The NLS Framework

Trevor Perrin (noise@trevp.net)

Revision 1, 2018-03-05, unofficial/unstable

Contents

1. Introduction

The **NLS** (or **NoiseLingoSocket**) framework builds on the Noise Protocol Framework and the NoiseSocket encoding layer to provide a rich set of features for building complex protocols. Protocols based on NLS can negotiate their choice

of Noise protocol, PSK, and transport phase options, and can also exchange **evidence blobs** for their static public keys (e.g. certificates).

This document also defines the **basic profiles** of NLS. Profiles will serve as the basis for implementation and interop in the Noise architecture.

2. Overview

The Noise architecture can be viewed as three layers:

- The core layer is the **Noise Protocol Framework**. This defines rules for constructing named Noise protocols.
- Below the core is an encoding layer, which can encode Noise protocol messages along with negotiation data. The **NoiseSocket** encoding layer can encode messages for delivery on some reliable, stream-based transport (like TCP). NoiseSocket is the only encoding considered in this document.
- Above the core is a negotiation language, which defines the contents of the negotiation data and handshake payloads. This document defines the **NoiseLingo** negotiation language.

Combinining all these layers gives us an expanded framework such as NLS ("NoiseLingoSocket").

Below we define the NoiseLingo language, the NLS framework based on it, and the notion of profiles based on NLS. Finally, we define a set of basic profiles for NLS (NoiseLink, NoiseZeroLink, NoiseTinyLink, NoiseAnonBox, and NoiseAuthBox).

3. The NoiseLingo negotiation language

3.1. NoiseLingo overview

NoiseLingo defines fields which can be used in negotiation data and handshake payloads. It's unlikely that any protocol will use all of these fields. Instead, **profiles** of NoiseLingo will use a subset of these fields.

3.2. NoiseLingo definitions

The NoiseLingo message contents are below, described and encoded using the protobuf version 3 ("proto3") language:

The negotiation data for the initial message and its response are below:

```
message NoiseLingoNegotiationDataRequest {
  string server_name = 1;
  string initial_protocol = 2;
  repeated string switch_protocol = 3;
  repeated string retry_protocol = 4;
  string rejected_protocol = 5;
  bytes psk_id = 6;
}

message NoiseLingoNegotiationDataResponse {
  oneof response {
    string switch_protocol = 3;
    string retry_protocol = 4;
    bool rejected = 5;
  }
}
```

Each handshake payload can use some subset of the following fields:

```
message NoiseLingoHandshakePayload {
  repeated string evidence_request_type = 1;
  repeated string evidence_blob_type = 2;
  repeated bytes evidence_blob = 3;
  bytes psk_id = 4;
  NoiseLingoTransportOptions transport_options = 5;
}

message NoiseLingoTransportOptions {
  uint32 max_send_length = 1;
  uint32 max_recv_length = 2;
  bool continuous_rekey = 3;
  bool short_terminated = 4;
}
```

3.3. NoiseLingo notes

NoiseLingo only uses field numbers in the range 1-6. Field numbers up to 10 in these messages are reserved for future use by NoiseLingo. User-defined extensions should use field numbers 11 and greater.

All NoiseLingo fields have acceptable default values, except `initial_protocol`. Thus, a zero-length protobuf message is valid in many cases.

3.4. NoiseLingo fields

This section explains the usage of each NoiseLingo field:

- `server_name`: This field states the intended recipient for the message, in case the message is being sent to a transport address (e.g. IP address) that might host multiple recipients (e.g. DNS names).
- `initial_protocol`: This field states the name of the initiator's initial Noise protocol. This field must be present.
- `switch_protocol`: In the initial request, this field indicates Noise protocol names which the initiator can support if the responder switches to one of them. It is OK to list the same value multiple times (this might happen when dealing with aliases; see below). In the response, this indicates the responder's decision to switch to the named protocol.
- `retry_protocol`: In the initial request, this field indicates Noise protocol names which the initiator can support if the responder requests it to retry with one of them. It is OK to list the same value multiple times (this might happen when dealing with aliases; see below). In the response, this indicates the responder's decision to ask the initiator to retry with the named protocol.
- `rejected_protocol`: This field lists a protocol name that the server previously returned a `rejected` response for. This allows a "rejected-retry" sequence where the client attempts a new handshake with a different protocol the server is more likely to support, but lists the `rejected_protocol` so that an upgraded server can detect rollback attacks. This is less secure than using a `switch_protocol` or `retry_protocol`, but might be preferred on low-end devices, or if retrofitting negotiation onto a protocol that didn't support it.
- `psk_id`: This field provides a PSK identifier indicating which PSK will be used if the responder chooses a PSK-based handshake. This field may appear once per handshake in either the initial message's negotiation data or handshake payload. Placing the PSK identifier in the handshake payload only makes sense if this payload is encrypted, and the PSK isn't required to decrypt it.
- `rejected`: If true, the responder found something wrong with the initiator's initial mesage, and will close the connection after sending this message. This message is optional for the responder to send (the responder might just close the connection immediatly).
- `evidence_request_type`: This lists identifiers defining various types of evidence that the sender is requesting for the recipient's static public key. Supported evidence types are "x509cert" and "x509chain", but users can define additional evidence types.

