 modules that are defined in [RFC5911] and [RFC5912].

```
<CODE BEGINS>

CMSORIforPSK-2019
  { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9)
    smime(16) modules(0) id-mod-cms-ori-psk-2019(TBD0) }

DEFINITIONS EXPLICIT TAGS ::=
BEGIN

-- EXPORTS All

IMPORTS

AlgorithmIdentifier{}, KEY-DERIVATION
  FROM AlgorithmInformation-2009  -- [RFC5912]
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-algorithmInformation-02(58) }

OTHER-RECIPIENT, OtherRecipientInfo, CMSVersion,
KeyTransRecipientInfo, OriginatorIdentifierOrKey,
UserKeyingMaterial, RecipientEncryptedKeys, EncryptedKey,
KeyDerivationAlgorithmIdentifier, KeyEncryptionAlgorithmIdentifier
  FROM CryptographicMessageSyntax-2010  -- [RFC6268]
    { iso(1) member-body(2) us(840) rsadsi(113549)
      pkcs(1) pkcs-9(9) smime(16) modules(0)
      id-mod-cms-2009(58) } ;

--
-- OtherRecipientInfo Types (ori-)
--

SupportedOtherRecipInfo OTHER-RECIPIENT ::= {
  ori-keyTransPSK |
  ori-keyAgreePSK,
  ... }

--
-- Key Transport with Pre-Shared Key
--

ori-keyTransPSK OTHER-RECIPIENT ::= {
  KeyTransPSKRecipientInfo IDENTIFIED BY id-ori-keyTransPSK }

id-ori OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) TBD1 }

id-ori-keyTransPSK OBJECT IDENTIFIER ::= { id-ori 1 }
```

```
   KeyTransPSKRecipientInfo ::= SEQUENCE {
     version CMSVersion,  -- always set to 0
     pskid PreSharedKeyIdentifier,
     kdfAlgorithm KeyDerivationAlgorithmIdentifier,
     keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
     ktris KeyTransRecipientInfos,
     encryptedKey EncryptedKey }

   PreSharedKeyIdentifier ::= OCTET STRING

   KeyTransRecipientInfos ::= SEQUENCE OF KeyTransRecipientInfo

   --
   -- Key Agreement with Pre-Shared Key
   --

   ori-keyAgreePSK OTHER-RECIPIENT ::= {
     KeyAgreePSKRecipientInfo IDENTIFIED BY id-ori-keyAgreePSK }

   id-ori-keyAgreePSK OBJECT IDENTIFIER ::= { id-ori 2 }

   KeyAgreePSKRecipientInfo ::= SEQUENCE {
     version CMSVersion,  -- always set to 0
     pskid PreSharedKeyIdentifier,
     originator [0] EXPLICIT OriginatorIdentifierOrKey,
     ukm [1] EXPLICIT UserKeyingMaterial OPTIONAL,
     kdfAlgorithm KeyDerivationAlgorithmIdentifier,
     keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
     recipientEncryptedKeys RecipientEncryptedKeys }

   --
   -- Structure to provide 'info' input to the KDF,
   -- including the Pre-Shared Key
   --

   CMSORIforPSKOtherInfo ::= SEQUENCE {
     psk                    OCTET STRING,
     keyMgmtAlgType         ENUMERATED {
       keyTrans               (5),
       keyAgree               (10) },
     keyEncryptionAlgorithm KeyEncryptionAlgorithmIdentifier,
     pskLength              INTEGER (1..MAX),
     kdkLength              INTEGER (1..MAX) }

   END

   <CODE ENDS>
```

7. Security Considerations

Implementations must protect the pre-shared key (PSK), key transport private key, the agreement private key, the key-derivation key, and the key-encryption key. Compromise of the PSK will make the encrypted content vulnerable to the future invention of a large-scale quantum computer. Compromise of the PSK and either the key transport private key or the agreement private key may result in the disclosure of all contents protected with that combination of keying material. Compromise of the PSK and the key-derivation key may result in disclosure of all contents protected with that combination of keying material. Compromise of the key-encryption key may result in the disclosure of all content-encryption keys or content-authenticated-encryption keys that were protected with that keying material, which in turn may result in the disclosure of the content.

A large-scale quantum computer will essentially negate the security provided by the key transport algorithm or the key agreement algorithm, which means that the attacker with a large-scale quantum computer can discover the key-derivation key. In addition a large-scale quantum computer effectively cuts the security provided by a symmetric key algorithm in half. Therefore, the PSK needs at least 256 bits of entropy to provide 128 bits of security. To match that same level of security, the key derivation function needs to be quantum-resistant and produce a key-encryption key that is at least 256 bits in length. Similarly, the content-encryption key or content-authenticated-encryption key needs to be at least 256 bits in length.

When using a PSK with a key transport or a key agreement algorithm, a key-encryption key is produced to encrypt the content-encryption key or content-authenticated-encryption key. If the key-encryption algorithm is different than the algorithm used to protect the content, then the effective security is determined by the weaker of the two algorithms. If, for example, content is encrypted with 256-bit AES, and the key is wrapped with 128-bit AES, then at most 128 bits of protection is provided. Implementers must ensure that the key-encryption algorithm is as strong or stronger than the content-encryption algorithm or content-authenticated-encryption algorithm.

