The NoiseSocket Protocol

Alexey Ermishkin Trevor Perrin

Revision 2draft, 2018-05-01, official/unstable

Contents

Abstract **1**

1. Overview **2**

2. Message Formats **2**
2.1. Handshake messages 3
2.2. Transport messages 3
2.3. Encrypted payloads 3

3. Negotiation **4**

4. Prologue **6**

5. IPR **7**

6. Acknowledgements **7**

7. References **7**

Abstract

NoiseSocket provides an encoding layer for the Noise Protocol Framework.

NoiseSocket can encode Noise messages and associated negotiation data into a form suitable for transmission over reliable, stream-based protocols such as TCP.

1. Overview

The Noise Protocol Framework [1] describes **Noise protocols**. A Noise protocol sends a fixed sequence of handshake messages based on a fixed set of cryptographic choices. In some situations 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 **compound protocols** where the initiator of the compound protocol (Alice) and the responder (Bob) negotiate a particular Noise protocol. This is a two-step process:

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

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

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 ciphertexts 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

Each Noise transport message consists of a single encrypted payload. Each Noise handshake message might contain a a single encrypted payload (or might contain a cleartext payload). When these payloads are 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 decrypted plaintext will be padding bytes, which may contain arbitrary data and must be ignored by the recipient.

3. Negotiation

The initiator of the NoiseSocket protocol (Alice) will choose the initial Noise protocol, and will indicate this to Bob using the `negotiation_data` field. Upon receiving an initial NoiseSocket message, Bob has five options:

- **Accept**: Bob accepts the initial Noise protocol. If this is an interactive protocol, Bob sends a NoiseSocket handshake message containing the next handshake message in the initial Noise protocol. The `negotiation_data` field of this response message must be empty.
- **Switch**: Bob sends a NoiseSocket handshake message containing a handshake message from a new Noise protocol (e.g. a fallback protocol), different from the initial Noise protocol. The `negotiation_data` field must be non-empty. The `noise_message` field must be non-empty.
- **Request Retry**: Bob requests Alice 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.
- **Explicit Reject**: Bob 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, Bob closes the network connection.
- **Silent Reject**: Bob closes the network connection.

The following table indicates the cases where the response `negotiation_data` and `noise_message` fields are non-empty.

	Negotiation	Message
Accept	-	Yes
Switch	Yes	Yes
Request Retry	Yes	-
Explicit Reject	Yes	-

Alice's first `negotiation_data` field must indicate the initial Noise protocol and what other Noise protocols Alice can support for the switch and retry cases. How this is encoded is up to the application.

If Bob's first `negotiation_data` field is empty, then the initial protocol was accepted. If the field is non-empty, it must encode values that distinguish betwen the "switch", "retry", and "reject" cases. In the first two cases, the `negotiation_data` must encode the Noise protocol that Alice should switch to, or the Noise protocol (or Noise protocols) that Bob is requesting retry with. In the last case, the `negotiation_data` must encode an error message.

In the retry case, Alice's second message may optionally contain `negotiation_data` which specifies which protocol Alice is retrying with. Aside from this, all handshake messages after Bob's response should contain empty `negotiation_data`, and any further errors should be handled by silent rejection and closing the connection.

Example negotiation flows:

- It's easy for Bob to change symmetric crypto options by switching to a different protocol. For example, if the initial Noise protocol is `Noise_XX_25519_AESGCM_SHA256`, Bob can switch to `Noise_XX+fallback_25519_ChaChaPoly_BLAKE2s`. This reuses the ephemeral public key from Alice's initial message.
- If Alice attempts 0-RTT encryption that Bob fails to decrypt, Bob can also switch to a fallback protocol. For example, if the initial Noise protocol is `Noise_IK_25519_AESGCM_SHA256`, Bob can fallback to `Noise_XX+fallback_25519_AESGCM_SHA256`. This reuses the ephemeral public key from Alice's initial message.
- If Bob wants to use a DH function that Alice supports but did not send an ephemeral public key for, in the initial message, then Bob might need to request a retry. For example, if the initial Noise protocol is `Noise_XX_25519_AESGCM_SHA256`, Bob can request retry with `Noise_XX_448_AESGCM_SHA256`, causing Alice 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 "NoiseSocketInit1" followed by all bytes transmitted in the NoiseSocket protocol prior to the `noise_message_len` in Alice's initial message. This consists of the following values concatenated together:

- The UTF-8 string "NoiseSocketInit1"
- The initial `negotiation_data_len`
- The initial `negotiation_data`

If Bob switches the Noise protocol, the prologue is set to the UTF-8 string "NoiseSocketInit2" followed by all bytes received and transmitted in the NoiseSocket protocol prior to the `noise_message_len` in Bob's response message. This consists of the following values concatenated together:

- The UTF-8 string "NoiseSocketInit2"
- The initial `negotiation_data_len`
- The initial `negotiation_data`
- The initial `noise_message_len`
- The initial `noise_message`
- The responding `negotiation_data_len`
- The responding `negotiation_data`

If Bob requests retry with a different Noise protocol, the prologue is set to the UTF-8 string "NoiseSocketInit3" followed by all bytes received and transmitted in the NoiseSocket protocol prior to the `noise_message_len` in Alice's retry message. This consists of the following values concatenated together:

- The UTF-8 string "NoiseSocketInit3"
- The initial `negotiation_data_len`
- The initial `negotiation_data`
- The initial `noise_message_len`
- The initial `noise_message`
- The responding `negotiation_data_len`
- The responding `negotiation_data`
- The responding `noise_message_len` (two bytes of zeros)
- The responding `noise_message` (zero-length, shown for completeness)
- The retry `negotiation_data_len`
- The retry `negotiation_data`

Finally, the application using NoiseSocket may append an arbitrary **application prologue** byte sequence following the above data.

5. IPR

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

6. Acknowledgements

Thanks to Gerardo di Giacomo, Nemanja Mijailovic, Rhys Weatherley, Christopher Wood, and Justin Cormack for helpful discussion.

7. References

[1] T. Perrin, "The Noise Protocol Framework." 2017. https://noiseprotocol.org