Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 12, 2019

R. Barnes
Cisco
K. Bhargavan
Inria

March 11, 2019

Hybrid Public Key Encryption
draft-barnes-cfrg-hpke-01

Abstract

This document describes a scheme for hybrid public-key encryption (HPKE). This scheme provides authenticated public key encryption of arbitrary-sized plaintexts for a recipient public key. HPKE works for any Diffie-Hellman group and has a strong security proof. We provide instantiations of the scheme using standard and efficient primitives.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2
2. Requirements Notation 3
3. Security Properties 3
4. Notation 3
5. Cryptographic Dependencies 3
5.1. DH-Based KEM 5
6. Hybrid Public Key Encryption 5
6.1. Encryption to a Public Key 6
6.2. Authentication using a Pre-Shared Key 7
6.3. Authentication using an Asymmetric Key 8
6.4. Encryption and Decryption 9
7. Ciphersuites 10
8. Security Considerations 11
9. IANA Considerations 11
10. References 11
10.1. Normative References 11
10.2. Informative References 12
Appendix A. Possible TODOs 13
Authors' Addresses 13

1. Introduction

Hybrid public-key encryption (HPKE) is a substantially more efficient solution than traditional public key encryption techniques such as those based on RSA or ElGamal. Encrypted messages convey a single ciphertext and authentication tag alongside a short public key, which may be further compressed. The key size and computational complexity of elliptic curve cryptographic primitives for authenticated encryption therefore make it compelling for a variety of use case. This type of public key encryption has many applications in practice, for example, in PGP [RFC6637] and in the developing Messaging Layer Security protocol [I-D.ietf-mls-protocol].

Currently, there are numerous competing and non-interoperable standards and variants for hybrid encryption, including ANSI X9.63 [ANSI], IEEE 1363a [IEEE], ISO/IEC 18033-2 [ISO], and SECG SEC 1 [SECG]. Lack of a single standard makes selection and deployment of a compatible, cross-platform and ecosystem solution difficult to define. This document defines an HPKE scheme that provides a subset of the functions provided by the collection of schemes above, but specified with sufficient clarity that they can be interoperably implemented and formally verified.

2. Requirements Notation

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 BCP14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Security Properties

As a hybrid authentication encryption algorithm, we desire security against (adaptive) chosen ciphertext attacks (IND-CCA2 secure). The HPKE variants described in this document achieve this property under the Random Oracle model assuming the gap Computational Diffie Hellman (CDH) problem is hard [S01].

4. Notation

The following terms are used throughout this document to describe the operations, roles, and behaviors of HPKE:

- Initiator (I): Sender of an encrypted message.
- Responder (R): Receiver of an encrypted message.
- Ephemeral (E): A fresh random value meant for one-time use.
- "(skX, pkX)": A KEM key pair used in role X; "skX" is the private key and "pkX" is the public key
- "pk(sk)": The public key corresponding to a private key
- "len(x)": The one-octet length of the octet string "x"
- "+": Concatenation of octet strings; "0x01 + 0x02 = 0x0102"
- "*": Repetition of an octet string; "0x01 * 4 = 0x01010101"
- "^": XOR of octet strings; "0xF0F0 ^ 0x1234 = 0xE2C4"

5. Cryptographic Dependencies

HPKE variants rely on the following primitives:

- A Key Encapsulation Mechanism (KEM):
  - GenerateKeyPair(): Generate a key pair (sk, pk)

  * Marshal(pk): Produce a fixed-length octet string encoding the public key "pk"

  * Unmarshal(enc): Parse a fixed-length octet string to recover a public key

  * Encap(pk): Generate an ephemeral symmetric key and a fixed-length encapsulation of that key that can be decapsulated by the holder of the private key corresponding to pk

  * Decap(enc, sk): Use the private key "sk" to recover the ephemeral symmetric key from its encapsulated representation "enc"

  * AuthEncap(pkR, skI) (optional): Same as Encap(), but the outputs encode an assurance that the ephemeral shared key is known only to the holder of the private key "skI"

  * AuthDecap(skI, pkR) (optional): Same as Decap(), but the holder of the private key "skI" is assured that the ephemeral shared key is known only to the holder of the private key corresponding to "pkI"

o A Key Derivation Function:

  * Extract(salt, IKM): Extract a pseudorandom key of fixed length from input keying material "IKM" and an optional octet string "salt"