Implementers should not mix quantum-resistant key management algorithms with their non-quantum-resistant counterparts. For example, the same content should not be protected with KeyTransRecipientInfo and KeyTransPSKRecipientInfo. Likewise, the same content should not be protected with KeyAgreeRecipientInfo and KeyAgreePSKRecipientInfo. Doing so would make the content vulnerable to the future invention of a large-scale quantum computer.

Implementers should not send the same content in different messages, one using a quantum-resistant key management algorithm and the other using a non-quantum-resistant key management algorithm, even if the content-encryption key is generated independently. Doing so may allow an eavesdropper to correlate the messages, making the content vulnerable to the future invention of a large-scale quantum computer.

This specification does not require that PSK is known only by the sender and recipients. The PSK may be known to a group. Since confidentiality depends on the key transport or key agreement algorithm, knowledge of the PSK by other parties does not enable eavesdropping. However, group members can record the traffic of other members, and then decrypt it if they ever gain access to a large-scale quantum computer. Also, when many parties know the PSK, there are many opportunities for theft of the PSK by an attacker. Once an attacker has the PSK, they can decrypt stored traffic if they ever gain access to a large-scale quantum computer in the same manner as a legitimate group member.

Sound cryptographic key hygiene is to use a key for one and only one purpose. Use of the recipient's public key for both the traditional CMS and the PSK-mixing variation specified in this document would be a violation of this principle; however, there is no known way for an attacker to take advantage of this situation. That said, an application should enforce separation whenever possible. For example, a purpose identifier for use in the X.509 extended key usage certificate extension [RFC5280] could be identified in the future to indicate that a public key should only be used in conjunction with a PSK, or only without.

Implementations must randomly generate key-derivation keys as well as the content-encryption keys or content-authenticated-encryption keys. Also, the generation of public/private key pairs for the key transport and key agreement algorithms rely on a random numbers. The use of inadequate pseudo-random number generators (PRNGs) to generate cryptographic keys can result in little or no security. An attacker may find it much easier to reproduce the PRNG environment that produced the keys, searching the resulting small set of possibilities, rather than brute force searching the whole key space. The generation of quality random numbers is difficult. [RFC4086] offers important guidance in this area.

Implementers should be aware that cryptographic algorithms become weaker with time. As new cryptoanalysis techniques are developed and computing performance improves, the work factor to break a particular cryptographic algorithm will be reduced. Therefore, cryptographic algorithm implementations should be modular, allowing new algorithms to be readily inserted. That is, implementers should be prepared for

the set of supported algorithms to change over time.

The security properties provided by the mechanisms specified in this document can be validated using formal methods. A ProVerif proof in [H2019] shows that an attacker with a large-scale quantum computer that is capable of breaking the Diffie-Hellman key agreement algorithm cannot disrupt the delivery of the content-encryption key to the recipient and the attacker cannot learn the content-encryption key from the protocol exchange.

8. Privacy Considerations

An observer can see which parties are using each PSK simply by watching the PSK key identifiers. However, the addition of these key identifiers is not really making privacy worse. When key transport is used, the RecipientIdentifier is always present, and it clearly identifies each recipient to an observer. When key agreement is used, either the IssuerAndSerialNumber or the RecipientKeyIdentifier is always present, and these clearly identify each recipient.

9. IANA Considerations

One object identifier for the ASN.1 module in Section 6 was assigned in the SMI Security for S/MIME Module Identifiers (1.2.840.113549.1.9.16.0) [IANA-MOD] registry:

```
id-mod-cms-ori-psk-2019 OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
   pkcs-9(9) smime(16) mod(0) TBD0 }
```

One new registry was created for Other Recipient Info Identifiers within the SMI Security for S/MIME Mail Security (1.2.840.113549.1.9.16) [IANA-SMIME] registry:

```
id-ori OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) TBD1 }
```

Two assignments were made in the new SMI Security for Other Recipient Info Identifiers (1.2.840.113549.1.9.16.TBD1) [IANA-ORI] registry with references to this document:

```
id-ori-keyTransPSK OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
   pkcs-9(9) smime(16) id-ori(TBD1) 1 }

id-ori-keyAgreePSK OBJECT IDENTIFIER ::= {
   iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
   pkcs-9(9) smime(16) id-ori(TBD1) 2 }
```

10. References

10.1. Normative References

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

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

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.

[X680] ITU-T, "Information technology -- Abstract Syntax Notation One (ASN.1): Specification of basic notation", ITU-T Recommendation X.680, 2015.

[X690] ITU-T, "Information technology -- ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ITU-T Recommendation X.690, 2015.

10.2. Informative References

[AES] National Institute of Standards and Technology, FIPS Pub 197: Advanced Encryption Standard (AES), 26 November 2001.

[C2PQ] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", work-in-progress, draft-hoffman-c2pq-04, August 2018.

[FGHT2016] Fried, J., Gaudry, P., Heninger, N., and E. Thome, "A kilobit hidden SNFS discrete logarithm computation", Cryptology ePrint Archive, Report 2016/961, 2016. https://eprint.iacr.org/2016/961.pdf.

[H2019] Hammell, J., "Re: [lamps] WG Last Call for draft-ietf-lamps-cms-mix-with-psk", <https://mailarchive.ietf.org/arch/msg/spasm/_6d_4jp3sOprAnbU2fp_yp_-6-k>, 27 May 2019.

