Internet-Draft
Intended status: Standards Track
Expires: 22 February 2018

R. Housley
Vigil Security

22 August 2017

Use of the Elliptic Curve Diffie-Hellman Key Agreement Algorithm with X25519 and X448 in the Cryptographic Message Syntax (CMS)

<draft-ietf-curdle-cms-ecdh-new-curves-10.txt>

Abstract

This document describes the conventions for using Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm using curve25519 and curve448 in the Cryptographic Message Syntax (CMS).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes the conventions for using Elliptic Curve Diffie-Hellman (ECDH) key agreement using curve25519 and curve448 [CURVES] in the Cryptographic Message Syntax (CMS) [CMS]. Key agreement is supported in three CMS content types: the enveloped-data content type [CMS], authenticated-data content type [CMS], and the authenticated-enveloped-data content type [AUTHENV].

The conventions for using some Elliptic Curve Cryptography (ECC) algorithms in CMS are described in [CMSECC]. These conventions cover the use of ECDH with some curves other than curve25519 and curve448 [CURVES]. Those other curves are not deprecated.

Using curve25519 with Diffie-Hellman key agreement is referred to as X25519. Using curve448 with Diffie-Hellman key agreement is referred to as X448.

1.1. Terminology

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 RFC 2119 [STDWORDS].

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].

2. Key Agreement

In 1976, Diffie and Hellman described a means for two parties to agree upon a shared secret value in manner that prevents eavesdroppers from learning the shared secret value [DH1976]. This secret may then be converted into pairwise symmetric keying material for use with other cryptographic algorithms. Over the years, many variants of this fundamental technique have been developed. This document describes the conventions for using Ephemeral-Static Elliptic Curve Diffie-Hellman (ECDH) key agreement using X25519 and X448 [CURVES].

The originator MUST use an ephemeral public/private key pair that is generated on the same elliptic curve as the public key of the recipient. The ephemeral key pair MUST be used for a single CMS protected content type, and then it MUST be discarded. The originator obtains the recipient’s static public key from the recipient’s certificate [PROFILE].

X25519 is described in Section 6.1 of [CURVES], and X448 is described in Section 6.2 of [CURVES]. Conforming implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value, and abort if so, as described in Section 6 of [CURVES]. If an alternative implementation of these elliptic curves to that documented in Section 6 of [CURVES] is employed, then the additional checks specified in Section 7 of [CURVES] SHOULD be performed.

In [CURVES], the shared secret value that is produced by ECDH is called K. (In some other specifications, the shared secret value is called Z.) A key derivation function (KDF) is used to produce a pairwise key-encryption key (KEK) from the shared secret value (K), the length of the key-encryption key, and the DER-encoded ECC-CMS-SharedInfo structure [CMSECC].

The ECC-CMS-SharedInfo definition from [CMSECC] is repeated here for convenience.

```
   ECC-CMS-SharedInfo ::= SEQUENCE {
     keyInfo         AlgorithmIdentifier,
     entityUInfo [0] EXPLICIT OCTET STRING OPTIONAL,
     suppPubInfo [2] EXPLICIT OCTET STRING  }
```

The ECC-CMS-SharedInfo keyInfo field contains the object identifier of the key-encryption algorithm and associated parameters. This algorithm will be used to wrap the content-encryption key. For example, the AES Key Wrap algorithm [AESKW] does not need parameters, so the algorithm identifier parameters are absent.

The ECC-CMS-SharedInfo entityUInfo field optionally contains additional keying material supplied by the sending agent. Note that [CMS] requires implementations to accept a KeyAgreeRecipientInfo SEQUENCE that includes the ukm field. If the ukm field is present, the ukm is placed in the entityUInfo field. By including the ukm, a different key-encryption key is generated even when the originator ephemeral private key is improperly used more than once. Therefore, if the ukm field is present, it MUST be selected in a manner that provides with very high probability a unique value; however, there is no security benefit to using a ukm value that is longer than the key-encryption key that will be produced by the KDF.

The ECC-CMS-SharedInfo suppPubInfo field contains the length of the generated key-encryption key, in bits, represented as a 32-bit number in network byte order. For example, the key length for AES-256 [AES] would be 0x00000100.

2.1. ANSI-X9.63-KDF

The ANSI-X9.63-KDF key derivation function is a simple construct based on a one-way hash function described in American National Standard X9.63 [X963]. This KDF is also described in Section 3.6.1 of [SEC1].

Three values are concatenated to produce the input string to the KDF:

1. The shared secret value generated by ECDH, K.
2. The iteration counter, starting with one, as described below.
3. The DER-encoded ECC-CMS-SharedInfo structure.

To generate a key-encryption key (KEK), the KDF generates one or more KM blocks, with the counter starting at 0x00000001, and incrementing the counter for each subsequent KM block until enough material has been generated. The 32-bit counter is represented in network byte order. The KM blocks are concatenated left to right, and then the leftmost portion of the result is used as the pairwise key-encryption key, KEK:

```
   KM(i) = Hash(K || INT32(counter=i) || DER(ECC-CMS-SharedInfo))

   KEK = KM(counter=1) || KM(counter=2) ...
```

2.2. HKDF

The HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is a robust construct based on a one-way hash function described in RFC 5869 [HKDF]. HKDF is comprised of two steps: HKDF-Extract followed by HKDF-Expand.

Three values are used as inputs to the HKDF:
   1. The shared secret value generated by ECDH, K.
   2. The length in octets of the keying data to be generated.
   3. The DER-encoded ECC-CMS-SharedInfo structure.

The ECC-CMS-SharedInfo structure optionally includes the ukm. If the ukm is present, the ukm is also used as the HKDF salt. HKDF uses an appropriate number of zero octets when no salt is provided.

The length of the generated key-encryption key is used in two places, once in bits, and once in octets. The ECC-CMS-SharedInfo structure includes the length of the generated key-encryption key in bits. The HKDF-Expand function takes an argument for the length of the generated key-encryption key in octets.

In summary, to produce the pairwise key-encryption key, KEK:

```
   if ukm is provided, then salt = ukm, else salt is not provided
   PRK = HKDF-Extract(salt, K)

   KEK = HKDF-Expand(PRK, DER(ECC-CMS-SharedInfo), SizeInOctets(KEK))
```

3. Enveloped-data Conventions

The CMS enveloped-data content type [CMS] consists of an encrypted content and wrapped content-encryption keys for one or more recipients. The ECDH key agreement algorithm is used to generate a pairwise key-encryption key between the originator and a particular recipient. Then, the key-encryption key is used to wrap the content-encryption key for that recipient. When there is more than one recipient, the same content-encryption key MUST be wrapped for each of them.

A compliant implementation MUST meet the requirements for constructing an enveloped-data content type in Section 6 of [CMS].

A content-encryption key MUST be randomly generated for each instance of an enveloped-data content type. The content-encryption key is used to encrypt the content.

3.1. EnvelopedData Fields

The enveloped-data content type is ASN.1 encoded using the EnvelopedData syntax. The fields of the EnvelopedData syntax MUST be populated as described in Section 6 of [CMS]. The RecipientInfo choice is described in Section 6.2 of [CMS], and repeated here for convenience.

```
RecipientInfo ::= CHOICE {
  ktri KeyTransRecipientInfo,
  kari [1] KeyAgreeRecipientInfo,
  kekri [2] KEKRecipientInfo,
  pwri [3] PasswordRecipientinfo,
  ori [4] OtherRecipientInfo }
```

For the recipients that use X25519 or X448 the RecipientInfo kari choice MUST be used.

3.2. KeyAgreeRecipientInfo Fields

The fields of the KeyAgreeRecipientInfo syntax MUST be populated as described in this section when X25519 or X448 is employed for one or more recipients.

The KeyAgreeRecipientInfo version MUST be 3.

The KeyAgreeRecipientInfo originator provides three alternatives for identifying the originator’s public key, and the originatorKey alternative MUST be used. The originatorKey MUST contain an ephemeral key for the originator. The originatorKey algorithm field MUST contain the id-X25519 or the id-X448 object identifier. The originator’s ephemeral public key MUST be encoded as an OCTET STRING.

The object identifiers for X25519 and X448 have been assigned in [ID.curdle-pkix]. They are repeated below for convenience.

When using X25519, the public key contains exactly 32 octets, and the id-X25519 object identifier is used:

```
id-X25519 OBJECT IDENTIFIER ::= { 1 3 101 110 }
```

When using X448, the public key contains exactly 56 octets, and the id-X448 object identifier is used:

```
id-X448 OBJECT IDENTIFIER ::= { 1 3 101 111 }
```

KeyAgreeRecipientInfo ukm is optional. The processing of the ukm with The ANSI-X9.63-KDF key derivation function is described in Section 2.1, and the processing of the ukm with the HKDF key derivation function is described in Section 2.2.

KeyAgreeRecipientInfo keyEncryptionAlgorithm MUST contain the object identifier of the key-encryption algorithm that will be used to wrap the content-encryption key. The conventions for using AES-128, AES-192, and AES-256 in the key wrap mode are specified in [CMSAES].

KeyAgreeRecipientInfo recipientEncryptedKeys includes a recipient identifier and encrypted key for one or more recipients. The RecipientEncryptedKey KeyAgreeRecipientIdentifier MUST contain either the issuerAndSerialNumber identifying the recipient’s certificate or the RecipientKeyIdentifier containing the subject key identifier from the recipient’s certificate. In both cases, the recipient’s certificate contains the recipient’s static X25519 or X448 public key. RecipientEncryptedKey EncryptedKey MUST contain the content-encryption key encrypted with the pairwise key-encryption key using the algorithm specified by the KeyWrapAlgorithm.

4. Authenticated-data Conventions

The CMS authenticated-data content type [CMS] consists an authenticated content, a message authentication code (MAC), and encrypted authentication keys for one or more recipients. The ECDH key agreement algorithm is used to generate a pairwise key-encryption key between the originator and a particular recipient. Then, the key-encryption key is used to wrap the authentication key for that recipient. When there is more than one recipient, the same authentication key MUST be wrapped for each of them.

A compliant implementation MUST meet the requirements for constructing an authenticated-data content type in Section 9 of [CMS].

A authentication key MUST be randomly generated for each instance of an authenticated-data content type. The authentication key is used to compute the MAC over the content.

4.1. AuthenticatedData Fields

The authenticated-data content type is ASN.1 encoded using the AuthenticatedData syntax. The fields of the AuthenticatedData syntax MUST be populated as described in [CMS]; for the recipients that use X25519 or X448 the RecipientInfo kari choice MUST be used.

4.2. KeyAgreeRecipientInfo Fields

The fields of the KeyAgreeRecipientInfo syntax MUST be populated as described in Section 3.2 of this document.

5. Authenticated-Enveloped-data Conventions

The CMS authenticated-enveloped-data content type [AUTHENV] consists of an authenticated and encrypted content and encrypted content-authenticated-encryption keys for one or more recipients. The ECDH key agreement algorithm is used to generate a pairwise key-encryption key between the originator and a particular recipient. Then, the key-encryption key is used to wrap the content-authenticated-encryption key for that recipient. When there is more than one recipient, the same content-authenticated-encryption key MUST be wrapped for each of them.

A compliant implementation MUST meet the requirements for constructing an authenticated-data content type in Section 2 of [AUTHENV].

A content-authenticated-encryption key MUST be randomly generated for each instance of an authenticated-enveloped-data content type. The content-authenticated-encryption key is used to authenticate and encrypt the content.

5.1. AuthEnvelopedData Fields

The authenticated-enveloped-data content type is ASN.1 encoded using the AuthEnvelopedData syntax. The fields of the AuthEnvelopedData syntax MUST be populated as described in [AUTHENV]; for the recipients that use X25519 or X448 the RecipientInfo kari choice MUST be used.

5.2. KeyAgreeRecipientInfo Fields

The fields of the KeyAgreeRecipientInfo syntax MUST be populated as described in Section 3.2 of this document.

6. Certificate Conventions

RFC 5280 [PROFILE] specifies the profile for using X.509 Certificates in Internet applications. A recipient static public key is needed for X25519 or X448, and the originator obtains that public key from the recipient’s certificate. The conventions for carrying X25519 and X448 public keys are specified in [ID.curdle-pkix].

7. Key Agreement Algorithm Identifiers

The following object identifiers are assigned in [CMSECC] to indicate ECDH with ANSI-X9.63-KDF using various one-way hash functions. These are expected to be used as AlgorithmIdentifiers with a parameter that specifies the key-encryption algorithm. These are repeated here for convenience.

```
secg-scheme OBJECT IDENTIFIER ::= {
  iso(1) identified-organization(3) certicom(132) schemes(1) }

dhSinglePass-stdDH-sha256kdf-scheme OBJECT IDENTIFIER ::= {
  secg-scheme 11 1 }

dhSinglePass-stdDH-sha384kdf-scheme OBJECT IDENTIFIER ::= {
  secg-scheme 11 2 }

dhSinglePass-stdDH-sha512kdf-scheme OBJECT IDENTIFIER ::= {
  secg-scheme 11 3 }
```

The following object identifiers are assigned to indicate ECDH with HKDF using various one-way hash functions. These are expected to be used as AlgorithmIdentifiers with a parameter that specifies the key-encryption algorithm.

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

dhSinglePass-stdDH-hkdf-sha256-scheme OBJECT IDENTIFIER ::= {
   smime-alg 19 }

dhSinglePass-stdDH-hkdf-sha384-scheme OBJECT IDENTIFIER ::= {
   smime-alg 20 }
```

```
      dhSinglePass-stdDH-hkdf-sha512-scheme OBJECT IDENTIFIER ::= {
         smime-alg 21 }
```

8. SMIMECapabilities Attribute Conventions

A sending agent MAY announce to other agents that it supports ECDH key agreement using the SMIMECapabilities signed attribute in a signed message [SMIME] or a certificate [CERTCAP]. Following the pattern established in [CMSECC], the SMIMECapabilities associated with ECDH carries a DER-encoded object identifier that identifies support for ECDH in conjunction with a particular KDF, and it includes a parameter that names the key wrap algorithm.

The following SMIMECapabilities values (in hexidecimal) from [CMSECC] might be of interest to implementations that support X25519 and X448:

```
      ECDH with ANSI-X9.63-KDF using SHA-256; uses AES-128 key wrap:
         30 15 06 06 2B 81 04 01 0B 01 30 0B 06 09 60 86 48 01 65 03 04
         01 05

      ECDH with ANSI-X9.63-KDF using SHA-384; uses AES-128 key wrap:
         30 15 06 06 2B 81 04 01 0B 02 30 0B 06 09 60 86 48 01 65 03 04
         01 05

      ECDH with ANSI-X9.63-KDF using SHA-512; uses AES-128 key wrap:
         30 15 06 06 2B 81 04 01 0B 03 30 0B 06 09 60 86 48 01 65 03 04
         01 05

      ECDH with ANSI-X9.63-KDF using SHA-256; uses AES-256 key wrap:
         30 15 06 06 2B 81 04 01 0B 01 30 0B 06 09 60 86 48 01 65 03 04
         01 2D

      ECDH with ANSI-X9.63-KDF using SHA-384; uses AES-256 key wrap:
         30 15 06 06 2B 81 04 01 0B 02 30 0B 06 09 60 86 48 01 65 03 04
         01 2D

      ECDH with ANSI-X9.63-KDF using SHA-512; uses AES-256 key wrap:
         30 15 06 06 2B 81 04 01 0B 03 30 0B 06 09 60 86 48 01 65 03 04
         01 2D
```

The following SMIMECapabilities values (in hexidecimal) based on the algorithm identifiers in Section 7 of this document might be of interest to implementations that support X25519 and X448:

```
      ECDH with HKDF using SHA-256; uses AES-128 key wrap:
         30 1A 06 0B 2A 86 48 86 F7 0D 01 09 10 03 13 30 0B 06 09 60 86
         48 01 65 03 04 01 05
      ECDH with HKDF using SHA-384; uses AES-128 key wrap:
         30 1A 06 0B 2A 86 48 86 F7 0D 01 09 10 03 14 30 0B 06 09 60 86
         48 01 65 03 04 01 05

