RFC 8696
Using Pre-Shared Key (PSK) in the Cryptographic Message Syntax (CMS)

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. 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 is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8696.

Copyright Notice
Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
Table of Contents

1. Introduction
   1.1. Terminology
   1.2. ASN.1
   1.3. Version Numbers

2. Overview

3. keyTransPSK

4. keyAgreePSK

5. Key Derivation

6. ASN.1 Module

7. Security Considerations

8. Privacy Considerations

9. IANA Considerations

10. References
   10.1. Normative References
   10.2. Informative References

Appendix A. Key Transport with PSK Example
   A.1. Originator Processing Example
   A.2. ContentInfo and AuthEnvelopedData
   A.3. Recipient Processing Example

Appendix B. Key Agreement with PSK Example
   B.1. Originator Processing Example
   B.2. ContentInfo and AuthEnvelopedData
   B.3. Recipient Processing Example

Acknowledgements

Author's Address
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 (ECDH) [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 (ECDH) [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 an HMAC-based key derivation function (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 classic public key algorithms, 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 offers 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; 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. It is best if each PSK has a unique identifier; however, if a recipient has more than one PSK with the same identifier, the recipient can try each of them in turn. A PSK is expected to be used with many messages, with a lifetime of weeks or months.
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 KDF is used to mix the 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 KDF is used to mix the 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 is 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. This pairwise key-encryption key 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:

```plaintext
id-ori OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) 13 }
id-ori-keyTransPSK OBJECT IDENTIFIER ::= { id-ori 1 }
```

The KeyTransPSKRecipientInfo type is:

```plaintext
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. That is, the encryptedKey field of KeyTransRecipientInfo contains the key-derivation key instead of the content-encryption 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 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 type 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 KDFs internally employ a one-way hash function. When this is the case, the hash function that is used is indirectly indicated by the KeyDerivationAlgorithmIdentifier. HKDF [RFC5869] is one example of a KDF that makes use of a hash function.

Other KDFs internally employ an encryption algorithm. When this is the case, the encryption that is used is indirectly indicated by the KeyDerivationAlgorithmIdentifier. For example, AES-128-CMAC can be used for randomness extraction in a KDF as described in [NIST2018].

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 specification [RFC5869] 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. Many KDFs do not require a salt, and the KeyDerivationAlgorithmIdentifier assignments for HKDF [RFC8619] do not offer a parameter for a salt. If a particular KDF requires a salt, then the salt value is provided as a parameter of the KeyDerivationAlgorithmIdentifier.
- 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:
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.

An acceptable KDF MUST accept IKM, L, and info inputs; an acceptable KDF MAY also accept salt and other inputs. All of these inputs MUST influence the output of the KDF. If the KDF requires a salt or other inputs, then those inputs MUST be provided as parameters of the KeyDerivationAlgorithmIdentifier.

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 [RFC5912] and [RFC6268].
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(69) }

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) 13 }

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

7. Security Considerations

The security considerations related to the CMS enveloped-data content type in [RFC5652] and the security considerations related to the CMS authenticated-enveloped-data content type in [RFC5083] continue to apply.

Implementations of the key derivation function must compute the entire result, which, in this specification, is a key-encryption key, before outputting any portion of the result. The resulting key-encryption key must be protected. 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; this, in turn, may result in the disclosure of the content. Note that there are two key-encryption keys when a PSK with a key agreement algorithm is used, with similar consequences for the compromise of either one of these keys.
Implementations must protect the PSK, key transport private key, agreement private key, and key-derivation 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 the disclosure of all contents protected with that combination of keying material.

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 are provided. Implementers must ensure that the key-encryption algorithm is as strong or stronger than the content-encryption algorithm or content-authenticated-encryption algorithm.

The selection of the key-derivation function imposes an upper bound on the strength of the resulting key-encryption key. The strength of the selected key-derivation function should be at least as strong as the key-encryption algorithm that is selected. NIST SP 800-56C Revision 1 [NIST2018] offers advice on the security strength of several popular key-derivation functions.

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 be 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 inherently 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 or without a PSK.

Implementations must randomly generate key-derivation keys as well as 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 random numbers. The use of inadequate pseudorandom 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 cryptanalysis 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 that 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 does not really weaken the privacy situation. 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 Identifier (1.2.840.113549.1.9.16.0)" registry [IANA]:

```
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) 69 }
```

One new entry has been added in the "SMI Security for S/MIME Mail Security (1.2.840.113549.1.9.16)" registry [IANA]:

```
id-ori OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) 13 }
```

A new registry titled "SMI Security for S/MIME Other Recipient Info Identifiers (1.2.840.113549.1.9.16.13)" has been created.

Updates to the new registry are to be made according to the Specification Required policy as defined in [RFC8126]. The expert is expected to ensure that any new values identify additional RecipientInfo structures for use with the CMS. Object identifiers for other purposes should not be assigned in this arc.

Two assignments were made in the new "SMI Security for S/MIME Other Recipient Info Identifiers (1.2.840.113549.1.9.16.13)" registry [IANA] 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(13) 1 }
```

```
id-ori-keyAgreePSK OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-ori(13) 2 }
```

10. References

10.1. Normative References


10.2. Informative References


[NAS2019]


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, in hexadecimal, is:

```plaintext
C244cdd1a0d1f39d961282770244fb0f6befb91ab7f96cb05213365cf95b15
```

The identifier assigned to the pre-shared key is:

```plaintext
ptf-kmc:13614122112
```

Alice obtains Bob's public key:

```plaintext
-----BEGIN PUBLIC KEY-----
MIIBANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAYEAgcICoCAQoQ42NgQ1GL
-----END PUBLIC KEY-----
```

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

```plaintext
9eeb67c9b95a74d4d4df16369680e801b5c14a9c
```

Alice randomly generates a content-encryption key:

```plaintext
c8adc30f4a3e2ac420caa76a68f5787c02ab42afea20d16726fd963a538e83
```

Alice randomly generates a key-derivation key:

```plaintext
df85af9e3cbeffde6e9b924263db31114d0a8e33a0d50e05eb64578ccde81e8
```

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

```
 52 69 3f 12 14 0c 91 dea 2b 44 c0 b7 93 6f 68 b4 4e 95 c9 db 15 89 3b 7 c 7 a 3 b 8 e 36 43 94 0 b a 5 4 3 c 0 b f 1 0 e 1 b
 5b 7 e a e f d 0 7 4 a f 3 8 0 6 6 5 2 d 0 4 f b 9 5 1 5 3 5 4 3 3 4 6 f 3 6 c 2 1 2 5 d b a 6 f 4 d 2 3 d 2 b c 6 1 4 3 4 b
 5 e 3 6 f f 7 2 b 3 e a e f 5 7 c 6 c f 7 7 f 4 9 2 4 c 3 0 9 f 1 7 4 b 0 b 8 7 5 3 5 5 4 b 5 8 e d 3 3 a 8 8 4 8 d 7 0 7 a 9 8
 c 0 c 2 b 1 d d c f d 0 9 e 3 1 f e 2 1 3 c a 0 a 4 8 d d 1 5 7 b d 7 d 8 4 2 e 8 5 c c 7 6 f 7 7 7 1 0 d 5 8 e f e a a 0 5 2 5
 c 6 5 1 b c d 1 4 1 0 f b 4 7 5 3 4 e c a b a f 5 a b 7 d a a b e d 8 0 9 d 4 b 9 7 2 2 0 c a f d 4 9 2 9 c 5 f b 6 8 4 f 7 b
 b 8 6 9 2 e 6 7 0 3 2 3 f f 9 b 3 f c 1 1 d 6 c a c 5 1 d 4 a 3 5 5 9 3 1 7 3 d 4 8 f 8 0 c a 8 4 3 b 8 9 7 8 9 d 6 2 5 e 7
 9 9 7 a d 6 7 4 d 2 5 a 2 a 7 d 1 6 5 a 5 f 3 9 b 3 c b 6 3 5 8 e 9 3 7 7 b d b 0 2 a c 8 a 5 2 4 a c 9 3 1 1 3 c e d 9 a d
 c 6 8 2 6 3 0 2 5 c 0 b b 9 9 7 d 7 1 6 e 5 8 d 4 d 7 b 6 9 7 3 9 b f 5 9 1 f 3 e 7 1 c 7 6 7 8 d c 0 d f 9 6 f 3 d f 9 e 8 a
 a 5 7 3 8 f 4 9 c e 2 1 4 8 9 f 3 0 0 e 4 0 8 8 9 1 b 2 8 b 2 a b 6 d 9 0 5 1 b 3 c 2 e 6 8 e f a 2 f a 9 7 9 9 a 7 0 6 8 7 8
d 5 f 4 6 2 0 1 8 c 0 2 1 d 6 6 9 e d 6 4 9 f 9 a c d f 7 8 4 7 6 8 1 0 1 9 8 b f b 8 b d 4 1 f f e d c 5 8 5 e a f a 9 5 7 e
 e a 1 d 3 6 2 5 e 4 b e d 3 7 6 7 e 7 a e 4 9 7 1 8 a e e 2 f 5 7 5 c 4 0 1 a 2 6 a 2 9 9 4 1 d 8 5 a 5 b 7 e e 9 9 a c 3 6 4 7 1
```

