Internet Engineering Task Force (IETF) Vigil Security
Intended Status: Proposed Standard
Expires: 18 March 2020 18 September 2019

Use of the HSS/LMS Hash-based Signature Algorithm in the Cryptographic Message Syntax (CMS)

<draft-ietf-lamps-cms-hash-sig-10>

Abstract

This document specifies the conventions for using the Hierarchical Signature System (HSS) / Leighton-Micali Signature (LMS) hash-based signature algorithm with the Cryptographic Message Syntax (CMS). In addition, the algorithm identifier and public key syntax are provided. The HSS/LMS algorithm is one form of hash-based digital signature; it is described in RFC 8554.

Table of Contents

1. Introduction

This document specifies the conventions for using the Hierarchical Signature System (HSS) / Leighton-Micali Signature (LMS) hash-based signature algorithm with the Cryptographic Message Syntax (CMS) [CMS] signed-data content type. The LMS system provides a one-time digital signature that is a variant of Merkle Tree Signatures (MTS). The HSS is built on top of the LMS system to efficiently scale for a larger numbers of signatures. The HSS/LMS algorithm is one form of hash-based digital signature, and it is described in [HASHSIG]. The HSS/LMS signature algorithm can only be used for a fixed number of signing operations with a given private key, and the number of signing operations depends upon the size of the tree. The HSS/LMS signature algorithm uses small public keys, and it has low computational cost; however, the signatures are quite large. The HSS/LMS private key can be very small when the signer is willing to perform additional computation at signing time; alternatively, the private key can consume additional memory and provide a faster signing time. The HSS/LMS signatures [HASHSIG] are currently defined to use exclusively SHA-256 [SHS].

1.1. ASN.1

CMS values are generated using ASN.1 [ASN1-B], using the Basic Encoding Rules (BER) and the Distinguished Encoding Rules (DER) [ASN1-E].

1.2. 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.

1.3. Motivation

Recent advances in cryptanalysis [BH2013] and progress in the development of quantum computers [NAS2019] pose a threat to widely deployed digital signature algorithms. As a result, there is a need to prepare for a day that cryptosystems such as RSA and DSA that depend on discrete logarithm and factoring cannot be depended upon.

If large-scale quantum computers are ever built, these computers will be able to break many of the public-key cryptosystems currently in use. A post-quantum cryptosystem [PQC] is a system that is secure against quantum computers that have more than a trivial number of quantum bits (qubits). It is open to conjecture when it will be feasible to build such computers; however, RSA, DSA, ECDSA, and EdDSA are all vulnerable if large-scale quantum computers come to pass.

Since the HSS/LMS signature algorithm does not depend on the difficulty of discrete logarithm or factoring, the HSS/LMS signature algorithm is considered to be post-quantum secure. One use of post-quantum secure signatures is the protection of software updates, perhaps using the format described in [FWPROT], to enable deployment of software that implements new cryptosystems.

2. HSS/LMS Hash-based Signature Algorithm Overview

Merkle Tree Signatures (MTS) are a method for signing a large but fixed number of messages. An MTS system depends on a one-time signature method and a collision-resistant hash function.

This specification makes use of the hash-based algorithm specified in [HASHSIG], which is the Leighton and Micali adaptation [LM] of the original Lamport-Diffie-Winternitz-Merkle one-time signature system [M1979][M1987][M1989a][M1989b].

As implied by the name, the hash-based signature algorithm depends on a collision-resistant hash function. The hash-based signature algorithm specified in [HASHSIG] uses only the SHA-256 one-way hash function [SHS], but it establishes an IANA registry [IANA-LMS] to permit the registration of additional one-way hash functions in the future.

2.1. Hierarchical Signature System (HSS)

The MTS system specified in [HASHSIG] uses a hierarchy of trees. The Hierarchical N-time Signature System (HSS) allows subordinate trees to be generated when needed by the signer. Otherwise, generation of the entire tree might take weeks or longer.

An HSS signature as specified in [HASHSIG] carries the number of signed public keys (Nspk), followed by that number of signed public keys, followed by the LMS signature as described in Section 2.2. The public key for the top-most LMS tree is the public key of the HSS system. The LMS private key in the parent tree signs the LMS public key in the child tree, and the LMS private key in the bottom-most tree signs the actual message. The signature over the public key and the signature over the actual message are LMS signatures as described in Section 2.2.

The elements of the HSS signature value for a stand-alone tree (a top tree with no children) can be summarized as:

```
u32str(0) ||
lms_signature  /* signature of message */
```

where, u32str() and || are used as defined in [HASHSIG].

The elements of the HSS signature value for a tree with Nspk signed public keys can be summarized as:

```
u32str(Nspk) ||
signed_public_key[0] ||
signed_public_key[1] ||
   ...
signed_public_key[Nspk-2] ||
signed_public_key[Nspk-1] ||
lms_signature  /* signature of message */
```

where, as defined in Section 3.3 of [HASHSIG], the signed_public_key structure contains the lms_signature over the public key followed by the public key itself. Note that Nspk is the number of levels in the hierarchy of trees minus 1.

2.2. Leighton-Micali Signature (LMS)

Each tree in the system specified in [HASHSIG] uses the Leighton-Micali Signature (LMS) system. LMS systems have two parameters. The first parameter is the height of the tree, h, which is the number of levels in the tree minus one. The [HASHSIG] specification supports five values for this parameter: h=5; h=10; h=15; h=20; and h=25. Note that there are 2^h leaves in the tree. The second parameter, m, is the number of bytes output by the hash function, and it is the amount of data associated with each node in the tree. The [HASHSIG] specification supports only the SHA-256 hash function [SHS], with m=32. As a result, the [HASHSIG] specification supports five tree sizes; they are identified as:

```
LMS_SHA256_M32_H5;
LMS_SHA256_M32_H10;
LMS_SHA256_M32_H15;
LMS_SHA256_M32_H20; and
LMS_SHA256_M32_H25.
```

The [HASHSIG] specification establishes an IANA registry [IANA-LMS] to permit the registration of additional hash functions and additional tree sizes in the future.

As specified in [HASHSIG], the LMS public key consists of four elements: the lms_algorithm_type from the list above, the otstype to identify the LM-OTS type as discussed in Section 2.3, the private key identifier (I) as described in Section 5.3 of [HASHSIG], and the m-byte string associated with the root node of the tree (T[1]).

The LMS public key can be summarized as:

```
u32str(lms_algorithm_type) || u32str(otstype) || I || T[1]
```

As specified in [HASHSIG], an LMS signature consists of four elements: the number of the leaf (q) associated with the LM-OTS signature, an LM-OTS signature as described in Section 2.3, a typecode indicating the particular LMS algorithm, and an array of values that is associated with the path through the tree from the leaf associated with the LM-OTS signature to the root. The array of values contains the siblings of the nodes on the path from the leaf to the root but does not contain the nodes on the path itself. The array for a tree with height h will have h values. The first value is the sibling of the leaf, the next value is the sibling of the parent of the leaf, and so on up the path to the root.

The four elements of the LMS signature value can be summarized as:

```
u32str(q) ||
ots_signature ||
u32str(type) ||
path[0] || path[1] || ... || path[h-1]
```

2.3. Leighton-Micali One-time Signature Algorithm (LM-OTS)

Merkle Tree Signatures (MTS) depend on a one-time signature method, and [HASHSIG] specifies the use of the LM-OTS, which has five parameters:

- n - The length in bytes of the hash function output. [HASHSIG] supports only SHA-256 [SHS], with n=32.

- H - A preimage-resistant hash function that accepts byte strings of any length, and returns an n-byte string.

- w - The width in bits of the Winternitz coefficients. [HASHSIG] supports four values for this parameter: w=1; w=2; w=4; and w=8.

- p - The number of n-byte string elements that make up the LM-OTS signature.

ls - The number of bits that are left-shifted in the final step of the checksum function, which is defined in Section 4.4 of [HASHSIG].

The values of p and ls are dependent on the choices of the parameters n and w, as described in Appendix B of [HASHSIG].

The [HASHSIG] specification supports four LM-OTS variants:

```
LMOTS_SHA256_N32_W1;
LMOTS_SHA256_N32_W2;
LMOTS_SHA256_N32_W4; and
LMOTS_SHA256_N32_W8.
```

The [HASHSIG] specification establishes an IANA registry [IANA-LMS] to permit the registration of additional variants in the future.

Signing involves the generation of C, an n-byte random value.

The LM-OTS signature value can be summarized as the identifier of the LM-OTS variant, the random value, and a sequence of hash values (y[0] through y[p-1]) that correspond to the elements of the public key as described in Section 4.5 of [HASHSIG]:

```
u32str(otstype) || C || y[0] || ... || y[p-1]
```

3. Algorithm Identifiers and Parameters

The algorithm identifier for an HSS/LMS hash-based signatures is:

```
id-alg-hss-lms-hashsig OBJECT IDENTIFIER ::= { iso(1)
    member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
    smime(16) alg(3) 17 }
```

When this object identifier is used for an HSS/LMS signature, the AlgorithmIdentifier parameters field MUST be absent (that is, the parameters are not present; the parameters are not set to NULL).

The signature value is a large OCTET STRING as described in Section 2 of this document. The signature format is designed for easy parsing. The HSS, LMS, and LMOTS component of the signature value each format include a counter and a type code that indirectly provide all of the information that is needed to parse the value during signature validation.

The signature value identifies the hash function used in the HSS/LMS tree. In [HASHSIG] uses only the SHA-256 hash function [SHS], but it establishes an IANA registry [IANA-LMS] to permit the registration of additional hash functions in the future.

4. HSS/LMS Public Key Identifier

The AlgorithmIdentifier for an HSS/LMS public key uses the id-alg-hss-lms-hashsig object identifier, and the parameters field MUST be absent.

When this AlgorithmIdentifier appears in the SubjectPublicKeyInfo field of an X.509 certificate [RFC5280], the certificate key usage extension MAY contain digitalSignature, nonRepudiation, keyCertSign, and cRLSign; however, it MUST NOT contain other values.

```
pk-HSS-LMS-HashSig PUBLIC-KEY ::= {
    IDENTIFIER id-alg-hss-lms-hashsig
    KEY HSS-LMS-HashSig-PublicKey
    PARAMS ARE absent
    CERT-KEY-USAGE
      { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

HSS-LMS-HashSig-PublicKey ::= OCTET STRING
```

Note that the id-alg-hss-lms-hashsig algorithm identifier is also referred to as id-alg-mts-hashsig. This synonym is based on the terminology used in an early draft of the document that became [HASHSIG].

The public key value is an OCTET STRING. Like the signature format, it is designed for easy parsing. The value is the number of levels in the public key, L, followed by the LMS public key.

The HSS/LMS public key value can be described as:

```
u32str(L) || lms_public_key
```

Note that the public key for the top-most LMS tree is the public key of the HSS system. When L=1, the HSS system is a single tree.

5. Signed-data Conventions

As specified in [CMS], the digital signature is produced from the message digest and the signer's private key. The signature is computed over different values depending on whether signed attributes are absent or present.

When signed attributes are absent, the HSS/LMS signature is computed over the content. When signed attributes are present, a hash is computed over the content using the same hash function that is used in the HSS/LMS tree, and then a message-digest attribute is constructed with the hash of the content, and then the HSS/LMS signature is computed over the DER-encoded set of signed attributes (which MUST include a content-type attribute and a message-digest attribute). In summary:

```
IF (signed attributes are absent)
THEN HSS_LMS_Sign(content)
ELSE message-digest attribute = Hash(content);
     HSS_LMS_Sign(DER(SignedAttributes))
```

When using [HASHSIG], the fields in the SignerInfo are used as follows:

digestAlgorithm MUST contain the one-way hash function used in the HSS/LMS tree. In [HASHSIG], SHA-256 is the only supported hash function, but other hash functions might be registered in the future. For convenience, the AlgorithmIdentifier for SHA-256 from [PKIXASN1] is repeated here:

```
mda-sha256 DIGEST-ALGORITHM ::= {
    IDENTIFIER id-sha256
    PARAMS TYPE NULL ARE preferredAbsent }

id-sha256 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2)
    country(16) us(840) organization(1) gov(101) csor(3)
    nistAlgorithms(4) hashalgs(2) 1 }
```

signatureAlgorithm MUST contain id-alg-hss-lms-hashsig, and the algorithm parameters field MUST be absent.

signature contains the single HSS signature value resulting from the signing operation as specified in [HASHSIG].

6. Security Considerations

Implementations MUST protect the private keys. Compromise of the private keys may result in the ability to forge signatures. Along with the private key, the implementation MUST keep track of which leaf nodes in the tree have been used. Loss of integrity of this tracking data can cause a one-time key to be used more than once. As a result, when a private key and the tracking data are stored on non-volatile media or stored in a virtual machine environment, failed writes, virtual machine snapshotting or cloning, and other operational concerns must be considered to ensure confidentiality and integrity.

When generating an LMS key pair, an implementation MUST generate each key pair independently of all other key pairs in the HSS tree.

An implementation MUST ensure that a LM-OTS private key is used to generate a signature only one time, and ensure that it cannot be used for any other purpose.

The generation of private keys relies on random numbers. The use of inadequate pseudo-random number generators (PRNGs) to generate these values 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, and [RFC4086] offers important guidance in this area.

The generation of hash-based signatures also depends on random numbers. While the consequences of an inadequate pseudo-random number generator (PRNG) to generate these values is much less severe than in the generation of private keys, the guidance in [RFC4086] remains important.

When computing signatures, the same hash function SHOULD be used to compute the message digest of the content and the signed attributes, if they are present.

7. IANA Considerations

SMI Security for S/MIME Module Identifier (1.2.840.113549.1.9.16.0) registry, change the reference for value 64 to point to this document.

In the SMI Security for S/MIME Algorithms (1.2.840.113549.1.9.16.3) registry, change the description for value 17 to "id-alg-hss-lms-hashsig" and change the reference to point to this document.

Also, add the following note to the registry:

> Value 17, "id-alg-hss-lms-hashsig", is also referred to as "id-alg-mts-hashsig".

8. References

8.1. Normative References

[ASN1-B] ITU-T, "Information technology -- Abstract Syntax Notation One (ASN.1): Specification of basic notation", ITU-T Recommendation X.680, 2015.

[ASN1-E] 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.

[CMS] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[HASHSIG] McGrew, D., Curcio, M., and S. Fluhrer, "Leighton-Micali Hash-Based Signatures", RFC 8554, April 2019, <https://rfc-editor.org/rfc/rfc8554.txt>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[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, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

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

8.2. Informative References

[BH2013] Ptacek, T., T. Ritter, J. Samuel, and A. Stamos, "The Factoring Dead: Preparing for the Cryptopocalypse", August 2013. <https://media.blackhat.com/us-13/us-13-Stamos-The-Factoring-Dead.pdf>

[CMSASN1] Hoffman, P. and J. Schaad, "New ASN.1 Modules for Cryptographic Message Syntax (CMS) and S/MIME", RFC 5911, DOI 10.17487/RFC5911, June 2010, <http://www.rfc-editor.org/info/rfc5911>.

[CMSASN1U] Schaad, J. and S. Turner, "Additional New ASN.1 Modules for the Cryptographic Message Syntax (CMS) and the Public Key Infrastructure Using X.509 (PKIX)", RFC 6268, DOI 10.17487/RFC6268, July 2011, <http://www.rfc-editor.org/info/rfc6268>.

[FWPROT] Housley, R., "Using Cryptographic Message Syntax (CMS) to Protect Firmware Packages", RFC 4108, DOI 10.17487/RFC4108, August 2005, <http://www.rfc-editor.org/info/rfc4108>.

[IANA-LMS] IANA Registry for Leighton-Micali Signatures (LMS). <https://www.iana.org/assignments/leighton-micali-signatures/leighton-micali-signatures.xhtml>.

[LM] Leighton, T. and S. Micali, "Large provably fast and secure digital signature schemes from secure hash functions", U.S. Patent 5,432,852, July 1995.

[M1979] Merkle, R., "Secrecy, Authentication, and Public Key Systems", Stanford University Information Systems Laboratory Technical Report 1979-1, 1979.

[M1987] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", Lecture Notes in Computer Science crypto87, 1988.

[M1989a] Merkle, R., "A Certified Digital Signature", Lecture Notes in Computer Science crypto89, 1990.

[M1989b] Merkle, R., "One Way Hash Functions and DES", Lecture Notes in Computer Science crypto89, 1990.

