# The Noise Protocol Framework: Book 2

Trevor Perrin (noise@trevp.net)

Revision 0draft, 2017-05-09

## Contents



## 1. Introduction

Noise is a framework for crypto protocols based on Diffie-Hellman key agreement. This specification defines additional features that can be used in combination with the features defined in Book 1 [1].

# 2. Modifiers

To specify extensions or modifications to the behavior in Book 1 [1], Noise uses **modifiers**.

A modifier is an ASCII string which is added to some component of the Noise protocol name. A modifier can be a **pattern modifier**, **DH modifier**, **cipher modifier**, or **hash modifier**, depending on which component of the protocol name it is added to.

The first modifier added onto a base name is simply appended. Thus, `fallback` is a pattern modifier added to the `XX` base name to produce `XXfallback`. Additional modifiers are separated with a plus sign. Thus, adding the `psk0` pattern modifier (below) would result in the pattern name `XXfallback+psk0`.

The final protocol name, including all modifiers, must be less than or equal to 255 bytes (e.g. `Noise_XXfallback+psk0_25519_AESGCM_SHA256`).

# 3. Pre-shared symmetric keys (PSK)

Noise provides a **pre-shared symmetric key** or **PSK** mode to support protocols where both parties have a shared secret key. When using PSK mode, the following changes are made:

## 3.1. Cryptographic functions

An additional function is defined based on the `HASH()` function:

- `HKDF3(chaining_key, input_key_material)`: Takes a `chaining_key` byte sequence of length `HASHLEN`, and an `input_key_material` byte sequence with length of 32 bytes or `DHLEN` bytes. Returns a triple of byte sequences each of length `HASHLEN`:
  - Sets `temp_key = HMAC-HASH(chaining_key, input_key_material)`.
  - Sets `output1 = HMAC-HASH(temp_key, byte(0x01))`.
  - Sets `output2 = HMAC-HASH(temp_key, output1 || byte(0x02))`.
  - Sets `output3 = HMAC-HASH(temp_key, output2 || byte(0x03))`.
  - Returns `(output1, output2, output3)`.

Note that `temp_key`, `output1`, `output2`, and `output3` are all `HASHLEN` bytes in length. Also note that the `HKDF3()` function is identical to `HKDF2()` from [1] except that a third output is produced.

An additional function is also defined in the `SymmetricState` object:

- `MixKeyAndHash(input_key_material)`: Executes the following steps:
  - Sets `ck, temp_h, temp_k = HKDF3(ck, input_key_material)`.
  - Calls `MixHash(temp_h)`.
  - If `HASHLEN` is 64, then truncates `temp_k` to 32 bytes.
  - Calls `InitializeKey(temp_k)`.

This function combines the functionality of `MixKey()` and `MixHash()` from [1]. Unlike those, this function ensures that its input affects both the encryption keys and the `h` value.

The third output from `HKDF3()` is used as the `k` value so that calculation of `k` may be skipped if `k` is not used.

## 3.2. Handshake tokens

To support PSKs, a new `"psk"` token is allowed to appear once (or multiple times) in a handshake pattern. This token can only appear in message patterns (not pre-message patterns). This token is processed by calling `MixKeyAndHash(psk)`, where `psk` is a 32-byte secret value provided by the application.

In [1], the `"e"` token in a pre-message pattern or message pattern always results in a call to `MixHash(e.public_key)`. When PSKs are used, all of these calls are replaced with `MixKeyAndHash(e.public_key)`. In conjunction with the validity rule in the next section, this ensures that PSK-based encryption uses encryption keys that are randomized using ephemeral public keys as nonces.

## 3.3. Validity rule

To prevent catastrophic key reuse, handshake patterns using the `"psk"` token must follow an additional validity rule:

- A party may not send any encrypted data after it processes a `"psk"` token unless it has previously sent an ephemeral public key (an `"e"` token), either before or after the `"psk"` token.

This rule guarantees that a `k` derived from a PSK will never be used for encryption unless it has also been randomized by `MixKeyAndHash(e.public_key)` using a self-chosen ephemeral public key.

## 3.4. Pattern modifiers

To indicate PSK mode and the placement of the `"psk"` token, pattern modifiers are used (see Section 2). The modifier `psk0` places a `"psk"` token at the beginning of the first handshake message. The modifiers `psk1`, `psk2`, etc., place a `"psk"` token at the end of the first, second, etc., handshake message.

