The NoiseSocket Protocol

Alexey Ermishkin (scratch@virgilsecurity.com)

Revision 0draft, 2017-07-12

Contents

Abstract

NoiseSocket is an extension of the Noise Protocol Framework that enables quick and seamless secure connections with minimal code size, small keys, modern ciphers and hash functions, and extremely fast speed. It can be used with raw public keys instead of X.509 infrastructure and targets IoT devices, microservices, and back-end applications such as datacenter-to-datacenter communications.

1. Overview

The Noise Protocol Framework describes simple **Noise protocols**. A Noise protocol sends a fixed sequence of handshake messages based on a fixed set of cryptographic choices. In some situations this is too rigid, and the responder needs flexibility to accept or reject the initiator's Noise protocol choice, or make its own choice based on options offered by the initiator.

The **NoiseSocket** framework allows the initiator and responder to negotiate a particular Noise protocol. This is a two-step process:

- The initiator begins executing an initial Noise protocol and sends an initial Noise handshake message. As a preamble to this message, the initiator can send some **negotiation data** which indicates the initial Noise protocol and can advertise support for other Noise protocols.
- The responder can **accept** the initiator's choice of initial Noise protocol, **change** to a different Noise protocol, or **reject** the initiator's message entirely. The responder indicates this choice by sending some negotiation data back to the initiator, or closing the connection.

NoiseSocket doesn't specify the contents of negotiation data, since different applications will encode versions and advertise protocol support in different ways. NoiseSocket just defines a message format to transport this data, and APIs to access it.

NoiseSocket handles two other low-level issues:

- NoiseSocket defines length fields for all messages, so NoiseSocket messages can be used with stream-based protocols like TCP.
- NoiseSocket defines padding fields which are included in every ciphertext, so that applications can pad their messages to avoid revealing plaintext lengths to an eavesdropper.

2. Message Formats

A NoiseSocket protocol begins with a **handshake phase**. During the handshake phase each NoiseSocket message contains a single **handshake message** from some underlying Noise protocol, plus optional negotiation data.

After the handshake completes, NoiseSocket enters the **transport phase** where each NoiseSocket message contains a **transport message** from some underlying Noise protocol.

All transport messages and some handshake messages contain an encrypted Noise **payload**. Each encrypted payload contains a plaintext with a **body** (its actual contents) followed by **padding**.

The following sections describe the format for NoiseSocket handshake and transport messages, and encrypted payloads.

2.1. Handshake messages

All NoiseSocket handshake messages have the same structure:

- negotiation_data_len (2 bytes)
- negotiation_data
- noise_message_len (2 bytes)
- noise_message

The `negotiation_data_len` and `noise_message_len` fields are 2-byte unsigned integers, encoded in big-endian, that store the number of bytes for the following `negotiation_data` and `noise_message` fields.

2.2. Transport messages

All NoiseSocket transport messages have the same structure:

- noise_message_len (2 bytes)
- noise_message

The `noise_message_len` field is a 2-byte unsigned integer, encoded in big-endian, that stores the number of bytes for the following `noise_message` field.

2.3. Encrypted payloads

Some Noise messages will carry an encrypted payload. When this payload is decrypted, the plaintext will have the following structure:

- body_len (2 bytes)
- body
- padding

The `body_len` field is a 2-byte unsigned integer, encoded in big-endian, that stores the number of bytes for the following `body` field. Following the `body` field the remainder of the plaintext will be padding bytes, which may contain arbitrary data and must be ignored by the recipient.

3. Negotiation

The initiator will choose the initial underlying Noise protocol, and will indicate this to the responder using the `negotiation_data` field.

Upon receiving an initial NoiseSocket message, the responder has five options:

- **Silent rejection**: The responder closes the network connection.
- **Explicit rejection**: The responder sends a single NoiseSocket handshake message. The `negotiation_data` field must be non-empty and contain an error message. The `noise_message` field must be empty. After sending this message, the responder closes the network connection.
- **Acceptance**: The responder sends a NoiseSocket handshake message containing the next handshake message in the initial Noise protocol. The `negotiation_data` field must be empty.
- **Change protocol and send fallback message**: The responder sends a NoiseSocket handshake message containing a handshake message from a new Noise protocol, different from the initial Noise protocol. The `negotiation_data` field must be non-empty. The `noise_message` field must be non-empty.
- **Change protocol and send reinitialization request**: The responder requests the initiator to send a NoiseSocket handshake message containing a handshake message from a new Noise protocol, different from the initial Noise protocol. The `negotiation_data` field must be non-empty. The `noise_message` field must be empty.

When the initiator receives the first NoiseSocket response message, and for all later handshake messages received by both parties, the only options are silent rejection, explicit rejection, or acceptance.