[IANA-MOD] https://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml#security-smime-0.

[IANA-SMIME] https://www.iana.org/assignments/smi-numbers/smi-

numbers.xhtml#security-smime.

[IANA-ORI] https://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml#security-smime-TBD1.

[NAS2019] National Academies of Sciences, Engineering, and Medicine, "Quantum Computing: Progress and Prospects", The National Academies Press, DOI 10.17226/25196, 2019.

[S1994] Shor, P., "Algorithms for Quantum Computation: Discrete Logarithms and Factoring", Proceedings of the 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124-134.

[RFC2631] Rescorla, E., "Diffie-Hellman Key Agreement Method", RFC 2631, June 1999.

[RFC4086] D. Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", RFC 4086, June 2005.

[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.

[RFC5753] Turner, S., and D. Brown, "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)", RFC 5753, January 2010.

[RFC5869] Krawczyk, H., and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

[RFC5911] Hoffman, P., and J. Schaad, "New ASN.1 Modules for Cryptographic Message Syntax (CMS) and S/MIME", RFC 5911, June 2010.

[RFC5912] Hoffman, P., and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, June 2010.

[RFC6268] Schaad, J., S. Turner, "Additional New ASN.1 Modules for the Cryptographic Message Syntax (CMS) and the Public Key Infrastructure Using X.509 (PKIX)", RFC 6268, July 2011.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2",

RFC 8017, November 2016.

Appendix A: Key Transport with PSK Example

This example shows the establishment of an AES-256 content-encryption key using:

- a pre-shared key of 256 bits;
- key transport using RSA PKCS#1 v1.5 with a 3072-bit key;
- key derivation using HKDF with SHA-384; and
- key wrap using AES-256-KEYWRAP.

In real-world use, the originator would encrypt the key-derivation key in their own RSA public key as well as the recipient's public key. This is omitted in an attempt to simplify the example.

A.1. Originator Processing Example

The pre-shared key known to Alice and Bob:

```
c244cdd11a0d1f39d9b61282770244fb0f6befb91ab7f96cb05213365cf95b15
```

The identifier assigned to the pre-shared key is:

```
ptf-kmc:13614122112
```

Alice obtains Bob's public key:

```
-----BEGIN PUBLIC KEY-----
MIIBojANBgkqhkiG9w0BAQEFAAOCAY8AMIIBigKCAYEA3ocW14cxncPJ47fnEjBZ
AyfC2lqapL3ET4jvV6C7gGeVrRQxWPDwl+cFYBBR2ej3j3/0ecDmu+XuVi2+s5JH
Keeza+itfuhsz3yifgeEpeK8T+SusHhn20/NBLhYKbh3kiAcCgQ56dpDrDvDcLqq
vS3jg/VO+OPnZbofoHOOevt8Q/roahJe1PlIyQ4udWB8zZezJ4mLLfbOA9YVaYXx
2AHHZJevo3nmRnlgJXo6mE00E/6qkhjDHKSMdl2WG6mO9TCDZc9qY3cAJDU6Ir0v
SH7qUl8/vN13y4UOFkn8hM4kmZ6bJqbZt5NbjHtY4uQ0VMW3RyESzhrO02mrp39a
uLNnH3EXdXaV1tk75H3qC7zJaeGWMJyQfOE3YfEGRKn8fxubji716D8UecAxAzFy
FL6m1JiOyV5acAiOpxN14qRYZdHnXOM9DqGIGpoeY1UuD4Mo05osOqOUpBJHA9fS
whSZG7VNf+vgNWTLNYSYLI04KiMdulnvU6ds+QPz+KKtAgMBAAE=
-----END PUBLIC KEY-----
```

Bob's RSA public key has the following key identifier:

```
9eeb67c9b95a74d44d2f16396680e801b5cba49c
```

Alice randomly generates a content-encryption key:

```
c8adc30f4a3e20ac420caa76a68f5787c02ab42afea20d19672fd963a5338e83
```

Alice randomly generates a key-derivation key:

```
df85af9e3cebffde6e9b9d24263db31114d0a8e33a0d50e05eb64578ccde81eb
```

Alice encrypts the key-derivation key in Bob's public key:

```
4e6200431ed95e0e28f7288dba56d4b90e75959e068884664c43368f3d978f3d
8179e5837e3c27bf8dc1f6e2827b9ede969be77417516de07d90e37c560add01
48deb0c9178088ccb72c068d8a9076b6a5e7ecc9093e30fdeaecc9e138d80626
74fcf685f3082b910839551cd8741beedeee6e87c08ff84f03ba87118730cdf7
667002316f1a29a6cc596c7ddf95a51e398927d1916bf27929945de080fc7c80
6af6281aed6492acffa4ef1b4f53e67fca9a417db2350a2277d586ee3cabefd3
b4a44f04d3c6803d54fe9a7159210dabedda9a94f310d303331da51c0218d92a
2efb003792259195a9fd4cc403af613fdf1a6265ea70bf702fd1c6f734264c9a
59196e8e8fd657fa028e272ef741eb7711fd5b3f4ea7da9c33df66bf487da710
1c9bbfddaf1c073900a3ea99da513d8aa32605db07dc1c47504cab30c9304a85
d87377f603ec3df4056ddcf3d756fb7ed98254421a4ae151f17ad4e28c5ea077
63358dfb1ef5f73435f337b21a38c1a3fa697a530dd97e462f6b5fb2052a2d53
```