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

[PKIXASN1] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <http://www.rfc-editor.org/info/rfc5912>.

[PQC] Bernstein, D., "Introduction to post-quantum cryptography", 2009. <http://www.pqcrypto.org/www.springer.com/cda/content/document/cda_downloaddocument/9783540887010-c1.pdf>

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <http://www.rfc-editor.org/info/rfc4086>.

Appendix: ASN.1 Module

```
<CODE STARTS>

MTS-HashSig-2013
  { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
    id-smime(16) id-mod(0) id-mod-mts-hashsig-2013(64) }

DEFINITIONS IMPLICIT TAGS ::= BEGIN

EXPORTS ALL;

IMPORTS
  PUBLIC-KEY, SIGNATURE-ALGORITHM, SMIME-CAPS
    FROM AlgorithmInformation-2009  -- RFC 5911 [CMSASN1]
      { iso(1) identified-organization(3) dod(6) internet(1)
        security(5) mechanisms(5) pkix(7) id-mod(0)
        id-mod-algorithmInformation-02(58) } ;

--
-- Object Identifiers
--

id-alg-hss-lms-hashsig OBJECT IDENTIFIER ::= { iso(1)
    member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
    smime(16) alg(3) 17 }

id-alg-mts-hashsig OBJECT IDENTIFIER ::= id-alg-hss-lms-hashsig

--
-- Signature Algorithm and Public Key
--

sa-HSS-LMS-HashSig SIGNATURE-ALGORITHM ::= {
    IDENTIFIER id-alg-hss-lms-hashsig
    PARAMS ARE absent
    PUBLIC-KEYS { pk-HSS-LMS-HashSig }
    SMIME-CAPS { IDENTIFIED BY id-alg-hss-lms-hashsig } }
```

```
   pk-HSS-LMS-HashSig PUBLIC-KEY ::= {
       IDENTIFIER id-alg-hss-lms-hashsig
       KEY HSS-LMS-HashSig-PublicKey
       PARAMS ARE absent
       CERT-KEY-USAGE
           { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

   HSS-LMS-HashSig-PublicKey ::= OCTET STRING

   --
   -- Expand the signature algorithm set used by CMS [CMSASN1U]
   --

   SignatureAlgorithmSet SIGNATURE-ALGORITHM ::=
       { sa-HSS-LMS-HashSig, ... }

   --
   -- Expand the S/MIME capabilities set used by CMS [CMSASN1]
   --

   SMimeCaps SMIME-CAPS ::=
       { sa-HSS-LMS-HashSig.&smimeCaps, ... }

   END

   <CODE ENDS>
```

Acknowledgements

Many thanks to Scott Fluhrer, Jonathan Hammell, Ben Kaduk, Panos Kampanakis, Barry Leiba, John Mattsson, Jim Schaad, Sean Turner, Daniel Van Geest, Roman Danyliw, Dale Worley, and Joe Clarke for their careful review and comments.

Author's Address

Russ Housley
Vigil Security, LLC
516 Dranesville Road
Herndon, VA 20170
USA

EMail: housley@vigilsec.com

INTERNET-DRAFT R. Housley
Internet Engineering Task Force (IETF) Vigil Security
Intended Status: Proposed Standard
Expires: 23 February 2020 23 August 2019

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

<draft-ietf-lamps-cms-mix-with-psk-07.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.

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. 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 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. 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 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 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 [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. 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:

```
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.

An acceptable KDF MUST accept IKM, L, and info inputs; and 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].

```
<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

The security considerations in 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, which 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 consequence for the compromise of either one of these keys.

Implementations must protect the pre-shared key (PSK), key transport private key, the agreement private key, and the 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 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 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.

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 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 inherently 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 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 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 }
```

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 additions 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 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.

[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.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017.

[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-05, 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.

[NIST2018] Barker, E., Chen, L., and R. Davis, "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", NIST Special Publication 800-56C Rev. 1, April 2018, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Cr1.pdf>.

[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.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, November 2016.

[RFC8619] Housley, R., "Algorithm Identifiers for the HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", June 2019.

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:

```
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
5cf95b150a0105300b060960864801650304012d020120020120
```

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, in hexadecimal:

```
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, 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
USA
EMail: housley@vigilsec.com

LAMPS WG
Internet-Draft
Updates: 3370 (if approved)
Intended status: Standards Track
Expires: March 19, 2020
P. Kampanakis
Cisco Systems
Q. Dang
NIST
September 16, 2019

Use of the SHAKE One-way Hash Functions in the Cryptographic Message Syntax (CMS)

draft-ietf-lamps-cms-shakes-18

Abstract

This document updates the "Cryptographic Message Syntax Algorithms" (RFC3370) and describes the conventions for using the SHAKE family of hash functions in the Cryptographic Message Syntax as one-way hash functions with the RSA Probabilistic signature and ECDSA signature algorithms. The conventions for the associated signer public keys in CMS are also described.

Status of This Memo

Copyright Notice

Table of Contents

1. Change Log

[EDNOTE: Remove this section before publication.]

- o draft-ietf-lamps-cms-shake-18:

  * Minor ASN.1 changes.

- o draft-ietf-lamps-cms-shake-17:

  * Minor updates for EDNOTE accuracy.

- o draft-ietf-lamps-cms-shake-16:

  * Minor nits.

  * Using bytes instead of bits for consistency.

- o draft-ietf-lamps-cms-shake-15:

  * Minor editorial nits.

- o draft-ietf-lamps-cms-shake-14:

  * Fixing error with incorrect preimage resistance bits for SHA128 and SHA256.

- o draft-ietf-lamps-cms-shake-13:

  * Addressing comments from Dan M.'s secdir review.

  * Addressing comment from Scott B.'s opsdir review about references in the abstract.

- o draft-ietf-lamps-cms-shake-12:

  * Nits identified by Roman, Barry L. in ballot position review.

- o draft-ietf-lamps-cms-shake-11:

  * Minor nits.

  * Nits identified by Roman in AD Review.

- o draft-ietf-lamps-cms-shake-10:

  * Updated IANA considerations section to request for OID assignments.

- o draft-ietf-lamps-cms-shake-09:

  * Fixed minor text nit.

  * Updates in Sec Considerations section.

- o draft-ietf-lamps-cms-shake-08:

  * id-shake128-len and id-shake256-len were replaced with id-sha128 with 32 bytes output length and id-shake256 with 64 bytes output length.

  * Fixed a discrepancy between section 3 and 4.4 about the KMAC OIDs that have parameters as optional.

- o draft-ietf-lamps-cms-shake-07:

  * Small nit from Russ while in WGLC.

- o draft-ietf-lamps-cms-shake-06:

  * Incorporated Eric's suggestion from WGLC.

- draft-ietf-lamps-cms-shake-05:

  * Added informative references.

  * Updated ASN.1 so it compiles.

  * Updated IANA considerations.

- draft-ietf-lamps-cms-shake-04:

  * Added RFC8174 reference and text.

  * Explicitly explained why RSASSA-PSS-params are omitted in section 4.2.1.

  * Simplified Public Keys section by removing redundant info from RFCs.

- draft-ietf-lamps-cms-shake-03:

  * Removed paragraph suggesting KMAC to be used in generating k in Deterministic ECDSA. That should be RFC6979-bis.

  * Removed paragraph from Security Considerations that talks about randomness of k because we are using deterministic ECDSA.

  * Completed ASN.1 module and fixed KMAC ASN.1 based on Jim's feedback.

  * Text fixes.

- draft-ietf-lamps-cms-shake-02:

  * Updates based on suggestions and clarifications by Jim.

  * Started ASN.1 module.

- draft-ietf-lamps-cms-shake-01:

  * Significant reorganization of the sections to simplify the introduction, the new OIDs and their use in CMS.

  * Added new OIDs for RSASSA-PSS that hardcodes hash, salt and MGF, according the WG consensus.

   * Updated Public Key section to use the new RSASSA-PSS OIDs and clarify the algorithm identifier usage.

   * Removed the no longer used SHAKE OIDs from section 3.1.

- draft-ietf-lamps-cms-shake-00:

   * Various updates to title and section names.

   * Content changes filling in text and references.

- draft-dang-lamps-cms-shakes-hash-00:

   * Initial version

2. Introduction

The "Cryptographic Message Syntax (CMS)" [RFC5652] is used to digitally sign, digest, authenticate, or encrypt arbitrary message contents. "Cryptographic Message Syntax (CMS) Algorithms" [RFC3370] defines the use of common cryptographic algorithms with CMS. This specification updates RFC3370 and describes the use of the SHAKE128 and SHAKE256 specified in [SHA3] as new hash functions in CMS. In addition, it describes the use of these functions with the RSASSA-PSS signature algorithm [RFC8017] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [X9.62] with the CMS signed-data content type.

In the SHA-3 family, two extendable-output functions (SHAKEs), SHAKE128 and SHAKE256, are defined. Four other hash function instances, SHA3-224, SHA3-256, SHA3-384, and SHA3-512, are also defined but are out of scope for this document. A SHAKE is a variable length hash function defined as SHAKE(M, d) where the output is a d-bits-long digest of message M. The corresponding collision and second-preimage-resistance strengths for SHAKE128 are min(d/2,128) and min(d,128) bits, respectively (Appendix A.1 [SHA3]). And the corresponding collision and second-preimage-resistance strengths for SHAKE256 are min(d/2,256) and min(d,256) bits, respectively. In this specification we use d=256 (for SHAKE128) and d=512 (for SHAKE256).

A SHAKE can be used in CMS as the message digest function (to hash the message to be signed) in RSASSA-PSS and ECDSA, message authentication code and as the mask generation function (MGF) in RSASSA-PSS. This specification describes the identifiers for SHAKEs to be used in CMS and their meaning.

2.1. 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.

3. Identifiers

This section identifies eight new object identifiers (OIDs) for using SHAKE128 and SHAKE256 in CMS.

Two object identifiers for SHAKE128 and SHAKE256 hash functions are defined in [shake-nist-oids] and we include them here for convenience.

```
  id-shake128 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2)
       country(16) us(840) organization(1) gov(101) csor(3)
       nistAlgorithm(4) 2 11 }

  id-shake256 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2)
       country(16) us(840) organization(1) gov(101) csor(3)
       nistAlgorithm(4) 2 12 }
```

In this specification, when using the id-shake128 or id-shake256 algorithm identifiers, the parameters MUST be absent. That is, the identifier SHALL be a SEQUENCE of one component, the OID.

[I-D.ietf-lamps-pkix-shake] [EDNOTE: Update reference with the RFC when it is published.] defines two identifiers for RSASSA-PSS signatures using SHAKEs which we include here for convenience.

```
  id-RSASSA-PSS-SHAKE128  OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6) 30 }

  id-RSASSA-PSS-SHAKE256  OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6) 31 }
```

The same RSASSA-PSS algorithm identifiers can be used for identifying public keys and signatures.

[I-D.ietf-lamps-pkix-shake] [EDNOTE: Update reference with the RFC when it is published.] also defines two algorithm identifiers of ECDSA signatures using SHAKEs which we include here for convenience.

```
id-ecdsa-with-shake128 OBJECT IDENTIFIER  ::=  { iso(1)
          identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) algorithms(6) 32 }

id-ecdsa-with-shake256 OBJECT IDENTIFIER  ::=  { iso(1)
          identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) algorithms(6) 33 }
```

The parameters for the four RSASSA-PSS and ECDSA identifiers MUST be absent. That is, each identifier SHALL be a SEQUENCE of one component, the OID.

Two object identifiers for KMACs using SHAKE128 and SHAKE256 as defined in by the National Institute of Standards and Technology (NIST) in [shake-nist-oids] and we include them here for convenience.

```
id-KmacWithSHAKE128 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2)
    country(16) us(840) organization(1) gov(101) csor(3)
    nistAlgorithm(4) 2 19 }

id-KmacWithSHAKE256 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2)
    country(16) us(840) organization(1) gov(101) csor(3)
    nistAlgorithm(4) 2 20 }
```

The parameters for id-KmacWithSHAKE128 and id-KmacWithSHAKE256 are OPTIONAL.

Section 4.1, Section 4.2.1, Section 4.2.2 and Section 4.4 specify the required output length for each use of SHAKE128 or SHAKE256 in message digests, RSASSA-PSS, ECDSA and KMAC.

4. Use in CMS

4.1. Message Digests

The id-shake128 and id-shake256 OIDs (Section 3) can be used as the digest algorithm identifiers located in the SignedData, SignerInfo, DigestedData, and the AuthenticatedData digestAlgorithm fields in CMS [RFC5652]. The OID encoding MUST omit the parameters field and the output length of SHAKE128 or SHAKE256 as the message digest MUST be 32 or 64 bytes, respectively.

The digest values are located in the DigestedData field and the Message Digest authenticated attribute included in the signedAttributes of the SignedData signerInfo. In addition, digest values are input to signature algorithms. The digest algorithm MUST be the same as the message hash algorithms used in signatures.

4.2. Signatures

In CMS, signature algorithm identifiers are located in the SignerInfo signatureAlgorithm field of SignedData content type and countersignature attribute. Signature values are located in the SignerInfo signature field of SignedData content type and countersignature attribute.

Conforming implementations that process RSASSA-PSS and ECDSA with SHAKE signatures when processing CMS data MUST recognize the corresponding OIDs specified in Section 3.

When using RSASSA-PSS or ECDSA with SHAKEs, the RSA modulus or ECDSA curve order SHOULD be chosen in line with the SHAKE output length. Refer to Section 6 for more details.

4.2.1. RSASSA-PSS Signatures

The RSASSA-PSS algorithm is defined in [RFC8017]. When id-RSASSA-PSS-SHAKE128 or id-RSASSA-PSS-SHAKE256 specified in Section 3 is used, the encoding MUST omit the parameters field. That is, the AlgorithmIdentifier SHALL be a SEQUENCE of one component, id-RSASSA-PSS-SHAKE128 or id-RSASSA-PSS-SHAKE256. [RFC4055] defines RSASSA-PSS-params that are used to define the algorithms and inputs to the algorithm. This specification does not use parameters because the hash, mask generation algorithm, trailer and salt are embedded in the OID definition.

The hash algorithm to hash a message being signed and the hash algorithm as the mask generation function used in RSASSA-PSS MUST be the same: both SHAKE128 or both SHAKE256. The output length of the hash algorithm which hashes the message SHALL be 32 (for SHAKE128) or 64 bytes (for SHAKE256).

The mask generation function takes an octet string of variable length and a desired output length as input, and outputs an octet string of the desired length. In RSASSA-PSS with SHAKEs, the SHAKEs MUST be used natively as the MGF function, instead of the MGF1 algorithm that uses the hash function in multiple iterations as specified in Section B.2.1 of [RFC8017]. In other words, the MGF is defined as the SHAKE128 or SHAKE256 with input being the mgfSeed for id-RSASSA-PSS- SHAKE128 and id-RSASSA-PSS-SHAKE256, respectively. The mgfSeed is the seed from which mask is generated, an octet string [RFC8017]. As explained in Step 9 of section 9.1.1 of [RFC8017], the output length of the MGF is emLen - hLen - 1 bytes. emLen is the maximum message length ceil((n-1)/8), where n is the RSA modulus in bits. hLen is 32 and 64-bytes for id-RSASSA-PSS-SHAKE128 and id-RSASSA-PSS-SHAKE256, respectively. Thus when SHAKE is used as the MGF, the

SHAKE output length maskLen is (8*emLen - 264) or (8*emLen - 520) bits, respectively. For example, when RSA modulus n is 2048, the output length of SHAKE128 or SHAKE256 as the MGF will be 1784 or 1528-bits when id-RSASSA-PSS-SHAKE128 or id-RSASSA-PSS-SHAKE256 is used, respectively.

The RSASSA-PSS saltLength MUST be 32 bytes for id-RSASSA-PSS-SHAKE128 or 64 bytes for id-RSASSA-PSS-SHAKE256. Finally, the trailerField MUST be 1, which represents the trailer field with hexadecimal value 0xBC [RFC8017].

4.2.2. ECDSA Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) is defined in [X9.62]. When the id-ecdsa-with-shake128 or id-ecdsa-with-shake256 (specified in Section 3) algorithm identifier appears, the respective SHAKE function is used as the hash. The encoding MUST omit the parameters field. That is, the AlgorithmIdentifier SHALL be a SEQUENCE of one component, the OID id-ecdsa-with-shake128 or id-ecdsa-with-shake256.