The DER encoding of CMSORIforPSKOtherInfo produces 58 octets:

```
30 38 04 20 c2 44 cdd 11 a 0 d 1 f39 9 b6 12 82 7 7 0 2 44 fb
0 f 6 b e f b9 1 a b 7 7 f 9 6 c0 b 5 2 1 3 3 6 c f 9 b 1 5 0 a 8 1 0 3 0 b 0 6 0 6 0 8 6 4 0 3 0 4 0 1 2 0 2 1 0 2 0 2 0
```

The HKDF output is 256 bits:

```
f319e9cebb35f1c6a7a9709b8760b9d03e30e16c5b2b69347e9f00ca540a232
```

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

```
ea0947250fa66cd525595e52a69aaade88efcf1b0f108abe291060391b1cdf59
07f36b4667e45342
```

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

```plaintext
cafebabefacedbaddecaf888
```

The content plaintext is:

```plaintext
48656c6c6f2c26776f726c6421
```

The resulting ciphertext is:

```plaintext
9af2d16f21547fceed9b3ef2d
```

The resulting 12-octet authentication tag is:

```plaintext
a0e5925cc184e0172463c44c
```

### A.2. ContentInfo and AuthEnvelopedData

Alice encodes the AuthEnvelopedData and the ContentInfo and sends the result to Bob. The resulting structure is:
SEQUENCE {
  OBJECT IDENTIFIER authEnvelopedData (1 2 840 113549 1 9 16 1 23)
}

SEQUENCE {
  INTEGER 0
}

SET {
  OBJECT IDENTIFIER keyTransPSK (1 2 840 113549 1 9 16 13 1)
  SEQUENCE {
    INTEGER 0
    OCTET STRING 'ptf-kmc:13614122112'
    SEQUENCE {
      OBJECT IDENTIFIER hkdf-with-sha384 (1 2 840 113549 1 9 16 3 29)
    }
    SEQUENCE {
      OBJECT IDENTIFIER aes256-wrap (2 16 840 1 101 3 4 1 45)
    }
    SEQUENCE {
      OBJECT IDENTIFIER rsaEncryption (1 2 840 113549 1 1 1)
      NULL
    }
    OCTET STRING 
      52 69 3f 12 14 0c 91 de a2 b4 4c 0b 79 36 f6 be 
      46 de 8a 7b fa b0 72 bc b6 ec fd 56 b0 6a 9f 65 
      1b d4 66 9d 33 6a ef 7b 44 9e 5c d9 b1 51 89 3b 
      7c 7a 3b 8e 36 43 94 84 0b 6a 54 34 cb f1 0e 1b 
      56 78 ae fd 07 4f af 38 06 65 d2 64 fb 95 15 35 
      43 34 6f 36 c2 12 5d ba 6f 4d 23 d2 bc 61 43 4b 
      5e 36 ff 72 b3 ea fe 57 c6 cf 7f 74 92 4c 30 9f 
      17 4b 0b 87 53 55 4b 58 ed 33 a8 84 8d 70 7a 9b 
      c0 c2 b1 dd cf d8 9e 31 fe 21 3c a0 a4 8d d1 57 
      bd 7d 84 2e 85 cc 76 f7 77 18 d5 8e fe aa 05 25 
      c6 51 bc d1 41 0f b4 75 34 ec ab af 5a b7 da ab 
      ed 80 9d 4b 97 22 0c af 6d 49 29 c5 fb 68 4f 7b 
      b8 69 2e 6e 70 33 2f f9 b3 f7 c1 1d 6c ac 51 d4 
      a3 55 93 17 3d 48 f8 0c a8 43 b8 97 89 6d 25 e7 
      99 7a d7 d6 74 d2 5a 2a 7d 16 5a 5f 39 b3 cb 63 
      58 e9 37 bd b0 2a c8 a5 24 ac 93 11 3c ed d9 ad 
      c6 82 63 02 5c 0b b0 99 7d 71 6e 58 d4 d7 b6 97 
      39 bf 59 1f 3e 71 c7 67 8d c0 df 96 f3 df 9e 8a 
      a5 73 8f 4f 9c e2 14 89 f3 0e 0e 40 89 1b 20 b2 
      ab 6d 90 51 b3 c2 e6 8e fa 2f a9 79 9a 70 68 7b 
      d5 f4 62 81 8c 82 1d 66 69 ed 64 9f 9a cd f7 84 
      76 81 01 98 bf b8 bd 41 ff ed c5 85 ea fa 95 7e 
      ea 1d 36 25 e4 be d3 76 e7 ae 49 71 8a ee 2f 57 
      5c 48 1a 26 a2 99 41 d8 da 5b 7e e9 ac a3 64 71 