Alice produces a 256-bit key-encryption key with HKDF using SHA-384; the secret value is the key-derivation key; the 'info' is the DER-encoded CMSORIforPSKOtherInfo structure with the following values:

```
    0   56: SEQUENCE {
    2   32:   OCTET STRING
          :     C2 44 CD D1 1A 0D 1F 39 D9 B6 12 82 77 02 44 FB
          :     0F 6B EF B9 1A B7 F9 6C B0 52 13 36 5C F9 5B 15
   36    1:   ENUMERATED 5
   39   11:   SEQUENCE {
   41    9:     OBJECT IDENTIFIER aes256-wrap
          :       { 2 16 840 1 101 3 4 1 45 }
          :     }
   52    1:   INTEGER 32
   55    1:   INTEGER 32
          :   }
```

The DER encoding of CMSORIforPSKOtherInfo produces 58 octets:

```
30380420c244cdd11a0d1f39d9b61282770244fb0f6befb91ab7f96cb0521336
5cf95b150a0105300b0609608648016503040 12d020120020120
```

The HKDF output is 256 bits:

```
a14d87451dfd11d83cd54ffe2bd38c49a2adfed3ac49f1d3e62bbdc64ae43b32
```

Alice uses AES-KEY-WRAP to encrypt the 256-bit content-encryption key with the key-encryption key:

```
ae4ea1d99e78fcdcea12d9f10d991ac71502939ee0c30ebdcc97dd1fc5ba3566
c83d0dd5d1b4faa5
```

Alice encrypts the content using AES-256-GCM with the content-encryption key. The 12-octet nonce used is:

```
cafebabefacedbaddecaf888
```

The content plaintext is:

```
48656c6c6f2c20776f726c6421
```

The resulting ciphertext is:

```
9af2d16f21547fcefed9b3ef2d
```

The resulting 12-octet authentication tag is:

```
a0e5925cc184e0172463c44c
```

A.2. ContentInfo and AuthEnvelopedData

Alice encodes the AuthEnvelopedData and the ContentInfo, and sends the result to Bob. The resulting structure is:

```
    0  650: SEQUENCE {
    4   11:  OBJECT IDENTIFIER authEnvelopedData
          :   { 1 2 840 113549 1 9 16 1 23 }
   17  633:  [0] {
   21  629:   SEQUENCE {
   25    1:    INTEGER 0
   28  551:    SET {
   32  547:     [4] {
   36   11:      OBJECT IDENTIFIER ** Placeholder **
          :       { 1 2 840 113549 1 9 16 TBD 1 }
   49  530:      SEQUENCE {
   53    1:       INTEGER 0
   56   19:       OCTET STRING 'ptf-kmc:13614122112'
   77   13:       SEQUENCE {
   79   11:        OBJECT IDENTIFIER ** Placeholder **
          :         { 1 2 840 113549 1 9 16 3 TBD }
          :        }
   92   11:       SEQUENCE {
   94    9:        OBJECT IDENTIFIER aes256-wrap
          :         { 2 16 840 1 101 3 4 1 45 }
          :        }
  105  432:       SEQUENCE {
  109  428:        SEQUENCE {
  113    1:         INTEGER 2
  116   20:         [0]
          :          9E EB 67 C9 B9 5A 74 D4 4D 2F 16 39 66 80 E8 01
          :          B5 CB A4 9C
  138   13:         SEQUENCE {
  140    9:          OBJECT IDENTIFIER rsaEncryption
          :           { 1 2 840 113549 1 1 1 }
  151    0:          NULL
          :          }
  153  384:         OCTET STRING
          :          18 09 D6 23 17 DF 2D 09 55 57 3B FE 75 95 EB 6A
          :          3D 57 84 6C 69 C1 49 0B F1 11 1A BB 40 0C D8 B5
          :          26 5F D3 62 4B E2 D8 E4 CA EC 6A 12 36 CA 38 E3
          :          A0 7D AA E0 5F A1 E3 BC 59 F3 AD A8 8D 95 A1 6B
          :          06 85 20 93 C7 C5 C0 05 62 ED DF 02 1D FE 68 7C
          :          18 A1 3A AB AA 59 92 30 6A 1B 92 73 D5 01 C6 5B
          :          FD 1E BB A9 B9 D2 7F 48 49 7F 3C 4F 3C 13 E3 2B
          :          2A 19 F1 7A CD BC 56 28 EF 7F CA 4F 69 6B 7E 92
          :          66 22 0D 13 B7 23 AD 41 9E 5E 98 2A 80 B7 6C 77
          :          FF 9B 76 B1 04 BA 30 6D 4B 4D F9 25 57 E0 7F 0E
```

```
          :          95 9A 43 6D 14 D5 72 3F AA 8F 66 35 40 D0 E3 71
          :          4B 7F 20 9D ED 67 EA 33 79 CD AB 84 16 72 07 D2
          :          AC 8D 3A DA 12 43 B7 2F 3A CF 91 3E F1 D9 58 20
          :          6D F2 9C 09 E1 EC D2 0B 82 BE 5D 69 77 6F FE F7
          :          EB F6 31 C0 D9 B7 15 BF D0 24 F3 05 1F FF 48 76
          :          1D 73 17 19 2C 38 C6 D5 86 BD 67 82 2D B2 61 AA
          :          08 C7 E4 37 34 D1 2D E0 51 32 15 4A AC 6B 2B 28
          :          5B CD FA 7C 65 89 2F A2 63 DB AB 64 88 43 CC 66
          :          27 84 29 AC 15 5F 3B 9E 5B DF 99 AE 4F 1B B2 BC
          :          19 6C 17 A1 99 A5 CF F7 80 32 11 88 F1 9D B3 6F
          :          4B 16 5F 3F 03 F7 D2 04 3D DE 5F 30 CD 8B BB 3A
          :          38 DA 9D EC 16 6C 36 4F 8B 7E 99 AA 99 FB 42 D6
          :          1A FF 3C 85 D7 A2 30 74 2C D3 AA F7 18 2A 25 3C
          :          B5 02 C4 17 62 21 97 F1 E9 81 83 D0 4E BF 5B 5D
          :         }
          :        }
      541  40:       OCTET STRING
          :          AE 4E A1 D9 9E 78 FC DC EA 12 D9 F1 0D 99 1A C7
          :          15 02 93 9E E0 C3 0E BD CC 97 DD 1F C5 BA 35 66
          :          C8 3D 0D D5 D1 B4 FA A5
          :       }
          :      }
          :     }
      583  55:    SEQUENCE {
      585   9:     OBJECT IDENTIFIER data { 1 2 840 113549 1 7 1 }
      596  27:     SEQUENCE {
      598   9:      OBJECT IDENTIFIER aes256-GCM
          :       { 2 16 840 1 101 3 4 1 46 }
      609  14:      SEQUENCE {
      611  12:       OCTET STRING CA FE BA BE FA CE DB AD DE CA F8 88
          :       }
          :      }
      625  13:     [0] 9A F2 D1 6F 21 54 7F CE FE D9 B3 EF 2D
          :     }
      640  12:    OCTET STRING A0 E5 92 5C C1 84 E0 17 24 63 C4 4C
          :    }
          :   }
          :  }
```