For simplicity and compliance with the ECDSA standard specification, the output length of the hash function must be explicitly determined. The output length for SHAKE128 or SHAKE256 used in ECDSA MUST be 32 or 64 bytes, respectively.

Conforming CA implementations that generate ECDSA with SHAKE signatures in certificates or CRLs SHOULD generate such signatures with a deterministically generated, non-random k in accordance with all the requirements specified in [RFC6979]. They MAY also generate such signatures in accordance with all other recommendations in [X9.62] or [SEC1] if they have a stated policy that requires conformance to those standards. Those standards have not specified SHAKE128 and SHAKE256 as hash algorithm options. However, SHAKE128 and SHAKE256 with output length being 32 and 64 octets, respectively can be used instead of 256 and 512-bit output hash algorithms such as SHA256 and SHA512.

4.3. Public Keys

In CMS, the signer’s public key algorithm identifiers are located in the OriginatorPublicKey’s algorithm attribute. The conventions and encoding for RSASSA-PSS and ECDSA public keys algorithm identifiers are as specified in Section 2.3 of [RFC3279], Section 3.1 of [RFC4055] and Section 2.1 of [RFC5480].

Traditionally, the rsaEncryption object identifier is used to identify RSA public keys. The rsaEncryption object identifier continues to identify the public key when the RSA private key owner does not wish to limit the use of the public key exclusively to RSASSA-PSS with SHAKEs. When the RSA private key owner wishes to limit the use of the public key exclusively to RSASSA-PSS, the AlgorithmIdentifier for RSASSA-PSS defined in Section 3 SHOULD be used as the algorithm attribute in the OriginatorPublicKey sequence. Conforming client implementations that process RSASSA-PSS with SHAKE public keys in CMS message MUST recognize the corresponding OIDs in Section 3.

Conforming implementations MUST specify and process the algorithms explicitly by using the OIDs specified in Section 3 when encoding ECDSA with SHAKE public keys in CMS messages.

The identifier parameters, as explained in Section 3, MUST be absent.

4.4. Message Authentication Codes

KMAC message authentication code (KMAC) is specified in [SP800-185]. In CMS, KMAC algorithm identifiers are located in the AuthenticatedData macAlgorithm field. The KMAC values are located in the AuthenticatedData mac field.

When the id-KmacWithSHAKE128 or id-KmacWithSHAKE256 OID is used as the MAC algorithm identifier, the parameters field is optional (absent or present). If absent, the SHAKE256 output length used in KMAC is 32 or 64 bytes, respectively, and the customization string is an empty string by default.

Conforming implementations that process KMACs with the SHAKEs when processing CMS data MUST recognize these identifiers.

When calculating the KMAC output, the variable N is 0xD2B282C2, S is an empty string, and L, the integer representing the requested output length in bits, is 256 or 512 for KmacWithSHAKE128 or KmacWithSHAKE256, respectively, in this specification.

5. IANA Considerations

One object identifier for the ASN.1 module in Appendix A was requested for the SMI Security for S/MIME Module Identifiers (1.2.840.113549.1.9.16.0) registry:

Decimal	Description	References
70	CMSAlgsForSHAKE-2019	[EDNOTE: THIS RFC]

6. Security Considerations

This document updates [RFC3370]. The security considerations section of that document applies to this specification as well.

NIST has defined appropriate use of the hash functions in terms of the algorithm strengths and expected time frames for secure use in Special Publications (SPs) [SP800-78-4] and [SP800-107]. These documents can be used as guides to choose appropriate key sizes for various security scenarios.

SHAKE128 with output length of 32 bytes offers 128-bits of collision and preimage resistance. Thus, SHAKE128 OIDs in this specification are RECOMMENDED with 2048 (112-bit security) or 3072-bit (128-bit security) RSA modulus or curves with group order of 256-bits (128-bit security). SHAKE256 with 64 bytes output length offers 256-bits of collision and preimage resistance. Thus, the SHAKE256 OIDs in this specification are RECOMMENDED with 4096-bit RSA modulus or higher or curves with group order of at least 512 bits such as NIST Curve P-521 (256-bit security). Note that we recommended 4096-bit RSA because we would need 15360-bit modulus for 256-bits of security which is impractical for today’s technology.

When more than two parties share the same message-authentication key, data origin authentication is not provided. Any party that knows the message-authentication key can compute a valid MAC, therefore the content could originate from any one of the parties.

7. Acknowledgements

This document is based on Russ Housley’s draft [I-D.housley-lamps-cms-sha3-hash]. It replaces SHA3 hash functions by SHAKE128 and SHAKE256 as the LAMPS WG agreed.

The authors would like to thank Russ Housley for his guidance and very valuable contributions with the ASN.1 module. Valuable feedback was also provided by Eric Rescorla.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3370] Housley, R., "Cryptographic Message Syntax (CMS) Algorithms", RFC 3370, DOI 10.17487/RFC3370, August 2002, <https://www.rfc-editor.org/info/rfc3370>.

[RFC4055] Schaad, J., Kaliski, B., and R. Housley, "Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 4055, DOI 10.17487/RFC4055, June 2005, <https://www.rfc-editor.org/info/rfc4055>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/info/rfc5652>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[SHA3] National Institute of Standards and Technology, U.S. Department of Commerce, "SHA-3 Standard - Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, August 2015.

[SP800-185] National Institute of Standards and Technology, "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash. NIST SP 800-185", December 2016, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf>.

8.2. Informative References

[I-D.housley-lamps-cms-sha3-hash] Housley, R., "Use of the SHA3 One-way Hash Functions in the Cryptographic Message Syntax (CMS)", draft-housley-lamps-cms-sha3-hash-00 (work in progress), March 2017.

[I-D.ietf-lamps-pkix-shake]
Kampanakis, P. and Q. Dang, "Internet X.509 Public Key Infrastructure: Additional Algorithm Identifiers for RSASSA-PSS and ECDSA using SHAKEs", draft-ietf-lamps-pkix-shake-15 (work in progress), July 2019.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, DOI 10.17487/RFC3279, April 2002, <https://www.rfc-editor.org/info/rfc3279>.

[RFC5753] Turner, S. and D. Brown, "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)", RFC 5753, DOI 10.17487/RFC5753, January 2010, <https://www.rfc-editor.org/info/rfc5753>.

[RFC5911] Hoffman, P. and J. Schaad, "New ASN.1 Modules for Cryptographic Message Syntax (CMS) and S/MIME", RFC 5911, DOI 10.17487/RFC5911, June 2010, <https://www.rfc-editor.org/info/rfc5911>.

[RFC6268] Schaad, J. and S. Turner, "Additional New ASN.1 Modules for the Cryptographic Message Syntax (CMS) and the Public Key Infrastructure Using X.509 (PKIX)", RFC 6268, DOI 10.17487/RFC6268, July 2011, <https://www.rfc-editor.org/info/rfc6268>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009, <http://www.secg.org/sec1-v2.pdf>.

[shake-nist-oids]
National Institute of Standards and Technology, "Computer Security Objects Register", October 2017, <https://csrc.nist.gov/Projects/Computer-Security-Objects-Register/Algorithm-Registration>.

[SP800-107]
National Institute of Standards and Technology (NIST), "SP800-107: Recommendation for Applications Using Approved Hash Algorithms", May 2014, <https://csrc.nist.gov/csrc/media/publications/sp/800-107/rev-1/final/documents/draft_revised_sp800-107.pdf>.

[SP800-78-4]
National Institute of Standards and Technology (NIST), "SP800-78-4: Cryptographic Algorithms and Key Sizes for Personal Identity Verification", May 2014, <https://csrc.nist.gov/csrc/media/publications/sp/800-78/4/final/documents/sp800_78-4_revised_draft.pdf>.

[X9.62] American National Standard for Financial Services (ANSI), "X9.62-2005 Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Standard (ECDSA)", November 2005.

Appendix A. ASN.1 Module

This appendix includes the ASN.1 modules for SHAKEs in CMS. This module includes some ASN.1 from other standards for reference.

```
CMSAlgsForSHAKE-2019 { iso(1) member-body(2) us(840)
     rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) modules(0)
     id-mod-cms-shakes-2019(70) }

DEFINITIONS EXPLICIT TAGS ::=

BEGIN

-- EXPORTS ALL;

IMPORTS

DIGEST-ALGORITHM, MAC-ALGORITHM, SMIME-CAPS
FROM AlgorithmInformation-2009
  { iso(1) identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7) id-mod(0)
    id-mod-algorithmInformation-02(58) }

RSAPublicKey, rsaEncryption, id-ecPublicKey
FROM PKIXAlgs-2009 { iso(1) identified-organization(3) dod(6)
     internet(1) security(5) mechanisms(5) pkix(7) id-mod(0)
     id-mod-pkix1-algorithms2008-02(56) }

sa-rsassapssWithSHAKE128, sa-rsassapssWithSHAKE256,
```

```
   sa-ecdsaWithSHAKE128, sa-ecdsaWithSHAKE256
   FROM PKIXAlgsForSHAKE-2019 {
      iso(1) identified-organization(3) dod(6)
      internet(1) security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkix1-shakes-2019(94) } ;

   -- Message Digest Algorithms (mda-)
   -- used in SignedData, SignerInfo, DigestedData,
   -- and the AuthenticatedData digestAlgorithm
   -- fields in CMS
   --
   --  This expands MessageAuthAlgs from [RFC5652] and
   --  MessageDigestAlgs in [RFC5753]
   --
   -- MessageDigestAlgs DIGEST-ALGORITHM ::= {
   --  mda-shake128   |
   --  mda-shake256,
   --  ...
   -- }

   --
   -- One-Way Hash Functions
   -- SHAKE128
   mda-shake128 DIGEST-ALGORITHM ::= {
     IDENTIFIER id-shake128  -- with output length 32 bytes.
   }
   id-shake128 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16)
                                       us(840) organization(1) gov(101)
                                       csor(3) nistAlgorithm(4)
                                       hashAlgs(2) 11 }

   -- SHAKE256
   mda-shake256 DIGEST-ALGORITHM ::= {
     IDENTIFIER id-shake256  -- with output length 64 bytes.
   }
   id-shake256 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16)
                                       us(840) organization(1) gov(101)
                                       csor(3) nistAlgorithm(4)
                                       hashAlgs(2) 12 }

   --
   -- Public key algorithm identifiers located in the
   -- OriginatorPublicKey's algorithm attribute in CMS.
   -- And Signature identifiers used in SignerInfo
   -- signatureAlgorithm field of SignedData content
   -- type and countersignature attribute in CMS.
   --
   -- From RFC5280, for reference.
```

```
   -- rsaEncryption OBJECT IDENTIFIER ::=  { pkcs-1 1 }
      -- When the rsaEncryption algorithm identifier is used
      -- for a public key, the AlgorithmIdentifier parameters
      -- field MUST contain NULL.
   --
   id-RSASSA-PSS-SHAKE128  OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6) 30 }

   id-RSASSA-PSS-SHAKE256  OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6) 31 }

      -- When the id-RSASSA-PSS-* algorithm identifiers are used
      -- for a public key or signature in CMS, the AlgorithmIdentifier
      -- parameters field MUST be absent. The message digest algorithm
      -- used in RSASSA-PSS MUST be SHAKE128 or SHAKE256 with a 32 or
      -- 64 byte outout length, respectively. The mask generation
      -- function MUST be SHAKE128 or SHAKE256 with an output length
      -- of (8*ceil((n-1)/8) - 264) or (8*ceil((n-1)/8) - 520) bits,
      -- respectively, where n is the RSA modulus in bits.
      -- The RSASSA-PSS saltLength MUST be 32 or 64 bytes, respectively.
      -- The trailerField MUST be 1, which represents the trailer
      -- field with hexadecimal value 0xBC. Regardless of
      -- id-RSASSA-PSS-* or rsaEncryption being used as the
      -- AlgorithmIdentifier of the OriginatorPublicKey, the RSA
      -- public key MUST be encoded using the RSAPublicKey type.

   -- From RFC4055, for reference.
   -- RSAPublicKey ::= SEQUENCE {
   --   modulus INTEGER, -- -- n
   --   publicExponent INTEGER } -- -- e

   id-ecdsa-with-shake128 OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6) 32 }

   id-ecdsa-with-shake256 OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6) 33 }

      -- When the id-ecdsa-with-shake* algorithm identifiers are
      -- used in CMS, the AlgorithmIdentifier parameters field
      -- MUST be absent and the signature algorithm should be
      -- deterministic ECDSA [RFC6979]. The message digest MUST
      -- be SHAKE128 or SHAKE256 with a 32 or 64 byte outout
      -- length, respectively. In both cases, the ECDSA public key,
      -- MUST be encoded using the id-ecPublicKey type.
```

```
   -- From RFC5480, for reference.
   -- id-ecPublicKey OBJECT IDENTIFIER ::= {
   --    iso(1) member-body(2) us(840) ansi-X9-62(10045) keyType(2) 1 }
        -- The id-ecPublicKey parameters must be absent or present
        -- and are defined as
   -- ECParameters ::= CHOICE {
   --     namedCurve         OBJECT IDENTIFIER
   --     -- -- implicitCurve   NULL
   --     -- -- specifiedCurve  SpecifiedECDomain
   --  }

   -- This expands SignatureAlgorithms from [RFC5912]
   --
   -- SignatureAlgs SIGNATURE-ALGORITHM ::= {
   --   sa-rsassapssWithSHAKE128 |
   --   sa-rsassapssWithSHAKE256 |
   --   sa-ecdsaWithSHAKE128 |
   --   sa-ecdsaWithSHAKE256,
   --   ...
   -- }

   -- This expands MessageAuthAlgs from [RFC5652] and [RFC6268]
   --
   -- Message Authentication (maca-) Algorithms
   -- used in AuthenticatedData macAlgorithm in CMS
   --
   MessageAuthAlgs MAC-ALGORITHM ::= {
       maca-KMACwithSHAKE128    |
       maca-KMACwithSHAKE256,
       ...
   }

   -- This expands SMimeCaps from [RFC5911]
   --
   SMimeCaps SMIME-CAPS ::= {
      -- sa-rsassapssWithSHAKE128.&smimeCaps |
      -- sa-rsassapssWithSHAKE256.&smimeCaps |
      -- sa-ecdsaWithSHAKE128.&smimeCaps |
      -- sa-ecdsaWithSHAKE256.&smimeCaps,
      maca-KMACwithSHAKE128.&smimeCaps   |
      maca-KMACwithSHAKE256.&smimeCaps,
      ...
    }

   --
   -- KMAC with SHAKE128
   maca-KMACwithSHAKE128 MAC-ALGORITHM ::= {
         IDENTIFIER id-KMACWithSHAKE128
```

```
         PARAMS TYPE KMACwithSHAKE128-params ARE optional
           -- If KMACwithSHAKE128-params parameters are absent
           -- the SHAKE128 output length used in KMAC is 256 bits
           -- and the customization string is an empty string.
         IS-KEYED-MAC TRUE
         SMIME-CAPS {IDENTIFIED BY id-KMACWithSHAKE128}
   }
   id-KMACWithSHAKE128 OBJECT IDENTIFIER ::=  { joint-iso-itu-t(2)
                             country(16) us(840) organization(1)
                             gov(101) csor(3) nistAlgorithm(4)
                             hashAlgs(2) 19 }
   KMACwithSHAKE128-params ::= SEQUENCE {
     kMACOutputLength     INTEGER DEFAULT 256, -- Output length in bits
     customizationString  OCTET STRING DEFAULT ''H
   }

   -- KMAC with SHAKE256
   maca-KMACwithSHAKE256 MAC-ALGORITHM ::= {
         IDENTIFIER id-KMACWithSHAKE256
         PARAMS TYPE KMACwithSHAKE256-params ARE optional
            -- If KMACwithSHAKE256-params parameters are absent
            -- the SHAKE256 output length used in KMAC is 512 bits
            -- and the customization string is an empty string.
         IS-KEYED-MAC TRUE
         SMIME-CAPS {IDENTIFIED BY id-KMACWithSHAKE256}
   }
   id-KMACWithSHAKE256 OBJECT IDENTIFIER ::=  { joint-iso-itu-t(2)
                            country(16) us(840) organization(1)
                            gov(101) csor(3) nistAlgorithm(4)
                            hashAlgs(2) 20 }
   KMACwithSHAKE256-params ::= SEQUENCE {
      kMACOutputLength     INTEGER DEFAULT 512, -- Output length in bits
      customizationString  OCTET STRING DEFAULT ''H
   }

   END
```

Authors' Addresses

Panos Kampanakis
Cisco Systems

Email: pkampana@cisco.com

Quynh Dang
NIST
100 Bureau Drive
Gaithersburg, MD 20899

Email: quynh.Dang@nist.gov

Network Working Group
Internet-Draft
Intended status: Informational
Expires: December 31, 2019

R. Housley
Vigil Security
June 29, 2019

Hash Of Root Key Certificate Extension
draft-ietf-lamps-hash-of-root-key-cert-extn-07

Abstract

This document specifies the Hash Of Root Key certificate extension. This certificate extension is carried in the self-signed certificate for a trust anchor, which is often called a Root Certification Authority (CA) certificate. This certificate extension unambiguously identifies the next public key that will be used at some point in the future as the next Root CA certificate, eventually replacing the current one.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document specifies the Hash Of Root Key X.509 version 3 certificate extension. The extension is an optional addition to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile [RFC5280]. The certificate extension facilitates the orderly transition from one Root Certification Authority (CA) public key to the next. It does so by publishing the hash value of the next generation public key in the current self-signed certificate. This hash value is a commitment to a particular public key in the next generation self-signed certificate. This commitment allows a relying party to unambiguously recognize the next generation self-signed certificate when it becomes available, install the new self-signed certificate in the trust anchor store, and eventually remove the previous one from the trust anchor store.