  * Expand(PRK, info, L): Expand a pseudorandom key "PRK" using optional string "info" into "L" bytes of output keying material

  * Nh: The output size of the Extract function

o An AEAD encryption algorithm [RFC5116]:

  * Seal(key, nonce, aad, pt): Encrypt and authenticate plaintext "pt" with associated data "aad" using secret key "key" and nonce "nonce", yielding ciphertext and tag "ct"

  * Open(key, nonce, aad, ct): Decrypt ciphertext "ct" using associated data "aad" with secret key "key" and nonce "nonce", returning plaintext message "pt" or the error value "OpenError"

  * Nk: The length in octets of a key for this algorithm

  * Nn: The length in octets of a nonce for this algorithm

A set of concrete instantiations of these primitives is provided in Section 7. Ciphersuite values are two octets long.

5.1. DH-Based KEM

Suppose we are given a Diffie-Hellman group that provides the following operations:

- GenerateKeyPair(): Generate an ephemeral key pair "(sk, pk)" for the DH group in use

- DH(sk, pk): Perform a non-interactive DH exchange using the private key sk and public key pk to produce a shared secret

- Marshal(pk): Produce a fixed-length octet string encoding the public key "pk"

Then we can construct a KEM (which we'll call "DHKEM") in the following way:

```
def Encap(pkR):
  skE, pkE = GenerateKeyPair()
  zz = DH(skE, pkR)
  enc = Marshal(pkE)
  return zz, enc

def Decap(enc, skR):
  pkE = Unmarshal(enc)
  return DH(skR, pkE)

def AuthEncap(pkR, skI):
  skE, pkE = GenerateKeyPair()
  zz = DH(skE, pkR) + DH(skI, pkR)
  enc = Marshal(pkE)
  return zz, enc

def AuthDecap(enc, skR, pkI):
  pkE = Unmarshal(enc)
  return DH(skR, pkE) + DH(skR, pkI)
```

The Marshal and GenerateKeyPair functions are the same as for the underlying DH group.

6. Hybrid Public Key Encryption

In this section, we define a few HPKE variants. All cases take a plaintext "pt" and a recipient public key "pkR" and produce an ciphertext "ct" and an encapsulated key "enc". These outputs are

constructed so that only the holder of the private key corresponding to "pkR" can decapsulate the key from "enc" and decrypt the ciphertext. All of the algorithms also take an "info" parameter that can be used to influence the generation of keys (e.g., to fold in identity information) and an "aad" parameter that provides Additional Authenticated Data to the AEAD algorithm in use.

In addition to the base case of encrypting to a public key, we include two authenticated variants, one of which authenticates possession of a pre-shared key, and one of which authenticates possession of a KEM private key. The following one-octet values will be used to distinguish between modes:

Mode	Value
mode_base	0x00
mode_psk	0x01
mode_auth	0x02

All of these cases follow the same basic two-step pattern:

1. Set up an encryption context that is shared between the sender and the recipient

2. Use that context to encrypt or decrypt content

A "context" encodes the AEAD algorithm and key in use, and manages the nonces used so that the same nonce is not used with multiple plaintexts.

The procedures described in this session are laid out in a Python-like pseudocode. The ciphersuite in use is left implicit.

6.1. Encryption to a Public Key

The most basic function of an HPKE scheme is to enable encryption for the holder of a given KEM private key. The "SetupBaseI()" and "SetupBaseR()" procedures establish contexts that can be used to encrypt and decrypt, respectively, for a given private key.

The the shared secret produced by the KEM is combined via the KDF with information describing the key exchange, as well as the explicit "info" parameter provided by the caller.

Note that the "SetupCore()" method is also used by the other HPKE variants describe below. The value "0*Nh" in the "SetupBase()" procedure represents an all-zero octet string of length "Nh".

```
def SetupCore(mode, secret, kemContext, info):
  context = ciphersuite + mode +
            len(kemContext) + kemContext +
            len(info) + info
  key = Expand(secret, "hpke key" + context, Nk)
  nonce = Expand(secret, "hpke nonce" + context, Nn)
  return Context(key, nonce)

def SetupBase(pkR, zz, enc, info):
  kemContext = enc + pkR
  secret = Extract(0\*Nh, zz)
  return SetupCore(mode_base, secret, kemContext, info)

def SetupBaseI(pkR, info):
  zz, enc = Encap(pkR)
  return SetupBase(pkR, zz, enc, info)

def SetupBaseR(enc, skR, info):
  zz = Decap(enc, skR)
  return SetupBase(pk(skR), zz, enc, info)
```