      ECDH with HKDF using SHA-512; uses AES-128 key wrap:
         30 1A 06 0B 2A 86 48 86 F7 0D 01 09 10 03 15 30 0B 06 09 60 86
         48 01 65 03 04 01 05

      ECDH with HKDF using SHA-256; uses AES-256 key wrap:
         30 1A 06 0B 2A 86 48 86 F7 0D 01 09 10 03 13 30 0B 06 09 60 86
         48 01 65 03 04 01 2D

      ECDH with HKDF using SHA-384; uses AES-256 key wrap:
         30 1A 06 0B 2A 86 48 86 F7 0D 01 09 10 03 14 30 0B 06 09 60 86
         48 01 65 03 04 01 2D

      ECDH with HKDF using SHA-512; uses AES-256 key wrap:
         30 1A 06 0B 2A 86 48 86 F7 0D 01 09 10 03 15 30 0B 06 09 60 86
         48 01 65 03 04 01 2D
```

9. Security Considerations

Please consult the security considerations of [CMS] for security considerations related to the enveloped-data content type and the authenticated-data content type.

Please consult the security considerations of [AUTHENV] for security considerations related to the authenticated-enveloped-data content type.

Please consult the security considerations of [CURVES] for security considerations related to the use of X25519 and X448.

The originator uses an ephemeral public/private key pair that is generated on the same elliptic curve as the public key of the recipient. The ephemeral key pair is used for a single CMS protected content type, and then it is discarded. If the originator wants to be able to decrypt the content (for enveloped-data and authenticated-enveloped-data) or check the authentication (for authenticated-data), then the originator needs to treat themselves as a recipient.

As specified in [CMS], implementations MUST support processing of the KeyAgreeRecipientInfo ukm field; this ensures that interoperability is not a concern whether the ukm is present or absent. The ukm is placed in the entityUInfo field of the ECC-CMS-SharedInfo structure. When present, the ukm ensures that a different key-encryption key is generated, even when the originator ephemeral private key is improperly used more than once.

10. IANA Considerations

One object identifier for the ASN.1 module in the Appendix 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-ecdh-alg-2017 OBJECT IDENTIFIER ::= {
      iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
      pkcs-9(9) smime(16) mod(0) 67 }
```

Three object identifiers for the Key Agreement Algorithm Identifiers in Sections 7 were assigned in the SMI Security for S/MIME Algorithms (1.2.840.113549.1.9.16.3) [IANA-ALG] registry:

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

   dhSinglePass-stdDH-hkdf-sha256-scheme OBJECT IDENTIFIER ::= {
      smime-alg 19 }

   dhSinglePass-stdDH-hkdf-sha384-scheme OBJECT IDENTIFIER ::= {
      smime-alg 20 }

   dhSinglePass-stdDH-hkdf-sha512-scheme OBJECT IDENTIFIER ::= {
      smime-alg 21 }
```

11. Normative References

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

[CERTCAP] Santesson, S., "X.509 Certificate Extension for Secure/Multipurpose Internet Mail Extensions (S/MIME) Capabilities", RFC 4262, December 2005.

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

[CMSASN1] Hoffman, P., and J. Schaad, "New ASN.1 Modules for Cryptographic Message Syntax (CMS) and S/MIME", RFC 5911, June 2010.

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

[CURVES]   Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, January 2016.

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

[ID.curdle-pkix]
Josefsson, S., and J. Schaad, "Algorithm Identifiers for Ed25519, Ed25519ph, Ed448, Ed448ph, X25519 and X448 for use in the Internet X.509 Public Key Infrastructure", 15 August 2016, Work-in-progress.

[PKIXALG] 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, April 2002.

[PROFILE] 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.

[SEC1]   Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", version 2.0, May 2009, <http://www.secg.org/sec1-v2.pdf>.

[SMIME]   Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.

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

[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.

12. Informative References

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

[AESKW] Schaad, J., and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, September 2002.

[CMSAES] Schaad, J., "Use of the Advanced Encryption Standard (AES) Encryption Algorithm in Cryptographic Message Syntax (CMS)", RFC 3565, July 2003.

[DH1976] Diffie, W., and M. E. Hellman, "New Directions in Cryptography", IEEE Trans. on Info. Theory, Vol. IT-22, Nov. 1976, pp. 644-654.

[IANA-ALG] https://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml#security-smime-3.

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

[X963] "Public-Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography", American National Standard X9.63-2001, 2001.

Appendix: ASN.1 Module

```
   CMSECDHAlgs-2017
     { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9)
       smime(16) modules(0) id-mod-cms-ecdh-alg-2017(67) }

   DEFINITIONS IMPLICIT TAGS ::=
   BEGIN

   -- EXPORTS ALL

   IMPORTS

     KeyWrapAlgorithm
       FROM CryptographicMessageSyntaxAlgorithms-2009  -- in [CMSASN1]
         { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
           pkcs-9(9) smime(16) modules(0) id-mod-cmsalg-2001-02(37) }

     KEY-AGREE, SMIME-CAPS
       FROM AlgorithmInformation-2009  -- in [CMSASN1]
         { iso(1) identified-organization(3) dod(6) internet(1)
           security(5) mechanisms(5) pkix(7) id-mod(0)
           id-mod-algorithmInformation-02(58) }

     dhSinglePass-stdDH-sha256kdf-scheme,
     dhSinglePass-stdDH-sha384kdf-scheme,
     dhSinglePass-stdDH-sha512kdf-scheme,
     kaa-dhSinglePass-stdDH-sha256kdf-scheme,
     kaa-dhSinglePass-stdDH-sha384kdf-scheme,
     kaa-dhSinglePass-stdDH-sha512kdf-scheme,
     cap-kaa-dhSinglePass-stdDH-sha256kdf-scheme,
     cap-kaa-dhSinglePass-stdDH-sha384kdf-scheme,
     cap-kaa-dhSinglePass-stdDH-sha512kdf-scheme
       FROM CMSECCAlgs-2009-02  -- in [CMSECC]
         { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1)
           pkcs-9(9) smime(16) modules(0)
           id-mod-cms-ecc-alg-2009-02(46) }
     ;
   --
   -- Object Identifiers
   --

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

   dhSinglePass-stdDH-hkdf-sha256-scheme OBJECT IDENTIFIER ::= {
      smime-alg 19 }

   dhSinglePass-stdDH-hkdf-sha384-scheme OBJECT IDENTIFIER ::= {
      smime-alg 20 }

   dhSinglePass-stdDH-hkdf-sha512-scheme OBJECT IDENTIFIER ::= {
      smime-alg 21 }

   --
   -- Extend the Key Agreement Algorithms in [CMSECC]
   --

   KeyAgreementAlgs KEY-AGREE ::= { ...,
     kaa-dhSinglePass-stdDH-sha256kdf-scheme   |
     kaa-dhSinglePass-stdDH-sha384kdf-scheme   |
     kaa-dhSinglePass-stdDH-sha512kdf-scheme   |
     kaa-dhSinglePass-stdDH-hkdf-sha256-scheme |
     kaa-dhSinglePass-stdDH-hkdf-sha384-scheme |
     kaa-dhSinglePass-stdDH-hkdf-sha512-scheme }

   kaa-dhSinglePass-stdDH-hkdf-sha256-scheme KEY-AGREE ::= {
     IDENTIFIER dhSinglePass-stdDH-hkdf-sha256-scheme
     PARAMS TYPE KeyWrapAlgorithm ARE required
     UKM -- TYPE unencoded data -- ARE preferredPresent
     SMIME-CAPS cap-kaa-dhSinglePass-stdDH-hkdf-sha256-scheme }

   kaa-dhSinglePass-stdDH-hkdf-sha384-scheme KEY-AGREE ::= {
     IDENTIFIER dhSinglePass-stdDH-hkdf-sha384-scheme
     PARAMS TYPE KeyWrapAlgorithm ARE required
     UKM -- TYPE unencoded data -- ARE preferredPresent
     SMIME-CAPS cap-kaa-dhSinglePass-stdDH-hkdf-sha384-scheme }

   kaa-dhSinglePass-stdDH-hkdf-sha512-scheme KEY-AGREE ::= {
     IDENTIFIER dhSinglePass-stdDH-hkdf-sha512-scheme
     PARAMS TYPE KeyWrapAlgorithm ARE required
     UKM -- TYPE unencoded data -- ARE preferredPresent
     SMIME-CAPS cap-kaa-dhSinglePass-stdDH-hkdf-sha512-scheme }
   --
   -- Extend the S/MIME CAPS in [CMSECC]
   --

   SMimeCAPS SMIME-CAPS ::= { ...,
     kaa-dhSinglePass-stdDH-sha256kdf-scheme.&smimeCaps   |
     kaa-dhSinglePass-stdDH-sha384kdf-scheme.&smimeCaps   |
     kaa-dhSinglePass-stdDH-sha512kdf-scheme.&smimeCaps   |
     kaa-dhSinglePass-stdDH-hkdf-sha256-scheme.&smimeCaps |
     kaa-dhSinglePass-stdDH-hkdf-sha384-scheme.&smimeCaps |
     kaa-dhSinglePass-stdDH-hkdf-sha512-scheme.&smimeCaps }

   cap-kaa-dhSinglePass-stdDH-hkdf-sha256-scheme SMIME-CAPS ::= {
     TYPE KeyWrapAlgorithm
     IDENTIFIED BY dhSinglePass-stdDH-hkdf-sha256-scheme }

   cap-kaa-dhSinglePass-stdDH-hkdf-sha384-scheme SMIME-CAPS ::= {
     TYPE KeyWrapAlgorithm
     IDENTIFIED BY dhSinglePass-stdDH-hkdf-sha384-scheme}

   cap-kaa-dhSinglePass-stdDH-hkdf-sha512-scheme SMIME-CAPS ::= {
     TYPE KeyWrapAlgorithm
     IDENTIFIED BY dhSinglePass-stdDH-hkdf-sha512-scheme }

   END
```

Acknowledgements

Many thanks to Roni Even, Daniel Migault, Eric Rescorla, Jim Schaad, Stefan Santesson, and Sean Turner for their review and insightful suggestions.

Author’s Address

Russ Housley
918 Spring Knoll Drive
Herndon, VA 20170
USA
housley@vigilsec.com

Internet-Draft R. Housley
Intended status: Standards Track Vigil Security
Expires: 11 April 2018 11 October 2017

Use of EdDSA Signatures in the Cryptographic Message Syntax (CMS)
<draft-ietf-curdle-cms-eddsa-signatures-08.txt>

Abstract

This document specifies the conventions for using Edwards-curve Digital Signature Algorithm (EdDSA) for curve25519 and curve448 in the Cryptographic Message Syntax (CMS). For each curve, EdDSA defines the PureEdDSA and HashEdDSA modes. However, the HashEdDSA mode is not used with the CMS. In addition, no context string is used with the CMS.

Status of This Memo

Copyright Notice

1. Introduction

This document specifies the conventions for using the Edwards-curve Digital Signature Algorithm (EdDSA) [RFC8032] for curve25519 [CURVE25519] and curve448 [CURVE448] with the Cryptographic Message Syntax (CMS) [RFC5652] signed-data content type. For each curve, [RFC8032] defines the PureEdDSA and HashEdDSA modes; however, the HashEdDSA mode is not used with the CMS. In addition, no context string is used with CMS. EdDSA with curve25519 is referred to as Ed25519, and EdDSA with curve448 is referred to as Ed448. The CMS conventions for PureEdDSA with Ed25519 and Ed448 are described in this document.

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].

2. EdDSA Signature Algorithm

The Edwards-curve Digital Signature Algorithm (EdDSA) [RFC8032] is a variant of Schnorr's signature system with (possibly twisted) Edwards curves. Ed25519 is intended to operate at around the 128-bit security level, and Ed448 at around the 224-bit security level.

One of the parameters of the EdDSA algorithm is the "prehash" function. This may be the identity function, resulting in an algorithm called PureEdDSA, or a collision-resistant hash function, resulting in an algorithm called HashEdDSA. In most situations the CMS SignedData includes signed attributes, including the message digest of the content. Since HashEdDSA offers no benefit when signed attributes are present, only PureEdDSA is used with the CMS.

2.1. Algorithm Identifiers

Each algorithm is identified by an object identifier, and the algorithm identifier may contain parameters if needed.

The ALGORITHM definition is repeated here for convenience:

```
   ALGORITHM ::= CLASS {
       &id    OBJECT IDENTIFIER UNIQUE,
       &Type  OPTIONAL }
     WITH SYNTAX {
       OID &id [PARMS &Type] }
```

2.2. EdDSA Algorithm Identifiers

The EdDSA signature algorithm is defined in [RFC8032], and the conventions for encoding the public key are defined in [CURDLE-PKIX].

The id-Ed25519 and id-Ed448 object identifiers are used to identify EdDSA public keys in certificates. The object identifiers are specified in [CURDLE-PKIX], and they are repeated here for convenience:

```
   sigAlg-Ed25519  ALGORITHM  ::=  { OID id-Ed25519 }

   sigAlg-Ed448    ALGORITHM  ::=  { OID id-Ed448 }

   id-Ed25519  OBJECT IDENTIFIER  ::=  { 1 3 101 112 }

   id-Ed448    OBJECT IDENTIFIER  ::=  { 1 3 101 113 }
```

2.3. Message Digest Algorithm Identifiers

When the signer includes signed attributes, a message digest algorithm is used to compute the message digest on the eContent value. When signing with Ed25519, the message digest algorithm MUST be SHA-512 [FIPS180]. Additional information on SHA-512 is available in RFC 6234 [RFC6234]. When signing with Ed448, the message digest algorithm MUST be SHAKE256 [FIPS202] with a 512-bit output value.

Signing with Ed25519 uses SHA-512 as part of the signing operation, and signing with Ed448 uses SHAKE256 as part of the signing operation.

For convenience, the object identifiers and parameter syntax for these algorithms are repeated here:

```
   hashAlg-SHA-512  ALGORITHM  ::=  { OID id-sha512 }

   hashAlg-SHAKE256  ALGORITHM  ::=  { OID id-shake256 }
```

```
      hashAlg-SHAKE256-LEN  ALGORITHM  ::=  { OID id-shake256-len
                               PARMS ShakeOutputLen }