Any pattern using one of these modifiers must process tokens according to the rules in Section 3.2, and must follow the validity rule in Section 3.3.

The table below lists some unmodified one-way patterns on the left, and the recommended PSK pattern on the right:

| | |
|---|---|
| `Noise_N(rs):`<br>`  <- s`<br>`  ...`<br>`  -> e, es` | `Noise_Npsk0(rs):`<br>`  <- s`<br>`  ...`<br>`  -> psk, e, es` |
| `Noise_K(s, rs):`<br>`  <- s`<br>`  ...`<br>`  -> e, es, ss` | `Noise_Kpsk0(s, rs):`<br>`  <- s`<br>`  ...`<br>`  -> psk, e, es, ss` |
| `Noise_X(s, rs):`<br>`  <- s`<br>`  ...`<br>`  -> e, es, s, ss` | `Noise_Xpsk1(s, rs):`<br>`  <- s`<br>`  ...`<br>`  -> e, es, s, ss, psk` |

Note that the `psk1` modifier is recommended for `Noise_X`. This is because `Noise_X` transmits the initiator's static public key. Because PSKs are typically pairwise, the responder likely cannot determine the PSK until it has decrypted the initiator's static public key. Thus, `psk1` is likely to be more useful here than `psk0`.

Following similar logic, we can define the most likely interactive PSK patterns:

| | |
|---|---|
| `Noise_NN():`<br>`  -> e`<br>`  <- e, ee` | `Noise_NNpsk0():`<br>`  -> psk, e`<br>`  <- e, ee` |
| `Noise_NN():`<br>`  -> e`<br>`  <- e, ee` | `Noise_NNpsk2():`<br>`  -> e`<br>`  <- e, ee, psk` |

```
Noise_NK(rs):
  <- s
  ...
  -> e, es
  <- e, ee
```

```
Noise_NKpsk0(rs):
  <- s
  ...
  -> psk, e, es
  <- e, ee
```

```
Noise_NK(rs):
  <- s
  ...
  -> e, es
  <- e, ee
```

```
Noise_NKpsk2(rs):
  <- s
  ...
  -> e, es
  <- e, ee, psk
```

```
Noise_NX(rs):
  -> e
  <- e, ee, s, es
```

```
Noise_NXpsk2(rs):
  -> e
  <- e, ee, s, es, psk
```

```
Noise_XN(s):
  -> e
  <- e, ee
  -> s, se
```

```
Noise_XNpsk3(s):
  -> e
  <- e, ee
  -> s, se, psk
```

```
Noise_XK(s, rs):
  <- s
  ...
  -> e, es
  <- e, ee
  -> s, se
```

```
Noise_XKpsk3(s, rs):
  <- s
  ...
  -> e, es
  <- e, ee
  -> s, se, psk
```

```
Noise_XX(s, rs):
  -> e
  <- e, ee, s, es
  -> s, se
```

```
Noise_XXpsk3(s, rs):
  -> e
  <- e, ee, s, es
  -> s, se, psk
```

```
Noise_KN(s):
  -> s
  ...
  -> e
  <- e, ee, se
```

```
Noise_KNpsk0(s):
  -> s
  ...
  -> psk, e
  <- e, ee, se
```

```
Noise_KN(s):
  -> s
  ...
  -> e
  <- e, ee, se
```

```
Noise_KNpsk2(s):
  -> s
  ...
  -> e
  <- e, ee, se, psk
```

```
Noise_KK(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se
```

```
Noise_KKpsk0(s, rs):
  -> s
  <- s
  ...
  -> psk, e, es, ss
  <- e, ee, se
```

```
Noise_KK(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se
```

```
Noise_KKpsk2(s, rs):
  -> s
  <- s
  ...
  -> e, es, ss
  <- e, ee, se, psk
```

```
Noise_KX(s, rs):
  -> s
  ...
  -> e
  <- e, ee, se, s, es
```

```
Noise_KXpsk2(s, rs):
  -> s
  ...
  -> e
  <- e, ee, se, s, es, psk
```

```
Noise_IN(s):
  -> e, s
  <- e, ee, se
```

```
Noise_INpsk1(s):
  -> e, s, psk
  <- e, ee, se
```

```
Noise_IN(s):
  -> e, s
  <- e, ee, se
```

```
Noise_INpsk2(s):
  -> e, s
  <- e, ee, se, psk
```

```
Noise_IK(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se
```

```
Noise_IKpsk1(s, rs):
  <- s
  ...
  -> e, es, s, ss, psk
  <- e, ee, se
```

```
Noise_IK(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se
```

```
Noise_IKpsk2(s, rs):
  <- s
  ...
  -> e, es, s, ss
  <- e, ee, se, psk
```

```
Noise_IX(s, rs):
  -> e, s
  <- e, ee, se, s, es
```

```
Noise_IXpsk2(s, rs):
  -> e, s
  <- e, ee, se, s, es, psk
```