A Root CA Certificate MAY include the Hashed Root Key certificate extension to provide the hash value of the next public key that will be used by the Root CA.

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

Certificates [RFC5280] use ASN.1 [X680]; Distinguished Encoding Rules (DER) [X690] are REQUIRED for certificate signing and validation.

2. Overview

Before the initial deployment of the Root CA, the following are generated:

R1 = The initial Root key pair
R2 = The second generation Root key pair
H2 = Thumbprint (hash) of the public key of R2
C1 = Self-signed certificate for R1, which also contains H2

C1 is a self-signed certificate, and it contains H2 within the HashOfRootKey extension. C1 is distributed as part of the initial the system deployment. The HashOfRootKey certificate extension is described in Section 3.

When the time comes to replace the initial Root CA certificate, R1, the following are generated:

R3 = The third generation Root key pair
H3 = Thumbprint (hash) the public key of R3
C2 = Self-signed certificate for R2, which contains H3

This is an iterative process. That is, R4 and H4 are generated when it is time for C3 to replace C2. And so on.

The successor to the Root CA self-signed certificate can be delivered by any means. Whenever a new Root CA self-signed certificate is received, the recipient is able to verify that the potential Root CA certificate links back to a previously authenticated Root CA certificate with the hashOfRootKey certificate extension. That is, the recipient verifies the signature on the self-signed certificate and verifies that the hash of the DER-encoded SubjectPublicKeyInfo from the potential Root CA certificate matches the value from the HashOfRootKey certificate extension of the current Root CA certificate. Checking the self-signed certificate signature ensures that the certificate contains the subject name, public key algorithm identifier, and public key algorithm parameters intended by the key owner; these are important inputs to certification path validation as defined in Section 6 of [RFC5280]. Checking the hash of the SubjectPublicKeyInfo ensures that the certificate contains the intended public key. If either check fails, then the potential Root CA certificate is not a valid replacement, and the recipient continues to use the current Root CA certificate. If both checks succeed, then the recipient adds the potential Root CA certificate to the trust anchor store. As discussed in Section 5, the recipient can remove the current Root CA certificate immediately in some situations. In other situations, the recipient waits an appropriate amount of time to ensure that existing certification paths continue to validate.

3. Hash Of Root Key Certificate Extension

The HashOfRootKey certificate extension MUST NOT be critical.

The following ASN.1 [X680][X690] syntax defines the HashOfRootKey certificate extension:

```
ext-HashOfRootKey EXTENSION ::= {    -- Only in Root CA certificates
   SYNTAX         HashedRootKey
   IDENTIFIED BY  id-ce-hashOfRootKey
   CRITICALITY    {FALSE} }

HashedRootKey ::= SEQUENCE {
   hashAlg        HashAlgorithm,         -- Hash algorithm used
   hashValue      OCTET STRING }         -- Hash of DER-encoded
                                         --   SubjectPublicKeyInfo

id-ce-hashOfRootKey  ::=  OBJECT IDENTIFIER { 1 3 6 1 4 1 51483 2 1 }
```

The definitions of EXTENSION and HashAlgorithm can be found in [RFC5912].

The hashAlg indicates the one-way hash algorithm that was used to compute the hash value.

The hashValue contains the hash value computed from the next generation public key. The public key is DER-encoded SubjectPublicKeyInfo as defined in [RFC5280].

4. IANA Considerations

This document makes no requests of the IANA.

5. Operational Considerations

Guidance on the transition from one trust anchor to another is available in Section 4.4 of [RFC4210]. In particular, the oldWithNew and newWithOld advice ensures that relying parties are able to validate certificates issued under the current Root CA certificate and the next generation Root CA certificate throughout the transition. The notAfter field in the oldWithNew certificate MUST cover the validity period of all unexpired certificates issued under the old Root CA private key. Further, this advice SHOULD be followed by Root CAs to avoid the need for all relying parties to make the transition at the same time.

After issuing the newWithOld certificate, the Root CA MUST stop using the old private key to sign certificates.

Some enterprise and application-specific environments offer a directory service or certificate repository to make certificate and CRLs available to relying parties. Section 3 in [RFC5280] describes a certificate repository. When a certificate repository is available, the oldWithNew and newWithOld certificates SHOULD be published before the successor to the current Root CA self-signed certificate is released. Recipients that are able to obtain the oldWithNew certificate SHOULD immediately remove the old Root CA self-signed certificate from the trust anchor store.

In environments without such a directory service or repository, like the Web PKI, recipients need a way to obtain the oldWithNew and newWithOld certificates. The Root CA SHOULD include the subject information access extension [RFC5280] with the accessMethod set to id-ad-caRepository and the assessLocation set to the HTTP URL that can be used to fetch a DER-encoded "certs-only" (simple PKI response) message as specified in [RFC5272] in all of their self-signed certificates. The Root CA SHOULD publish the "certs-only" message with the oldWithNew certificate and the newWithOld certificate before the subsequent Root CA self-signed certificate is released. The "certs-only" message format allows certificates to be added and removed from the bag of certificates over time, so the same HTTP URL can be used throughout the lifetime of the Root CA.

In environments without such a directory service or repository, recipients SHOULD keep both the old and replacement Root CA self-signed certificates in the trust anchor store for some amount of time to ensure that all end-entity certificates can be validated until they expire. The recipient MAY keep the old Root CA self-signed certificate until all of the certificates in the local cache that are subordinate to it have expired.

Certification path construction is more complex when the trust anchor store contains multiple self-signed certificates with the same distinguished name. For this reason, the replacement Root CA self-signed certificate SHOULD contain a different distinguished name than the one it is replacing. One approach is to include a number as part of the name that is incremented with each generation, such as "Example CA", "Example CA G2", "Example CA G3", and so on.

Changing names from one generation to another can lead to confusion when reviewing the history of a trust anchor store. To assist with such review, a recipient MAY create an audit entry to capture the old and replacement self-signed certificates.

The Root CA must securely back up the yet-to-be-deployed key pair. If the Root CA stores the key pair in a hardware security module, and that module fails, the Root CA remains committed to the key pair that is no longer available. This leaves the Root CA with no alternative but to deploy a new self-signed certificate that contains a newly-generated key pair in the same manner as the initial self-signed certificate, thus losing the benefits of the Hash Of Root Key certificate extension altogether.

6. Security Considerations

The security considerations from [RFC5280] apply, especially the discussion of self-issued certificates.

The Hash Of Root Key certificate extension facilitates the orderly transition from one Root CA public key to the next by publishing the hash value of the next generation public key in the current certificate. This allows a relying party to unambiguously recognize the next generation public key when it becomes available; however, the full public key is not disclosed until the Root CA releases the next generation certificate. In this way, attackers cannot begin to analyze the public key before the next generation Root CA self-signed certificate is released.

The Root CA needs to ensure that the public key in the next generation certificate is as strong or stronger than the key that it is replacing. Of course, a significant advance in cryptoanalytic capability can break the yet-to-be-deployed key pair. Such advances are rare and difficult to predict. If such an advance occurs, the Root CA remains committed to the now broken key. This leaves the Root CA with no alternative but to deploy a new self-signed certificate that contains a newly-generated key pair, most likely using a different signature algorithm, in the same manner as the initial self-signed certificate, thus losing the benefits of the Hash Of Root Key certificate extension altogether.

The Root CA needs to employ a hash function that is resistant to preimage attacks [RFC4270]. A first-preimage attack against the hash function would allow an attacker to find another input that results published hash value. For the attack to be successful, the input would have to be a valid SubjectPublicKeyInfo that contains a public key that corresponds to a private key known to the attacker. A second-preimage attack becomes possible once the Root CA releases the next generation public key, which makes the input to the hash function available to the attacker and everyone else. Again, the attacker needs to find a valid SubjectPublicKeyInfo that contains the public key that corresponds to a private key known to the attacker. If the employed hash function is broken after the Root CA publishes the self-signed certificate with the HashOfRootKey certificate extension, an attacker would be able to trick the recipient into installing the incorrect next generation certificate in the trust anchor store.

If an early release of the next generation public key occurs and the Root CA is concerned that attackers were given too much lead time to analyze that public key, then the Root CA can transition to a freshly generated key pair by rapidly performing two transitions. The first transition takes the Root CA to the key pair that suffered the early release, and it causes the Root CA to generate the subsequent Root key pair. The second transition occurs when the Root CA is confident that the population of relying parties have completed the first transition, and it takes the Root CA to the freshly generated key pair. Of course, the second transition also causes the Root CA to generate another key pair that is reserved for future use. Queries for the CRLs associated with certificates that are subordinate to the self-signed certificate can give some indication for the number of relying parties that are still actively using the self-signed certificates.

7. Acknowledgements

The Secure Electronic Transaction (SET) [SET] specification published by MasterCard and VISA in 1997 includes a very similar certificate extension. The SET certificate extension has essentially the same semantics, but the syntax fairly different.

CTIA - The Wireless Association - is developing a public key infrastructure that will make use of the certificate extension described in this document, and the object identifiers used in the ASN.1 module were assigned by CTIA.

Many thanks to Stefan Santesson, Jim Schaad, Daniel Kahn Gillmor, Joel Halpern, Paul Hoffman, Rich Salz, and Ben Kaduk. Their review and comments have greatly improved the document, especially the Operational Considerations and Security Considerations sections.

8. References

8.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4210] Adams, C., Farrell, S., Kause, T., and T. Mononen, "Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)", RFC 4210, DOI 10.17487/RFC4210, September 2005, <https://www.rfc-editor.org/info/rfc4210>.

[RFC4270] Hoffman, P. and B. Schneier, "Attacks on Cryptographic Hashes in Internet Protocols", RFC 4270, DOI 10.17487/RFC4270, November 2005, <https://www.rfc-editor.org/info/rfc4270>.

[RFC5272] Schaad, J. and M. Myers, "Certificate Management over CMS (CMC)", RFC 5272, DOI 10.17487/RFC5272, June 2008, <https://www.rfc-editor.org/info/rfc5272>.

[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, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <https://www.rfc-editor.org/info/rfc5912>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[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.

8.2. Informative References

[SET] MasterCard and VISA, "SET Secure Electronic Transaction Specification -- Book 2: Programmer's Guide, Version 1.0", May 1997.

Appendix A. ASN.1 Module

The following ASN.1 module provides the complete definition of the HashOfRootKey certificate extension.

```
   HashedRootKeyCertExtn { 1 3 6 1 4 1 51483 0 1 }

   DEFINITIONS IMPLICIT TAGS ::=
   BEGIN

   -- EXPORTS All

   IMPORTS

   HashAlgorithm
     FROM PKIX1-PSS-OAEP-Algorithms-2009  -- [RFC5912]
          { iso(1) identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) id-mod(0)
            id-mod-pkix1-rsa-pkalgs-02(54) }

   EXTENSION
     FROM PKIX-CommonTypes-2009
       { iso(1) identified-organization(3) dod(6) internet(1)
         security(5) mechanisms(5) pkix(7) id-mod(0)
         id-mod-pkixCommon-02(57) } ;

   --
   -- Expand the certificate extensions list in [RFC5912]
   --

   CertExtensions EXTENSION ::= {
      ext-HashOfRootKey, ... }

   --
   -- HashOfRootKey Certificate Extension
   --

   ext-HashOfRootKey EXTENSION ::= {    -- Only in Root CA certificates
      SYNTAX          HashedRootKey
      IDENTIFIED BY  id-ce-hashOfRootKey
      CRITICALITY    {FALSE} }

   HashedRootKey  ::=  SEQUENCE {
      hashAlg        HashAlgorithm,     -- Hash algorithm used
      hashValue      OCTET STRING }     -- Hash of DER-encoded
                                        --   SubjectPublicKeyInfo

   id-ce-hashOfRootKey OBJECT IDENTIFIER  ::=  { 1 3 6 1 4 1 51483 2 1 }

   END
```

Author's Address

Russ Housley
Vigil Security
516 Dranesville Road
Herndon, VA 20170
US

Email: housley@vigilsec.com

LAMPS WG P. Kampanakis
Internet-Draft Cisco Systems
Updates: 3279 (if approved) Q. Dang
Intended status: Standards Track NIST
Expires: January 22, 2020 July 21, 2019

Internet X.509 Public Key Infrastructure: Additional Algorithm Identifiers for RSASSA-PSS and ECDSA using SHAKEs
draft-ietf-lamps-pkix-shake-15

Abstract

Digital signatures are used to sign messages, X.509 certificates and CRLs. This document updates the "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List Profile" (RFC3279) and describes the conventions for using the SHAKE function family in Internet X.509 certificates and revocation lists as one-way hash functions with the RSA Probabilistic signature and ECDSA signature algorithms. The conventions for the associated subject public keys are also described.

Status of This Memo

Copyright Notice

Table of Contents

1. Change Log

[EDNOTE: Remove this section before publication.]

- draft-ietf-lamps-pkix-shake-15:

  * Minor editorial nits.

- draft-ietf-lamps-pkix-shake-14:

  * Fixing error with incorrect preimage resistance bits for SHA128 and SHA256.

- draft-ietf-lamps-pkix-shake-13:

  * Addressing one applicable comment from Dan M. about sec levels while in secdir review of draft-ietf-lamps-cms-shakes.

  * Addressing comment from Scott B.'s opsdir review about references in the abstract.

- draft-ietf-lamps-pkix-shake-12:

  * Nits identified by Roman, Eric V. Ben K., Barry L. in ballot position review.

- draft-ietf-lamps-pkix-shake-11:

  * Nits identified by Roman in AD Review.

- draft-ietf-lamps-pkix-shake-10:

  * Updated IANA considerations section to request for OID assignments.

- draft-ietf-lamps-pkix-shake-09:

  * Fixed minor text nits.

  * Added text name allocation for SHAKEs in IANA considerations.

  * Updates in Sec Considerations section.

- draft-ietf-lamps-pkix-shake-08:

  * Small nits from Russ while in WGLC.

- draft-ietf-lamps-pkix-shake-07:

  * Incorporated Eric's suggestion from WGLC.

- draft-ietf-lamps-pkix-shake-06:

  * Added informative references.

  * Updated ASN.1 so it compiles.

  * Updated IANA considerations.

- draft-ietf-lamps-pkix-shake-05:

  * Added RFC8174 reference and text.

  * Explicitly explained why RSASSA-PSS-params are omitted in section 5.1.1.

  * Simplified Public Keys section by removing redundant info from RFCs.

- draft-ietf-lamps-pkix-shake-04:

   * Removed paragraph suggesting KMAC to be used in generating k in Deterministic ECDSA. That should be RFC6979-bis.

   * Removed paragraph from Security Considerations that talks about randomness of k because we are using deterministic ECDSA.

   * Various ASN.1 fixes.

   * Text fixes.

- draft-ietf-lamps-pkix-shake-03:

   * Updates based on suggestions and clarifications by Jim.

   * Added ASN.1.

- draft-ietf-lamps-pkix-shake-02:

   * Significant reorganization of the sections to simplify the introduction, the new OIDs and their use in PKIX.

