The NoiseSocket Protocol

Alexey Ermishkin (scratch@virgilsecurity.com)

Revision 0draft, 2017-05-01

Contents

Abstract

NoiseSocket is an extension of the Noise Protocol Framework (developed by the authors of Signal and currently used by WhatsApp) that enables quick and seamless Transport Layer Security (TLS) between multiple parties with minimal code space overhead, small keys, and extremely fast speed. NoiseSocket is designed to overcome the shortcomings of existing TLS implementations and

targets IoT devices, microservices, back-end applications such as datacenter-to-datacenter communications, and use cases where third-party certificate of authority infrastructure is not optimal.

1. Crypto

Each party maintains the following variables:

- **`s, e`**: The local party's static and ephemeral key pairs (which may be empty).
- **`rs, re`**: The remote party's static and ephemeral public keys (which may be empty).
- **`h`**: A **handshake hash** value that hashes all the handshake data that's been sent and received.
- **`ck`**: A **chaining key** that hashes all previous DH outputs. Once the handshake completes, the chaining key will be used to derive the encryption keys for transport messages.
- **`k, n`**: An encryption key `k` (which may be empty) and a counter-based nonce `n`. Whenever a new DH output causes a new `ck` to be calculated, a new `k` is also calculated. The key `k` and nonce `n` are used to encrypt static public keys and handshake payloads. Encryption with `k` uses some **AEAD** cipher mode (in the sense of Rogaway [1]) and uses the current `h` value as **associated data** which is covered by the AEAD authentication. Encryption of static public keys and payloads provides some confidentiality and key confirmation during the handshake phase.

NoiseSocket protocol is instantiated with a concrete set of **DH functions**, **cipher functions**, and a **hash function**. The signature for these functions is defined below.

1.1. DH functions

Noise depends on the following **DH functions** (and an associated constant):

- **`GENERATE_KEYPAIR()`**: Generates a new Diffie-Hellman key pair. A DH key pair consists of `public_key` and `private_key` elements. A `public_key` represents an encoding of a DH public key into a byte sequence of length `DHLEN`. The `public_key` encoding details are specific to each set of DH functions.
- **`DH(key_pair, public_key)`**: Performs a Diffie-Hellman calculation between the private key in `key_pair` and the `public_key` and returns an output sequence of bytes of length `DHLEN`.

The `public_key` either encodes some value in a large prime-order group (which may have multiple equivalent encodings), or is an invalid value. Implementations must handle invalid public keys either by returning some output which is purely a function of the public key and does not depend on the private key, or by signaling an error to the caller. The DH function may define more specific rules for handling invalid values.

- **`DHLEN`** = A constant specifying the size in bytes of public keys and DH outputs. For security reasons, `DHLEN` must be 32 or greater.

1.2. Cipher functions

Noise depends on the following **cipher functions**:

- **`ENCRYPT(k, n, ad, plaintext)`**: Encrypts `plaintext` using the cipher key `k` of 32 bytes and an 8-byte unsigned integer nonce `n` which must be unique for the key `k`. Returns the ciphertext. Encryption must be done with an "AEAD" encryption mode with the associated data `ad` (using the terminology from [1]) and returns a ciphertext that is the same size as the plaintext plus 16 bytes for authentication data. The entire ciphertext must be indistinguishable from random if the key is secret.

- **`DECRYPT(k, n, ad, ciphertext)`**: Decrypts `ciphertext` using a cipher key `k` of 32 bytes, an 8-byte unsigned integer nonce `n`, and associated data `ad`. Returns the plaintext, unless authentication fails, in which case an error is signaled to the caller.

1.3. Hash functions

Noise depends on the following **hash function** (and associated constants):

- **`HASH(data)`**: Hashes some arbitrary-length data with a collision-resistant cryptographic hash function and returns an output of `HASHLEN` bytes.

- **`HASHLEN`** = A constant specifying the size in bytes of the hash output. Must be 32 or 64.

- **`BLOCKLEN`** = A constant specifying the size in bytes that the hash function uses internally to divide its input for iterative processing. This is needed to use the hash function with HMAC (`BLOCKLEN` is `B` in [2]).