A.3. Recipient Processing Example

Bob's private key:

```
-----BEGIN RSA PRIVATE KEY-----
MIIG5AIBAAKCAYEA3ocW14cxncPJ47fnEjBZAyfC2lqapL3ET4jvV6C7gGeVrRQx
WPDwl+cFYBBR2ej3j3/0ecDmu+XuVi2+s5JHKeeza+itfuhsz3yifgeEpeK8T+Su
sHhn20/NBLhYKbh3kiAcCgQ56dpDrDvDcLqqvS3jg/VO+OPnZbofoHOOevt8Q/ro
ahJe1PlIyQ4udWB8zZezJ4mLLfbOA9YVaYXx2AHHZJevo3nmRnlgJXo6mE00E/6q
khjDHKSMdl2WG6mO9TCDZc9qY3cAJDU6Ir0vSH7qUl8/vN13y4UOFkn8hM4kmZ6b
JqbZt5NbjHtY4uQ0VMW3RyESzhrO02mrp39auLNnH3EXdXaV1tk75H3qC7zJaeGW
MJyQfOE3YfEGRKn8fxubji716D8UecAxAzFyFL6m1JiOyV5acAiOpxN14qRYZdHn
XOM9DqGIGpoeY1UuD4Mo05osOqOUpBJHA9fSwhSZG7VNf+vgNWTLNYSYLI04KiMd
ulnvU6ds+QPz+KKtAgMBAAECggGATFfkSkUjjJCjLvDk4aScpSx6+Rakf2hrdS3x
jwqhyUfAXgTTeUQQBs1HVtHCgxQd+qlXYn3/qu8TeZVwG4NPztyi/Z5yB1wOGJEV
3k8N/ytul6pJFFn6p48VM01bUdTrkMJbXERe6g/rr6dBQeeItCaOK7N5SIJH3Oqh
9xYuB5tH4rquCdYLmt17Tx8CaVqU9qPY3vOdQEOwIjjMV8uQUR8rHSO9KkSj8AGs
Lq9kcuPpvgJc2oqMRcNePS2WVh8xPFktRLLRazgLP8STHAtjT6SlJ2UzkUqfDHGK
q/BoXxBDu6L1VDwdnIS5HXtL54ElcXWsoOyKF8/ilmhRUIUWRZFmlS1ok8IC5IgX
UdL9rJVZFTRLyAwmcCEvRM1asbBrhyEyshSOuN5nHJi2WVJ+wSHijeKl1qeLlpMk
HrdIYBq4Nz7/zXmiQphpAy+yQeanhP8O4O6C8e7RwKdpxe44su4Z8fEgA5yQx0u7
8yR1EhGKydX5bhBLR5Cm1VM7rT2BAoHBAP/+e5gZLNf/ECtEBZjeiJ0VshszOoUq
haUQPA+9Bx9pytsoKm5oQhB7QDaxAvrn8/FUW2aAkaXsaj9F+/q30AYSQtExai9J
fdKKook3oimN8/yNRsKmhfjGOj8hd4+GjX0qoMSBCEVdT+bAjjry8wgQrqReuZnu
oXU85dmb3jvv0uIczIKvTIeyjXE5afjQIJLmZFXsBm09BG87Ia5EFUKly96BOMJh
/QWEzuYYXDqOFfzQtkAefXNFW21Kz4Hw2QKBwQDeiGh4lxCGTjECvG7fauMGlu+q
DSdYyMHif6t6mx57eS16EjvOrlXKItYhIyzW8Kw0rf/CSB2j8ig1GkMLTOgrGIJ1
0322o50FOr5oOmZPueeR4pOyAP0fgQ8DD1L3JBpY68/8MhYbsizVrR+Ar4jM0f96
W2bF5Xj3h+fQTDMkx6VrCCQ6miRmBUzH+ZPs5n/lYOzAYrqiKOanaiHy4mjRvlsy
mjZ6z5CG8sISqcLQ/k3Qli5pOY/v0rdBjgwAW/UCgcEAqGVYGjKdXCzuDvf9EpV4
mpTWB6yIV2ckaPOn/tZi5BgsmEPwvZYZt0vMbu28Px7sSpkqUuBKbzJ4pcy8uC3I
SuYiTAhMiHS4rxIBX3BYXSuDD2RD4vG1+XM0h6jVRHXHh0nOXdVfgnmigPGz3jVJ
B8oph/jD8O2YCk4YCTDOXPEi8Rjusxzro+whvRR+kG0gsGGcKSVNCPj1fNISEte4
gJId7O1mUAAzeDjn/VaS/PXQovEMolssPPKn9NocbKbpAoHBAJnFHJunl22W/lrr
ppmPnIzjI30YVcYOA5vlqLKyGaAsnfYqP1WUNgfVhq2jRsrHx9cnHQI9Hu442PvI
x+c5H30YFJ4ipE3eRRRmAUi4ghY5WgD+1hw8fqyUW7E7l5LbSbGEUVXtrkU5G64T
UR91LEyMF8OPATdiV/KD4PWYkgaqRm3tVEuCVACDTQkqNsOOi3YPQcm270w6gxfQ
SOEy/kdhCFexJFA8uZvmh6Cp2crczxyBilR/yCxqKOONqlFdOQKBwFbJk5eHPjJz
AYueKMQESPGYCrwIqxgZGCxaqeVArHvKsEDx5whI6JWoFYVkFA8F0MyhukoEb/2x
2qB5T88Dg3EbqjTiLg3qxrWJ2OxtUo8pBP2I2wbl2NOwzcbrlYhzEZ8bJyxZu5i1
sYILC8PJ4Qzw6jS4Qpm4y1WHz8e/ElW6VyfmljZYA7f9WMntdfeQVqCVzNTvKn6f
hg6GSpJTzp4LV3ougi9nQuWXZF2wInsXkLYpsiMbL6Fz34RwohJtYA==
-----END RSA PRIVATE KEY-----
```