   * Added new OIDs for RSASSA-PSS that hardcode hash, salt and MGF, according the WG consensus.

   * Updated Public Key section to use the new RSASSA-PSS OIDs and clarify the algorithm identifier usage.

   * Removed the no longer used SHAKE OIDs from section 3.1.

   * Consolidated subsection for message digest algorithms.

   * Text fixes.

- draft-ietf-lamps-pkix-shake-01:

   * Changed titles and section names.

   * Removed DSA after WG discussions.

   * Updated shake OID names and parameters, added MGF1 section.

   * Updated RSASSA-PSS section.

   * Added Public key algorithm OIDs.

   * Populated Introduction and IANA sections.

- draft-ietf-lamps-pkix-shake-00:

* Initial version

2. Introduction

[RFC3279] defines cryptographic algorithm identifiers for the Internet X.509 Certificate and Certificate Revocation Lists (CRL) profile [RFC5280]. This document updates RFC3279 and defines identifiers for several cryptographic algorithms that use variable length output SHAKE functions introduced in [SHA3] which can be used with .

In the SHA-3 family, two extendable-output functions (SHAKEs), SHAKE128 and SHAKE256, are defined. Four other hash function instances, SHA3-224, SHA3-256, SHA3-384, and SHA3-512, are also defined but are out of scope for this document. A SHAKE is a variable length hash function defined as SHAKE(M, d) where the output is a d-bits-long digest of message M. The corresponding collision and second-preimage-resistance strengths for SHAKE128 are min(d/2,128) and min(d,128) bits, respectively (Appendix A.1 [SHA3]). And the corresponding collision and second-preimage-resistance strengths for SHAKE256 are min(d/2,256) and min(d,256) bits, respectively.

A SHAKE can be used as the message digest function (to hash the message to be signed) in RSASSA-PSS [RFC8017] and ECDSA [X9.62] and as the hash in the mask generation function (MGF) in RSASSA-PSS.

3. 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.

4. Identifiers

This section defines four new object identifiers (OIDs), for RSASSA-PSS and ECDSA with each of SHAKE128 and SHAKE256. The same algorithm identifiers can be used for identifying a public key in RSASSA-PSS.

The new identifiers for RSASSA-PSS signatures using SHAKEs are below.

```
id-RSASSA-PSS-SHAKE128  OBJECT IDENTIFIER  ::=  { iso(1)
          identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) algorithms(6)
          TBD1 }

id-RSASSA-PSS-SHAKE256  OBJECT IDENTIFIER  ::=  { iso(1)
          identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) algorithms(6)
          TBD2 }
```

The new algorithm identifiers of ECDSA signatures using SHAKEs are below.

```
id-ecdsa-with-shake128 OBJECT IDENTIFIER  ::=  { iso(1)
          identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) algorithms(6)
          TBD3 }

id-ecdsa-with-shake256 OBJECT IDENTIFIER  ::=  { iso(1)
          identified-organization(3) dod(6) internet(1)
          security(5) mechanisms(5) pkix(7) algorithms(6)
          TBD4 }
```

The parameters for the four identifiers above MUST be absent. That is, the identifier SHALL be a SEQUENCE of one component, the OID.

Section 5.1.1 and Section 5.1.2 specify the required output length for each use of SHAKE128 or SHAKE256 in RSASSA-PSS and ECDSA. In summary, when hashing messages to be signed, output lengths of SHAKE128 and SHAKE256 are 256 and 512 bits respectively. When the SHAKEs are used as mask generation functions RSASSA-PSS, their output length is (8*ceil((n-1)/8) - 264) or (8*ceil((n-1)/8) - 520) bits, respectively, where n is the RSA modulus size in bits.

5. Use in PKIX

5.1. Signatures

Signatures are used in a number of different ASN.1 structures. As shown in the ASN.1 representation from [RFC5280] below, in an X.509 certificate, a signature is encoded with an algorithm identifier in the signatureAlgorithm attribute and a signatureValue attribute that contains the actual signature.

```
Certificate  ::=  SEQUENCE  {
   tbsCertificate       TBSCertificate,
   signatureAlgorithm   AlgorithmIdentifier,
   signatureValue       BIT STRING  }
```

The identifiers defined in Section 4 can be used as the AlgorithmIdentifier in the signatureAlgorithm field in the sequence Certificate and the signature field in the sequence TBSCertificate in X.509 [RFC5280]. The parameters of these signature algorithms are absent as explained in Section 4.

Conforming CA implementations MUST specify the algorithms explicitly by using the OIDs specified in Section 4 when encoding RSASSA-PSS or ECDSA with SHAKE signatures in certificates and CRLs. Conforming client implementations that process certificates and CRLs using RSASSA-PSS or ECDSA with SHAKE MUST recognize the corresponding OIDs. Encoding rules for RSASSA-PSS and ECDSA signature values are specified in [RFC4055] and [RFC5480], respectively.

When using RSASSA-PSS or ECDSA with SHAKEs, the RSA modulus and ECDSA curve order SHOULD be chosen in line with the SHAKE output length. Refer to Section 7 for more details.

5.1.1. RSASSA-PSS Signatures

The RSASSA-PSS algorithm is defined in [RFC8017]. When id-RSASSA-PSS-SHAKE128 or id-RSASSA-PSS-SHAKE256 specified in Section 4 is used, the encoding MUST omit the parameters field. That is, the AlgorithmIdentifier SHALL be a SEQUENCE of one component, id-RSASSA-PSS-SHAKE128 or id-RSASSA-PSS-SHAKE256. [RFC4055] defines RSASSA-PSS-params that are used to define the algorithms and inputs to the algorithm. This specification does not use parameters because the hash, mask generation algorithm, trailer and salt are embedded in the OID definition.

The hash algorithm to hash a message being signed and the hash algorithm used as the mask generation function in RSASSA-PSS MUST be the same: both SHAKE128 or both SHAKE256. The output length of the hash algorithm which hashes the message SHALL be 32 (for SHAKE128) or 64 bytes (for SHAKE256).

The mask generation function takes an octet string of variable length and a desired output length as input, and outputs an octet string of the desired length. In RSASSA-PSS with SHAKEs, the SHAKEs MUST be used natively as the MGF function, instead of the MGF1 algorithm that uses the hash function in multiple iterations as specified in Section B.2.1 of [RFC8017]. In other words, the MGF is defined as the SHAKE128 or SHAKE256 output of the mgfSeed for id-RSASSA-PSS-

SHAKE128 and id-RSASSA-PSS-SHAKE256, respectively. The mgfSeed is the seed from which mask is generated, an octet string [RFC8017]. As explained in Step 9 of section 9.1.1 of [RFC8017], the output length of the MGF is emLen - hLen - 1 bytes. emLen is the maximum message length ceil((n-1)/8), where n is the RSA modulus in bits. hLen is 32 and 64-bytes for id-RSASSA-PSS-SHAKE128 and id-RSASSA-PSS-SHAKE256, respectively. Thus when SHAKE is used as the MGF, the SHAKE output length maskLen is (8*emLen - 264) or (8*emLen - 520) bits, respectively. For example, when RSA modulus n is 2048, the output length of SHAKE128 or SHAKE256 as the MGF will be 1784 or 1528-bits when id-RSASSA-PSS-SHAKE128 or id-RSASSA-PSS-SHAKE256 is used, respectively.

The RSASSA-PSS saltLength MUST be 32 bytes for id-RSASSA-PSS-SHAKE128 or 64 bytes for id-RSASSA-PSS-SHAKE256. Finally, the trailerField MUST be 1, which represents the trailer field with hexadecimal value 0xBC [RFC8017].

5.1.2. ECDSA Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) is defined in [X9.62]. When the id-ecdsa-with-shake128 or id-ecdsa-with-shake256 (specified in Section 4) algorithm identifier appears, the respective SHAKE function (SHAKE128 or SHAKE256) is used as the hash. The encoding MUST omit the parameters field. That is, the AlgorithmIdentifier SHALL be a SEQUENCE of one component, the OID id-ecdsa-with-shake128 or id-ecdsa-with-shake256.

For simplicity and compliance with the ECDSA standard specification, the output length of the hash function must be explicitly determined. The output length, d, for SHAKE128 or SHAKE256 used in ECDSA MUST be 256 or 512 bits, respectively.

Conforming CA implementations that generate ECDSA with SHAKE signatures in certificates or CRLs SHOULD generate such signatures with a deterministically generated, non-random k in accordance with all the requirements specified in [RFC6979]. They MAY also generate such signatures in accordance with all other recommendations in [X9.62] or [SEC1] if they have a stated policy that requires conformance to those standards. Those standards have not specified SHAKE128 and SHAKE256 as hash algorithm options. However, SHAKE128 and SHAKE256 with output length being 32 and 64 octets, respectively, can be used instead of 256 and 512-bit output hash algorithms such as SHA256 and SHA512.

5.2. Public Keys

Certificates conforming to [RFC5280] can convey a public key for any public key algorithm. The certificate indicates the public key algorithm through an algorithm identifier. This algorithm identifier is an OID and optionally associated parameters. The conventions and encoding for RSASSA-PSS and ECDSA public keys algorithm identifiers are as specified in Section 2.3.1 and 2.3.5 of [RFC3279], Section 3.1 of [RFC4055] and Section 2.1 of [RFC5480].

Traditionally, the rsaEncryption object identifier is used to identify RSA public keys. The rsaEncryption object identifier continues to identify the subject public key when the RSA private key owner does not wish to limit the use of the public key exclusively to RSASSA-PSS with SHAKEs. When the RSA private key owner wishes to limit the use of the public key exclusively to RSASSA-PSS with SHAKEs, the AlgorithmIdentifiers for RSASSA-PSS defined in Section 4 SHOULD be used as the algorithm field in the SubjectPublicKeyInfo sequence [RFC5280]. Conforming client implementations that process RSASSA-PSS with SHAKE public keys when processing certificates and CRLs MUST recognize the corresponding OIDs.

Conforming CA implementations MUST specify the X.509 public key algorithm explicitly by using the OIDs specified in Section 4 when encoding ECDSA with SHAKE public keys in certificates and CRLs. Conforming client implementations that process ECDSA with SHAKE public keys when processing certificates and CRLs MUST recognize the corresponding OIDs.

The identifier parameters, as explained in Section 4, MUST be absent.

6. IANA Considerations

One object identifier for the ASN.1 module in Appendix A is requested for the SMI Security for PKIX Module Identifiers (1.3.6.1.5.5.7.0) registry:

Decimal	Description	References
TBD	id-mod-pkix1-shakes-2019	[EDNOTE: THIS RFC]

IANA is requested to update the SMI Security for PKIX Algorithms [SMI-PKIX] (1.3.6.1.5.5.7.6) registry with four additional entries:

Decimal	Description	References
TBD1	id-RSASSA-PSS-SHAKE128	[EDNOTE: THIS RFC]
TBD2	id-RSASSA-PSS-SHAKE256	[EDNOTE: THIS RFC]
TBD3	id-ecdsa-with-shake128	[EDNOTE: THIS RFC]
TBD4	id-ecdsa-with-shake256	[EDNOTE: THIS RFC]

IANA is also requested to update the Hash Function Textual Names Registry [Hash-Texts] with two additional entries for SHAKE128 and SHAKE256:

Hash Function Name	OID	Reference
shake128	2.16.840.1.101.3.4.2.11	[EDNOTE: THIS RFC]
shake256	2.16.840.1.101.3.4.2.12	[EDNOTE: THIS RFC]

7. Security Considerations

This document updates [RFC3279]. The security considerations section of that document applies to this specification as well.

NIST has defined appropriate use of the hash functions in terms of the algorithm strengths and expected time frames for secure use in Special Publications (SPs) [SP800-78-4] and [SP800-107]. These documents can be used as guides to choose appropriate key sizes for various security scenarios.

SHAKE128 with output length of 256-bits offers 128-bits of collision and preimage resistance. Thus, SHAKE128 OIDs in this specification are RECOMMENDED with 2048 (112-bit security) or 3072-bit (128-bit security) RSA modulus or curves with group order of 256-bits (128-bit security). SHAKE256 with 512-bits output length offers 256-bits of collision and preimage resistance. Thus, the SHAKE256 OIDs in this specification are RECOMMENDED with 4096-bit RSA modulus or higher or curves with group order of at least 521-bits (256-bit security). Note that we recommended 4096-bit RSA because we would need 15360-bit modulus for 256-bits of security which is impractical for today’s technology.

8. Acknowledgements

We would like to thank Sean Turner, Jim Schaad and Eric Rescorla for their valuable contributions to this document.

The authors would like to thank Russ Housley for his guidance and very valuable contributions with the ASN.1 module.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, DOI 10.17487/RFC3279, April 2002, <https://www.rfc-editor.org/info/rfc3279>.

[RFC4055] Schaad, J., Kaliski, B., and R. Housley, "Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 4055, DOI 10.17487/RFC4055, June 2005, <https://www.rfc-editor.org/info/rfc4055>.

[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, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/info/rfc5480>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[SHA3] National Institute of Standards and Technology (NIST), "SHA-3 Standard - Permutation-Based Hash and Extendable-Output Functions FIPS PUB 202", August 2015, <https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions>.

9.2. Informative References

[Hash-Texts]
IANA, "Hash Function Textual Names", July 2017, <https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xhtml>.

[RFC5912] Hoffman, P. and J. Schaad, "New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)", RFC 5912, DOI 10.17487/RFC5912, June 2010, <https://www.rfc-editor.org/info/rfc5912>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009, <http://www.secg.org/sec1-v2.pdf>.

[SMI-PKIX]
IANA, "SMI Security for PKIX Algorithms", March 2019, <https://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml#smi-numbers-1.3.6.1.5.5.7.6>.

[SP800-107]
National Institute of Standards and Technology (NIST), "SP800-107: Recommendation for Applications Using Approved Hash Algorithms", May 2014, <https://csrc.nist.gov/csrc/media/publications/sp/800-107/rev-1/final/documents/draft_revised_sp800-107.pdf>.

[SP800-78-4]
National Institute of Standards and Technology (NIST), "SP800-78-4: Cryptographic Algorithms and Key Sizes for Personal Identity Verification", May 2014, <https://csrc.nist.gov/csrc/media/publications/sp/800-78/4/final/documents/sp800_78-4_revised_draft.pdf>.

[X9.62] American National Standard for Financial Services (ANSI), "X9.62-2005: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Standard (ECDSA)", November 2005.