---

Of course, the above list does not exhaust all possible patterns that can be formed with these modifiers, nor does it exhaust all the ways that `"psk"` tokens could be used outside of these modifiers (e.g. multiple PSKs per handshake). Defining additional PSK patterns is outside the scope of this document.

# 4. Rekey

Parties might wish to periodically update their cipherstate keys using a one-way function, so that a compromise of cipherstate keys will not decrypt older messages. Periodic rekey might also be used to reduce the volume of data encrypted under a single cipher key (this is usually not important with good ciphers, though note the discussion on `AESGCM` data volumes in Section 14 of [1]).

To enable this, Noise supports a `Rekey()` function which may be called on a `CipherState`.

It is up to to the application if and when to perform rekey. For example:

- Applications might perform continuous rekey, where they rekey the relevant cipherstate after every transport message sent or received. This is simple and gives good protection to older ciphertexts, but might be difficult for implementations where changing keys is expensive.
- Applications might rekey a cipherstate automatically after it has has been used to send or receive some number of messages.

- Applications might choose to rekey based on arbitrary criteria, in which case they signal this to the other party by sending a message.

Applications must make these decisions on their own; there are no modifiers which specify rekey behavior.

Note that rekey only updates the cipherstate's `k` value, it doesn't reset the cipherstate's `n` value, so applications performing rekey must still perform a new handshake if sending $2^{64}$ or more transport messages.

### 4.1. Cryptographic functions

To support rekey, an additional cipher function is defined:

- `REKEY(k)`: Returns a new 32-byte cipher key as a pseudorandom function of `k`. If this function is not specifically defined for some set of cipher functions, then it defaults to returning the first 32 bytes from `ENCRYPT(k, maxnonce, zerolen, zeros)`, where `MAXNONCE` equals $2^{64}$-1, `zerolen` is a zero-length byte sequence, and `zeros` is a sequence of 32 bytes filled with zeros.

An additional function is defined in the `CipherState` object:

- `Rekey()`: Sets `k = REKEY(k)`.

## 5. Security Considerations

- **Pre-shared symmetric keys**: Pre-shared symmetric keys must be secret values with 256 bits of entropy.

## 6. Rationales

This section collects various design rationales.

### 5.1. Ciphers and encryption

Pre-shared symmetric keys are 256 bits because:

- Pre-shared key length is fixed to simplify testing and implementation, and to deter users from mistakenly using low-entropy passwords as pre-shared keys.

Rekey defaults to using encryption with the nonce $2^{64}$-1 because:

- With `AESGCM` and `ChaChaPoly` rekey can be computed efficiently (the "encryption" just needs to apply the cipher, and can skip calculation of the authentication tag).

Rekey doesn't reset `n` to zero because:

- Leaving `n` unchanged is simple.
- If the cipher has a weakness such that repeated rekeying gives rise to a cycle of keys, then letting `n` advance will avoid catastrophic reuse of the same `k` and `n` values.
- Letting `n` advance puts a bound on the total number of encryptions that can be performed with a set of derived keys.

# 7. IPR

The Noise specification Book 2 (this document) is hereby placed in the public domain.

# 8. Acknowledgements

The PSK approach was largely motivated and designed by Jason Donenfeld, based on his experience with PSKs in WireGuard.

The rekey design benefited from discussions with Rhys Weatherley, Alexey Ermishkin, and Olaoluwa Osuntokun.

# 9. References

[1] Trevor Perrin, "The Noise Protocol Framework: Book 1." 2017. https://noiseprotocol.org/noise_book1.pdf