Noise defines additional functions based on the above `HASH()` function:

- **`HMAC-HASH(key, data)`**: Applies `HMAC` from [2] using the `HASH()` function. This function is only called as part of `HKDF()`, below.

- **`HKDF(chaining_key, input_key_material, num_outputs)`**: Takes a `chaining_key` byte sequence of length `HASHLEN`, and an `input_key_material` byte sequence with length either zero bytes, 32 bytes, or `DHLEN` bytes. Returns a pair or triple of byte sequences each of length `HASHLEN`, depending on whether `num_outputs` is two or three:

  - 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))`.
  - If `num_outputs == 2` then returns the pair `(output1, output2)`.
  - Sets `output3 = HMAC-HASH(temp_key, output2 || byte(0x03))`.
  - Returns the triple `(output1, output2, output3)`.

Note that `temp_key`, `output1`, `output2`, and `output3` are all `HASHLEN` bytes in length. Also note that the `HKDF()` function is simply `HKDF` from [3] with the `chaining_key` as HKDF `salt`, and zero-length HKDF `info`.

2. Messages

There are four types of messages, each prefixed by a 2-byte length. * Section 2.1 The handshake initiation message that starts the handshake process for establishing a secure session * Section 2.2 The handshake response to the initiation message * Section 2.3 The optional third and subsequent handshake messages * Section 2.4 Transport message

2.1. First handshake message

In the **First handshake message** client offers server a set of sub-messages, each identified by its own protocol name, ex: `Noise_XX_25519_AESGCM_SHA256`.

packet length (2 bytes)	number of sub-messages N (1 byte)
len (1 byte)	protocol name
length (2 bytes)	Sub message body
len (1 byte)	protocol name
length (2 bytes)	Sub message body
....	
len (1 byte)	protocol name
length (2 bytes)	Sub message body

Each handshake sub-message contains following fields:

- 1 byte length of the following string, indicating the ciphersuite/protocol used, i.e. message type
- String indicating message type
- 2 bytes big-endian length of following sub-message
- **Sub message body**

length (1 byte)	protocol name
length (2 bytes)	Sub message body

2.2. Second handshake message

In the **Second handshake message** server responds to client with the following structure:

- 1 byte sub-message index server responds to
- Handshake message

2.3. Third handshake message

If Noise_XX pattern is used, then there's a need to send the third handshake message from client to server

2.4. Data message

After handshake is complete all following packets must be encrypted using those cipherstates.

3. Prologue

Noise prologue is calculated as follows:

- 1 byte number of message types (N)
- N times:
  - 1 byte message type length
  - Message type (ex. Noise protocol string)

An example of such prologue could be found in Appendix

5. API

We present a set of methods which will help to implement NoiseSocket flow

- **`ReadString(buffer)`** reads 1 byte `len` of the following string from `buffer` and then `len` bytes string itself. Advances read position to `len + 1`
- **`WriteString(string, buffer)`** writes 1 byte `len` of the following string to `buffer` and then string itself.
- **`ReadData(buffer)`** reads 2 byte `len` of the following data from `buffer` and then `len` bytes of data. Advances position to `len + 2`
- **`WriteData(data, buffer)`** writes 2 bytes `len` of the following data to `buffer` and then the data itself.
- **`CalculatePrologue(protocols)`** takes a list of protocol names in the order they will be used in the handshake.

  Variables:

  - **`prologue_buffer`** - a byte buffer to write to

  Algorithm:

  - Writes 1 byte number of protocols `N` to `prologue_buffer`
  - Does `N` times:
    - Takes the next `protocol_name` from `protocols`
    - Calls `WriteString(protocol_name, prologue_buffer)`

  Returns:

  - `prologue_buffer`
- **`ComposeInitiatorHandshakeMessages(s, data, protocols)`** takes client's static key `s`, optional payload `data` and a list of protocols, same that was used for caling `CalculatePrologue` Variables:

  - **`result_buffer`** - buffer, containing the resulting byte sequence

- **message_buffer** - temporary buffer to hold the result of calling WriteMessage on the current handshake_state
  - **handshake_states** - an array of all instances of HandshakeState objects, created during this method