Bob decrypts the key-derivation key with his RSA private key:

```
df85af9e3cebffde6e9b9d24263db31114d0a8e33a0d50e05eb64578ccde81eb
```

Bob produces a 256-bit key-encryption key with HKDF using SHA-384; the secret value is the key-derivation key; the 'info' is the DER-encoded CMSORIforPSKOtherInfo structure with the same values as shown in A.1. The HKDF output is 256 bits:

a14d87451dfd11d83cd54ffe2bd38c49a2adfed3ac49f1d3e62bbdc64ae43b32

Bob uses AES-KEY-WRAP to decrypt the content-encryption key with the key-encryption key; the content-encryption key is:

c8adc30f4a3e20ac420caa76a68f5787c02ab42afea20d19672fd963a5338e83

Bob decrypts the content using AES-256-GCM with the content-encryption key, and checks the received authentication tag. The 12-octet nonce used is:

cafebabefacedbaddecaf888

The 12-octet authentication tag is:

a0e5925cc184e0172463c44c

The received ciphertext content is:

9af2d16f21547fcefed9b3ef2d

The resulting plaintext content is:

48656c6c6f2c20776f726c6421

Appendix B: Key Agreement with PSK Example

This example shows the establishment of an AES-256 content-encryption key using:

- a pre-shared key of 256 bits;
- key agreement using ECDH on curve P-384 and X9.63 KDF with SHA-384;
- key derivation using HKDF with SHA-384; and
- key wrap using AES-256-KEYWRAP.

In real-world use, the originator would treat themselves as an additional recipient by performing key agreement with their own static public key and the ephemeral private key generated for this message. This is omitted in an attempt to simplify the example.