Appendix A. ASN.1 module

This appendix includes the ASN.1 module for SHAKEs in X.509. This module does not come from any existing RFC.

```
 PKIXAlgsForSHAKE-2019 { iso(1) identified-organization(3) dod(6)
   internet(1) security(5) mechanisms(5) pkix(7) id-mod(0)
   id-mod-pkix1-shakes-2019(TBD) }

 DEFINITIONS EXPLICIT TAGS ::=

 BEGIN

 -- EXPORTS ALL;

 IMPORTS

 -- FROM [RFC5912]

 PUBLIC-KEY, SIGNATURE-ALGORITHM, DIGEST-ALGORITHM, SMIME-CAPS
 FROM AlgorithmInformation-2009
   { iso(1) identified-organization(3) dod(6) internet(1) security(5)
     mechanisms(5) pkix(7) id-mod(0)
     id-mod-algorithmInformation-02(58) }

 -- FROM [RFC5912]

 RSAPublicKey, rsaEncryption, pk-rsa, pk-ec,
 CURVE, id-ecPublicKey, ECPoint, ECParameters, ECDSA-Sig-Value
 FROM PKIXAlgs-2009 { iso(1) identified-organization(3) dod(6)
      internet(1) security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkix1-algorithms2008-02(56) }
;

 --
 -- Message Digest Algorithms (mda-)
 --
 DigestAlgorithms DIGEST-ALGORITHM ::= {
   -- This expands DigestAlgorithms from [RFC5912]
   mda-shake128    |
   mda-shake256,
   ...
 }
```

```
   --
   -- One-Way Hash Functions
   --

   -- SHAKE128
   mda-shake128 DIGEST-ALGORITHM ::= {
     IDENTIFIER id-shake128  -- with output length 32 bytes.
   }
   id-shake128 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16)
                                       us(840) organization(1) gov(101)
                                       csor(3) nistAlgorithm(4)
                                       hashAlgs(2) 11 }

   -- SHAKE256
   mda-shake256 DIGEST-ALGORITHM ::= {
     IDENTIFIER id-shake256  -- with output length 64 bytes.
   }
   id-shake256 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16)
                                       us(840) organization(1) gov(101)
                                       csor(3) nistAlgorithm(4)
                                       hashAlgs(2) 12 }

   --
   -- Public Key (pk-) Algorithms
   --
   PublicKeys PUBLIC-KEY ::= {
     -- This expands PublicKeys from [RFC5912]
     pk-rsaSSA-PSS-SHAKE128 |
     pk-rsaSSA-PSS-SHAKE256,
     ...
   }

   -- The hashAlgorithm is mda-shake128
   -- The maskGenAlgorithm is id-shake128
   -- Mask Gen Algorithm is SHAKE128 with output length
   -- (8*ceil((n-1)/8) - 264) bits, where n is the RSA
   -- modulus in bits.
   -- The saltLength is 32. The trailerField is 1.
   pk-rsaSSA-PSS-SHAKE128 PUBLIC-KEY ::= {
     IDENTIFIER id-RSASSA-PSS-SHAKE128
     KEY RSAPublicKey
     PARAMS ARE absent
     -- Private key format not in this module --
     CERT-KEY-USAGE { nonRepudiation, digitalSignature,
                      keyCertSign, cRLSign }
   }

   -- The hashAlgorithm is mda-shake256
```

```
    -- The maskGenAlgorithm is id-shake256
    -- Mask Gen Algorithm is SHAKE256 with output length
    -- (8*ceil((n-1)/8) - 520)-bits, where n is the RSA
    -- modulus in bits.
    -- The saltLength is 64. The trailerField is 1.
    pk-rsaSSA-PSS-SHAKE256 PUBLIC-KEY ::= {
      IDENTIFIER id-RSASSA-PSS-SHAKE256
      KEY RSAPublicKey
      PARAMS ARE absent
      -- Private key format not in this module --
      CERT-KEY-USAGE { nonRepudiation, digitalSignature,
                       keyCertSign, cRLSign }
    }

    --
    -- Signature Algorithms (sa-)
    --
    SignatureAlgs SIGNATURE-ALGORITHM ::= {
      -- This expands SignatureAlgorithms from [RFC5912]
      sa-rsassapssWithSHAKE128 |
      sa-rsassapssWithSHAKE256 |
      sa-ecdsaWithSHAKE128 |
      sa-ecdsaWithSHAKE256,
      ...
    }

    --
    -- SMIME Capabilities (sa-)
    --
    SMimeCaps SMIME-CAPS ::= {
      -- The expands SMimeCaps from [RFC5912]
      sa-rsassapssWithSHAKE128.&smimeCaps |
      sa-rsassapssWithSHAKE256.&smimeCaps |
      sa-ecdsaWithSHAKE128.&smimeCaps |
      sa-ecdsaWithSHAKE256.&smimeCaps,
      ...
    }

    -- RSASSA-PSS with SHAKE128
    sa-rsassapssWithSHAKE128 SIGNATURE-ALGORITHM ::= {
      IDENTIFIER id-RSASSA-PSS-SHAKE128
      PARAMS ARE absent
          -- The hashAlgorithm is mda-shake128
          -- The maskGenAlgorithm is id-shake128
          -- Mask Gen Algorithm is SHAKE128 with output length
          -- (8*ceil((n-1)/8) - 264) bits, where n is the RSA
          -- modulus in bits.
          -- The saltLength is 32. The trailerField is 1
```

```
      HASHES { mda-shake128 }
      PUBLIC-KEYS { pk-rsa | pk-rsaSSA-PSS-SHAKE128 }
      SMIME-CAPS { IDENTIFIED BY id-RSASSA-PSS-SHAKE128 }
    }
    id-RSASSA-PSS-SHAKE128  OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6)
            TBD1 }

    -- RSASSA-PSS with SHAKE256
    sa-rsassapssWithSHAKE256 SIGNATURE-ALGORITHM ::= {
      IDENTIFIER id-RSASSA-PSS-SHAKE256
      PARAMS ARE absent
          -- The hashAlgorithm is mda-shake256
          -- The maskGenAlgorithm is id-shake256
          -- Mask Gen Algorithm is SHAKE256 with output length
          -- (8*ceil((n-1)/8) - 520)-bits, where n is the
          -- RSA modulus in bits.
          -- The saltLength is 64. The trailerField is 1.
     HASHES { mda-shake256 }
     PUBLIC-KEYS { pk-rsa | pk-rsaSSA-PSS-SHAKE256 }
     SMIME-CAPS { IDENTIFIED BY id-RSASSA-PSS-SHAKE256 }
    }
    id-RSASSA-PSS-SHAKE256  OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6)
            TBD2 }

    -- ECDSA with SHAKE128
    sa-ecdsaWithSHAKE128 SIGNATURE-ALGORITHM ::= {
      IDENTIFIER id-ecdsa-with-shake128
      VALUE ECDSA-Sig-Value
      PARAMS ARE absent
      HASHES { mda-shake128 }
      PUBLIC-KEYS { pk-ec }
      SMIME-CAPS { IDENTIFIED BY id-ecdsa-with-shake128 }
    }
    id-ecdsa-with-shake128 OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6)
            TBD3 }

    -- ECDSA with SHAKE256
    sa-ecdsaWithSHAKE256 SIGNATURE-ALGORITHM ::= {
      IDENTIFIER id-ecdsa-with-shake256
      VALUE ECDSA-Sig-Value
      PARAMS ARE absent
      HASHES { mda-shake256 }
```

```
      PUBLIC-KEYS { pk-ec }
      SMIME-CAPS { IDENTIFIED BY id-ecdsa-with-shake256 }
    }
    id-ecdsa-with-shake256 OBJECT IDENTIFIER  ::=  { iso(1)
            identified-organization(3) dod(6) internet(1)
            security(5) mechanisms(5) pkix(7) algorithms(6)
            TBD4 }

    END
```

Authors' Addresses

Panos Kampanakis
Cisco Systems

Email: pkampana@cisco.com

Quynh Dang
NIST
100 Bureau Drive, Stop 8930
Gaithersburg, MD 20899-8930
USA

Email: quynh.dang@nist.gov

Network Working Group
Internet-Draft
Obsoletes: 6844 (if approved)
Intended status: Standards Track
Expires: December 1, 2019

P. Hallam-Baker

R. Stradling
Sectigo
J. Hoffman-Andrews
Let's Encrypt
May 30, 2019

DNS Certification Authority Authorization (CAA) Resource Record
draft-ietf-lamps-rfc6844bis-07

Abstract

The Certification Authority Authorization (CAA) DNS Resource Record allows a DNS domain name holder to specify one or more Certification Authorities (CAs) authorized to issue certificates for that domain name. CAA Resource Records allow a public Certification Authority to implement additional controls to reduce the risk of unintended certificate mis-issue. This document defines the syntax of the CAA record and rules for processing CAA records by certificate issuers.

This document obsoletes RFC 6844.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Certification Authority Authorization (CAA) DNS Resource Record allows a DNS domain name holder to specify the Certification Authorities (CAs) authorized to issue certificates for that domain name. Publication of CAA Resource Records allows a public

Certification Authority to implement additional controls to reduce the risk of unintended certificate mis-issue.

Like the TLSA record defined in DNS-Based Authentication of Named Entities (DANE) [RFC6698], CAA records are used as a part of a mechanism for checking PKIX [RFC6698] certificate data. The distinction between the two specifications is that CAA records specify an authorization control to be performed by a certificate issuer before issue of a certificate and TLSA records specify a verification control to be performed by a relying party after the certificate is issued.

Conformance with a published CAA record is a necessary but not sufficient condition for issuance of a certificate.

Criteria for inclusion of embedded trust anchor certificates in applications are outside the scope of this document. Typically, such criteria require the CA to publish a Certification Practices Statement (CPS) that specifies how the requirements of the Certificate Policy (CP) are achieved. It is also common for a CA to engage an independent third-party auditor to prepare an annual audit statement of its performance against its CPS.

A set of CAA records describes only current grants of authority to issue certificates for the corresponding DNS domain name. Since certificates are valid for a period of time, it is possible that a certificate that is not conformant with the CAA records currently published was conformant with the CAA records published at the time that the certificate was issued. Relying parties MUST NOT use CAA records as part of certificate validation.

CAA records MAY be used by Certificate Evaluators as a possible indicator of a security policy violation. Such use SHOULD take account of the possibility that published CAA records changed between the time a certificate was issued and the time at which the certificate was observed by the Certificate Evaluator.

2. Definitions

2.1. Requirements Language

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.

2.2. Defined Terms

The following terms are used in this document:

Certificate: An X.509 Certificate, as specified in [RFC5280].

Certificate Evaluator: A party other than a Relying Party that evaluates the trustworthiness of certificates issued by Certification Authorities.

Certification Authority (CA): An Issuer that issues certificates in accordance with a specified Certificate Policy.

Certificate Policy (CP): Specifies the criteria that a Certification Authority undertakes to meet in its issue of certificates. See [RFC3647].

Certification Practices Statement (CPS): Specifies the means by which the criteria of the Certificate Policy are met. In most cases, this will be the document against which the operations of the Certification Authority are audited. See [RFC3647].

Domain Name: The label assigned to a node in the Domain Name System.

Domain Name System (DNS): The Internet naming system specified in [RFC1034] and [RFC1035].

DNS Security (DNSSEC): Extensions to the DNS that provide authentication services as specified in [RFC4033], [RFC4034], [RFC4035], [RFC5155], and revisions.

Fully-Qualified Domain Name (FQDN): A Domain Name that includes the labels of all superior nodes in the Domain Name System.

Issuer: An entity that issues certificates. See [RFC5280].

Property: The tag-value portion of a CAA Resource Record.

Property Tag: The tag portion of a CAA Resource Record.

Property Value: The value portion of a CAA Resource Record.

Resource Record (RR): A particular entry in the DNS including the owner name, class, type, time to live, and data, as defined in [RFC1034] and [RFC2181].

Resource Record Set (RRSet): A set of Resource Records of a particular owner name, class, and type. The time to live on all RRs within an RRSet is always the same, but the data may be different among RRs in the RRSet.

Relevant Resource Record Set (Relevant RRSet): A set of CAA Resource Records resulting from applying the algorithm in Section 3 to a specific Fully-Qualified Domain Name or Wildcard Domain Name.

Relying Party: A party that makes use of an application whose operation depends on use of a certificate for making a security decision. See [RFC5280].

Wildcard Domain Name: A Domain Name consisting of a single asterisk character followed by a single full stop character ("*.") followed by a Fully-Qualified Domain Name.

3. Relevant Resource Record Set

Before issuing a certificate, a compliant CA MUST check for publication of a Relevant RRSet. If such an RRSet exists, a CA MUST NOT issue a certificate unless the CA determines that either (1) the certificate request is consistent with the applicable CAA Resource Record set or (2) an exception specified in the relevant Certificate Policy or Certification Practices Statement applies. If the Relevant RRSet for a Fully-Qualified Domain Name or Wildcard Domain Name contains no Property Tags that restrict issuance (for instance, if it contains only iodef Property Tags, or only Property Tags unrecognized by the CA), CAA does not restrict issuance.

A certificate request MAY specify more than one Fully-Qualified Domain Name and MAY specify Wildcard Domain Names. Issuers MUST verify authorization for all the Fully-Qualified Domain Names and Wildcard Domain Names specified in the request.

The search for a CAA RRSet climbs the DNS name tree from the specified label up to but not including the DNS root '.' until a CAA RRSet is found.

Given a request for a specific Fully-Qualified Domain Name X, or a request for a Wildcard Domain Name *.X, the Relevant Resource Record Set RelevantCAASet(X) is determined as follows (in pseudocode):

Let CAA(X) be the RRSet returned by performing a CAA record query for the Fully-Qualified Domain Name X, according to the lookup algorithm specified in RFC 1034 section 4.3.2 (in particular chasing aliases). Let Parent(X) be the Fully-Qualified Domain Name produced by removing the leftmost label of X.

```
RelevantCAASet(domain):
  while domain is not ".":
    if CAA(domain) is not Empty:
      return CAA(domain)
    domain = Parent(domain)
  return Empty
```

For example, processing CAA for the Fully-Qualified Domain Name "X.Y.Z" where there are no CAA records at any level in the tree RelevantCAASet would have the following steps:

```
CAA("X.Y.Z.") = Empty; domain = Parent("X.Y.Z.") = "Y.Z."
CAA("Y.Z.")   = Empty; domain = Parent("Y.Z.")   = "Z."
CAA("Z.")     = Empty; domain = Parent("Z.")     = "."
return Empty
```

Processing CAA for the Fully-Qualified Domain Name "A.B.C" where there is a CAA record "issue example.com" at "B.C" would terminate early upon finding the CAA record:

```
CAA("A.B.C.") = Empty; domain = Parent("A.B.C.") = "B.C."
CAA("B.C.")   = "issue example.com"
return "issue example.com"
```

4. Mechanism

4.1. Syntax

A CAA Resource Record contains a single Property consisting of a tag-value pair. A Fully-Qualified Domain Name MAY have multiple CAA RRs associated with it and a given Property Tag MAY be specified more than once across those RRs.

The RDATA section for a CAA Resource Record contains one Property. A Property consists of the following:

```
+0-1-2-3-4-5-6-7-|0-1-2-3-4-5-6-7-|
| Flags          | Tag Length = n |
+----------------|----------------+...+---------------+
| Tag char 0     | Tag char 1     |...| Tag char n-1  |
+----------------|----------------+...+---------------+
+----------------|----------------+.....+----------------+
| Value byte 0   | Value byte 1   |.....| Value byte m-1 |
+----------------|----------------+.....+----------------+
```

Where n is the length specified in the Tag length field and m is the remaining octets in the Value field. They are related by (m = d - n - 2) where d is the length of the RDATA section.

The fields are defined as follows:

Flags: One octet containing the following field:

Bit 0, Issuer Critical Flag: If the value is set to '1', the Property is critical. A Certification Authority MUST NOT issue certificates for any FQDN the Relevant RRSet for that FQDN contains a CAA critical Property for an unknown or unsupported Property Tag.

Note that according to the conventions set out in [RFC1035], bit 0 is the Most Significant Bit and bit 7 is the Least Significant Bit. Thus, the Flags value 1 means that bit 7 is set while a value of 128 means that bit 0 is set according to this convention.

All other bit positions are reserved for future use.

To ensure compatibility with future extensions to CAA, DNS records compliant with this version of the CAA specification MUST clear (set to "0") all reserved flags bits. Applications that interpret CAA records MUST ignore the value of all reserved flag bits.

Tag Length: A single octet containing an unsigned integer specifying the tag length in octets. The tag length MUST be at least 1.

Tag: The Property identifier, a sequence of US-ASCII characters.

Tags MAY contain US-ASCII characters 'a' through 'z', 'A' through 'Z', and the numbers 0 through 9. Tags MUST NOT contain any other characters. Matching of tags is case insensitive.

Tags submitted for registration by IANA MUST NOT contain any characters other than the (lowercase) US-ASCII characters 'a' through 'z' and the numbers 0 through 9.

Value: A sequence of octets representing the Property Value. Property Values are encoded as binary values and MAY employ sub-formats.

The length of the value field is specified implicitly as the remaining length of the enclosing RDATA section.

4.1.1. Canonical Presentation Format

The canonical presentation format of the CAA record is:

CAA <flags> <tag> <value>

Where:

Flags: Is an unsigned integer between 0 and 255.

Tag: Is a non-zero-length sequence of US-ASCII letters and numbers in lower case.

Value: The value field, expressed as a contiguous set of characters without interior spaces, or as a quoted string. See the <character-string> format specified in [RFC1035], Section 5.1, but note that the value field contains no length byte and is not limited to 255 characters.

4.2. CAA issue Property

If the issue Property Tag is present in the Relevant RRSet for a Fully-Qualified Domain Name, it is a request that Issuers

1. Perform CAA issue restriction processing for the FQDN, and

2. Grant authorization to issue certificates containing that FQDN to the holder of the issuer-domain-name or a party acting under the explicit authority of the holder of the issuer-domain-name.