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

      id-sha512  OBJECT IDENTIFIER  ::=  { hashAlgs 3 }

      id-shake256  OBJECT IDENTIFIER  ::=  { hashAlgs 12 }

      id-shake256-len  OBJECT IDENTIFIER  ::=  { hashAlgs 18 }

      ShakeOutputLen  ::=  INTEGER  -- Output length in bits
```

When using the id-sha512 or id-shake256 algorithm identifier, the parameters MUST be absent.

When using the id-shake256-len algorithm identifier, the parameters MUST be present, and the parameter MUST contain 512, encoded as a positive integer value.

2.4. EdDSA Signatures

The id-Ed25519 and id-Ed448 object identifiers are also used for signature values. When used to identify signature algorithms, the AlgorithmIdentifier parameters field MUST be absent.

The data to be signed is processed using PureEdDSA, and then a private key operation generates the signature value. As described in Section 3.3 of [RFC8032], the signature value is the opaque value ENC(R) || ENC(S), where || represents concatenation. As described in Section 5.3 of [RFC5652], the signature value is ASN.1 encoded as an OCTET STRING and included in the signature field of SignerInfo.

3. Signed-data Conventions

The processing depends on whether the signer includes signed attributes.

The inclusion of signed attributes is preferred, but the conventions for signed-data without signed attributes are provided for completeness.

3.1. Signed-data Conventions With Signed Attributes

The SignedData digestAlgorithms field includes the identifiers of the message digest algorithms used by one or more signer. There MAY be any number of elements in the collection, including zero. When signing with Ed25519, the digestAlgorithm SHOULD include id-sha512, and if present, the algorithm parameters field MUST be absent. When signing with Ed448, the digestAlgorithm SHOULD include id-shake256-len, and if present, the algorithm parameters field MUST also be present, and the parameter MUST contain 512, encoded as a positive integer value.

The SignerInfo digestAlgorithm field includes the identifier of the message digest algorithms used by the signer. When signing with Ed25519, the digestAlgorithm MUST be id-sha512, and the algorithm parameters field MUST be absent. When signing with Ed448, the digestAlgorithm MUST be id-shake256-len, the algorithm parameters field MUST be present, and the parameter MUST contain 512, encoded as a positive integer value.

The SignerInfo signedAttributes MUST include the message-digest attribute as specified in Section 11.2 of [RFC5652]. When signing with Ed25519, the message-digest attribute MUST contain the message digest computed over the eContent value using SHA-512. When signing with Ed448, the message-digest attribute MUST contain the message digest computed over the eContent value using SHAKE256 with an output length of 512 bits.

The SignerInfo signatureAlgorithm field MUST contain either id-Ed25519 or id-Ed448, depending on the elliptic curve that was used by the signer. The algorithm parameters field MUST be absent.

The SignerInfo signature field contains the octet string resulting from the EdDSA private key signing operation.

3.2. Signed-data Conventions Without Signed Attributes

The SignedData digestAlgorithms field includes the identifiers of the message digest algorithms used by one or more signer. There MAY be any number of elements in the collection, including zero. When signing with Ed25519, list of identifiers MAY include id-sha512, and if present, the algorithm parameters field MUST be absent. When signing with Ed448, list of identifiers MAY include id-shake256, and if present, the algorithm parameters field MUST be absent.

The SignerInfo digestAlgorithm field includes the identifier of the message digest algorithms used by the signer. When signing with Ed25519, the digestAlgorithm MUST be id-sha512, and the algorithm parameters field MUST be absent. When signing with Ed448, the digestAlgorithm MUST be id-shake256, and the algorithm parameters field MUST be absent.

NOTE: Either id-sha512 or id-shake256 is used as part to the private key signing operation. However, the private key signing operation does not take a message digest computed with one of these algorithms as an input.

The SignerInfo signatureAlgorithm field MUST contain either id-Ed25519 or id-Ed448, depending on the elliptic curve that was used by the signer. The algorithm parameters field MUST be absent.

The SignerInfo signature field contains the octet string resulting from the EdDSA private key signing operation.

4. Implementation Considerations

The EdDSA specification [RFC8032] includes the following warning. It deserves highlighting, especially when signed-data is used without signed attributes and the content to be signed might be quite large:

PureEdDSA requires two passes over the input. Many existing APIs, protocols, and environments assume digital signature algorithms only need one pass over the input, and may have API or bandwidth concerns supporting anything else.

5. Security Considerations

Implementations must protect the EdDSA private key. Compromise of the EdDSA private key may result in the ability to forge signatures.

The generation of EdDSA private key 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. RFC 4086 [RANDOM] offers important guidance in this area.

Unlike DSA and ECDSA, EdDSA does not require the generation of a random value for each signature operation.

Using the same private key with different algorithms has the potential to leak extra information about the private key to an attacker. For this reason, the same private key SHOULD NOT be used with more than one set of EdDSA parameters, although it appears that there are no security concerns when using the same private key with PureEdDSA and HashEdDSA [RFC8032].

When computing signatures, the same hash function SHOULD be used for all operations. This reduces the number of failure points in the signature process.

6. IANA Considerations

This document requires no actions by IANA.

7. Acknowledgements

Many thanks to Jim Schaad, Daniel Migault, and Adam Roach for the careful review and comments on the draft document. Thanks to Quynh Dang for coordinating the object identifiers assignment by NIST.

8. Normative References

[CURDLE-PKIX]
Josefsson, S., and J. Schaad, "Algorithm Identifiers for Ed25519, Ed25519ph, Ed448, Ed448ph, X25519 and X448 for use in the Internet X.509 Public Key Infrastructure", draft-ietf-curdle-pkix-02, 31 October 2016, Work-in-progress.

[FIPS180] National Institute of Standards and Technology, U.S. Department of Commerce, "Secure Hash Standard", Federal Information Processing Standard (FIPS) 180-3, October 2008.

[FIPS202] National Institute of Standards and Technology, U.S. Department of Commerce, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Federal Information Processing Standard (FIPS) 202, August 2015.

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

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

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", RFC 8032, January 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.

9. Informative References

[CURVE25519] Bernstein, D., "Curve25519: new Diffie-Hellman speed records", DOI 10.1007/11745853_14, February 2006, <http://cr.yp.to/ecdh.html>.

[CURVE448] Hamburg, M., "Ed448-Goldilocks, a new elliptic curve", June 2015, <http://eprint.iacr.org/2015/625>.

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

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

Author’s Address

Russ Housley
918 Spring Knoll Drive
Herndon, VA 20170
USA
housley@vigilsec.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 9, 2018

S. Josefsson
SJD AB
J. Schaad
August Cellars

May 8, 2018

Algorithm Identifiers for Ed25519, Ed448, X25519 and X448 for use in the Internet X.509 Public Key Infrastructure
draft-ietf-curdle-pkix-10

Abstract

This document specifies algorithm identifiers and ASN.1 encoding formats for Elliptic Curve constructs using the curve25519 and curve448 curves. The signature algorithms covered are Ed25519 and Ed448. The key agreement algorithm covered are X25519 and X448. The encoding for Public Key, Private Key and EdDSA digital signature structures is provided.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

In [RFC7748], the elliptic curves curve25519 and curve448 are described. They are designed with performance and security in mind. The curves may be used for Diffie-Hellman and Digital Signature operations.

[RFC7748] describes the operations on these curves for the Diffie-Hellman operation. A convention has developed that when these two curves are used with the Diffie-Hellman operation, they are referred to as X25519 and X448. This RFC defines the ASN.1 Object Identifiers (OIDs) for the operations X25519 and X448 along with the associated parameters. The use of these OIDs is described for public and private keys.

In [RFC8032] the elliptic curve signature system Edwards-curve Digital Signature Algorithm (EdDSA) is described along with a recommendation for the use of the curve25519 and curve448. EdDSA has defined two modes, the PureEdDSA mode without pre-hashing, and the HashEdDSA mode with pre-hashing. The convention used for identifying the algorithm/curve combinations is to use "Ed25519" and "Ed448" for the PureEdDSA mode. The document does not provide the conventions needed for the pre-hash versions of the signature algorithm. The use of the OIDs is described for public keys, private keys and signatures.

[RFC8032] additionally defined the concept of a context. Contexts can be used to differentiate signatures generated for different purposes with the same key. The use of contexts is not defined in this document for the following reasons:

- o The current implementations of Ed25519 do not support the use of contexts, thus if specified it will potentially delay the use of these algorithms further.

- o The EdDSA algorithms are the only IETF algorithms that currently support the use of contexts, however there is a possibility that there will be confusion between which algorithms need to have separate keys and which do not. This may result in a decrease of security for those other algorithms.

- o There are still ongoing discussions among the cryptographic community about how effective the use of contexts is for preventing attacks.

- o There needs to be discussions about the correct way to identify when context strings are to be used. It is not clear if different OIDs should be used for different contexts, or the OID should merely note that a context string needs to be provided.

2. Requirements 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. Curve25519 and Curve448 Algorithm Identifiers

Certificates conforming to [RFC5280] can convey a public key for any public key algorithm. The certificate indicates the algorithm through an algorithm identifier. This algorithm identifier is an OID and optionally associated parameters.

The AlgorithmIdentifier type, which is included for convenience, is defined as follows:

```
   AlgorithmIdentifier  ::=  SEQUENCE  {
       algorithm   OBJECT IDENTIFIER,
       parameters  ANY DEFINED BY algorithm OPTIONAL
   }
```

The fields in AlgorithmIdentifier have the following meanings:

- o algorithm identifies the cryptographic algorithm with an object identifier. Four such OIDs are defined below.

- o parameters, which are optional, are the associated parameters for the algorithm identifier in the algorithm field.

In this document we define four new OIDs for identifying the different curve/algorithm pairs. The curves being curve25519 and curve448. The algorithms being ECDH and EdDSA in pure mode. For all of the OIDs, the parameters MUST be absent.

It is possible to find systems that require the parameters to be present. This can be either due to a defect in the original 1997 syntax or a programming error where developers never got input where this was not true. The optimal solution is to fix these systems, where this is not possible the problem needs to be restricted to that subsystem and not propigated to the internet.

The same algorithm identifiers are used for identifying a public key, identifying a private key and identifying a signature (for the two EdDSA related OIDs). Additional encoding information is provided below for each of these locations.

```
   id-X25519    OBJECT IDENTIFIER ::= { 1 3 101 110 }
   id-X448      OBJECT IDENTIFIER ::= { 1 3 101 111 }
   id-Ed25519   OBJECT IDENTIFIER ::= { 1 3 101 112 }
   id-Ed448     OBJECT IDENTIFIER ::= { 1 3 101 113 }
```

4. Subject Public Key Fields

In the X.509 certificate, the subjectPublicKeyInfo field has the SubjectPublicKeyInfo type, which has the following ASN.1 syntax:

```
   SubjectPublicKeyInfo  ::=  SEQUENCE  {
       algorithm         AlgorithmIdentifier,
       subjectPublicKey  BIT STRING
   }
```

The fields in SubjectPublicKeyInfo have the following meanings:

- algorithm is the algorithm identifier and parameters for the public key (see above).

- subjectPublicKey contains the byte stream of the public key. The algorithms defined in this document always encode the public key as an exact multiple of 8-bits.

Both [RFC7748] and [RFC8032] define the public key value as being a byte string. It should be noted that the public key is computed differently for each of these documents, thus the same private key will not produce the same public key.

The following is an example of a public key encoded using the textual encoding defined in [RFC7468].

```
-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEAGb9ECWmEzf6FQbrBZ9w7lshQhqowtrbLDFw4rXAxZuE=
-----END PUBLIC KEY-----
```

5. Key Usage Bits

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

If the keyUsage extension is present in a certificate that indicates id-X25519 or id-X448 in SubjectPublicKeyInfo, then the following MUST be present:

```
        keyAgreement;
```

one of the following MAY also be present:

```
          encipherOnly; or
          decipherOnly.
```

If the keyUsage extension is present in an end-entity certificate that indicates id-Ed25519 or id-Ed448, 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-Ed25519 or id-Ed448, then the keyUsage extension MUST contain one or more of the following values:

```
         nonRepudiation;
         digitalSignature;
         keyCertSign; and
         cRLSign.
```

6. EdDSA Signatures

Signatures can be placed in a number of different ASN.1 structures. The top level structure for a certificate is given below as being illustrative of how signatures are frequently encoded with an algorithm identifier and a location for the signature.

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

The same algorithm identifiers are used for signatures as are used for public keys. When used to identify signature algorithms, the parameters MUST be absent.

The data to be signed is prepared for EdDSA. Then, a private key operation is performed to generate the signature value. This value is the opaque value ENC(R) || ENC(S) described in section 3.3 of [RFC8032]. The octet string representing the signature is encoded directly in the BIT STRING without adding any additional ASN.1 wrapping. For the Certificate structure, the signature value is wrapped in the "signatureValue" BIT STRING field.

7. Private Key Format

Asymmetric Key Packages [RFC5958] describes how to encode a private key in a structure that both identifies what algorithm the private key is for, but allows for the public key and additional attributes about the key to be included as well. For illustration, the ASN.1 structure OneAsymmetricKey is replicated below. The algorithm specific details of how a private key is encoded is left for the document describing the algorithm itself.

```
   OneAsymmetricKey ::= SEQUENCE {
      version Version,
      privateKeyAlgorithm PrivateKeyAlgorithmIdentifier,
      privateKey PrivateKey,
      attributes [0] IMPLICIT Attributes OPTIONAL,
      ...,
      [[2: publicKey [1] IMPLICIT PublicKey OPTIONAL ]],
      ...
   }

   PrivateKey ::= OCTET STRING

   PublicKey ::= BIT STRING
```

For the keys defined in this document, the private key is always an opaque byte sequence. The ASN.1 type CurvePrivateKey is defined in this document to hold the byte sequence. Thus when encoding a OneAsymmetricKey object, the private key is wrapped in an CurvePrivateKey object and wrapped by the OCTET STRING of the "privateKey" field.

```
   CurvePrivateKey ::= OCTET STRING
```

To encode a EdDSA, X25519 or X448 private key, the "privateKey" field will hold the encoded private key. The "privateKeyAlgorithm" field uses the AlgorithmIdentifier structure. The structure is encoded as defined above. If present, the "publicKey" field will hold the encoded key as defined in [RFC7748] and [RFC8032].

The following is an example of a private key encoded using the textual encoding defined in [RFC7468].

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEINTuctv5E1hK1bbY8fdp+K06/nwoy/HU++CXqI9EdVhC
-----END PRIVATE KEY-----
```

The following example, in addition to encoding the private key, additionally has an attribute included as well as the public key. As with the prior example, the textual encoding defined in [RFC7468] is used.

```
-----BEGIN PRIVATE KEY-----
MHICAQEwBQYDK2VwBCIEINTuctv5E1hK1bbY8fdp+K06/nwoy/HU++CXqI9EdVhC
oB8wHQYKKoZIhvcNAQkJFDEPDA1DdXJkbGUgQ2hhaXJzgSEAGb9ECWmEzf6FQbrB
Z9w7lshQhqowtrbLDFw4rXAxZuE=
-----END PRIVATE KEY------
```

NOTE: There exist some private key import functions that have not picked up the new ASN.1 structure OneAsymmetricKey that is defined in [RFC7748]. This means that they will not accept a private key structure which contains the public key field. This means a balancing act needs to be done between being able to do a consistency check on the key pair and widest ability to import the key.

8. Human Readable Algorithm Names

For the purpose of consistent cross-implementation naming, this section establishes human readable names for the algorithms specified in this document. Implementations SHOULD use these names when referring to the algorithms. If there is a strong reason to deviate from these names -- for example, if the implementation has a different naming convention and wants to maintain internal consistency -- it is encouraged to deviate as little as possible from the names given here.

Use the string "ECDH" when referring to a public key of type "X25519" or "X448" when the curve is not known or relevant.

When the curve is known, use the more specific string of "X25519" or "X448".

Use the string "EdDSA" when referring to a signing public key or signature when the curve is not known or relevant.

When the curve is known, use a more specific string. For the id-Ed25519 value use the string "Ed25519". For id-Ed448 use "Ed448".