The initiator's first `negotiation_data` field must indicate the initial Noise protocol and what other Noise protocols the initiator can support. How this is encoded is up to the application.

If the responder's first `negotiation_data` field is non-empty, the `negotiation_data` must distinguish betwen the "explicit rejection", "fallback", and "reinitialization request" cases. In the first case, the `negotiation_data` must encode an error message. In the latter two cases, the `negotiation_data` must encode the Noise protocol the initiator should fallback to or reinitialize with.

Below are some example negotiation flows:

- It's easy for the responder to change symmetric crypto options using a fallback protocol. For example, if the initial Noise protocol is `Noise_XX_25519_AESGCM_SHA256`, the responder can fallback to `Noise_XX+fallback_25519_ChaChaPoly_BLAKE2s`. This reuses the ephemeral public key from the initiator's initial message.
- If the initiator attempts 0-RTT encryption that the responder fails to decrypt, the responder can use a fallback protocol. For example, if the initial Noise protocol is `Noise_IK_25519_AESGCM_SHA256`, the responder

can fallback to `Noise_XX+fallback_25519_AESGCM_SHA256`. This reuses the ephemeral public key from the initiator's initial message.

- If the responder wants to use a DH function that the initiator supports but did not send an ephemeral public key for, in the initial message, then the responder might need to request reinitialization. For example, if the initial Noise protocol is `Noise_XX_25519_AESGCM_SHA256`, the responder can request reinitialization to `Noise_XX_448_AESGCM_SHA256`, causing the initiator to respond with a NoiseSocket message containing the initial message from the `Noise_XX` pattern with a Curve448 ephemeral public key.

4. Prologue

Noise protocols take a **prologue** input. The prologue is cryptographically authenticated to make sure both parties have the same view of it.

The prologue for the initial Noise protocol is set to the UTF-8 string "NoiseSocketInit" followed by all bytes transmitted prior to the `noise_message_len`. This consists of the following values concatenated together:

- The UTF-8 string "NoiseSocketInit"
- The initial message's `negotiation_data_len`
- The initial message's `negotiation_data`

If the responder changes the Noise protocol, the prologue is set to the UTF-8 string "NoiseSocketReInit" followed by all bytes received and transmitted prior to the `noise_message_len`. This consists of the following values concatenated together:

- The UTF-8 string "NoiseSocketReInit"
- The initial message's `negotiation_data_len`
- The initial message's `negotiation_data`
- The initial message's `noise_message_len`
- The initial message's `noise_message`
- The responding message's `negotiation_data_len`
- The responding message's `negotiation_data`

5. API

The initiator uses the following functions during the handshake phase. These functions are described in the order they would typically be used to send the initial handshake message and process the first response. In particular, the initiator would "peek" at the negotiation data, then decide whether reinitialization is necessary (based on whether the negotiation data indicates a reinitialization request or a fallback message).

`Initialize`:

- INPUT: pattern, dh, cipher, hash
- OUTPUT: session object

`WriteHandshakeMessage`:

- INPUT: negotiation_data, message_body, padded_len
  - `negotiation_data` is zero-length if omitted
  - `message_body` is zero-length if omitted
  - If this message has an encrypted payload and `noise_message_len` would be less than `padded_len`, padding is added to make `noise_message_len` equal `padded_len`.
- OUTPUT: handshake_message

`PeekHandshakeMessage`:

- INPUT: handshake_message
- OUTPUT: negotiation_data

`Reinitialize`:

- INPUT: fallback pattern or reinitialization pattern, dh, cipher, hash
- OUTPUT: session object

`ReadHandshakeMessage`:

- INPUT: handshake_message
- OUTPUT: message_body

The server will use the same functions, except it will first "peek" at the initial message, then call `Initialize` if it is accepting the initial protocol, or `Reinitialize` if it is changing protocols with a fallback message or reinitialization request.

If the responder is sending an explicit rejection or reinitialization request, it will use the following function:

`WriteEmptyHandshakeMessage`:

- INPUT: negotiation_data
- OUTPUT: handshake_message

Following the first exchange of handshake message, the parties will continue calling `ReadHandshakeMessage` and `WriteHandshakeMessage` until the handshake is complete.

After the handshake is complete, both parties will call `WriteMessage` and `ReadMessage` to send transport messages. Every call to `WriteMessage` will produce a NoiseSocket transport message, and every call to `ReadMessage` will decrypt a NoiseSocket transport message and return its body.

`WriteMessage`:

- INPUT: message_body, padded_len
  - `padded_len` is zero (no padding) if omitted
  - If `noise_message_len` would be less than `padded_len`, padding is added to make `noise_message_len` equal `padded_len`.
- OUTPUT: transport_message

`ReadMessage`:

- INPUT: transport_message
- OUTPUT: message_body

6. IPR

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