Note that the context construction in the SetupCore procedure is equivalent to serializing a structure of the following form in the TLS presentation syntax:

```
struct {
    uint16 ciphersuite;
    uint8 mode;
    opaque kemContext<0..255>;
    opaque info<0..255>;
} HPKEContext;
```

6.2. Authentication using a Pre-Shared Key

This variant extends the base mechansism by allowing the recipient to authenticate that the sender possessed a given pre-shared key (PSK). We assume that both parties have been provisioned with both the PSK value "psk" and another octet string "pskID" that is used to identify which PSK should be used.

The primary differences from the base case are:

o The PSK is used as the "salt" input to the KDF (instead of 0)

- o The PSK ID is added to the context string used as the "info" input to the KDF

This mechanism is not suitable for use with a low-entropy password as the PSK. A malicious recipient that does not possess the PSK can use decryption of a plaintext as an oracle for performing offline dictionary attacks.

```
def SetupPSK(pkR, psk, pskID, zz, enc, info):
  kemContext = enc + pkR + pskID
  secret = Extract(psk, zz)
  return SetupCore(mode_psk, secret, kemContext, info)

def SetupPSKI(pkR, psk, pskID, info):
  zz, enc = Encap(pkR)
  return SetupPSK(pkR, psk, pskID, zz, enc, info)

def SetupPSKR(enc, skR, psk, pskID, info):
  zz = Decap(enc, skR)
  return SetupPSK(pk(skR), psk, pskID, zz, enc, info)
```

6.3. Authentication using an Asymmetric Key

This variant extends the base mechansism by allowing the recipient to authenticate that the sender possessed a given KEM private key. This assurance is based on the assumption that "AuthDecap(enc, skR, pkI)" produces the correct shared secret only if the encapsulated value "enc" was produced by "AuthEncap(pkR, skI)", where "skI" is the private key corresponding to "pkI". In other words, only two people could have produced this secret, so if the recipient is one, then the sender must be the other.

The primary differences from the base case are:

- o The calls to "Encap" and "Decap" are replaced with calls to "AuthEncap" and "AuthDecap".

- o The initiator public key is added to the context string used as the "info" input to the KDF

Obviously, this variant can only be used with a KEM that provides "AuthEncap()" and "AuthDecap()" procuedures.

This mechanism authenticates only the key pair of the initiator, not any other identity. If an application wishes to authenticate some other identity for the sender (e.g., an email address or domain name), then this identity should be included in the "info" parameter to avoid unknown key share attacks.

```
def SetupAuth(pkR, pkI, zz, enc, info):
  kemContext = enc + pkR + pkI
  secret = Extract(0*Nh, zz)
  return SetupCore(mode_auth, secret, kemContext, info)

def SetupAuthI(pkR, skI, info):
  zz, enc = AuthEncap(pkR, skI)
  return SetupAuth(pkR, pk(skI), zz, enc, info)

def SetupAuthR(enc, skR, pkI, info):
  zz = AuthDecap(enc, skR, pkI)
  return SetupAuth(pk(skR), pkI, zz, enc, info)
```

6.4. Encryption and Decryption

HPKE allows multiple encryption operations to be done based on a given setup transaction. Since the public-key operations involved in setup are typically more expensive than symmetric encryption or decryption, this allows applications to "amortize" the cost of the public-key operations, reducing the overall overhead.

In order to avoid nonce reuse, however, this decryption must be stateful. Each of the setup procedures above produces a context object that stores the required state:

- The AEAD algorithm in use
- The key to be used with the AEAD algorithm
- A base nonce value
- A sequence number (initially 0)

All of these fields except the sequence number are constant. The sequence number is used to provide nonce uniqueness: The nonce used for each encryption or decryption operation is the result of XORing the base nonce with the current sequence number, encoded as a big-endian integer of the same length as the nonce. Implementations MAY use a sequence number that is shorter than the nonce (padding on the left with zero), but MUST return an error if the sequence number overflows.

Each encryption or decryption operation increments the sequence number for the context in use. A given context SHOULD be used either only for encryption or only for decryption.