9. ASN.1 Module

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

```
-- ASN.1 Module

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

DEFINITIONS EXPLICIT TAGS ::=
BEGIN

IMPORTS
  SIGNATURE-ALGORITHM, KEY-AGREE, PUBLIC-KEY, KEY-WRAP,
  KeyUsage, AlgorithmIdentifier
  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)}

     mda-sha512
     FROM PKIX1-PSS-OAEP-Algorithms-2009
       { 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) }

     kwa-aes128-wrap, kwa-aes256-wrap
     FROM CMSAesRsaesOaep-2009
       { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9)
         smime(16) modules(0) id-mod-cms-aes-02(38) }
     ;


   id-edwards-curve-algs OBJECT IDENTIFIER ::= { 1 3 101 }

   id-X25519        OBJECT IDENTIFIER ::= { id-edwards-curve-algs 110 }
   id-X448          OBJECT IDENTIFIER ::= { id-edwards-curve-algs 111 }
   id-Ed25519       OBJECT IDENTIFIER ::= { id-edwards-curve-algs 112 }
   id-Ed448         OBJECT IDENTIFIER ::= { id-edwards-curve-algs 113 }


    sa-Ed25519 SIGNATURE-ALGORITHM ::= {
       IDENTIFIER id-Ed25519
        PARAMS ARE absent
        PUBLIC-KEYS {pk-Ed25519}
        SMIME-CAPS { IDENTIFIED BY id-Ed25519 }
    }

    pk-Ed25519 PUBLIC-KEY ::= {
        IDENTIFIER id-Ed25519
        -- KEY no ASN.1 wrapping --
        PARAMS ARE absent
        CERT-KEY-USAGE {digitalSignature, nonRepudiation,
                        keyCertSign, cRLSign}
        PRIVATE-KEY CurvePrivateKey
    }

    kaa-X25519 KEY-AGREE ::= {
        IDENTIFIER id-X25519
        PARAMS ARE absent
        PUBLIC-KEYS {pk-X25519}
        UKM -- TYPE no ASN.1 wrapping -- ARE preferredPresent
        SMIME-CAPS {
           TYPE AlgorithmIdentifier{KEY-WRAP, {KeyWrapAlgorithms}}
           IDENTIFIED BY id-X25519 }
    }
    pk-X25519 PUBLIC-KEY ::= {
        IDENTIFIER id-X25519
        -- KEY no ASN.1 wrapping --
        PARAMS ARE absent
        CERT-KEY-USAGE { keyAgreement }
        PRIVATE-KEY CurvePrivateKey
    }

    KeyWrapAlgorithms KEY-WRAP ::= {
        kwa-aes128-wrap | kwa-aes256-wrap,
        ...
    }

    kaa-X448 KEY-AGREE ::= {
        IDENTIFIER id-X448
        PARAMS ARE absent
        PUBLIC-KEYS {pk-X448}
        UKM -- TYPE no ASN.1 wrapping  -- ARE preferredPresent
        SMIME-CAPS {
           TYPE AlgorithmIdentifier{KEY-WRAP, {KeyWrapAlgorithms}}
           IDENTIFIED BY id-X448 }
    }

    pk-X448 PUBLIC-KEY ::= {
        IDENTIFIER id-X448
        -- KEY no ASN.1 wrapping --
        PARAMS ARE absent
        CERT-KEY-USAGE { keyAgreement }
        PRIVATE-KEY CurvePrivateKey
    }

   CurvePrivateKey ::= OCTET STRING


   END
```

10. Examples

This section contains illustrations of EdDSA public keys and certificates, illustrating parameter choices.

10.1. Example Ed25519 Public Key

An example of a Ed25519 public key:

```
         Public Key Information:
             Public Key Algorithm: Ed25519
             Algorithm Security Level: High

         Public Key Usage:

         Public Key ID: 9b1f5eeded043385e4f7bc623c5975b90bc8bb3b

         -----BEGIN PUBLIC KEY-----
         MCowBQYDK2VwAyEAGb9ECWmEzf6FQbrBZ9w7lshQhqowtrbLDFw4rXAxZuE=
         -----END PUBLIC KEY-----
```

10.2. Example X25519 Certificate

An example of a self issued PKIX certificate using Ed25519 to sign a X25519 public key would be:

```
     0 300: SEQUENCE {
     4 223:   SEQUENCE {
     7   3:     [0] {
     9   1:       INTEGER 2
          :       }
    12   8:     INTEGER 56 01 47 4A 2A 8D C3 30
    22   5:     SEQUENCE {
    24   3:       OBJECT IDENTIFIER
          :         Ed 25519 signature algorithm { 1 3 101 112 }
          :       }
    29  25:     SEQUENCE {
    31  23:       SET {
    33  21:         SEQUENCE {
    35   3:           OBJECT IDENTIFIER commonName (2 5 4 3)
    40  14:           UTF8String 'IETF Test Demo'
          :           }
          :         }
          :       }
    56  30:     SEQUENCE {
    58  13:       UTCTime 01/08/2016 12:19:24 GMT
    73  13:       UTCTime 31/12/2040 23:59:59 GMT
          :       }
    88  25:     SEQUENCE {
    90  23:       SET {
    92  21:         SEQUENCE {
    94   3:           OBJECT IDENTIFIER commonName (2 5 4 3)
    99  14:           UTF8String 'IETF Test Demo'
          :           }
          :         }
          :       }
   115  42:     SEQUENCE {
```

```
   117   5:       SEQUENCE {
   119   3:         OBJECT IDENTIFIER
          :           ECDH 25519 key agreement { 1 3 101 110 }
          :         }
   124  33:       BIT STRING
          :         85 20 F0 09 89 30 A7 54 74 8B 7D DC B4 3E F7 5A
          :         0D BF 3A 0D 26 38 1A F4 EB A4 A9 8E AA 9B 4E 6A
          :       }
   159  69:     [3] {
   161  67:       SEQUENCE {
   163  15:         SEQUENCE {
   165   3:           OBJECT IDENTIFIER basicConstraints (2 5 29 19)
   170   1:           BOOLEAN TRUE
   173   5:           OCTET STRING, encapsulates {
   175   3:             SEQUENCE {
   177   1:               BOOLEAN FALSE
          :               }
          :             }
          :           }
   180  14:         SEQUENCE {
   182   3:           OBJECT IDENTIFIER keyUsage (2 5 29 15)
   187   1:           BOOLEAN FALSE
   190   4:           OCTET STRING, encapsulates {
   192   2:             BIT STRING 3 unused bits
          :               '10000'B (bit 4)
          :             }
          :           }
   196  32:         SEQUENCE {
   198   3:           OBJECT IDENTIFIER subjectKeyIdentifier (2 5 29 14)
   203   1:           BOOLEAN FALSE
   206  22:           OCTET STRING, encapsulates {
   208  20:             OCTET STRING
          :               9B 1F 5E ED ED 04 33 85 E4 F7 BC 62 3C 59 75
          :               B9 0B C8 BB 3B
          :             }
          :           }
          :         }
          :       }
          :     }
   230   5:   SEQUENCE {
   232   3:     OBJECT IDENTIFIER
          :       Ed 25519 signature algorithm { 1 3 101 112 }
          :     }
   237  65:   BIT STRING
          :     AF 23 01 FE DD C9 E6 FF C1 CC A7 3D 74 D6 48 A4
          :     39 80 82 CD DB 69 B1 4E 4D 06 EC F8 1A 25 CE 50
          :     D4 C2 C3 EB 74 6C 4E DD 83 46 85 6E C8 6F 3D CE
          :     1A 18 65 C5 7A C2 7B 50 A0 C3 50 07 F5 E7 D9 07
          :   }
```

```
-----BEGIN CERTIFICATE-----
MIIBLDCB36ADAgECAghWAUdKKo3DMDAFBgMrZXAwGTEXMBUGA1UEAwwOSUVURiBUZX
N0IERlbW8wHhcNMTYwODAxMTIxOTI0WhcNNDAxMjMxMjM1OTU5WjAZMRcwFQYDVQQD
DA5JRVRGIFRlc3QgRGVtbzAqMAUGAytlbgMhAIUg8AmJMKdUdIt93LQ+91oNvzoNJj
ga9OukqY6qm05qo0UwQzAPBgNVHRMBAf8EBTADAQEAMA4GA1UdDwEBAAQEAwIDCDAg
BgNVHQ4BAQAEFgQUmx9e7e0EM4Xk97xiPFl1uQvIuzswBQYDK2VwA0EAryMB/t3J5v
/BzKc9dNZIpDmAgs3babFOTQbs+BolzlDUwsPrdGxO3YNGhW7Ibz3OGhhlxXrCe1Cg
w1AH9efZBw==
-----END CERTIFICATE-----
```

10.3. Examples of Ed25519 Private Key

An example of an Ed25519 private key without the public key:

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEINTuctv5E1hK1bbY8fdp+K06/nwoy/HU++CXqI9EdVhC
-----END PRIVATE KEY-----
```

The same item dumped as ASN.1 yields:

```
 0 30   46: SEQUENCE {
 2 02    1:   INTEGER 0
 5 30    5:   SEQUENCE {
 7 06    3:     OBJECT IDENTIFIER
          :       Ed 25519 signature algorithm { 1 3 101 112 }
          :     }
12 04   34:   OCTET STRING
          :     04 20 D4 EE 72 DB F9 13 58 4A D5 B6 D8 F1 F7 69
          :     F8 AD 3A FE 7C 28 CB F1 D4 FB E0 97 A8 8F 44 75
          :     58 42
          :   }
```

Note that the value of the private key is:

```
D4 EE 72 DB F9 13 58 4A D5 B6 D8 F1 F7 69 F8 AD
3A FE 7C 28 CB F1 D4 FB E0 97 A8 8F 44 75 58 42
```

An example of the same Ed25519 private key encoded with an attribute and the public key:

```
-----BEGIN PRIVATE KEY-----
MHICAQEwBQYDK2VwBCIEINTuctv5E1hK1bbY8fdp+K06/nwoy/HU++CXqI9EdVhC
oB8wHQYKKoZIhvcNAQkJFDEPDA1DdXJkbGUgQ2hhaXJzgSEAGb9ECWmEzf6FQbrB
Z9w7lshQhqowtrbLDFw4rXAxZuE=
-----END PRIVATE KEY-----
```

The same item dumped as ASN.1 yields:

```
     0 114: SEQUENCE {
     2   1:   INTEGER 1
     5   5:   SEQUENCE {
     7   3:     OBJECT IDENTIFIER '1 3 101 112'
          :     }
    12  34:   OCTET STRING, encapsulates {
          :     04 20 D4 EE 72 DB F9 13 58 4A D5 B6 D8 F1 F7 69
          :     F8 AD 3A FE 7C 28 CB F1 D4 FB E0 97 A8 8F 44 75
          :     58 42
          :     }
    48  31:   [0] {
    50  29:     SEQUENCE {
    52  10:       OBJECT IDENTIFIER '1 2 840 113549 1 9 9 20'
    64  15:       SET {
    66  13:         UTF8String 'Curdle Chairs'
          :         }
          :       }
          :     }
   81  33:   [1] 00 19 BF 44 09 69 84 CD FE 85 41 BA C1 67 DC 3B
                 96 C8 50 86 AA 30 B6 B6 CB 0C 5C 38 AD 70 31 66
                 E1
          :   }
```

11. Acknowledgments

Text and/or inspiration were drawn from [RFC5280], [RFC3279], [RFC4055], [RFC5480], and [RFC5639].

The following people discussed the document and provided feedback: Klaus Hartke, Ilari Liusvaara, Erwann Abalea, Rick Andrews, Rob Stradling, James Manger, Nikos Mavrogiannopoulos, Russ Housley, David Benjamin, Brian Smith, and Alex Wilson.

A big thank you to Symantec for kindly donating the OIDs used in this draft.

12. 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 9.

The OIDs are being independently registered in the IANA registry "SMI Security for Cryptographic Algorithms" in [I-D.schaad-curdle-oid-registry].

13. Security Considerations

The security considerations of [RFC5280], [RFC7748], and [RFC8032] apply accordingly.

The procedures for going from a private key to a public key are different for when used with Diffie-Hellman and when used with Edwards Signatures. This means that the same public key cannot be used for both ECDH and EdDSA.

14. References

14.1. 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>.

[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>.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, DOI 10.17487/RFC5958, August 2010, <https://www.rfc-editor.org/info/rfc5958>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[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>.

14.2. Informative References

[I-D.schaad-curdle-oid-registry]
Schaad, J. and R. Andrews, "IANA Registration for new Cryptographic Algorithm Object Identifier Range", draft-schaad-curdle-oid-registry-03 (work in progress), January 2018.

[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>.

[RFC5639] Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <https://www.rfc-editor.org/info/rfc5639>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

Appendix A. Invalid Encodings

There are a number of things that need to be dealt with when a new key part is decoded and imported into the system. A partial list of these includes:

- o ASN.1 encoding errors: Two items are highlighted here. First, the use of an OCTET STRING rather than a BIT STRING for the public key. This was an incorrect copy of the structure from [RFC5958] which was corrected before publication. However, any early implementation may have this wrong. Second, the value of the version field is required to be 0 if the publicKey is absent and 1 if present. This is called out in [RFC5958] but is not duplicated in the main text.

- o Key encoding errors: Both [RFC7748] and [RFC8032] have formatting requirements for keys that need to be enforced. In some cases the enforcement is done at the time of importing, for example doing masking or a mod p operation. In other cases the enforcement is done by rejecting the keys and having an import failure.

- Key mismatch errors: If a public key is provided, it may not agree with the private key either because it is wrong or the wrong algorithm was used.

Some systems are also going to be stricter on what they accept. As stated in [RFC5958], BER decoding of OneAsymmetricKey objects is a requirement for compliance. Despite this requirement, some acceptors will only decode DER formats. The following is a BER encoding of a private key, as such is is valid, but it may not be accepted by many systems.

```
-----BEGIN PRIVATE KEY-----
MIACAQAwgAYDK2VwAAAEIgQg1O5y2/kTWErVttjx92n4rTr+fCjL8dT74Jeoj0R1W
EIAAA==
-----END PRIVATE KEY-----
```

What follows here is a brief sampling of some incorrect keys.

In the following example, the private key does not match the masking requirements for X25519. For this example the top bits are set to zero and the bottom three bits are set to 001.

```
-----BEGIN PRIVATE KEY-----
MFMCAQEwBQYDK2VuBCIEIPj///////////////////////////////////////8/oS
MDIQCEfA0sN1I082XmYJVRh6NzWg92E9FgnTpqTYxTrqpaIg==
-----END PRIVATE KEY-----
```

In the following examples, the key is the wrong length because an all zero byte has been removed. In one case the first byte has been removed, in the other case the last byte has been removed.

```
-----BEGIN PRIVATE KEY-----
MFICAQEwBQYDK2VwBCIEIC3GfeUYbZGTAhwLEE2cbvJL7ivTlcy17VottfN6L8HwoS
IDIADBfk2Lv/J8H7YYwj/OmIcDx++jzVkKrKwS0/HjyQyM
-----END PRIVATE KEY-----
```

```
-----BEGIN PRIVATE KEY-----
MFICAQEwBQYDK2VwBCIEILJXn1VaLqvausjUaZexwI/ozmOFjfEk78KcYN+7hsNJoS
IDIACdQhJwzi/MCGcsQeQnIUh2JFybDxSrZxuLudJmpJLk
-----END PRIVATE KEY-----
```

Authors' Addresses

Simon Josefsson
SJD AB

Email: simon@josefsson.org

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

Internet-Draft
Updates: 4252, 4253 (if approved)
Intended status: Standards Track
Expires: April 12, 2018

D. Bider
Bitvise Limited
October 12, 2017

Use of RSA Keys with SHA-256 and SHA-512 in Secure Shell (SSH)
draft-ietf-curdle-rsa-sha2-12.txt

Abstract

This memo updates RFC 4252 and RFC 4253 to define new public key algorithms for use of RSA keys with SHA-256 and SHA-512 for server and client authentication in SSH connections.

Status

Copyright

1. Overview and Rationale

Secure Shell (SSH) is a common protocol for secure communication on the Internet. In [RFC4253], SSH originally defined the public key algorithms "ssh-rsa" for server and client authentication using RSA with SHA-1, and "ssh-dss" using 1024-bit DSA and SHA-1. These algorithms are now considered deficient. For US government use, NIST has disallowed 1024-bit RSA and DSA, and use of SHA-1 for signing [800-131A].

This memo updates RFC 4252 and RFC 4253 to define new public key algorithms allowing for interoperable use of existing and new RSA keys with SHA-256 and SHA-512.

1.1. Requirements Terminology

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].

1.2. Wire Encoding Terminology

The wire encoding types in this document - "boolean", "byte", "string", "mpint" - have meanings as described in [RFC4251].

2. Public Key Format vs. Public Key Algorithm

In [RFC4252], the concept "public key algorithm" is used to establish a relationship between one algorithm name, and:

A. Procedures used to generate and validate a private/public keypair.
B. A format used to encode a public key.
C. Procedures used to calculate, encode, and verify a signature.

This document uses the term "public key format" to identify only A and B in isolation. The term "public key algorithm" continues to identify all three aspects A, B, and C.

3. New RSA Public Key Algorithms

This memo adopts the style and conventions of [RFC4253] in specifying how use of a public key algorithm is indicated in SSH.

The following new public key algorithms are defined:

rsa-sha2-256	RECOMMENDED	sign	Raw RSA key
rsa-sha2-512	OPTIONAL	sign	Raw RSA key

These algorithms are suitable for use both in the SSH transport layer [RFC4253] for server authentication, and in the authentication layer [RFC4252] for client authentication.

Since RSA keys are not dependent on the choice of hash function, the new public key algorithms reuse the "ssh-rsa" public key format as defined in [RFC4253]:

```
string    "ssh-rsa"
mpint     e
mpint     n
```

All aspects of the "ssh-rsa" format are kept, including the encoded string "ssh-rsa". This allows existing RSA keys to be used with the new public key algorithms, without requiring re-encoding, or affecting already trusted key fingerprints.

Signing and verifying using these algorithms is performed according to the RSASSA-PKCS1-v1_5 scheme in [RFC8017] using SHA-2 [SHS] as hash.

For the algorithm "rsa-sha2-256", the hash used is SHA-256.
For the algorithm "rsa-sha2-512", the hash used is SHA-512.

The resulting signature is encoded as follows:

```
string    "rsa-sha2-256" / "rsa-sha2-512"
string    rsa_signature_blob
```

The value for ’rsa_signature_blob’ is encoded as a string containing S - an octet string which is the output of RSASSA-PKCS1-v1_5, of length equal to the length in octets of the RSA modulus.

3.1. Use for server authentication

To express support and preference for one or both of these algorithms for server authentication, the SSH client or server includes one or both algorithm names, "rsa-sha2-256" and/or "rsa-sha2-512", in the name-list field "server_host_key_algorithms" in the SSH_MSG_KEXINIT packet [RFC4253]. If one of the two host key algorithms is negotiated, the server sends an "ssh-rsa" public key as part of the negotiated key exchange method (e.g. in SSH_MSG_KEXDH_REPLY), and encodes a signature with the appropriate signature algorithm name - either "rsa-sha2-256", or "rsa-sha2-512".

3.2. Use for client authentication

To use this algorithm for client authentication, the SSH client sends an SSH_MSG_USERAUTH_REQUEST message [RFC4252] encoding the "publickey" method, and encoding the string field "public key algorithm name" with the value "rsa-sha2-256" or "rsa-sha2-512". The "public key blob" field encodes the RSA public key using the "ssh-rsa" public key format.

For example, as defined in [RFC4252] and [RFC4253], an SSH "publickey" authentication request using an "rsa-sha2-512" signature would be properly encoded as follows:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    user name
string    service name
string    "publickey"
boolean   TRUE
string    "rsa-sha2-512"
string    public key blob:
    string    "ssh-rsa"
    mpint     e
    mpint     n
string    signature:
    string    "rsa-sha2-512"
    string    rsa_signature_blob
```

If the client includes the signature field, the client MUST encode the same algorithm name in the signature as in SSH_MSG_USERAUTH_REQUEST - either "rsa-sha2-256", or "rsa-sha2-512". If a server receives a mismatching request, it MAY apply arbitrary authentication penalties, including but not limited to authentication failure or disconnect.

OpenSSH 7.2 (but not 7.2p2) incorrectly encodes the algorithm in the signature as "ssh-rsa" when the algorithm in SSH_MSG_USERAUTH_REQUEST is "rsa-sha2-256" or "rsa-sha2-512". In this case, the signature does actually use either SHA-256 or SHA-512. A server MAY, but is not required to, accept this variant, or another variant that corresponds to a good-faith implementation, and is decided to be safe to accept.

3.3. Discovery of public key algorithms supported by servers

Implementation experience has shown that there are servers which apply authentication penalties to clients attempting public key algorithms which the SSH server does not support.

Servers that accept rsa-sha2-* signatures for client authentication SHOULD implement the extension negotiation mechanism defined in [EXT-INFO], including especially the "server-sig-algs" extension.

When authenticating with an RSA key against a server that does not implement the "server-sig-algs" extension, clients MAY default to an "ssh-rsa" signature to avoid authentication penalties. When the new rsa-sha2-* algorithms have been sufficiently widely adopted to warrant disabling "ssh-rsa", clients MAY default to one of the new algorithms.

4. IANA Considerations

IANA is requested to update the "Secure Shell (SSH) Protocol Parameters" registry established with [RFC4250], to extend the table Public Key Algorithm Names [IANA-PKA]:

- To the immediate right of the column Public Key Algorithm Name, a new column is to be added, titled Public Key Format. For existing entries, the column Public Key Format should be assigned the same value found under Public Key Algorithm Name.

- Immediately following the existing entry for "ssh-rsa", two sibling entries are to be added:

P. K. Alg. Name	P. K. Format	Reference	Note
rsa-sha2-256	ssh-rsa	[this document]	Section 3
rsa-sha2-512	ssh-rsa	[this document]	Section 3

5. Security Considerations

The security considerations of [RFC4251] apply to this document.

5.1. Key Size and Signature Hash

The National Institute of Standards and Technology (NIST) Special Publication 800-131A, Revision 1 [800-131A], disallows the use of RSA and DSA keys shorter than 2048 bits for US government use. The same document disallows the SHA-1 hash function for digital signature generation, except under NIST’s protocol-specific guidance.

It is prudent to follow this advice also outside of US government use.

5.2. Transition

This document is based on the premise that RSA is used in environments where a gradual, compatible transition to improved algorithms will be better received than one that is abrupt and incompatible. It advises that SSH implementations add support for new RSA public key algorithms along with SSH_MSG_EXT_INFO and the "server-sig-algs" extension to allow coexistence of new deployments with older versions that support only "ssh-rsa". Nevertheless, implementations SHOULD start to disable "ssh-rsa" in their default configurations as soon as they have reason to believe that new RSA signature algorithms have been widely adopted.

5.3. PKCS#1 v1.5 Padding and Signature Verification

This document prescribes RSASSA-PKCS1-v1_5 signature padding because:

(1) RSASSA-PSS is not universally available to all implementations;
(2) PKCS#1 v1.5 is widely supported in existing SSH implementations;
(3) PKCS#1 v1.5 is not known to be insecure for use in this scheme.

Implementers are advised that a signature with PKCS#1 v1.5 padding MUST NOT be verified by applying the RSA key to the signature, and then parsing the output to extract the hash. This may give an attacker opportunities to exploit flaws in the parsing and vary the encoding. Verifiers MUST instead apply PKCS#1 v1.5 padding to the expected hash, then compare the encoded bytes with the output of the RSA operation.

6. References

6.1. Normative References

[SHS] National Institute of Standards and Technology (NIST), United States of America, "Secure Hash Standard (SHS)", FIPS Publication 180-4, August 2015, <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.

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

[RFC4251] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[RFC4252] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, January 2006.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

[EXT-INFO] Bider, D., "Extension Negotiation in Secure Shell (SSH)", draft-ietf-curdle-ssh-ext-info-15.txt, September 2017, <https://tools.ietf.org/html/draft-ietf-curdle-ssh-ext-info-15>.

6.2. Informative References

[800-131A] National Institute of Standards and Technology (NIST), "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths", NIST Special Publication 800-131A, Revision 1, November 2015, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf>.

[RFC4250] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, January 2006.

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

[IANA-PKA] "Secure Shell (SSH) Protocol Parameters", <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-19>.

Author's Address

Denis Bider
Bitvise Limited
4105 Lombardy Court
Colleyville, Texas 76034
United States of America

Email: ietf-ssh3@denisbider.com
URI: https://www.bitvise.com/

Acknowledgments

Thanks to Jon Bright, Niels Moeller, Stephen Farrell, Mark D. Baushke, Jeffrey Hutzelman, Hanno Boeck, Peter Gutmann, Damien Miller, Mat Berchtold, Roumen Petrov, Daniel Migault, Eric Rescorla, Russ Housley, Alissa Cooper, Adam Roach, and Ben Campbell for reviews, comments, and suggestions.

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: July 5, 2018

A. Adamantiadis
libssh
S. Josefsson
SJD AB
M. Baushke
Juniper Networks, Inc.

January 1, 2018

Secure Shell (SSH) Key Exchange Method using Curve25519 and Curve448
draft-ietf-curdle-ssh-curves-07

Abstract

This document describes the conventions for using Curve25519 and Curve448 key exchange methods in the Secure Shell (SSH) protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Secure Shell (SSH) [RFC4251] is a secure remote login protocol. The key exchange protocol described in [RFC4253] supports an extensible set of methods. [RFC5656] describes how elliptic curves are integrated in SSH, and this document reuses those protocol messages.

This document describes how to implement key exchange based on Curve25519 and Ed448-Goldilocks [RFC7748] in SSH. For Curve25519 with SHA-256 [RFC6234], the algorithm we describe is equivalent to the privately defined algorithm "curve25519-sha256@libssh.org", which is currently implemented and widely deployed in libssh and OpenSSH. The Curve448 key exchange method is novel but similar in spirit, and we chose to couple it with SHA-512 [RFC6234] to further separate it from the Curve25519 alternative.

This document provide Curve25519 as the preferred choice, but suggests that the fall back option Curve448 is implemented to provide an hedge against unforeseen analytical advances against Curve25519 and SHA-256. Due to different implementation status of these two curves (high-quality free implementations of Curve25519 has been in deployed use for several years, while Curve448 implementations are slowly appearing), it is accepted that adoption of Curve448 will be slower.

2. 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.

3. Key Exchange Methods

The key exchange procedure is similar to the ECDH method described in chapter 4 of [RFC5656], though with a different wire encoding used for public values and the final shared secret. Public ephemeral keys are encoded for transmission as standard SSH strings.

The protocol flow, the SSH_MSG_KEX_ECDH_INIT and SSH_MSG_KEX_ECDH_REPLY messages, and the structure of the exchange hash are identical to chapter 4 of [RFC5656].

The method names registered by this document are "curve25519-sha256" and "curve448-sha512".

The methods are based on Curve25519 and Curve448 scalar multiplication, as described in [RFC7748]. Private and public keys are generated as described therein. Public keys are defined as strings of 32 bytes for Curve25519 and 56 bytes for Curve448. Clients and servers MUST fail the key exchange if the length of the received public keys are not the expected lengths, or if the derived shared secret only consists of zero bits. No further validation is required beyond what is discussed in [RFC7748]. The derived shared secret is 32 bytes when Curve25519 is used and 56 bytes when Curve448 is used. The encodings of all values are defined in [RFC7748]. The hash used is SHA-256 for Curve25519 and SHA-512 for Curve448.

3.1. Shared Secret Encoding

The following step differs from [RFC5656], which uses a different conversion. This is not intended to modify that text generally, but only to be applicable to the scope of the mechanism described in this document.

The shared secret, K, is defined in [RFC4253] and [RFC5656] as an integer encoded as a multiple precision integer (mpint). Curve25519/448 outputs a binary string X, which is the 32 or 56 byte point obtained by scalar multiplication of the other side’s public key and the local private key scalar. The 32 or 56 bytes of X are converted into K by interpreting the octets as an unsigned fixed-length integer encoded in network byte order.

The integer K is then encoded as an mpint using the process described in section 5 of [RFC4251] and the resulting bytes are fed as described in [RFC4253] to the key exchange method’s hash function to generate encryption keys.

When performing the X25519 or X448 operations, the integer values there will be encoded into byte strings by doing a fixed-length unsigned little-endian conversion, per [RFC7748]. It is only later when these byte strings are then passed to the ECDH code in SSH that the bytes are re-interpreted as a fixed-length unsigned big-endian integer value K, and then later that K value is encoded as a variable-length signed "mpint" before being fed to the hash algorithm used for key generation. The mpint K is then fed along with other data to the key exchange method's hash function to generate encryption keys.

4. Acknowledgements

The "curve25519-sha256" key exchange method is identical to the "curve25519-sha256@libssh.org" key exchange method created by Aris Adamantiadis and implemented in libssh and OpenSSH.

Thanks to the following people for review and comments: Denis Bider, Damien Miller, Niels Moeller, Matt Johnston, Eric Rescorla, Ron Frederick, Stefan Buehler.

5. Security Considerations

The security considerations of [RFC4251], [RFC5656], and [RFC7748] are inherited.

Curve25519 provide strong security and is efficient on a wide range of architectures, and has properties that allows better implementation properties compared to traditional elliptic curves. Curve448 with SHA-512 is similar, but has not received the same cryptographic review as Curve25519, and is slower, but it is provided as an hedge to combat unforeseen analytical advances against Curve25519 and SHA-256.

The way the derived binary secret string is encoded into a mpint before it is hashed (i.e., adding or removing zero-bytes for encoding) raises the potential for a side-channel attack which could determine the length of what is hashed. This would leak the most significant bit of the derived secret, and/or allow detection of when the most significant bytes are zero. For backwards compatibility reasons it was decided not to address this potential problem.

6. IANA Considerations

IANA is requested to add "curve25519-sha256" and "curve448-sha512" to the "Key Exchange Method Names" registry for SSH [IANA-KEX] that was created in RFC 4250 section 4.10 [RFC4250].

7. References

7.1. 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>.

[RFC4250] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, DOI 10.17487/RFC4250, January 2006, <https://www.rfc-editor.org/info/rfc4250>.

[RFC4251] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, DOI 10.17487/RFC4251, January 2006, <https://www.rfc-editor.org/info/rfc4251>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC5656] Stebila, D. and J. Green, "Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer", RFC 5656, DOI 10.17487/RFC5656, December 2009, <https://www.rfc-editor.org/info/rfc5656>.

[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>.

7.2. Informative References

[IANA-KEX] Internet Assigned Numbers Authority (IANA), "Secure Shell (SSH) Protocol Parameters: Key Exchange Method Names", January 2018, <http://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

Appendix A. Copying conditions

Authors' Addresses

Aris Adamantiadis
libssh

Email: aris@badcode.be

Simon Josefsson
SJD AB

Email: simon@josefsson.org

Mark D. Baushke
Juniper Networks, Inc.

Email: mdb@juniper.net

Internet-Draft   D. Bider
Updates: 4252, 4253, 4254 (if approved)   Bitvise Limited
Intended status: Standards Track   September 23, 2017
Expires: March 23, 2018

Extension Negotiation in Secure Shell (SSH)
draft-ietf-curdle-ssh-ext-info-15.txt

Abstract

This memo updates RFC 4252, RFC 4253, and RFC 4254 to define a mechanism for SSH clients and servers to exchange information about supported protocol extensions confidentially after SSH key exchange.

Status

Copyright

Table of Contents

1. Overview and Rationale

Secure Shell (SSH) is a common protocol for secure communication on the Internet. The original design of the SSH transport layer [RFC4253] lacks proper extension negotiation. Meanwhile, diverse implementations take steps to ensure that known message types contain no unrecognized information. This makes it difficult for implementations to signal capabilities and negotiate extensions without risking disconnection. This obstacle has been recognized in relationship with [SSH-RSA-SHA2], where the need arises for a client to discover public key algorithms a server accepts, to avoid authentication penalties and trial-and-error.

This memo updates RFC 4252, RFC 4253, and RFC 4254.

1.1. Requirements Terminology

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].

1.2. Wire Encoding Terminology

The wire encoding types in this document - "byte", "uint32", "string", "boolean", "name-list" - have meanings as described in [RFC4251].

2. Extension Negotiation Mechanism

2.1. Signaling of Extension Negotiation in SSH_MSG_KEXINIT

Applications implementing this mechanism MUST add one of the following indicator names to the field "kex_algorithms" in the SSH_MSG_KEXINIT message sent by the application in the first key exchange:

- When acting as server: "ext-info-s"
- When acting as client: "ext-info-c"

The indicator name is added without quotes, and MAY be added at any position in the name-list, subject to proper separation from other names as per name-list conventions.

The names are added to the "kex_algorithms" field because this is one of two name-list fields in SSH_MSG_KEXINIT that do not have a separate copy for each data direction.