A.3. Recipient Processing Example

Bob’s private key is:
Bob decrypts the key-derivation key with his RSA private key:

```plaintext
df85af9e3cebfde6e9b9d2463db3114d0a8e33a0d50e05eb6457fcede81eb
```

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

```plaintext
-----BEGIN RSA PRIVATE KEY-----
MIIG5AIBAAKCAYEA3ocW14cxncjP47fnEjBZayfC2lqapL3ET4jvV6C7gGeVrQRx
WPdvw-cfYBBR2e3j3j-0ecDmu+XuVi2+s5JHkeeza+itfu/zs3y1geEpeK8T76+Su
shhn2/BNLhYbk3l1AcGQ56dpDrDvLcLqV3Sj6j/0+V0+PnZbofoM0O0e8T80Q/ro
ahJe1P1lyQ4udwB8zz1e趣味jMlLFlboA9YVYaXx2AHZHj3eovomnRlJGx0mE00E/eq
khiJbHKSmldL2G60mO9TCDz3cQy3o9cAJDU61r0svHTqUL8/vN13j40UFkXn8H4kmK6b
JqzbTsngHjt4yUq01WVWM3ySzhZr80rmp39auLhNn3EkdXaVtik75HqzC7JaeGW
M3jQf0E3YfEGRkNhxvubj7i1D8uecAaAzfYfLe6mlj1Oy5vSaci1OpXH1Naq4RZy7d
HnxQM9DqgipoJeYlUd4Mo05os0q0UppBHA595Sw5hSGZ7BNVf+vNWTLNY5LUnL40I4Md
ulVnU6ds+QpZ+KkTa6MBAAEcGgGATFfKsKuujj3cJLVdk4a4cSpSx6+Rak2hrdS3x
jwqhyUTaqTeUQ81s1HvTHCgqxdq+gLkXvn3/Yq8TeZvW4NpZtyj/Z35B1wO0G3E
3k8S/ytviL6p3F6n6p49VM01UbDrTmJbXeRe6g/rr6dBQee1tCaOK7N5SiJ03Qh
9xYuB5th4roquDCLynt17Tx8caVQ9pYq3vdOEwojJmv8uUQR8hSO9KKSj8A8Gs
Lq9kcuPvpvJc2oqMrc5cPsn25Wv8xPfktllRazglPL6STH4tjT6SI2jUzkUqDFDGK
q/boXXbU6bU1LVdwnI5NSHxtL54ElcWoswAoK8F8/1mlnHRIuUWRFbMs10k8IC5gX
Udl9J3VZFLRby4vdcCeVRai46bhrvYyshOu5NjH3i2WJj+wsHjikeKl1qelPPK
Hrd1yBq4nNz7/zXmiQophaPy+yQeast1P80406C8e7RwKdpexe44su428fEga5yQx0u7
8yR1ehGkY5xbhBLR5cm1V Mt7r2AoAHBAP/+/e5gZLNF+ECTEByjzj+Vsh0ssoUuQ
haUQPA+9b9pytosKm5oQhB7QoxAvnr8/FUw2aAkaXxaj9F/+/q30AYSxExa1J9
fdKkook3oimN8/yNrsKhmfjioj8hvd4+GjXq0oMSBCEvdT+ab7jyj8wRqReuZnu
oXU85dmb3jv0UiczTVIyejX5EafjQIIjLmZFXsBom9B9G8Ia5EFUyKly9B0MOm
/Qw6zuYYDqOQFzQtkAaeFxNFw21zK4HwQKBWqDe1gh4lCgtjECvF7fanaMGilq+
D5DyymHh6f66m56e16Ejv0rLXKityHyzW8KhWrf/C5B2jBi1GKMLTQOGIRJ1
6323o50fo5o0nZpueaRe04oPA0f0q8D61L3JBPY68/8MhYb1sZvRr+Ar4jM9F6
W2bF5Xj3v+hvQFDkx6vCrCC66mICmB2hZ+Zp5n5/Yl0AzYrykiQOanaHyjm4jRvLy
mjZ665C68S15cQkJ/Mqlyj5vO/v0rDbjgwAm/UCegAqOvVYjKdCZudVf99EvP4
mpTw6bgyV2ckaP0n/tzI5bgsmEwpwZyt8Mbu28P7s5PkoUuKBzbJ4pcy8uC3I
SuYtTAmH4sarXIr3Bh3xIV3D4vg1/XMO6hjVRKHash0nOdXvFmg1P9z3vJ3
B80oph/jB0802Vckyr4YCTD0PEx8RIjxuszr+whvsh1rk+Kg0gsGCkSVSCPjlnFNEt4e
Cj71071uMAaAeDyJua/PXQV0EmOl5sSSPknNocbkBpAhBjAMNfHJln12ZW/lrr
ppmnPnZj1I3o8vC5O1q6LkGaAsnfyQpWUNGfVhqj2rSjRHiXcHjQ9I0U442P1
+x5cHi30yF34iplE3eRRRmNjIAi4hy5Wgd+1h8wqyfquWE7L5l5bShGUEVUTxrkU5G64
URi9L1eMY0F0PATdI/KD4PKw1kagam3rTeUvCVADCTQkNsOio3YPQcm270w96gxf
SOEy/8kdhCfexJALu2Vmn6c2ccrzxyBiLR/yCqkK00bQfd0QKBWfjB5eElPEHjz
AyueKMQ6ESP6rCwLqXzg0XCaqexArvKsEdx5whiJ6WYoFyVkAF8nGmyuoe2/Bx
2qB5T88D3eBqzTl3q3xWj20xUo8bP83B2w2b20Wnzcbr1yhzbEz8bJxjuZ51i
sYI8L84Qz66S4Qpm4Wy1Whz8e/El6WvymJjMYAT9f9MntdeQfQvCvZkTnvKn6f
hgg6SpJtzp4LVuog1n9QWxKF2wZnsXkLYpS1mB6F3z4RwohJrTyA==
-----END RSA PRIVATE KEY-----
```
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:

```
cafebabeacedbaddecaf888
```