  Algorithm:

  - Calls CalculatePrologue(protocols) to receive prologue
  - Writes the 1 byte number of protocols N to result_buffer
  - Does N times
    - Takes the next protocol from protocols
    - Calls WriteString(protocol_name, result_buffer)
    - Calls GENERATE_KEYPAIR() to generate new e
    - Initializes new HandshakeState instance with DH functions, Cipher functions and Hash functions, described in protocol and also s, e and prologue to receive handshake_state
    - initializes new message_buffer
    - Calls WriteMessage(data, message_buffer) on handshake_state
    - Calls WriteData(message_buffer, result_buffer)
    - Adds handshake_state to handshake_states

  Returns:

  - result_buffer
  - handshake_states

- **ParseFirstMessage(message, s)** receives message, created by calling ComposeInitiatorHandshakeMessages and static keypair s Variables:

  - **protocols** - a list of protocol names, parsed from message
  - **sub_messages** - a list of byte sequences in order they were written to message by calling WriteMessage
  - **handshake_state** - a state that was created when server chose one of the incoming messages
  - **message_index** - an index of the message that server chose
  - **payload** - an optional payload, provided in the first message

  Algorithm:

  - Reads 1 byte number of sub-messages N
  - Does N times:
    - Calls ReadString(message) to receive protocol_name
    - Calls ReadData(message) to receive sub_message
    - Appends protocol_name to protocols
    - Appends sub_message to sub_messages
  - Calls CalculatePrologue(protocols) to receive prologue
  - Chooses a protocol, according to server protocol priority and the corresponding sub_message from sub_messages.
  - Writes index of the chosen protocol to message_index
  - Calls GENERATE_KEYPAIR() to generate new e

- Initializes new `HandshakeState` instance with `DH functions`, `Cipher functions` and `Hash functions`, described in `protocol` and also `s`, `e` and `prologue` to receive `handshake_state`
- Calls `ReadMessage(sub_messages)` on `handshake_state` to receive an optional `payload`

Returns:

- `message_index`
- `handshake_state`
- `payload`

- **`ComposeServerResponseMessage(handshake_state, index, data)`** receives `handshake_state`, `index` and optional `data` to send to client

  Variables:

  - **`result_buffer`** - buffer, containing the resulting byte sequence

  Algorithm:

  - Writes `index` to `result_buffer`
  - Calls WriteMessage(result_buffer, data) on handshake_state

  Returns:

  - `result_buffer`

- **`SetMaxPacketLen(len)`** Sets the global valiable **`max_packet_len`** to `len`. `len` must satisfy the following condition: $127 <$ `len` < 65536 . The default value of **`max_packet_len`** is 65535

- **`GetMaxPacketLen`** Returns the number of bytes of the plaintext that can be placed into the current packet

  Algorithm:

  - If handshake is not complete return **`max_packet_len`**
  - Else return **`max_packet_len - CIPHER_OVERHEAD`** . `CIPHER_OVERHEAD` is the number of bytes added by the chosen symmetric cipher

- **`Write(data)`** writes data to outgoing socket

  Variables:

  - **`buffer`** - temporary buffer

  Algorithm:

  - While length(data) > 0
    - *

  Returns:

  - `result_buffer`

6. Test vectors

Test vectors consist of one initial message, prologue and a set of private keys. Two for initiator (static, ephemeral) and two for responder.

Initial message contains 16 sub-messages each correspond to a specific Noise protocol. The order of protocols can be seen in `Protocols` array.

"Server" chooses which sub-message to answer and this forms a `session`. Each session contains an array of transport messages which consist of raw wire data ("Packet" field), and payload

7. IPR

The NoiseSocket specification (this document) is hereby placed in the public domain.

8. References

[1] P. Rogaway, "Authenticated-encryption with Associated-data," in Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002. http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf

[2] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication." Internet Engineering Task Force; RFC 2104 (Informational); IETF, Feb-1997. http://www.ietf.org/rfc/rfc2104.txt

[3] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. http://www.ietf.org/rfc/rfc5869.txt