KEM-based Hybrid Forward Secrecy for Noise

Trevor Perrin (noise@trevp.net)

Revision 1, 2018-11-17, unofficial/unstable

Contents

1. Introduction

This document describes the "hfs" modifier for Noise. This modifier adds KEM-based hybrid forward secrecy to a Noise handshake.

2. Overview

Noise handshakes depend on some Diffie-Hellman function (usually an Elliptic Curve Diffie-Hellman function such as X25519). If this function is broken by future cryptanalysis then previously-recorded traffic could be decrypted.

To hedge against this possibility, one could use a different public-key algorithm to add *hybrid forward secrecy*, so that both this new algorithm and the DH algorithm would have to be broken to decrypt old traffic. At time of writing, there's some interest in using *post-quantum cryptographic algorithms* which might be secure against a *quantum computer* which could break DH.

These post-quantum algorithms often come in the form of a **KEM**, or Key Encapsulation Mechanism. Unlike a DH algorithm where both parties exchange public keys in any order, a KEM algorithm requires one party to send a ciphertext based on the other party's public key.

This document describes a Noise handshake modifer (`"hfs"`) that adds hybrid forward secrecy using a KEM algorithm.

3. KEM functions

KEM functions are similar to DH functions, but require an exchange of a public key followed by a ciphertext, rather than an exchange of public keys.

The signature for these functions is defined below.

- `GENERATE_KEM_KEYPAIR()`: Generates a new KEM key pair. A KEM key pair consists of `public_key` and `private_key` elements. A `public_key` represents an encoding of a KEM public key into a byte sequence of length `KEMPUBLICKEYLEN`. The `public_key` encoding details are specific to each set of KEM functions.
- `GENERATE_KEM_CIPHERTEXT(public_key)`: Generates both a KEM ciphertext and a KEM output based on the recipient's public key. A `ciphertext` represents an encoding of a KEM ciphertext into a byte sequence of length `KEMCIPHERTEXTLEN`. A `kem_output` represents an encoding of a KEM output into a byte sequence of length `KEMOUTPUTLEN`. The `ciphertext` and `kem_output` encoding details are specific to each set of KEM functions.
- `KEM(key_pair, ciphertext)`: Performs a KEM calculation between the private key in `key_pair` and the `ciphertext` and returns a KEM output byte sequence of length `KEMOUTPUTLEN`. The `kem_output` matches the value that was generated by `GENERATE_KEM_CIPHERTEXT()`.
- `KEMPUBLICKEYLEN` = A constant specifying the size in bytes of KEM public keys.
- `KEMCIPHERTEXTLEN` = A constant specifying the size in bytes of KEM ciphertexts.
- `KEMOUTPUTLEN` = A constant specifying the size in bytes of KEM outputs.

4. KEM tokens

Two new tokens are introduced by this document:

- `"e1"` = This token directs the sender to call `GENERATE_KEM_KEYPAIR()`. The resulting KEM public key is transferred to the recipient as if it was a static DH public key (i.e. using `EncryptAndHash()` if a key is available).
- `"ekem1"` = This token directs the sender to call `GENERATE_KEM_CIPHERTEXT()` using a previously received KEM public key. The resulting KEM ciphertext is transferred to the recipient as if it was a static DH public key (i.e. using `EncryptAndHash()` if a key is available). On receiving this token, the recipient will call `KEM(key_pair, ciphertext)` to derive the same `kem_output` as the sender possesses. On sending or receiving this token, the parties call `MixKey(kem_output)`.

5. The `"hfs"` modifier

The `"hfs"` pattern modifier adds an `"e1"` token directly following the first occurrence of `"e"`, unless there is a DH operation in this same message, in which case the `"hfs"` token is placed directly after this DH (so that the public key will be encrypted).

The `"hfs"` modifier also adds an `"ekem1"` token directly following the first occurrence of `"ee"`.

When the `"hfs"` modifier is used, the DH name section must contain a KEM algorithm name directly following the DH algorithm name, separated by a plus sign.

The table below lists some example unmodified patterns on the left, and some HFS patterns on the right:

```
NN:
  -> e
  <- e, ee
```

```
NNhfs:
  -> e, e1
  <- e, ee, ekem1
```

```
NK:
  <- s
  ...
  -> e, es
  <- e, ee
```

```
NKhfs:
  <- s
  ...
  -> e, es, e1
  <- e, ee, ekem1
```

```
NX:
  -> e
  <- e, ee, s, es
```

```
NXhfs:
  -> e, e1
  <- e, ee, ekem1, s, es
```

```
XN:
  -> e
  <- e, ee
  -> s, se
```

```
XNhfs:
  -> e, e1
  <- e, ee, ekem1
  -> s, se
```

```
XK:
  <- s
  ...
  -> e, es
  <- e, ee
  -> s, se
```

```
XKhfs:
  <- s
  ...
  -> e, es, e1
  <- e, ee, ekem1
  -> s, se
```

```
XX:
  -> e
  <- e, ee, s, es
  -> s, se
```

```
XXhfs:
  -> e, e1
  <- e, ee, ekem1, s, es
  -> s, se
```

```
KN:
  -> s
  ...
  -> e
  <- e, ee, se
```

```
KNhfs:
  -> s
  ...
  -> e, e1
  <- e, ee, ekem1, se
```

```
KK:
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se
```

```
KKhfs:
  -> s
  <- s
  ...
  -> e, e1, es, ss
  <- e, ee, ekem1, se
```

```
KX:
  -> s
  ...
  -> e
  <- e, ee, se, s, es
```

```
KXhfs:
  -> s
  ...
  -> e, e1
  <- e, ee, ekem1, se, s, es
```

```
IN:
  -> e, s
  <- e, ee, se
```

```
INhfs:
  -> e, e1, s
  <- e, ekem1, se
```

```
IK:
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se
```

```
IKhfs:
  <- s
  ...
  -> e, es, e1, s, ss
  <- e, ee, ekem1, se
```

```
IX:
  -> e, s
  <- e, ee, se, s, es
```

```
IXhfs:
  -> e, e1, s
  <- e, ee, ekem1, se, s, es
```

---

6. Selecting postquantum KEMs

Postquantum algorithms are an active area of research. This document doesn't list any such algorithms, because it's unclear which algorithms will remain secure into the future (against both quantum and nonquantum attacks).

It's even more unclear how to balance the security strength of these algorithms against performance considerations, and even more unclear which algorithms are likely to become widely-supported.

7. Security considerations

Due to the Noise key derivation, even using a weak KEM algorithm will not hurt the security of the protocol. However it's difficult to assess how much security current postquantum algorithms provide.

Note that KEM public keys are sometimes sent in clear. If making the handshake indistinguishable from random bytes is a goal, some additional method will have to be used to obscure these values (and the DH ephemeral public keys).

6. Rationales

KEM public keys and ciphertexts are encrypted when possible because:

- This maintains consistency with current Noise behavior, which is to encrypt everything except the ephemeral DH public keys.
- This behavior for tokens makes it easier to obscure the type of protocol being executed, since more of the handshake is encrypted.
- This behavior might add other small security benefits (e.g. if an attacker could break some KEM public keys but not all of them, then encrypting the KEM public key might force them to perform an expensive attack against DH to determine if the KEM public key was vulnerable).

7. IPR

8. Acknowledgements

This document draws on discussions and an earlier spec from Rhys Weatherley.