The CAA issue Property Value has the following sub-syntax (specified in ABNF as per [RFC5234]).

```
issue-value = *WSP [issuer-domain-name *WSP] [";" *WSP [parameters *WSP]]

issuer-domain-name = label *("." label)
label = (ALPHA / DIGIT) *( *("-") (ALPHA / DIGIT))

parameters = (parameter *WSP ";" *WSP parameters) / parameter
parameter = tag *WSP "=" *WSP value
tag = (ALPHA / DIGIT) *( *("-") (ALPHA / DIGIT))
value = *(%x21-3A / %x3C-7E)
```

For consistency with other aspects of DNS administration, FQDN values are specified in letter-digit-hyphen Label (LDH-Label) form.

The following CAA record set requests that no certificates be issued for the FQDN 'certs.example.com' by any Issuer other than ca1.example.net or ca2.example.org.

```
certs.example.com         CAA 0 issue "ca1.example.net"
certs.example.com         CAA 0 issue "ca2.example.org"
```

Because the presence of an issue Property Tag in the Relevant RRSet for an FQDN restricts issuance, FQDN owners can use an issue Property Tag with no issuer-domain-name to request no issuance.

For example, the following RRSet requests that no certificates be issued for the FQDN 'nocerts.example.com' by any Issuer.

```
nocerts.example.com       CAA 0 issue ";"
```

An issue Property Tag where the issue-value does not match the ABNF grammar MUST be treated the same as one specifying an empty issuer-domain-name. For example, the following malformed CAA RRSet forbids issuance:

```
malformed.example.com     CAA 0 issue "%%%%%"
```

CAA authorizations are additive; thus, the result of specifying both an empty issuer-domain-name and a non-empty issuer-domain-name is the same as specifying just the non-empty issuer-domain-name.

An Issuer MAY choose to specify parameters that further constrain the issue of certificates by that Issuer, for example, specifying that certificates are to be subject to specific validation polices, billed to certain accounts, or issued under specific trust anchors.

For example, if ca1.example.net has requested its customer accountable.example.com to specify their account number "230123" in each of the customer's CAA records using the (CA-defined) "account" parameter, it would look like this:

```
accountable.example.com   CAA 0 issue "ca1.example.net; account=230123"
```

The semantics of parameters to the issue Property Tag are determined by the Issuer alone.

4.3. CAA issuewild Property

The issuewild Property Tag has the same syntax and semantics as the issue Property Tag except that it only grants authorization to issue certificates that specify a Wildcard Domain Name and issuewild properties take precedence over issue properties when specified. Specifically:

issuewild properties MUST be ignored when processing a request for a Fully-Qualified Domain Name that is not a Wildcard Domain Name.

If at least one issuewild Property is specified in the Relevant RRSet for a Wildcard Domain Name, all issue properties MUST be ignored when processing a request for that Wildcard Domain Name.

For example, the following RRSet requests that _only_ ca1.example.net issue certificates for "wild.example.com" or "sub.wild.example.com", and that _only_ ca2.example.org issue certificates for "*.wild.example.com" or "*.sub.wild.example.com). Note that this presumes there are no CAA RRs for sub.wild.example.com.

```
wild.example.com          CAA 0 issue "ca1.example.net"
wild.example.com          CAA 0 issuewild "ca2.example.org"
```

The following RRSet requests that _only_ ca1.example.net issue certificates for "wild2.example.com", "*.wild2.example.com" or "*.sub.wild2.example.com".

```
wild2.example.com         CAA 0 issue "ca1.example.net"
```

The following RRSet requests that _only_ ca2.example.org issue certificates for "*.wild3.example.com" or "*.sub.wild3.example.com". It does not permit any Issuer to issue for "wild3.example.com" or "sub.wild3.example.com".

```
wild3.example.com         CAA 0 issuewild "ca2.example.org"
wild3.example.com         CAA 0 issue ";"
```

The following RRSet requests that _only_ ca2.example.org issue certificates for "*.wild3.example.com" or "*.sub.wild3.example.com". It permits any Issuer to issue for "wild3.example.com" or "sub.wild3.example.com".

```
wild3.example.com         CAA 0 issuewild "ca2.example.org"
```

4.4. CAA iodef Property

The iodef Property specifies a means of reporting certificate issue requests or cases of certificate issue for domains for which the Property appears in the Relevant RRSet, when those requests or issuances violate the security policy of the Issuer or the FQDN holder.

The Incident Object Description Exchange Format (IODEF) [RFC7970] is used to present the incident report in machine-readable form.

The iodef Property Tag takes a URL as its Property Value. The URL scheme type determines the method used for reporting:

mailto: The IODEF incident report is reported as a MIME email attachment to an SMTP email that is submitted to the mail address specified. The mail message sent SHOULD contain a brief text message to alert the recipient to the nature of the attachment.

http or https: The IODEF report is submitted as a Web service request to the HTTP address specified using the protocol specified in [RFC6546].

These are the only supported URL schemes.

The following RRSet specifies that reports may be made by means of email with the IODEF data as an attachment, a Web service [RFC6546], or both:

```
report.example.com         CAA 0 issue "ca1.example.net"
report.example.com         CAA 0 iodef "mailto:security@example.com"
report.example.com         CAA 0 iodef "http://iodef.example.com/"
```

4.5. Critical Flag

The critical flag is intended to permit future versions of CAA to introduce new semantics that MUST be understood for correct processing of the record, preventing conforming CAs that do not recognize the new semantics from issuing certificates for the indicated FQDNs.

In the following example, the Property with a Property Tag of 'tbs' is flagged as critical. Neither the ca1.example.net CA nor any other Issuer is authorized to issue for "new.example.com" (or any other domains for which this is the Relevant RRSet) unless the Issuer has implemented the processing rules for the 'tbs' Property Tag.

```
new.example.com       CAA 0 issue "ca1.example.net"
new.example.com       CAA 128 tbs "Unknown"
```

5. Security Considerations

CAA records assert a security policy that the holder of an FDQN wishes to be observed by Issuers. The effectiveness of CAA records as an access control mechanism is thus dependent on observance of CAA constraints by Issuers.

The objective of the CAA record properties described in this document is to reduce the risk of certificate mis-issue rather than avoid reliance on a certificate that has been mis-issued. DANE [RFC6698] describes a mechanism for avoiding reliance on mis-issued certificates.

5.1. Use of DNS Security

Use of DNSSEC to authenticate CAA RRs is strongly RECOMMENDED but not required. An Issuer MUST NOT issue certificates if doing so would conflict with the Relevant RRSet, irrespective of whether the corresponding DNS records are signed.

DNSSEC provides a proof of non-existence for both DNS Fully-Qualified Domain Names and RRSets within FQDNs. DNSSEC verification thus enables an Issuer to determine if the answer to a CAA record query is empty because the RRSet is empty or if it is non-empty but the response has been suppressed.

Use of DNSSEC allows an Issuer to acquire and archive a proof that they were authorized to issue certificates for the FQDN. Verification of such archives may be an audit requirement to verify CAA record processing compliance. Publication of such archives may be a transparency requirement to verify CAA record processing compliance.

5.2. Non-Compliance by Certification Authority

CAA records offer CAs a cost-effective means of mitigating the risk of certificate mis-issue: the cost of implementing CAA checks is very small and the potential costs of a mis-issue event include the removal of an embedded trust anchor.

5.3. Mis-Issue by Authorized Certification Authority

Use of CAA records does not prevent mis-issue by an authorized Certification Authority, i.e., a CA that is authorized to issue certificates for the FQDN in question by CAA records.

FQDN holders SHOULD verify that the CAs they authorize to issue certificates for their FQDNs employ appropriate controls to ensure that certificates are issued only to authorized parties within their organization.

Such controls are most appropriately determined by the FQDN holder and the authorized CA(s) directly and are thus out of scope of this document.

5.4. Suppression or Spoofing of CAA Records

Suppression of the CAA record or insertion of a bogus CAA record could enable an attacker to obtain a certificate from an Issuer that was not authorized to issue for an affected FQDN.

Where possible, Issuers SHOULD perform DNSSEC validation to detect missing or modified CAA record sets.

In cases where DNSSEC is not deployed for a corresponding FQDN, an Issuer SHOULD attempt to mitigate this risk by employing appropriate DNS security controls. For example, all portions of the DNS lookup process SHOULD be performed against the authoritative name server. Data cached by third parties MUST NOT be relied on as the sole source of DNS CAA information but MAY be used to support additional anti-spoofing or anti-suppression controls.

5.5. Denial of Service

Introduction of a malformed or malicious CAA RR could in theory enable a Denial-of-Service (DoS) attack. This could happen by modification of authoritative DNS records or by spoofing inflight DNS responses.

This specific threat is not considered to add significantly to the risk of running an insecure DNS service.

An attacker could, in principle, perform a DoS attack against an Issuer by requesting a certificate with a maliciously long DNS name. In practice, the DNS protocol imposes a maximum name length and CAA processing does not exacerbate the existing need to mitigate DoS attacks to any meaningful degree.

5.6. Abuse of the Critical Flag

A Certification Authority could make use of the critical flag to trick customers into publishing records that prevent competing Certification Authorities from issuing certificates even though the customer intends to authorize multiple providers. This could happen if the customers were setting CAA records based on data provided by the CA rather than generating those records themselves.

In practice, such an attack would be of minimal effect since any competent competitor that found itself unable to issue certificates due to lack of support for a Property marked critical should investigate the cause and report the reason to the customer. The customer will thus discover that they had been deceived.

6. Deployment Considerations

A CA implementing CAA may find that they receive errors looking up CAA records. The following are some common causes of such errors, so that CAs may provide guidance to their subscribers on fixing the underlying problems.

6.1. Blocked Queries or Responses

Some middleboxes, in particular anti-DDoS appliances, may be configured to drop DNS packets of unknown types, or may start dropping such packets when they consider themselves under attack. This generally manifests as a timed-out DNS query, or a SERVFAIL at a local recursive resolver.

6.2. Rejected Queries and Malformed Responses

Some authoritative nameservers respond with REJECTED or NOTIMP when queried for a Resource Record type they do not recognize. At least one authoritative resolver produces a malformed response (with the QR bit set to 0) when queried for unknown Resource Record types. Per RFC 1034, the correct response for unknown Resource Record types is NOERROR.

6.3. Delegation to Private Nameservers

Some FQDN administrators make the contents of a subdomain unresolvable on the public Internet by delegating that subdomain to a nameserver whose IP address is private. A CA processing CAA records for such subdomains will receive SERVFAIL from its recursive resolver. The CA MAY interpret that as preventing issuance. FQDN administrators wishing to issue certificates for private FQDNs SHOULD use split-horizon DNS with a publicly available nameserver, so that CAs can receive a valid, empty CAA response for those FQDNs.

6.4. Bogus DNSSEC Responses

Queries for CAA Resource Records are different from most DNS RR types, because a signed, empty response to a query for CAA RRs is meaningfully different from a bogus response. A signed, empty response indicates that there is definitely no CAA policy set at a given label. A bogus response may mean either a misconfigured zone, or an attacker tampering with records. DNSSEC implementations may have bugs with signatures on empty responses that go unnoticed, because for more common Resource Record types like A and AAAA, the difference to an end user between empty and bogus is irrelevant; they both mean a site is unavailable.

In particular, at least two authoritative resolvers that implement live signing had bugs when returning empty Resource Record sets for DNSSEC-signed zones, in combination with mixed-case queries. Mixed-case queries, also known as DNS 0x20, are used by some recursive resolvers to increase resilience against DNS poisoning attacks. DNSSEC-signing authoritative resolvers are expected to copy the same capitalization from the query into their ANSWER section, but sign the response as if they had used all lowercase. In particular, PowerDNS versions prior to 4.0.4 had this bug.

7. Differences versus RFC6844

This document obsoletes RFC6844. The most important change is to the Certification Authority Processing section. RFC6844 specified an algorithm that performed DNS tree-climbing not only on the FQDN being processed, but also on all CNAMEs and DNAMEs encountered along the way. This made the processing algorithm very inefficient when used on FQDNs that utilize many CNAMEs, and would have made it difficult for hosting providers to set CAA policies on their own FQDNs without setting potentially unwanted CAA policies on their customers' FQDNs. This document specifies a simplified processing algorithm that only performs tree climbing on the FQDN being processed, and leaves processing of CNAMEs and DNAMEs up to the CA's recursive resolver.

This document also includes a "Deployment Considerations" section detailing experience gained with practical deployment of CAA enforcement among CAs in the WebPKI.

This document clarifies the ABNF grammar for the issue and issuewild tags and resolves some inconsistencies with the document text. In particular, it specifies that parameters are separated with semicolons. It also allows hyphens in Property Tags.

This document also clarifies processing of a CAA RRset that is not empty, but contains no issue or issuewild tags.

This document removes the section titled "The CAA RR Type," merging it with "Mechanism" because the definitions were mainly duplicates. It moves the "Use of DNS Security" section into Security Considerations. It renames "Certification Authority Processing" to "Relevant Resource Record Set," and emphasizes the use of that term to more clearly define which domains are affected by a given RRset.

8. IANA Considerations

IANA is requested to add [[[RFC Editor: Please replace with this RFC]]] as a reference for the Certification Authority Restriction Flags and Certification Authority Restriction Properties registries, and update references to [RFC6844] within those registries to refer to [[[RFC Editor: Please replace with this RFC]]]. IANA is also requested to update the CAA TYPE in the DNS Parameters registry with a reference to [[[RFC Editor: Please replace with this RFC]]].

9. Acknowledgements

The authors would like to thank the following people who contributed to the design and documentation of this work item: Corey Bonnell, Chris Evans, Stephen Farrell, Jeff Hodges, Paul Hoffman, Tim Hollebeek, Stephen Kent, Adam Langley, Ben Laurie, James Manger, Chris Palmer, Scott Schmit, Sean Turner, and Ben Wilson.

10. References

10.1. Normative References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2181] Elz, R. and R. Bush, "Clarifications to the DNS Specification", RFC 2181, DOI 10.17487/RFC2181, July 1997, <https://www.rfc-editor.org/info/rfc2181>.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

[RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, DOI 10.17487/RFC4034, March 2005, <https://www.rfc-editor.org/info/rfc4034>.

[RFC4035] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Protocol Modifications for the DNS Security Extensions", RFC 4035, DOI 10.17487/RFC4035, March 2005, <https://www.rfc-editor.org/info/rfc4035>.

[RFC5155] Laurie, B., Sisson, G., Arends, R., and D. Blacka, "DNS Security (DNSSEC) Hashed Authenticated Denial of Existence", RFC 5155, DOI 10.17487/RFC5155, March 2008, <https://www.rfc-editor.org/info/rfc5155>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[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, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6546] Trammell, B., "Transport of Real-time Inter-network Defense (RID) Messages over HTTP/TLS", RFC 6546, DOI 10.17487/RFC6546, April 2012, <https://www.rfc-editor.org/info/rfc6546>.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012, <https://www.rfc-editor.org/info/rfc6698>.

[RFC6844] Hallam-Baker, P. and R. Stradling, "DNS Certification Authority Authorization (CAA) Resource Record", RFC 6844, DOI 10.17487/RFC6844, January 2013, <https://www.rfc-editor.org/info/rfc6844>.

[RFC7970] Danyliw, R., "The Incident Object Description Exchange Format Version 2", RFC 7970, DOI 10.17487/RFC7970, November 2016, <https://www.rfc-editor.org/info/rfc7970>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[RFC3647] Chokhani, S., Ford, W., Sabett, R., Merrill, C., and S. Wu, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework", RFC 3647, DOI 10.17487/RFC3647, November 2003, <https://www.rfc-editor.org/info/rfc3647>.

Authors' Addresses

Phillip Hallam-Baker

Email: phill@hallambaker.com

Rob Stradling
Sectigo Ltd.

Email: rob@sectigo.com

Jacob Hoffman-Andrews
Let’s Encrypt

Email: jsha@letsencrypt.org

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 12, 2019
D. Van Geest
ISARA Corporation
S. Fluhrer
Cisco Systems
March 11, 2019

Algorithm Identifiers for HSS and XMSS for Use in the Internet X.509 Public Key Infrastructure
draft-vangeest-x509-hash-sigs-03

Abstract

This document specifies algorithm identifiers and ASN.1 encoding formats for the Hierarchical Signature System (HSS), eXtended Merkle Signature Scheme (XMSS), and XMSS^MT, a multi-tree variant of XMSS. This specification applies to the Internet X.509 Public Key infrastructure (PKI) when digital signatures are used to sign certificates and certificate revocation lists (CRLs).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Hierarchical Signature System (HSS) is described in [I-D.mcgrew-hash-sigs].