The indicator names inserted by the client and server are different to ensure these names will not produce a match, and therefore not affect the algorithm chosen in key exchange algorithm negotiation.

The inclusion of textual indicator names is intended to provide a clue for implementers to discover this mechanism.

2.2. Enabling Criteria

If a client or server offers "ext-info-c" or "ext-info-s" respectively, it MUST be prepared to accept an SSH_MSG_EXT_INFO message from the peer.

A server only needs to send "ext-info-s" if it intends to process SSH_MSG_EXT_INFO from the client. A client only needs to send "ext-info-c" if it plans to process SSH_MSG_EXT_INFO from the server.

If a server receives an "ext-info-c", or a client receives an "ext-info-s", it MAY send an SSH_MSG_EXT_INFO message, but is not required to do so.

Neither party needs to wait for the other's SSH_MSG_KEXINIT in order to decide whether to send the appropriate indicator in its own SSH_MSG_KEXINIT.

Implementations MUST NOT send an incorrect indicator name for their role. Implementations MAY disconnect if the counter-party sends an incorrect indicator. If "ext-info-c" or "ext-info-s" ends up being negotiated as a key exchange method, the parties MUST disconnect.

2.3. SSH_MSG_EXT_INFO Message

A party that received the "ext-info-c" or "ext-info-s" indicator MAY send the following message:

```
byte       SSH_MSG_EXT_INFO (value 7)
uint32     nr-extensions
repeat the following 2 fields "nr-extensions" times:
  string   extension-name
  string   extension-value (binary)
```

Implementers' attention is called to Section 2.5., in particular the requirement to tolerate any sequence of bytes - including null bytes at any position - in an unknown extension's extension-value.

2.4. Message Order

If a client sends SSH_MSG_EXT_INFO, it MUST send it as the next packet following the client's first SSH_MSG_NEWKEYS message to the server.

If a server sends SSH_MSG_EXT_INFO, it MAY send it at zero, one, or both of the following opportunities:

- As the next packet following the server's first SSH_MSG_NEWKEYS.

  Where clients need information in the server's SSH_MSG_EXT_INFO to authenticate, it is helpful if the server sends its SSH_MSG_EXT_INFO not only as next packet after SSH_MSG_NEWKEYS, but without delay.

Clients cannot rely on this because the server is not required to send the message at this time; and if sent, it may be delayed by the network. However, if a timely SSH_MSG_EXT_INFO is received, a client can pipeline an authentication request after its SSH_MSG_SERVICE_REQUEST, even when it needs extension information.

- Immediately preceding the server’s SSH_MSG_USERAUTH_SUCCESS, as defined in [RFC4252].

  The server MAY send SSH_MSG_EXT_INFO at this second opportunity, whether or not it sent it at the first. A client that sent "ext-info-c" MUST accept a server’s SSH_MSG_EXT_INFO at both opportunities, but MUST NOT require it.

  This allows a server to reveal support for additional extensions that it was unwilling to reveal to an unauthenticated client. If a server sends a second SSH_MSG_EXT_INFO, this replaces any initial one, and both the client and the server re-evaluate extensions in effect. The server’s second SSH_MSG_EXT_INFO is matched against the client’s original.

  The timing of the second opportunity is chosen for the following reasons. If the message was sent earlier, it would not allow the server to withhold information until the client has authenticated. If it was sent later, a client that needs information from the second SSH_MSG_EXT_INFO immediately after it authenticates would have no way to reliably know whether to expect the message.

2.5. Interpretation of Extension Names and Values

Each extension is identified by its extension-name, and defines the conditions under which the extension is considered to be in effect. Applications MUST ignore unrecognized extension-names.

An extension MAY dictate, where it is specified, that in order to take effect, both parties must include it in their SSH_MSG_EXT_INFO; or it can be sufficient that only one party includes it; or other rules MAY be specified. The relative order in which extensions appear in an SSH_MSG_EXT_INFO message MUST be ignored.

Extension-value fields are interpreted as defined by their respective extension. This field MAY be empty if permitted by the extension. Applications that do not implement or recognize an extension MUST ignore its extension-value, regardless of its size or content. Applications MUST tolerate any sequence of bytes - including null bytes at any position - in an unknown extension’s extension-value.

The cumulative size of an SSH_MSG_EXT_INFO message is limited only by the maximum packet length that an implementation may apply in accordance with [RFC4253]. Implementations MUST accept well-formed SSH_MSG_EXT_INFO messages up to the maximum packet length they accept.

3. Initially Defined Extensions

3.1. "server-sig-algs"

This extension is sent with the following extension name and value:

```
string      "server-sig-algs"
name-list   public-key-algorithms-accepted
```

The name-list type is a strict subset of the string type, and is thus permissible as an extension-value. See [RFC4251] for more information.

This extension is sent by the server, and contains a list of public key algorithms that the server is able to process as part of a "publickey" authentication request. If a client sends this extension, the server MAY ignore it, and MAY disconnect.

In this extension, a server MUST enumerate all public key algorithms it might accept during user authentication. However, there exist early server implementations which do not enumerate all accepted algorithms. For this reason, a client MAY send a user authentication request using a public key algorithm not included in "server-sig-algs".

A client that wishes to proceed with public key authentication MAY wait for the server's SSH_MSG_EXT_INFO so it can send a "publickey" authentication request with an appropriate public key algorithm, rather than resorting to trial and error.

Servers that implement public key authentication SHOULD implement this extension.

If a server does not send this extension, a client MUST NOT make any assumptions about the server's public key algorithm support, and MAY proceed with authentication requests using trial and error. Note that implementations are known to exist that apply authentication penalties (*) if the client attempts to use an unexpected public key algorithm.

(*) Authentication penalties are applied by servers to deter brute force password guessing, username enumeration, and other types of behavior deemed suspicious by server administrators or implementers. Penalties may include automatic IP address throttling or blocking, and may trigger email alerts or auditing.

3.2. "delay-compression"

This extension MAY be sent by both parties as follows:

```
  string         "delay-compression"
  string:
    name-list    compression_algorithms_client_to_server
    name-list    compression_algorithms_server_to_client
```

The extension-value is a string that encodes two name-lists. The name-lists themselves have the encoding of strings. For example: to indicate a preference for algorithms "foo,bar" in the client-to-server direction, and "bar,baz" in the server-to-client direction, a sender encodes the extension-value as follows (including its length):

```
  00000016 00000007 666f6f2c626172 00000007 6261722c62617a
```

This same encoding could be sent by either party - client or server.

This extension allows the server and client to renegotiate compression algorithm support without having to conduct a key re-exchange, putting new algorithms into effect immediately upon successful authentication.

This extension takes effect only if both parties send it. Name-lists MAY include any compression algorithm that could have been negotiated in SSH_MSG_KEXINIT, except algorithms that define their own delayed compression semantics. This means "zlib,none" is a valid algorithm list in this context; but "zlib@openssh.com" is not.

If both parties send this extension, but the name-lists do not contain a common algorithm in either direction, the parties MUST disconnect in the same way as if negotiation failed as part of SSH_MSG_KEXINIT.

If this extension takes effect, the renegotiated compression algorithm is activated for the very next SSH message after the trigger message:

- Sent by the server, the trigger message is SSH_MSG_USERAUTH_SUCCESS.
- Sent by the client, the trigger message is SSH_MSG_NEWCOMPRESS.

If this extension takes effect, the client MUST send the following message within a reasonable number of outgoing SSH messages after receiving SSH_MSG_USERAUTH_SUCCESS - but not necessarily as the first such outgoing message:

```
  byte       SSH_MSG_NEWCOMPRESS (value 8)
```

The purpose of SSH_MSG_NEWCOMPRESS is to avoid a race condition where the server cannot reliably know whether a message sent by the client was sent before or after receiving the server’s SSH_MSG_USERAUTH_SUCCESS. For example, clients may send keep-alive messages during logon processing.

As is the case for all extensions unless otherwise noted, the server MAY delay including this extension until its secondary SSH_MSG_EXT_INFO, sent before SSH_MSG_USERAUTH_SUCCESS. This allows the server to avoid advertising compression until the client has authenticated.

If the parties re-negotiate compression using this extension in a session where compression is already enabled; and the re-negotiated algorithm is the same in one or both directions; then the internal compression state MUST be reset for each direction at the time the re-negotiated algorithm takes effect.

3.2.1. Awkwardly Timed Key Re-Exchange

A party that has signaled, or intends to signal, support for this extension in an SSH session, MUST NOT initiate key re-exchange in that session until either of the following occurs:

- This extension was negotiated, and the party that’s about to start key re-exchange already sent its trigger message for compression.

- The party has sent (if server) or received (if client) the message SSH_MSG_USERAUTH_SUCCESS, and this extension was not negotiated.

If a party violates this rule, the other party MAY disconnect.

In general, parties SHOULD NOT start key re-exchange before successful user authentication, but MAY tolerate it if not using this extension.

3.2.2. Subsequent Re-Exchange

In subsequent key re-exchanges that unambiguously begin after the compression trigger messages, the compression algorithms negotiated in re-exchange override the algorithms negotiated with this extension.

3.2.3. Compatibility Note: OpenSSH up to 7.5

This extension uses a binary extension-value encoding. OpenSSH clients up to and including version 7.5 advertise support to receive SSH_MSG_EXT_INFO, but disconnect on receipt of an extension-value containing null bytes. This is an error fixed in OpenSSH version 7.6.

Implementations that wish to interoperate with OpenSSH 7.5 and earlier are advised to check the remote party’s SSH version string, and omit this extension if an affected version is detected. Affected versions do not implement this extension, so there is no harm in omitting it. The extension SHOULD NOT be omitted if the detected OpenSSH version is 7.6 or higher. This would make it harder for the OpenSSH project to implement this extension in a higher version.

3.3. "no-flow-control"

This extension is sent with the following extension name and value:

```
  string      "no-flow-control"
  string      choice of: "p" for preferred | "s" for supported
```

A party SHOULD send "s" if it supports "no-flow-control", but does not prefer to enable it. A party SHOULD send "p" if it prefers to enable the extension if the other party supports it. Parties MAY disconnect if they receive a different extension value.

To take effect, this extension MUST be:

- Sent by both parties.
- At least one party MUST have sent the value "p" (preferred).

If this extension takes effect, the "initial window size" fields in SSH_MSG_CHANNEL_OPEN and SSH_MSG_CHANNEL_OPEN_CONFIRMATION, as defined in [RFC4254], become meaningless. The values of these fields MUST be ignored, and a channel behaves as if all window sizes are infinite. Neither side is required to send any SSH_MSG_CHANNEL_WINDOW_ADJUST messages, and if received, such messages MUST be ignored.

This extension is intended, but not limited to, use by file transfer applications that are only going to use one channel, and for which the flow control provided by SSH is an impediment, rather than a feature.

Implementations MUST refuse to open more than one simultaneous channel when this extension is in effect. Nevertheless, server implementations SHOULD support clients opening more than one non-simultaneous channel.

3.3.1. Prior "No Flow Control" Practice

Before this extension, some applications would simply not implement SSH flow control, sending an initial channel window size of 2^32 - 1. Applications SHOULD NOT do this for the following reasons:

- It is plausible to transfer more than 2^32 bytes over a channel. Such a channel will hang if the other party implements SSH flow control according to [RFC4254].

- There exist implementations which cannot handle large channel window sizes, and can exhibit non-graceful behaviors, including disconnect.

3.4. "elevation"

The terms "elevation" and "elevated" refer to an operating system mechanism where an administrator user’s logon session is associated with two security contexts: one limited, and one with administrative rights. To "elevate" such a session is to activate the security context with full administrative rights. For more information about this mechanism on Windows, see also [WINADMIN] and [WINTOKEN].

This extension MAY be sent by the client as follows:

```
  string      "elevation"
  string      choice of: "y" | "n" | "d"
```

A client sends "y" to indicate its preference that the session should be elevated; "n" to not be elevated; and "d" for the server to use its default behavior. The server MAY disconnect if it receives a different

extension value. If a client does not send the "elevation" extension, the server SHOULD act as if "d" was sent.

If a client has included this extension, then after authentication, a server that supports this extension SHOULD indicate to the client whether elevation was done by sending the following global request:

```
  byte        SSH_MSG_GLOBAL_REQUEST
  string      "elevation"
  boolean     want reply = false
  boolean     elevation performed
```

Clients that implement this extension help reduce attack surface for Windows servers that handle administrative logins. Where clients do not support this extension, servers must elevate sessions to allow full access by administrative users always. Where clients support this extension, sessions can be created without elevation unless requested.

4. IANA Considerations

4.1. Additions to existing tables

IANA is requested to insert the following entries into the table Message Numbers [IANA-M] under Secure Shell (SSH) Protocol Parameters [RFC4250]:

Value	Message ID	Reference
7	SSH_MSG_EXT_INFO	[this document]
8	SSH_MSG_NEWCOMPRESS	[this document]

IANA is requested to insert the following entries into the table Key Exchange Method Names [IANA-KE]:

Method Name	Reference	Note
ext-info-s	[this document]	Section 2.2
ext-info-c	[this document]	Section 2.2

4.2. New table: Extension Names

Also under Secure Shell (SSH) Protocol Parameters, IANA is requested to create a new table, Extension Names, with initial content:

Extension Name	Reference	Note
server-sig-algs	[this document]	Section 3.1
delay-compression	[this document]	Section 3.2
no-flow-control	[this document]	Section 3.3
elevation	[this document]	Section 3.4

4.2.1. Future Assignments to Extension Names

Names in the Extension Names table MUST follow the Conventions for Names defined in [RFC4250], Section 4.6.1.

Requests for assignments of new non-local names in the Extension Names table (i.e. names not including the '@' character) MUST be done through the IETF CONSENSUS method, as described in [RFC8126].

5. Security Considerations

Security considerations are discussed throughout this document. This document updates the SSH protocol as defined in [RFC4251] and related documents. The security considerations of [RFC4251] apply.

6. References

6.1. Normative References

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

[RFC4250] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, January 2006.

[RFC4251] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.

[RFC4252] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", RFC 4252, January 2006.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, January 2006.

[RFC4254] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

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

6.2. Informative References

[SSH-RSA-SHA2]
Bider, D., "Use of RSA Keys with SHA-2 256 and 512 in Secure Shell (SSH)", draft-ietf-curdle-rsa-sha2-10.txt, August 2017, <https://tools.ietf.org/html/draft-ietf-curdle-rsa-sha2-10>.

[IANA-M] "Secure Shell (SSH) Protocol Parameters", <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-1>.

[IANA-KE] "Secure Shell (SSH) Protocol Parameters", <https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>.

[WINADMIN] "How to launch a process as a Full Administrator when UAC is enabled?", <https://blogs.msdn.microsoft.com/winsdk/2013/03/22/how-to-launch-a-process-as-a-full-administrator-when-uac-is-enabled/>.

[WINTOKEN] "TOKEN_ELEVATION_TYPE enumeration", <https://msdn.microsoft.com/en-us/library/windows/desktop/bb530718.aspx>.

Author's Address

Denis Bider
Bitvise Limited
4105 Lombardy Court
Colleyville, Texas 76034
United States of America

EMail: ietf-ssh3@denisbider.com
URI: https://www.bitvise.com/

Acknowledgments

Thanks to Markus Friedl and Damien Miller for comments and initial implementation. Thanks to Peter Gutmann, Roumen Petrov, Mark D. Baushke, Daniel Migault, Eric Rescorla, Matthew A. Miller, Mirja Kuehlewind, Adam Roach, Spencer Dawkins, Alexey Melnikov, and Ben Campbell for reviews and feedback.

Internet Engineering Task Force
Internet-Draft
Updates: 4250 (if approved)
Intended status: Standards Track
Expires: July 6, 2018

M. Baushke
Juniper Networks, Inc.
January 2, 2018

Key Exchange (KEX) Method Updates and Recommendations for Secure Shell (SSH)
draft-ietf-curdle-ssh-kex-sha2-10

Abstract

This document is intended to update the recommended set of key exchange methods for use in the Secure Shell (SSH) protocol to meet evolving needs for stronger security. This document updates RFC 4250.