The 12-octet authentication tag is:

```
a0e5925cc184e0172463c44c
```

The received ciphertext content is:

```
9af2d16f21547fced9b3ef2d
```

The resulting plaintext content is:

```
48656c6c6f2c0776f726c6421
```

### 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, in hexadecimal, is:

```
4aa53cbf500850dd583a5d9821605c6fa228fb5917f87c078660214e2d83e4
```
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-----
MHYWwEAYHKoZIj0CAQYFK4EEACIDYgAE/ccGPB09nmUwGrgbmEoFY9HR/bCo8WyeY
/dePQVrwZmwN2yMJm02d1kWCvLTz8U7atinxyIRe9CV5ydau1KWU/wbkhPDnzuSM
YkcpxMG032zz7EloW5aF0ja13vv/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+PwlzRF+Wt1Ab25qUh80B70ePWjxp
/b8P6ioU66gBwYFK4FAACKhZANiAAQ5G0EmJk/2ks8sXY1kzbuG3U3ttWwQRX4
LFDJICjvYfr+yTpQQvkchm88FAh9MEkw4NKctokKNgpsqXyrT3D0g76oIYEnpPb
GE5lJdp9x9sB5d4ABwlsU0Zb7P/7i8=
-----END EC PRIVATE KEY-----
```

Alice computes a shared secret called "Z" using 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:
The DER encoding of ECC-CMS-SharedInfo produces 23 octets:

```plaintext
3015300b060960864801650304012da2060404000000020
```

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

```plaintext
27dc25dddb0b425f7a968ceada80a8f73c6ccaab115baafcce4a22a45d6b8f3da
```

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