It is up to the application to ensure that encryptions and decryptions are done in the proper sequence, so that the nonce values used for encryption and decryption line up.

```
def Context.Nonce(seq):
  encSeq = encode\_big\_endian(seq, len(self.nonce))
  return self.nonce ^ encSeq

def Context.Seal(aad, pt):
  ct = Seal(self.key, self.Nonce(self.seq), aad, pt)
  self.seq += 1
  return ct

def Context.Open(aad, ct):
  pt = Open(self.key, self.Nonce(self.seq), aad, pt)
  if pt == OpenError:
    return OpenError
  self.seq += 1
  return pt
```

7. Ciphersuites

The HPKE variants as presented will function correctly for any combination of primitives that provides the functions described above. In this section, we provide specific instantiations of these primitives for standard groups, including: Curve25519, Curve448 [RFC7748], and the NIST curves P-256 and P-512.

Value	KEM	KDF	AEAD
0x0001	DHKEM(P-256)	HKDF-SHA256	AES-GCM-128
0x0002	DHKEM(P-256)	HKDF-SHA256	ChaCha20Poly1305
0x0002	DHKEM(Curve25519)	HKDF-SHA256	AES-GCM-128
0x0002	DHKEM(Curve25519)	HKDF-SHA256	ChaCha20Poly1305
0x0001	DHKEM(P-521)	HKDF-SHA512	AES-GCM-256
0x0002	DHKEM(P-521)	HKDF-SHA512	ChaCha20Poly1305
0x0002	DHKEM(Curve448)	HKDF-SHA512	AES-GCM-256
0x0002	DHKEM(Curve448)	HKDF-SHA512	ChaCha20Poly1305

For the NIST curves P-256 and P-521, the Marshal function of the DH scheme produces the normal (non-compressed) representation of the public key, according to [SECG]. When these curves are used, the recipient of an HPKE ciphertext MUST validate that the ephemeral public key "pkE" is on the curve. The relevant validation procedures are defined in [keyagreement]

For the CFRG curves Curve25519 and Curve448, the Marshal function is the identity function, since these curves already use fixed-length octet strings for public keys.

The values "Nk" and "Nn" for the AEAD algorithms referenced above are as follows:

AEAD	Nk	Nn
AES-GCM-128	16	12
AES-GCM-256	32	12
ChaCha20Poly1305	32	12

8. Security Considerations

[[TODO]]

9. IANA Considerations

[[OPEN ISSUE: Should the above table be in an IANA registry?]]

10. References

10.1. Normative References

[ANSI] "Public Key Cryptography for the Financial Services Industry -- Key Agreement and Key Transport Using Elliptic Curve Cryptography", n.d..

[IEEE] "IEEE 1363a, Standard Specifications for Public Key Cryptography - Amendment 1 -- Additional Techniques", n.d..

[ISO] "ISO/IEC 18033-2, Information Technology - Security Techniques - Encryption Algorithms - Part 2 -- Asymmetric Ciphers", n.d..

[keyagreement]
Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013.

[MAEA10] "A Comparison of the Standardized Versions of ECIES", n.d., <http://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF>.

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[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>.

[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>.

[S01] "A Proposal for an ISO Standard for Public Key Encryption (verison 2.1)", n.d., <http://www.shoup.net/papers/iso-2_1.pdf>.

[SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", n.d., <http://www.secg.org/download/aid-780/sec1-v2.pdf>.

10.2. Informative References

[I-D.ietf-mls-protocol]
Barnes, R., Millican, J., Omara, E., Cohn-Gordon, K., and R. Robert, "The Messaging Layer Security (MLS) Protocol", draft-ietf-mls-protocol-03 (work in progress), January 2019.

[RFC6637] Jivsov, A., "Elliptic Curve Cryptography (ECC) in OpenPGP", RFC 6637, DOI 10.17487/RFC6637, June 2012, <https://www.rfc-editor.org/info/rfc6637>.

Appendix A. Possible TODOs

The following extensions might be worth specifying:

- Multiple recipients - It might be possible to add some simplifications / assurances for the case where the same value is being encrypted to multiple recipients.

- Test vectors - Obviously, we can provide decryption test vectors in this document. In order to provide known-answer tests, we would have to introduce a non-secure deterministic mode where the ephemeral key pair is derived from the inputs. And to do that safely, we would need to augment the decrypt function to detect the deterministic mode and fail.

- A reference implementation in hacspec or similar

Authors' Addresses

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

Karthik Bhargavan
Inria

Email: karthikeyan.bhargavan@inria.fr