The eXtended Merkle Signature Scheme (XMSS), and its multi-tree variant XMSS^MT, are described in [RFC8391].

These signature algorithms are based on well-studied Hash Based Signature (HBS) schemes, which can withstand known attacks using quantum computers. They combine Merkle Trees with One Time Signature (OTS) schemes in order to create signature systems which can sign a large but limited number of messages per private key. The private keys are stateful; a key's state must be updated and persisted after signing to prevent reuse of OTS keys. If an OTS key is reused, cryptographic security is not guaranteed for that key.

Due to the statefulness of the private key and the limited number of signatures that can be created, these signature algorithms might not be appropriate for use in interactive protocols. While the right selection of algorithm parameters would allow a private key to sign a virtually unbounded number of messages (e.g. 2^60), this is at the cost of a larger signature size and longer signing time. Since these algorithms are already known to be secure against quantum attacks, and because roots of trust are generally long-lived and can take longer to be deployed than end-entity certificates, these signature algorithms are more appropriate to be used in root and subordinate CA certificates. They are also appropriate in non-interactive contexts such as code signing. In particular, there are multi-party IoT ecosystems where publicly trusted code signing certificates are useful.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Subject Public Key Algorithms

Certificates conforming to [RFC5280] can convey a public key for any public key algorithm. The certificate indicates the algorithm through an algorithm identifier. An algorithm identifier consists of an OID and optional parameters.

In this document, we define new OIDs for identifying the different hash-based signature algorithms. An additional OID is defined in [I-D.ietf-lamps-cms-hash-sig] and repeated here for convenience. For all of the OIDs, the parameters MUST be absent.

2.1. HSS Public Keys

The object identifier and public key algorithm identifier for HSS is defined in [I-D.ietf-lamps-cms-hash-sig]. The definitions are repeated here for reference.

The object identifier for an HSS public key is id-alg-hss-lms-hashsig:

```
id-alg-hss-lms-hashsig  OBJECT IDENTIFIER ::= { iso(1)
   member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
   smime(16) alg(3) 17 }
```

Note that the id-alg-hss-lms-hashsig algorithm identifier is also referred to as id-alg-mts-hashsig. This synonym is based on the terminology used in an early draft of the document that became [I-D.mcgrew-hash-sigs].

The HSS public key's properties are defined as follows:

```
      pk-HSS-LMS-HashSig PUBLIC-KEY ::= {
         IDENTIFIER id-alg-hss-lms-hashsig
         KEY HSS-LMS-HashSig-PublicKey
         PARAMS ARE absent
         CERT-KEY-USAGE
            { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

      HSS-LMS-HashSig-PublicKey ::= OCTET STRING
```

[I-D.ietf-lamps-cms-hash-sig] contains more information on the contents and format of an HSS public key.

2.2. XMSS Public Keys

The object identifier for an XMSS public key is id-alg-xmss:

```
      id-alg-xmss  OBJECT IDENTIFIER ::= { itu-t(0)
         identified-organization(4) etsi(0) reserved(127)
         etsi-identified-organization(0) isara(15) algorithms(1)
         asymmetric(1) xmss(13) 0 }
```

The XMSS public key's properties are defined as follows:

```
      pk-XMSS PUBLIC-KEY ::= {
         IDENTIFIER id-alg-xmss
         KEY XMSS-PublicKey
         PARAMS ARE absent
         CERT-KEY-USAGE
            { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

      XMSS-PublicKey ::= OCTET STRING
```

The format of an XMSS public key is is formally defined using XDR [RFC4506] and is defined in Appendix B.3 of [RFC8391]. In particular, the first 4 bytes represents the big-ending encoding of the XMSS algorithm type.

2.3. XMSS^MT Public Keys

The object identifier for an XMSS^MT public key is id-alg-xmssmt:

```
      id-alg-xmssmt  OBJECT IDENTIFIER ::= { itu-t(0)
         identified-organization(4) etsi(0) reserved(127)
         etsi-identified-organization(0) isara(15) algorithms(1)
         asymmetric(1) xmssmt(14) 0 }
```

The XMSS^MT public key's properties are defined as follows:

```
pk-XMSSMT PUBLIC-KEY ::= {
   IDENTIFIER id-alg-xmssmt
   KEY XMSSMT-PublicKey
   PARAMS ARE absent
   CERT-KEY-USAGE
      { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

XMSSMT-PublicKey ::= OCTET STRING
```

The format of an XMSS^MT public key is is formally defined using XDR [RFC4506] and is defined in Appendix C.3 of [RFC8391]. In particular, the first 4 bytes represents the big-ending encoding of the XMSS^MT algorithm type.

3. Key Usage Bits

The intended application for the key is indicated in the keyUsage certificate extension.

If the keyUsage extension is present in an end-entity certificate that indicates id-alg-xmss or id-alg-xmssmt in SubjectPublicKeyInfo, then the keyUsage extension MUST contain one or both of the following values:

nonRepudiation; and
digitalSignature.

If the keyUsage extension is present in a certification authority certificate that indicates id-alg-xmss or id-alg-xmssmt, then the keyUsage extension MUST contain one or more of the following values:

nonRepudiation;
digitalSignature;
keyCertSign; and
cRLSign.

[I-D.ietf-lamps-cms-hash-sig] defines the key usage for id-alg-hss-lms-hashsig, which is the same as for the keys above.

4. Signature Algorithms

This section identifies OIDs for signing using HSS, XMSS, and XMSS^MT. When these algorithm identifiers appear in the algorithm field as an AlgorithmIdentifier, the encoding MUST omit the parameters field. That is, the AlgorithmIdentifier SHALL be a SEQUENCE of one component, one of the OIDs defined below.

The data to be signed is prepared for signing. For the algorithms used in this document, the data is signed directly by the signature algorithm, the data is not hashed before processing. Then, a private key operation is performed to generate the signature value. For HSS, the signature value is described in section 3.3 of [I-D.mcgrew-hash-sigs]. For XMSS and XMSS^MT the signature values are described in sections B.2 and C.2 of [RFC8391] respectively. The octet string representing the signature is encoded directly in the BIT STRING without adding any additional ASN.1 wrapping. For the Certificate and CertificateList structures, the signature value is wrapped in the "signatureValue" BIT STRING field.

4.1. HSS Signature Algorithm

The HSS public key OID is also used to specify that an HSS signature was generated on the full message, i.e. the message was not hashed before being processed by the HSS signature algorithm.

```
id-alg-hss-lms-hashsig OBJECT IDENTIFIER ::= { iso(1)
    member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
    smime(16) alg(3) 17 }
```

[I-D.ietf-lamps-cms-hash-sig] contains more information on the contents and format of an HSS signature.

4.2. XMSS Signature Algorithm

The XMSS public key OID is also used to specify that an XMSS signature was generated on the full message, i.e. the message was not hashed before being processed by the XMSS signature algorithm.

```
id-alg-xmss  OBJECT IDENTIFIER ::= { itu-t(0)
   identified-organization(4) etsi(0) reserved(127)
   etsi-identified-organization(0) isara(15) algorithms(1)
   asymmetric(1) xmss(13) 0 }
```

The format of an XMSS signature is is formally defined using XDR [RFC4506] and is defined in Appendix B.2 of [RFC8391].

4.3. XMSS^MT Signature Algorithm

The XMSS^MT public key OID is also used to specify that an XMSS^MT signature was generated on the full message, i.e. the message was not hashed before being processed by the XMSS^MT signature algorithm.

```
      id-alg-xmssmt  OBJECT IDENTIFIER ::= { itu-t(0)
         identified-organization(4) etsi(0) reserved(127)
         etsi-identified-organization(0) isara(15) algorithms(1)
         asymmetric(1) xmssmt(14) 0 }
```

The format of an XMSS^MT signature is is formally defined using XDR [RFC4506] and is defined in Appendix C.2 of [RFC8391].

5. ASN.1 Module

For reference purposes, the ASN.1 syntax is presented as an ASN.1 module here.

```
   -- ASN.1 Module

   Hashsigs-pkix-0 -- TBD - IANA assigned module OID

   DEFINITIONS EXPLICIT TAGS ::=
   BEGIN

   IMPORTS
     PUBLIC-KEY, SIGNATURE-ALGORITHM
     FROM AlgorithmInformation-2009
       {iso(1) identified-organization(3) dod(6) internet(1) security(5)
       mechanisms(5) pkix(7) id-mod(0)
       id-mod-algorithmInformation-02(58)}
   ;

   -- Object Identifiers

   --
   -- id-alg-hss-lms-hashsig is defined in [ietf-lamps-cms-hash-sig]
   --
   -- id-alg-hss-lms-hashsig OBJECT IDENTIFIER ::= { iso(1)
   --     member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs9(9)
   --     smime(16) alg(3) 17 }

   id-alg-xmss  OBJECT IDENTIFIER ::= { itu-t(0)
      identified-organization(4) etsi(0) reserved(127)
      etsi-identified-organization(0) isara(15) algorithms(1)
      asymmetric(1) xmss(13) 0 }

   id-alg-xmssmt  OBJECT IDENTIFIER ::= { itu-t(0)
      identified-organization(4) etsi(0) reserved(127)
      etsi-identified-organization(0) isara(15) algorithms(1)
      asymmetric(1) xmssmt(14) 0 }
```

```
   -- Signature Algorithms and Public Keys

   --
   -- sa-HSS-LMS-HashSig is defined in [ietf-lamps-cms-hash-sig]
   --
   -- sa-HSS-LMS-HashSig SIGNATURE-ALGORITHM ::= {
   --     IDENTIFIER id-alg-hss-lms-hashsig
   --     PARAMS ARE absent
   --     PUBLIC-KEYS { pk-HSS-LMS-HashSig }
   --     SMIME-CAPS { IDENTIFIED BY id-alg-hss-lms-hashsig } }

   --
   -- pk-HSS-LMS-HashSig is defined in [ietf-lamps-cms-hash-sig]
   --
   -- pk-HSS-LMS-HashSig PUBLIC-KEY ::= {
   --    IDENTIFIER id-alg-hss-lms-hashsig
   --    KEY HSS-LMS-HashSig-PublicKey
   --    PARAMS ARE absent
   --    CERT-KEY-USAGE
   --       { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }
   --
   -- HSS-LMS-HashSig-PublicKey ::= OCTET STRING

   sa-XMSS SIGNATURE-ALGORITHM ::= {
       IDENTIFIER id-alg-xmss
       PARAMS ARE absent
       PUBLIC-KEYS { pk-XMSS }
       SMIME-CAPS { IDENTIFIED BY id-alg-xmss } }

   pk-XMSS PUBLIC-KEY ::= {
      IDENTIFIER id-alg-xmss
      KEY XMSS-PublicKey
      PARAMS ARE absent
      CERT-KEY-USAGE
         { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

   XMSS-PublicKey ::= OCTET STRING

   sa-XMSSMT SIGNATURE-ALGORITHM ::= {
       IDENTIFIER id-alg-xmssmt
       PARAMS ARE absent
       PUBLIC-KEYS { pk-XMSSMT }
       SMIME-CAPS { IDENTIFIED BY id-alg-xmssmt } }
```

```
   pk-XMSSMT PUBLIC-KEY ::= {
      IDENTIFIER id-alg-xmssmt
      KEY XMSSMT-PublicKey
      PARAMS ARE absent
      CERT-KEY-USAGE
         { digitalSignature, nonRepudiation, keyCertSign, cRLSign } }

   XMSSMT-PublicKey ::= OCTET STRING

   END
```

6. Security Considerations

6.1. Algorithm Security Considerations

The cryptographic security of the signatures generated by the algorithms mentioned in this document depends only on the hash algorithms used within the signature algorithms and the pre-hash algorithm used to create an X.509 certificate's message digest. Grover's algorithm [Grover96] is a quantum search algorithm which gives a quadratic improvement in search time to brute-force pre-image attacks. The results of [BBBV97] show that this improvement is optimal, however [Fluhrer17] notes that Grover's algorithm doesn't parallelize well. Thus, given a bounded amount of time to perform the attack and using a conservative estimate of the performance of a real quantum computer, the pre-image quantum security of SHA-256 is closer to 190 bits. All parameter sets for the signature algorithms in this document currently use SHA-256 internally and thus have at least 128 bits of quantum pre-image resistance, or 190 bits using the security assumptions in [Fluhrer17].

[Zhandry15] shows that hash collisions can be found using an algorithm with a lower bound on the number of oracle queries on the order of 2^(n/3) on the number of bits, however [DJB09] demonstrates that the quantum memory requirements would be much greater. Therefore a parameter set using SHA-256 would have at least 128 bits of quantum collision-resistance as well as the pre-image resistance mentioned in the previous paragraph.

Given the quantum collision and pre-image resistance of SHA-256 estimated above, the current parameter sets used by id-alg-hss-lms-hashsig, id-alg-xmss and id-alg-xmssmt provide 128 bits or more of quantum security. This is believed to be secure enough to protect X.509 certificates for well beyond any reasonable certificate lifetime.

6.2. Implementation Security Considerations

Implementations MUST protect the private keys. Compromise of the private keys may result in the ability to forge signatures. Along with the private key, the implementation MUST keep track of which leaf nodes in the tree have been used. Loss of integrity of this tracking data can cause a one-time key to be used more than once. As a result, when a private key and the tracking data are stored on non-volatile media or stored in a virtual machine environment, care must be taken to preserve confidentiality and integrity.

The generation of private keys relies on random numbers. The use of inadequate pseudo-random number generators (PRNGs) to generate these values 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.

The generation of hash-based signatures also depends on random numbers. While the consequences of an inadequate pseudo-random number generator (PRNGs) to generate these values is much less severe than the generation of private keys, the guidance in [RFC4086] remains important.

7. Acknowledgements

Thanks for Russ Housley for the helpful suggestions.

This document uses a lot of text from similar documents ([RFC3279] and [RFC8410]) as well as [I-D.ietf-lamps-cms-hash-sig]. Thanks go to the authors of those documents. "Copying always makes things easier and less error prone" - [RFC8411].

8. IANA Considerations

IANA is requested to assign a module OID from the "SMI for PKIX Module Identifier" registry for the ASN.1 module in Section 5.

9. References

9.1. Normative References

[I-D.ietf-lamps-cms-hash-sig]
Housley, R., "Use of the HSS/LMS Hash-based Signature Algorithm in the Cryptographic Message Syntax (CMS)", draft-ietf-lamps-cms-hash-sig-07 (work in progress), March 2019.

[I-D.mcgrew-hash-sigs]
McGrew, D., Curcio, M., and S. Fluhrer, "Hash-Based Signatures", draft-mcgrew-hash-sigs-15 (work in progress), January 2019.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[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, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC8391] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/info/rfc8391>.

9.2. Informative References

[BBBV97] Bennett, C., Bernstein, E., Brassard, G., and U. Vazirani, "Strengths and weaknesses of quantum computing", SIAM J. Comput. 26(5), 1510-1523, 1997.

[DJB09] Bernstein, D., "Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?", SHARCS 9, p. 105, 2009.

[Fluhrer17]
Fluhrer, S., "Reassessing Grover's Algorithm", Cryptology ePrint Archive Report 2017/811, August 2017, <https://eprint.iacr.org/2017/811.pdf>.

[Grover96]
Grover, L., "A fast quantum mechanical algorithm for database search", 28th ACM Symposium on the Theory of Computing p. 212, 1996.

[RFC3279] Bassham, L., Polk, W., and R. Housley, "Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3279, DOI 10.17487/RFC3279, April 2002, <https://www.rfc-editor.org/info/rfc3279>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC8410] Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure", RFC 8410, DOI 10.17487/RFC8410, August 2018, <https://www.rfc-editor.org/info/rfc8410>.

[RFC8411] Schaad, J. and R. Andrews, "IANA Registration for the Cryptographic Algorithm Object Identifier Range", RFC 8411, DOI 10.17487/RFC8411, August 2018, <https://www.rfc-editor.org/info/rfc8411>.

[Zhandry15]
Zhandry, M., "A note on the quantum collision and set equality problems", Quantum Information & Computation 15, 7-8, 557-567, May 2015.

Authors' Addresses

Daniel Van Geest
ISARA Corporation
560 Westmount Rd N
Waterloo, Ontario N2L 0A9
Canada

Email: daniel.vangeest@isara.com

Scott Fluhrer
Cisco Systems
170 West Tasman Drive
San Jose, CA 95134
USA

Email: sfluhrer@cisco.com