```plaintext
0 56: SEQUENCE {
2 32: OCTET STRING
   4A A5 3C BF 50 08 50 DD 58 3A 5D 98 21 60 5C 6F
   A2 2B FB 50 91 FB 1C 07 50 08 60 21 5D 20 5E
36 1: ENUMERATED 10
39 11: SEQUENCE {
40 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:

```plaintext
303804204aa53cbf500850dd583a5d9821605c6fa228fb5917f87c1c07866021
4e2d83e40a010a300b060960864801650304012d020120020120
```

The HKDF output is the 256-bit KEK2:

```plaintext
7de693ee30ae22b5f8f6cd026c2164103f4e1430f1ab135dc1fb98954f9830bb
```

Alice uses AES-KEY-WRAP to encrypt the content-encryption key with the KEK2; the wrapped key is:
Alice encrypts the content using AES-256-GCM with the content-encryption key. The 12-octet nonce used is:

```
dbaddecaf888cfebabeface
```

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:
SEQUENCE {
  OBJECT IDENTIFIER authEnvelopedData (1 2 840 113549 1 9 16 1 23)
  [0] {
    INTEGER 0
    SET {
      [4] {
        OBJECT IDENTIFIER keyAgreePSK (1 2 840 113549 1 9 16 13 2)
        SEQUENCE {
          INTEGER 0
          OCTET STRING 'ptf-kmc:216840110121'
          [0] {
            [1] {
              SEQUENCE {
                OBJECT IDENTIFIER ecdhX963KDF-SHA256 (1 3 132 1 11 1)
                OBJECT IDENTIFIER aes256-wrap (2 16 840 1 101 3 4 1 45)
              }
            }
            SEQUENCE {
              OBJECT IDENTIFIER hkdf-with-sha384 (1 2 840 113549 1 9 16 3 29)
            }
            SEQUENCE {
              OBJECT IDENTIFIER aes256-wrap (2 16 840 1 101 3 4 1 45)
            }
          SEQUENCE {
            OCTET STRING encapsulates {
              OCTET STRING
              E8 21 8B 98 B8 B7 D8 6B 5E 9E BD C8 AE B8 C4 EC
              DC 05 C5 29
            }
            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
          }
          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 C9 2C 56 C9 28 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
        }
      }
      [0] {
        OCTET STRING
        E8 21 8B 98 B8 B7 D8 6B 5E 9E BD C8 AE B8 C4 EC
        DC 05 C5 29
      }
    }
  }
  SEQUENCE {
    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
  }
  OBJECT IDENTIFIER data (1 2 840 113549 1 7 1)
}
B.3. Recipient Processing Example

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

```
-----BEGIN PUBLIC KEY-----
MHYwEAYKoZIzj0CAQYFk4E4ACIDYGaeAEORtBJiZP9pLPLF2NZM27ht1Lt7bVsEEV
wCQySao72H6F/sk67kFZHIZvPBQIfTBJMODSnaJCjYkbKl8q09w7To0+qCGBDaT
2xhOZSYz8fbAbGUHQAqJbFNGW+/4v
-----END PUBLIC KEY-----
```

Bob's static ECDH private key is:

```
-----BEGIN EC PRIVATE KEY-----
MIGkAgEBBDAnJ4hb+tTUN9X03/W0RsrY+yqcptlRSYkhaDISqYFXfTU0ugjJEmRk
NTPj4y1IRjegBwYFK4EEEACKhZANiAARJwY88Z72eZTAAuBsYSgVj0dh9sJk7bJ5j9
149BWvBmbA3bIwmY7Z3WRYK8tPPxTq2KfIHf70JXnjJq7UpZT/BuSE80f05Ix1
RynEwajfbPc60SWhbdLoU6Nre+x/9bk=
-----END EC PRIVATE KEY-----
```

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

```
3f015ed8ff4b995b3a95157bbee7e9cc0ee52f0b7e41eac79d1c116c556
19cf8807e6d800c2e40240fe0e26adc
```

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

```
27dc25db0b425f7a968ceada80a8f73c6caab115baafcece4a22a45d6b8f3da
```
Bob produces the 256-bit KEK2 with HKDF using SHA-384; the secret value is KEK1; and the ‘info’ is the DER-encoded CMSORIforPSKOtherInfo structure with the values shown in Appendix 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:

```
550268c42e5b29719426c1ff
```

The received ciphertext content is:

```
fc6d6f823e3ed2d209d0c6ffcf
```

The resulting plaintext content is:

```
48656c6f6f2c28776f726c6421
```

**Acknowledgements**

Many thanks to Roman Danyliw, Ben Kaduk, 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
United States of America
Email: housley@vigilsec.com