B.1. Originator Processing Example

The pre-shared key known to Alice and Bob:

4aa53cbf500850dd583a5d9821605c6fa228fb5917f87c1c078660214e2d83e4

The identifier assigned to the pre-shared key is:

ptf-kmc:216840110121

Alice randomly generates a content-encryption key:
937b1219a64d57ad81c05cc86075e86017848c824d4e85800c731c5b7b091033

Alice obtains Bob's static ECDH public key:

```
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAEScGPBO9nmUwGrgbGEoFY9HR/bCo0WyeY
/dePQVrwZmwN2yMJmO2d1kWCvLTz8U7atinxyIRe9CV54yau1KWU/wbkhPDnzuSM
YkcpxMGo32z3JetEloW5aFOja13vv/W5
-----END PUBLIC KEY-----
```

It has a key identifier of:
e8218b98b8b7d86b5e9ebdc8aeb8c4ecdc05c529

Alice generates an ephemeral ECDH key pair on the same curve:

```
-----BEGIN EC PRIVATE KEY-----
MIGkAgEBBDCMiWLG44ik+L8cYVvJrQdLcFA+PwlgRF+Wt1Ab25qUh8OB7OePWjxp
/b8P6IOuI6GgBwYFK4EEACKhZANiAAQ5G0EmJk/2ks8sXY1kzbuG3Uu3ttWwQRXA
LFDJICjvYfr+yTpOQVkchm88FAh9MEkw4NKctokKNgpsqXyrT3DtOg76oIYENpPb
GE5lJdjPx9sBsZQdABwlsU0Zb7P/7i8=
-----END EC PRIVATE KEY-----
```

Alice computes a shared secret, called Z, using the Bob's static ECDH public key and her ephemeral ECDH private key; Z is:
3f015ed0ff4b99523a95157bbe77e9cc0ee52fcffeb7e41eac79d1c11b6cc556
19cf8807e6d800c2de40240fe0e26adc

Alice computes the pairwise key-encryption key, called KEK1, from Z using the X9.63 KDF with the ECC-CMS-SharedInfo structure with the following values:

```
    0   21: SEQUENCE {
    2   11:   SEQUENCE {
    4    9:     OBJECT IDENTIFIER aes256-wrap
          :       { 2 16 840 1 101 3 4 1 45 }
          :     }
   15    6:   [2] {
   17    4:     OCTET STRING 00 00 00 20
          :     }
          :   }
```

The DER encoding of ECC-CMS-SharedInfo produces 23 octets:
3015300b060960864801650304012da206040400000020

The X9.63 KDF output is the 256-bit KEK1:
27dc25ddb0b425f7a968ceada80a8f73c6ccaab115baafcce4a22a45d6b8f3da

Alice produces the 256-bit KEK2 with HKDF using SHA-384; the secret value is KEK1; the 'info' is the DER-encoded CMSORIforPSKOtherInfo structure with the following values:

```
    0   56: SEQUENCE {
    2   32:   OCTET STRING
          :     4A A5 3C BF 50 08 50 DD 58 3A 5D 98 21 60 5C 6F
          :     A2 28 FB 59 17 F8 7C 1C 07 86 60 21 4E 2D 83 E4
   36    1:   ENUMERATED 10
   39   11:   SEQUENCE {
   41    9:     OBJECT IDENTIFIER aes256-wrap
          :       { 2 16 840 1 101 3 4 1 45 }
          :     }
   52    1:   INTEGER 32
   55    1:   INTEGER 32
          :   }
```

The DER encoding of CMSORIforPSKOtherInfo produces 58 octets:

```
   303804204aa53cbf500850dd583a5d9821605c6fa228fb5917f87c1c07866021
   4e2d83e40a010a300b060960864801650304012d020120020120
```

The HKDF output is the 256-bit KEK2:

```
   7de693ee30ae22b5f8f6cd026c2164103f4e1430f1ab135dc1fb98954f9830bb
```

Alice uses AES-KEY-WRAP to encrypt the content-encryption key with the KEK2; the wrapped key is:

```
   229fe0b45e40003e7d8244ec1b7e7ffb2c8dca16c36f5737222553a71263a92b
   de08866a602d63f4
```

Alice encrypts the content using AES-256-GCM with the content-encryption key. The 12-octet nonce used is:

```
   dbaddecaf888cafebabeface
```

The plaintext is:

```
   48656c6c6f2c20776f726c6421
```

The resulting ciphertext is:

```
   fc6d6f823e3ed2d209d0c6ffcf
```

The resulting 12-octet authentication tag is:

```
   550260c42e5b29719426c1ff
```