Status of This Memo

Copyright Notice

Table of Contents

1. Overview and Rationale

Secure Shell (SSH) is a common protocol for secure communication on the Internet. In [RFC4253], SSH originally defined two Key Exchange Method Names that MUST be implemented. Over time, what was once considered secure, is no longer considered secure. The purpose of this RFC is to recommend that some published key exchanges be deprecated as well as recommending some that SHOULD and one that MUST be adopted. This document updates [RFC4250].

This document adds recommendations for adoption of Key Exchange Methods which MUST, SHOULD, MAY, SHOULD NOT, and MUST NOT be implemented. New key exchange methods will use the SHA-2 family of hashes found in [RFC6234] and are drawn from these ssh-curves from [I-D.ietf-curdle-ssh-curves] and DH MODP primes from the [RFC8268] and gss-keyex [I-D.ietf-curdle-gss-keyex-sha2].

2. Requirements Language

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 RFC 2119 [RFC2119].

3. Key Exchange Methods

This memo adopts the style and conventions of [RFC4253] in specifying how the use of data key exchange is indicated in SSH.

This RFC also collects Key Exchange Method Names in various existing RFCs [RFC4253], [RFC4419], [RFC4432], [RFC4462], [RFC5656], [RFC8268], [I-D.ietf-curdle-gss-keyex-sha2], and [I-D.ietf-curdle-ssh-curves] and provides a suggested suitability for implementation of MUST, SHOULD, SHOULD NOT, and MUST NOT. Any method not explicitly listed, MAY be implemented.

This document is intended to provide guidance as to what Key Exchange Algorithms are to be considered for new or updated SSH implementations. This document will be superseded when one or more of the listed algorithms are considered too weak to continue to use securely, in which case they will likely be downgraded to SHOULD NOT or MUST NOT. Or, when newer methods have been analyzed and found to be secure with wide enough adoption to upgrade their recommendation from MAY to SHOULD or MUST.

3.1. curve25519-sha256

The Curve25519 provides strong security and is efficient on a wide range of architectures with properties that allow better implementation properties compared to traditional elliptic curves. The use of SHA2-256 (also known as SHA-256) as defined in [RFC6234] for integrity is a reasonable one for this method. This Key Exchange Method is described in [I-D.ietf-curdle-ssh-curves] and is similar to the IKEv2 Key Agreement described in [RFC8031]. This Key Exchange Method has multiple implementations and SHOULD be implemented in any SSH interested in using elliptic curve based key exchanges.

3.2. curve448-sha512

The Curve448 provides very strong security. It uses SHA2-512 (also known as SHA-256) defined in [RFC6234] for integrity. It is probably stronger and more work than is currently needed. This Key Exchange Method is described in [I-D.ietf-curdle-ssh-curves] and is similar to the IKEv2 Key Agreement described in [RFC8031]. This method MAY be implemented.

3.3. diffie-hellman-group-exchange-sha1

This set of ephemerally generated key exchange groups uses SHA-1 as defined in [RFC4419]. However, SHA-1 has security concerns provided in [RFC6194], so it would be better to use a key exchange method which uses a SHA-2 hash as in [RFC6234] for integrity. This key exchange SHOULD NOT be used.

3.4. diffie-hellman-group-exchange-sha256

This set of ephemerally generated key exchange groups uses SHA2-256 as defined in [RFC4419]. [RFC8270] mandates implementations avoid any MODP group with less than 2048 bits. This key exchange MAY be used.

3.5. diffie-hellman-group1-sha1

This method is decribed in [RFC4253] and uses [RFC7296] Oakley Group 2 (a 1024-bit MODP group) and SHA-1 [RFC3174]. Due to recent security concerns with SHA-1 [RFC6194] and with MODP groups with less than 2048 bits (see [LOGJAM] and [NIST-SP-800-131Ar1]), this method is considered insecure. This method is being moved from MUST to SHOULD NOT instead of MUST NOT only to allow a transition time to get off of it. There are many old implementations out there that may still need to use this key exchange, it should be removed from server implementations as quickly as possible.

3.6. diffie-hellman-group14-sha1

This method uses [RFC3526] group14 (a 2048-bit MODP group) which is still a reasonable size. This key exchange group uses SHA-1 which has security concerns [RFC6194]. However, this group is still strong enough and is widely deployed. This method is being moved from MUST to SHOULD to aid in transition to stronger SHA-2 based hashes. This method will transition to SHOULD NOT when SHA-2 alternatives are more generally available.

3.7. diffie-hellman-group14-sha256

This key exchange method is defined in [RFC8268] and uses the group14 (a 2048-bit MODP group) along with a SHA-2 (SHA2-256) hash as in [RFC6234] for integrity. This represents the smallest Finite Field Cryptography (FFC) Diffie-Hellman (DH) key exchange method considered to be secure. It is a reasonably simple transition to move from SHA-1 to SHA-2. This method MUST be implemented.

3.8. diffie-hellman-group15-sha512

This key exchange method is defined in [RFC8268] and uses group15 along with a SHA-2 (SHA2-512) hash as in [RFC6234] for integrity. Note: The use of this 3072-bit MODP group would be equally justified to use SHA2-384 as the hash rather than SHA2-512. However, some small implementations would rather only worry about two rather than three new hashing functions. This group does not really provide much additional head room over the 2048-bit group14 FFC DH and the predominate open source implementations are not adopting it. This method MAY be implemented.

3.9. diffie-hellman-group16-sha512

This key exchange method is defined in [RFC8268] and uses group16 along with a SHA-2 (SHA2-512) hash as in [RFC6234] for integrity. The use of FFC DH is well understood and trusted. Adding larger modulus sizes and protecting with SHA2-512 should give enough head room to be ready for the next scare that someone has pre-computed it. This modulus (4096-bit) is larger than that required by [CNSA-SUITE] and should be sufficient to inter-operate with more paranoid nation-states. This method SHOULD be implemented.

3.10. diffie-hellman-group17-sha512

This key exchange method is defined in [RFC8268] and uses group17 along with a SHA-2 (SHA2-512) hash as in [RFC6234] for integrity. The use of this 6144-bit MODP group is going to be slower than what may be desirable. It is provided to help those who wish to avoid using ECC algorithms. This method MAY be implemented.

3.11. diffie-hellman-group18-sha512

This key exchange method is defined in [RFC8268] and uses group18 along with a SHA-2 (SHA2-512) hash as in [RFC6234] for integrity. The use of this 8192-bit MODP group is going to be slower than what may be desirable. It is provided to help those who wish to avoid using ECC algorithms. This method MAY be implemented.

3.12. ecdh-sha2-nistp256

This key exchange method is defined in [RFC5656]. Elliptic Curve Diffie-Hellman (ECDH) are often implemented because they are smaller and faster than using large FFC primes with traditional Diffie-Hellman (DH). However, given [CNSA-SUITE] and [safe-curves], this curve may not be as useful and strong as desired for handling TOP SECRET information for some applications. The SSH development community is divided on this and many implementations do exist. If traditional ECDH key exchange methods are implemented, then this method SHOULD be implemented.

It is advisable to match the ECDSA and ECDH algorithms to use the same curve for both.

3.13. ecdh-sha2-nistp384

This key exchange method is defined in [RFC5656]. This ECDH method should be implemented because it is smaller and faster than using large FFC primes with traditional Diffie-Hellman (DH). Given [CNSA-SUITE], it is considered good enough for TOP SECRET. If traditional ECDH key exchange methods are implemented, then this method SHOULD be implemented.

Research into ways of breaking ECDSA continues. Papers such as [ECDSA-Nonce-Leak] as well as concerns raised in [safe-curves] may mean that this algorithm will need to be downgraded in the future along the other ECDSA nistp curves.

3.14. ecdh-sha2-nistp521

This key exchange method is defined in [RFC5656]. This ECDH method may be implemented because it is smaller and faster than using large FFC primes with traditional Diffie-Hellman (DH). It is not listed in [CNSA-SUITE], so it is not currently appropriate for TOP SECRET. It is possible that the mismatch between the 521-bit key and the 512-bit hash could mean that as many as nine bits of this key could be at risk of leaking if appropriate padding measures are not taken. This method MAY be implemented, but is not recommended.

3.15. gss-gex-sha1-*

This key exchange method is defined in [RFC4462]. This set of ephemerally generated key exchange groups uses SHA-1 which has security concerns [RFC6194]. It is recommended that these key exchange groups NOT be used. This key exchange SHOULD NOT be used. It is intended that it move to MUST NOT as soon as the majority of server implementations no longer offer it. It should be removed from server implementations as quickly as possible.

3.16. gss-group1-sha1-*

This key exchange method is defined in [RFC4462]. This method suffers from the same problems of diffie-hellman-group1-sha1. It uses [RFC7296] Oakley Group 2 (a 1024-bit MODP group) and SHA-1 [RFC3174]. Due to recent security concerns with SHA-1 [RFC6194] and with MODP groups with less than 2048 bits (see [LOGJAM] and [NIST-SP-800-131Ar1]), this method is considered insecure. This method SHOULD NOT be implemented. It is intended that it move to MUST NOT as soon as the majority of server implementations no longer offer it. It should be removed from server implementations as quickly as possible.

3.17. gss-group14-sha1-*

This key exchange method is defined in [RFC4462]. This generated key exchange groups uses SHA-1 which has security concerns [RFC6194]. If GSS-API key exchange methods are being used, then this one SHOULD be implemented until such time as SHA-2 variants may be implemented and deployed. This method will transition to SHOULD NOT when SHA-2 alternatives are more generally available. No other standard indicated that this method was anything other than optional even though it was implemented in all GSS-API systems. This method MAY be implemented.

3.18. gss-group14-sha256-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. This key exchange uses the group14 (a 2048-bit MODP group) along with a SHA-2 (SHA2-256) hash. This represents the smallest Finite Field Cryptography (FFC) Diffie-Hellman (DH) key exchange method considered to be secure. It is a reasonably simple transition to move from SHA-1 to SHA-2. If the GSS-API is to be used, then this method SHOULD be implemented.

3.19. gss-group15-sha512-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. The use of this 3072-bit MODP group does not really provide much additional head room over the 2048-bit group14 FFC DH. If the GSS-API is to be used, then this method MAY be implemented.

3.20. gss-group16-sha512-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. The use of FFC DH is well understood and trusted. Adding larger modulus sizes and protecting with SHA2-512 should give enough head room to be ready for the next scare that someone has pre-computed. This modulus (4096-bit) is larger than that required by [CNSA-SUITE] and should be sufficient to inter-operate with more paranoid nation-states. If the GSS-API is to be used, then this method SHOULD be implemented.

3.21. gss-group17-sha512-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. The use of this 6144-bit MODP group is going to be slower than what may be desirable. It is provided to help those who wish to avoid using ECC algorithms. If the GSS-API is to be used, then this method MAY be implemented.

3.22. gss-group18-sha512-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. The use of this 8192-bit MODP group is going to be slower than what may be desirable. It is provided to help those who prefer to avoid using ECC algorithms. If the GSS-API is to be used, then this method MAY be implemented.

3.23. gss-nistp256-sha256-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. If the GSS-API is to be used with ECC algorithms, then this method SHOULD be implemented.

3.24. gss-nistp384-sha384-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. If the GSS-API is to be used with ECC algorithms, then this method SHOULD be implemented to permit TOP SECRET information to be communicated.

3.25. gss-nistp521-sha512-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. If the GSS-API is to be used with ECC algorithms, then this method MAY be implemented.

3.26. gss-curve25519-sha256-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. If the GSS-API is to be used with ECC algorithms, then this method SHOULD be implemented.

3.27. gss-curve448-sha512-*

This key exchange method is defined in [I-D.ietf-curdle-gss-keyex-sha2]. If the GSS-API is to be used with ECC algorithms, then this method MAY be implemented.

3.28. rsa1024-sha1

This key exchange method is defined in [RFC4432]. The security of RSA 1024-bit modulus keys is not good enough any longer. A key size should be 2048-bits. This generated key exchange groups uses SHA-1 which has security concerns [RFC6194]. This method MUST NOT be implemented.

3.29. rsa2048-sha256

This key exchange method is defined in [RFC4432]. An RSA 2048-bit modulus key with a SHA2-256 hash. At the present time, a 2048-bit RSA key is considered to be suffiently strong in [NIST-SP-800-131Ar1] to be permitted. In addition, the use of a SHA-2 hash as defined in [RFC6234] is a good integrity measure. This method MAY be implemented.

4. Selecting an appropriate hashing algorithm

As may be seen from the above, the Key Exchange Methods area all using either SHA256 or SHA512 with the exception of the ecdh-sha2-nistp384 which uses SHA384.

The cited CNSA Suite specifies the use of SHA384 and says that SHA256 is no longer good enough for TOP SECRET. Nothing is said about the use of SHA512. It may be that the internal state of 1024 bits in both SHA384 and SHA512 makes the SHA384 more secure because it does not leak an additional 128 bits of state. Of course, the use of SHA384 also reduces the security strength to 384 bits instead of being 512 bits. This seems to contradict the desire to double the symmetric key strength in order to try to be safe from Post Quantum Computing (PQC) attacks given a session key derived from the key exchange will be limited to the security strength of the hash being used.

The move away from SHA256 to SHA512 for the newer key exchange methods is more to try to slow Grover’s algorithm (a PQC attack) slightly. It is also the case that SHA2-512 may, in many modern CPUs, be implemented more efficiently using 64-bit arithmetic than SHA256 which is faster on 32-bit CPUs. The selection of SHA384 vs SHA512 is more about reducing the number of code point alternatives to negotiate. There seemed to be consensus in favor of SHA2-512 over SHA2-384 for key exchanges.

5. Summary Guidance for Key Exchange Method Names

The Implement column is the current recommendations of this RFC. Key Exchange Method Names are listed alphabetically.

Key Exchange Method Name	Reference	Implement
curve25519-sha256	ssh-curves	SHOULD
diffie-hellman-group-exchange-sha1	RFC4419	SHOULD NOT
diffie-hellman-group1-sha1	RFC4253	SHOULD NOT
diffie-hellman-group14-sha1	RFC4253	SHOULD
diffie-hellman-group14-sha256	RFC8268	MUST
diffie-hellman-group16-sha512	RFC8268	SHOULD
ecdh-sha2-nistp256	RFC5656	SHOULD
ecdh-sha2-nistp384	RFC5656	SHOULD
gss-gex-sha1-*	RFC4462	SHOULD NOT
gss-group1-sha1-*	RFC4462	SHOULD NOT
gss-group14-sha256-*	gss-keyex	SHOULD
gss-group16-sha512-*	gss-keyex	SHOULD
gss-nistp256-sha256-*	gss-keyex	SHOULD
gss-nistp384-sha384-*	gss-keyex	SHOULD
gss-curve25519-sha256-*	gss-keyex	SHOULD
rsa1024-sha1	RFC4432	MUST NOT

The full set of official [IANA-KEX] key algorithm method names not otherwise mentioned in this document MAY be implemented.

The guidance of this document is that the SHA-1 algorithm hashing SHOULD NOT be used. If it is used in implementations, it should only be provided for backwards compatibility, should not be used in new designs, and should be phased out of existing key exchanges as quickly as possible because of its known weaknesses. Any key exchange using SHA-1 should not be in a default key exchange list if at all possible. If they are needed for backward compatibility, they SHOULD be listed after all of the SHA-2 based key exchanges.

The [RFC4253] MUST diffie-hellman-group14-sha1 method SHOULD be retained for compatibility with older Secure Shell implementations. It is intended that this key exchange method be phased out as soon as possible. It SHOULD be listed after all possible SHA-2 based key exchanges.

It is believed that all current SSH implementations should be able to achieve an implementation of the "diffie-hellman-group14-sha256" method. To that end, this is one method that MUST be implemented.

[TO BE REMOVED: This registration should take place at the following location: <http://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>]

6. Acknowledgements

Thanks to the following people for review and comments: Denis Bider, Peter Gutmann, Damien Miller, Niels Moeller, Matt Johnston, Iwamoto Kouichi, Simon Josefsson, Dave Dugal, Daniel Migault, Anna Johnston, and Tero Kivinen.

Thanks to the following people for code to implement inter-operable exchanges using some of these groups as found in an this draft: Darren Tucker for OpenSSH and Matt Johnston for Dropbear. And thanks to Iwamoto Kouichi for information about RLogin, Tera Term (ttssh) and Poderosa implementations also adopting new Diffie-Hellman groups based on this draft.

7. Security Considerations

This SSH protocol provides a secure encrypted channel over an insecure network. It performs server host authentication, key exchange, encryption, and integrity protection. It also derives a unique session ID that may be used by higher-level protocols.

Full security considerations for this protocol are provided in [RFC4251]

It is desirable to deprecate or remove key exchange method name that are considered weak. A key exchange method may be weak because too few bits are used, or the hashing algorithm is considered too weak.

The diffie-hellman-group1-sha1 is being moved from MUST to MUST NOT. This method used [RFC7296] Oakley Group 2 (a 1024-bit MODP group) and SHA-1 [RFC3174]. Due to recent security concerns with SHA-1

[RFC6194] and with MODP groups with less than 2048 bits [NIST-SP-800-131Ar1], this method is no longer considered secure.

The United States Information Assurance Directorate (IAD) at the National Security Agency (NSA) has published a FAQ [MFQ-U-OO-815099-15] suggesting that the use of Elliptic Curve Diffie-Hellman (ECDH) using the nistp256 curve and SHA-2 based hashes less than SHA2-384 are no longer sufficient for transport of TOP SECRET information. If your systems need to be concerned with TOP SECRET information, then the guidance for supporting lesser security strength key exchanges may be omitted for your implementations.

The MODP group14 is already required for SSH implementations and most implementations already have a SHA2-256 implementation, so diffie-hellman-group14-sha256 is provided as an easy to implement and faster to use key exchange. Small embedded applications may find this KEX desirable to use.

The NSA Information Assurance Directorate (IAD) has also published the Commercial National Security Algorithm Suite (CNSA Suite) [CNSA-SUITE] in which the 3072-bit MODP Group 15 in [RFC3526] is explicitly mentioned as the minimum modulus to protect TOP SECRET communications.

It has been observed in [safe-curves] that the NIST Elliptic Curve Prime Curves (P-256, P-384, and P-521) are perhaps not the best available for Elliptic Curve Cryptography (ECC) Security. For this reason, none of the [RFC5656] curves are mandatory to implement. However, the requirement that "every compliant SSH ECC implementation MUST implement ECDH key exchange" is now taken to mean that if ecdsa-sha2-[identifier] is implemented, then ecdh-sha2-[identifier] MUST be implemented.

In a Post-Quantum Computing (PQC) world, it will be desirable to use larger cyclic subgroups. To do this using Elliptic Curve Cryptography will require much larger prime base fields, greatly reducing their efficiency. Finite Field based Cryptography already requires large enough base fields to accommodate larger cyclic subgroups. Until such time as a PQC method of key exchange is developed and adopted, it may be desirable to generate new and larger DH groups to avoid pre-calculation attacks that are provably not backdoored.

8. IANA Considerations

IANA is requested to annotate entries in [IANA-KEX] which MUST NOT be implemented as being deprecated by this document.

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>.

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, DOI 10.17487/RFC3526, May 2003, <https://www.rfc-editor.org/info/rfc3526>.

[RFC4250] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, DOI 10.17487/RFC4250, January 2006, <https://www.rfc-editor.org/info/rfc4250>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC8031] Nir, Y. and S. Josefsson, "Curve25519 and Curve448 for the Internet Key Exchange Protocol Version 2 (IKEv2) Key Agreement", RFC 8031, DOI 10.17487/RFC8031, December 2016, <https://www.rfc-editor.org/info/rfc8031>.

[RFC8268] Baushke, M., "More Modular Exponentiation (MODP) Diffie-Hellman (DH) Key Exchange (KEX) Groups for Secure Shell (SSH)", RFC 8268, DOI 10.17487/RFC8268, December 2017, <https://www.rfc-editor.org/info/rfc8268>.

[RFC8270] Velvindron, L. and M. Baushke, "Increase the Secure Shell Minimum Recommended Diffie-Hellman Modulus Size to 2048 Bits", RFC 8270, DOI 10.17487/RFC8270, December 2017, <https://www.rfc-editor.org/info/rfc8270>.

9.2. Informative References

[CNSA-SUITE] "Information Assurance by the National Security Agency", "Commercial National Security Algorithm Suite", September 2016, <https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm>.

[ECDSA-Nonce-Leak]
De Mulder, Hutter, Marson, and Pearson, "Using Bleichenbacher’s Solution to the Hidden Number Problem to Attack Nonce Leaks in 384-Bit ECDSA", IACR Cryptology ePrint Archive 2013, August 2013, <https://eprint.iacr.org/2013/346.pdf>.

[I-D.ietf-curdle-gss-keyex-sha2]
Sorce, S. and H. Kario, "GSS-API Key Exchange with SHA2", draft-ietf-curdle-gss-keyex-sha2-03 (work in progress), December 2017.

[I-D.ietf-curdle-ssh-curves]
Adamantiadis, A., Josefsson, S., and M. Baushke, "Secure Shell (SSH) Key Exchange Method using Curve25519 and Curve448", draft-ietf-curdle-ssh-curves-06 (work in progress), November 2017.

[IANA-KEX]
Internet Assigned Numbers Authority (IANA), "Secure Shell (SSH) Protocol Parameters: Key Exchange Method Names", January 2018, <http://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>.

[LOGJAM] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J., Heninger, N., Springall, D., Thome, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Beguelin, S., and P. Zimmermann, "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice", ACM Conference on Computer and Communications Security (CCS) 2015, 2015, <https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf>.

[MFQ-U-OO-815099-15]
"National Security Agency/Central Security Service", "CNSA Suite and Quantum Computing FAQ", January 2016, <https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm>.

[NIST-SP-800-131Ar1]
Barker and Roginsky, "Transitions: Recommendation for the Transitioning of the Use of Cryptographic Algorithms and Key Lengths", NIST Special Publication 800-131A Revision 1, November 2015, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf>.

[RFC3174] Eastlake 3rd, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, DOI 10.17487/RFC3174, September 2001, <https://www.rfc-editor.org/info/rfc3174>.

[RFC4251] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", RFC 4251, DOI 10.17487/RFC4251, January 2006, <https://www.rfc-editor.org/info/rfc4251>.

[RFC4419] Friedl, M., Provos, N., and W. Simpson, "Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol", RFC 4419, DOI 10.17487/RFC4419, March 2006, <https://www.rfc-editor.org/info/rfc4419>.

[RFC4432] Harris, B., "RSA Key Exchange for the Secure Shell (SSH) Transport Layer Protocol", RFC 4432, DOI 10.17487/RFC4432, March 2006, <https://www.rfc-editor.org/info/rfc4432>.

[RFC4462] Hutzelman, J., Salowey, J., Galbraith, J., and V. Welch, "Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol", RFC 4462, DOI 10.17487/RFC4462, May 2006, <https://www.rfc-editor.org/info/rfc4462>.

[RFC5656] Stebila, D. and J. Green, "Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer", RFC 5656, DOI 10.17487/RFC5656, December 2009, <https://www.rfc-editor.org/info/rfc5656>.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <https://www.rfc-editor.org/info/rfc6194>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <https://www.rfc-editor.org/info/rfc7296>.

[safe-curves]
Bernstein and Lange, "SafeCurves: choosing safe curves for elliptic-curve cryptography.", February 2016, <https://safecurves.cr.yp.to/>.

Author's Address

Mark D.   Baushke
Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, CA 94089-1228
US

Email: mdb@juniper.net
URI: http://www.juniper.net/

```
Internet Engineering Task Force                               M. Baushke
Internet-Draft                                    Juniper Networks, Inc.
Updates: 4250, 4253 (if approved)                     September 15, 2017
Intended status: Standards Track
Expires: March 19, 2018
```

More Modular Exponential (MODP) Diffie-Hellman (DH) Key Exchange (KEX) Groups for Secure Shell (SSH)

draft-ietf-curdle-ssh-modp-dh-sha2-09

Abstract

This document defines added Modular Exponential (MODP) Groups for the Secure Shell (SSH) protocol using SHA-2 hashes. This document updates RFC 4250. This document updates RFC 4253 including an errata fix for checking the Peer’s DH Public Key.

Status of This Memo

Copyright Notice

1. Overview and Rationale

Secure Shell (SSH) is a common protocol for secure communication on the Internet. Security protocols and primitives are an active area for research and help to suggest updates to SSH.

Section 3 of the [RFC4253] contains a small errata for checking the Peer’s DH Public key. Section 4 of this document provides the correction.

Due to security concerns with SHA-1 [RFC6194] and with MODP groups with less than 2048 bits [NIST-SP-800-131Ar1] implementer and users request support for larger Diffie Hellman (DH) MODP group sizes with data integrity verification using the SHA-2 family of secure hash algorithms as well as MODP groups providing more security. The use of larger MODP groups and the move to the SHA-2 family of hashes are important features to strengthen the key exchange algorithms available to the SSH client and server.

DH primes being adopted by this document are all "safe primes" such that p = 2q + 1 where q is also a prime. New MODP groups are being introduced starting with the MODP 3072-bit group 15. All use SHA512 as the hash algorithm.

The DH 2048-bit MODP group 14 is already present in most SSH implementations and most implementations already have a SHA256 implementation, so diffie-hellman-group14-sha256 is provided as easy to implement.

It is intended that these new MODP groups with SHA-2 based hashes update the [RFC4253] section 6.4 and [RFC4250] section 4.10 standards.

The United States Information Assurance Directorate (IAD) at the National Security Agency (NSA) has published "Commercial National Security Algorithm (CNSA) Suite and Quantum Computing Frequently Asked Questions (FAQ)" [MFQ-U-OO-815099-15] addressed to organizations that run classified or unclassified national security systems (NSS) and vendors that build products used in NSS.

This FAQ document indicates that NSS should no longer use:

o ECDH and ECDSA with NIST P-256

o SHA-256

- o AES-128
- o RSA with 2048-bit keys
- o Diffie-Hellman with 2048-bit keys

The FAQ also states that NSS users should select DH groups based upon well established and validated parameter sets that comply with the minimum required sizes. Some specific examples include:

- o Elliptic Curves are currently restricted to the NIST P-384 group only for both ECDH and ECDSA, in accordance with existing NIST and NIAP standards.
- o RSA moduli should have a minimum size of 3072 bits (other than the noted PKI exception), and keys should be generated in accordance with all relevant NIST standards.
- o For Diffie-Hellman use a Diffie-Hellman prime modulus of at least 3072 bits as specified in IETF RFC 3526 [RFC3526] (Groups 15-18).

Although SSH may not always be used to protect Top Secret communications, this document adopts the use of the DH groups provided as an example in the FAQ as well as the use of SHA512 rather than SHA256 for the new DH groups.

[TO BE REMOVED: Please send comments on this draft to curdle@ietf.org.]

2. Requirements Language

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 RFC 2119 [RFC2119].

3. Key Exchange Algorithms

This document adds some new Key Exchange Algorithm Method Names in [RFC4253] and [RFC4250].

This document adopts the style and conventions of [RFC4253] in specifying how the use of new data key exchange is indicated in SSH.

The following new key exchange method algorithms are defined:

- o diffie-hellman-group14-sha256
- o diffie-hellman-group15-sha512

- diffie-hellman-group16-sha512
- diffie-hellman-group17-sha512
- diffie-hellman-group18-sha512

The SHA-2 family of secure hash algorithms are defined in [RFC6234].

The method of key exchange used for the name "diffie-hellman-group14-sha256" is the same as that for "diffie-hellman-group14-sha1" except that the SHA256 hash algorithm is used. It is recommended that diffie-hellman-group14-sha256 SHOULD be supported to smooth the transition to newer group sizes.

The group15 through group18 names are the same as those specified in [RFC3526] 3072-bit MODP Group 15, 4096-bit MODP Group 16, 6144-bit MODP Group 17, and 8192-bit MODP Group 18.

The SHA512 algorithm is to be used when "sha512" is specified as a part of the key exchange method name.

4. Checking the Peer’s DH Public Key

Section 3 of [RFC4253] contains a small errata. When checking e (client public key) and f (server public key) values, an incorrect range is provided. The erroneous text is:

> Values of ’e’ or ’f’ that are not in the range [1, p-1] MUST NOT be sent or accepted by either side. If this condition is violated, the key exchange fails.

The errata is that the range should have been an open interval excluding the end point values. (i.e "(1, p-1)"). This document amends that document text as follows:

> DH Public key values MUST be checked and both conditions:
>
> $1 < e < p-1$
>
> $1 < f < p-1$
>
> MUST be true. Values not within these bounds MUST NOT be sent or accepted by either side. If either one of these condition is violated, then the key exchange fails.

This simple check ensures:

- The remote peer behaves properly.

o The local system is not forced into the two-element subgroup.

5. IANA Considerations

IANA is requested to add to the Key Exchange Method Names algorithm registry [IANA-KEX] with the following entries:

Key Exchange Method Name	Reference
diffie-hellman-group14-sha256	This Draft
diffie-hellman-group15-sha512	This Draft
diffie-hellman-group16-sha512	This Draft
diffie-hellman-group17-sha512	This Draft
diffie-hellman-group18-sha512	This Draft

[TO BE REMOVED: This registration should take place at the following location: <http://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>]

6. Acknowledgements

Thanks to the following people for review and comments: Denis Bider, Peter Gutmann, Damien Miller, Niels Moeller, Matt Johnston, Iwamoto Kouichi, Dave Dugal, Daniel Migault, Anna Johnston, Ron Frederick, Rich Salz, Travis Finkenauer, Eric Rescorla.

7. Security Considerations

The security considerations of [RFC4253] apply to this document.

The security considerations of [RFC3526] suggest that MODP group14 through group18 have security strengths that range between 110 bits of security through 310 bits of security. They are based on [RFC3766] Determining Strengths For Public Keys Used For Exchanging Symmetric Keys. Care should be taken to use sufficient entropy and/or DRBG algorithms to maximize the true security strength of the key exchange and ciphers selected.

Using a fixed set of Diffie-Hellman parameters makes them a high value target for pre-computation. Generating additional sets of primes to be used, or moving to larger values is a mitigation against this issue.

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>.

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, DOI 10.17487/RFC3526, May 2003, <https://www.rfc-editor.org/info/rfc3526>.

[RFC4250] Lehtinen, S. and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Assigned Numbers", RFC 4250, DOI 10.17487/RFC4250, January 2006, <https://www.rfc-editor.org/info/rfc4250>.

[RFC4253] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <https://www.rfc-editor.org/info/rfc4253>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

8.2. Informative References

[IANA-KEX]
Internet Assigned Numbers Authority (IANA), "Secure Shell (SSH) Protocol Parameters: Key Exchange Method Names", March 2017, <http://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml#ssh-parameters-16>.

[MFQ-U-OO-815099-15]
"National Security Agency/Central Security Service", "CNSA Suite and Quantum Computing FAQ", January 2016, <https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm>.

[NIST-SP-800-131Ar1]
Barker and Roginsky, "Transitions: Recommendation for the Transitioning of the Use of Cryptographic Algorithms and Key Lengths", NIST Special Publication 800-131A Revision 1, November 2015, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf>.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, DOI 10.17487/RFC3766, April 2004, <https://www.rfc-editor.org/info/rfc3766>.

[RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <https://www.rfc-editor.org/info/rfc6194>.

Author’s Address

Mark D. Baushke
Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, CA 94089-1228
US

Phone: +1 408 745 2952
Email: mdb@juniper.net
URI: http://www.juniper.net/