B.2. ContentInfo and AuthEnvelopedData

Alice encodes the AuthEnvelopedData and the ContentInfo, and sends the result to Bob. The resulting structure is:

```
    0  327: SEQUENCE {
    4   11:  OBJECT IDENTIFIER authEnvelopedData
          :   { 1 2 840 113549 1 9 16 1 23 }
   17  310:  [0] {
   21  306:   SEQUENCE {
   25    1:    INTEGER 0
   28  229:    SET {
   31  226:     [4] {
   34   11:      OBJECT IDENTIFIER ** Placeholder **
          :       { 1 2 840 113549 1 9 16 TBD 2 }
   47  210:      SEQUENCE {
   50    1:       INTEGER 0
   53   20:       OCTET STRING 'ptf-kmc:216840110121'
   75   85:       [0] {
   77   83:        [1] {
   79   19:         SEQUENCE {
   81    6:          OBJECT IDENTIFIER
          :           dhSinglePass-stdDH-sha256kdf-scheme
          :           { 1 3 132 1 11 1 }
   89    9:          OBJECT IDENTIFIER aes256-wrap
          :           { 2 16 840 1 101 3 4 1 45 }
          :          }
  100   60:         BIT STRING, encapsulates {
  103   57:          OCTET STRING
          :           1B 41 26 26 4F F6 92 CF 2C 5D 8D 64 CD BB 86 DD
          :           4B B7 B6 D5 B0 41 15 C0 2C 50 C9 20 28 EF 61 FA
          :           FE C9 3A 4E 41 59 1C 86 6F 3C 14 08 7D 30 49 30
          :           E0 D2 9C B6 89 0A 36 0A 6C
          :          }
          :         }
          :        }
  162   13:       SEQUENCE {
  164   11:        OBJECT IDENTIFIER ** Placeholder **
          :         { 1 2 840 113549 1 9 16 3 TBD }
          :        }
  177   11:       SEQUENCE {
  179    9:        OBJECT IDENTIFIER aes256-wrap
          :         { 2 16 840 1 101 3 4 1 45 }
          :        }
  190   68:       SEQUENCE {
  192   66:        SEQUENCE {
  194   22:         [0] {
  196   20:          OCTET STRING
```

```
          :          E8 21 8B 98 B8 B7 D8 6B 5E 9E BD C8 AE B8 C4 EC
          :          DC 05 C5 29
          :          }
  218   40:         OCTET STRING
          :         22 9F E0 B4 5E 40 00 3E 7D 82 44 EC 1B 7E 7F FB
          :         2C 8D CA 16 C3 6F 57 37 22 25 53 A7 12 63 A9 2B
          :         DE 08 86 6A 60 2D 63 F4
          :         }
          :        }
          :       }
          :      }
          :     }
  260   55:    SEQUENCE {
  262    9:     OBJECT IDENTIFIER data { 1 2 840 113549 1 7 1 }
  273   27:     SEQUENCE {
  275    9:      OBJECT IDENTIFIER aes256-GCM
          :       { 2 16 840 1 101 3 4 1 46 }
  286   14:      SEQUENCE {
  288   12:       OCTET STRING DB AD DE CA F8 88 CA FE BA BE FA CE
          :       }
          :      }
  302   13:     [0] FC 6D 6F 82 3E 3E D2 D2 09 D0 C6 FF CF
          :     }
  317   12:    OCTET STRING 55 02 60 C4 2E 5B 29 71 94 26 C1 FF
          :    }
          :   }
          :  }
```

B.3. Recipient Processing Example

Bob obtains Alice's ephemeral ECDH public key from the message:

```
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAEORtBJiZP9pLPLF2NZM27ht1Lt7bVsEEV
wCxQySAo72H6/sk6TkFZHIZvPBQIfTBJMODSnLaJCjYKbKl8q09w7ToO+qCGBDaT
2xhOZSXYz8fbAbGUHQAcJbFNGW+z/+4v
-----END PUBLIC KEY-----
```

Bob's static ECDH private key:

```
-----BEGIN EC PRIVATE KEY-----
MIGkAgEBBDAnJ4hB+tTUN9X03/W0RsrYy+qcptlRSYkhaDIsQYPXfTU0ugjJEmRk
NTPj4y1IRjegBwYFK4EEACKhZANiAARJwY8E72eZTAauBsYSgVj0dH9sKjRbJ5j9
149BWvBmbA3bIwmY7Z3WRYK8tPPxTtq2KfHIhF70JXnjJq7UpZT/BuSE8OfO5Ixi
RynEwajfbPcl60SWhbloU6NrXe+/9bk=
-----END EC PRIVATE KEY-----
```

Bob computes a shared secret, called Z, using the Alice's ephemeral ECDH public key and his static ECDH private key; Z is:

3f015ed0ff4b99523a95157bbe77e9cc0ee52fcffeb7e41eac79d1c11b6cc556
19cf8807e6d800c2de40240fe0e26adc

Bob computes the pairwise key-encryption key, called KEK1, from Z using the X9.63 KDF with the ECC-CMS-SharedInfo structure with the values shown in B.1. The X9.63 KDF output is the 256-bit KEK1:

27dc25ddb0b425f7a968ceada80a8f73c6ccaab115baafcce4a22a45d6b8f3da

Bob produces the 256-bit KEK2 with HKDF using SHA-384; the secret value is KEK1; the 'info' is the DER-encoded CMSORIforPSKOtherInfo structure with the values shown in B.1. The HKDF output is the 256-bit KEK2:

7de693ee30ae22b5f8f6cd026c2164103f4e1430f1ab135dc1fb98954f9830bb

Bob uses AES-KEY-WRAP to decrypt the content-encryption key with the KEK2; the content-encryption key is:

937b1219a64d57ad81c05cc86075e86017848c824d4e85800c731c5b7b091033

Bob decrypts the content using AES-256-GCM with the content-encryption key, and checks the received authentication tag. The 12-octet nonce used is:

dbaddecaf888cafebabeface

The 12-octet authentication tag is:

550260c42e5b29719426c1ff

The received ciphertext content is:

fc6d6f823e3ed2d209d0c6ffcf

The resulting plaintext content is:

48656c6c6f2c20776f726c6421

Acknowledgements

Many thanks to Roman Danyliw, Burt Kaliski, Panos Kampanakis, Jim Schaad, Robert Sparks, Sean Turner, and Daniel Van Geest for their review and insightful comments. They have greatly improved the design, clarity, and implementation guidance.

Author's Address

Russ Housley
Vigil Security, LLC
516 Dranesville Road
Herndon, VA 20170
USA
EMail: housley@vigilsec.com