- `evidence_blob_type`: This lists identifiers defining various types of evidence that the sender is providing for its static public key. Supported evidence types are "x509cert" and "x509chain", but users can define additional evidence types. These values will typically match the values in `evidence_request_type`, but this isn't strictly required. For example, a single `evidence_request_type` may allow several different `evidence_blob_type` responses. The indexes in this list correspond to indexes for `evidence_blob`.
- `evidence_blob`: This lists evidence blobs for the sender's static public key. The type of each blob is determined by the corresponding `evidence_blob_type` element. Blobs are likely to contain certificates or signatures for the sender's static public key. This list can contain one additional element which does not correspond to an `evidence_blob_type` but is assumed to match an implicit evidence type the parties are preconfigured with. For example, this allows sending a single `evidence_blob` even if no `evidence_blob_type` fields are present.
- `max_send_length`: Indicates the maximum size of transport messages that the sender will send. Zero is interpreted as 65535 (the maximum), so a nonzero value must be less than 65535.
- `max_recv_length`: Indicates the maximum size of transport messages that the recipient can send. Zero is interpreted as 65535 (the maximum), so a nonzero value must be less than 65535.
- `continuous_rekey`: Indicates that the sender will rekey the sending cipherstate after sending each transport message.
- `short_terminated`: Indicates that the sender will transmit maximum-length transport messages except for a "short" final transport message, at which point the stream is terminated. If the sender has no plaintext to send in the final transport message, a transport message with zero-length plaintext will be sent. See NoiseAuthBox and NoiseAnonBox for usage of this feature.

3.5. Protocol aliases

Noise protocol names might be long strings, so alias strings are allowed in `initial_protocol`, `switch_protocol`, `retry_protocol`, and `rejected_protocol`. An alias is any string that is not a Noise protocol name (e.g "1", "2", "aes_protocols", etc.)

An alias may be shorthand for a single protocol name (for `initial_protocol`, or for `switch_protocol` and `retry_protocol` in Response1). An alias might also represent a sequence of protocol names, which are substituted for the alias. If the receiver doesn't recognize an alias they will ignore it, unless that makes

the resulting message invalid (e.g. an unrecognized alias in `initial_protocol`, or in Response1's `switch_protocol` or `retry_protocol`).

An alias should begin with either an ASCII lowercase character or digit, to distinguish it from explicitly-named protocols which will start with capital letters (e.g. "Noise_XX_25519_AESGCM_SHA256").

Aliases should be used with caution, as they can prevent interoperability unless both parties have agreed on the same aliases.

4. The NLS framework

NLS combines NoiseLingo with NoiseSocket. The NoiseSocket application prologue is set to "NLS(revision1)". This will change with every revision of this document to emphasize that this is a work-in-progress, and not to be used except for testing.

The NoiseLingoNegotiationData messages are transmitted inside the NoiseSocket `negotiation_data` fields. If the responder accepts the initiator's `initial_protocol`, the response `negotiation_data` is zero-length (according to NoiseSocket).

If the responder requests the initiator to retry a different protocol, the initiator's retry message will reset to the NoiseLingo Request1 message, and so on.

If the responder requests to switch to a different protocol, the NoiseLingo messages will not reset, unless the profile indicates otherwise.

5. Basic profiles

5.1. NoiseLink

NoiseLink is intended to be the default and "entry-level" use of Noise.

The NoiseLink profile uses the following Noise protocols:

- `Noise_XX_25519_AESGCM_SHA256`
- `Noise_XX_25519_ChaChaPoly_SHA256`
- `Noise_XXfallback_25519_AESGCM_SHA256`
- `Noise_XXfallback_25519_ChaChaPoly_SHA256`

The initiator's `initial_protocol` will choose one of the first two. Thus, an initiator only needs to implement one of these protocols. For future-proofing, a server must support all of them, and must support the `XX` protocols for `retry_protocol`, and the `XXfallback` protocols for `switch_protocol`.

```
message NoiseLinkNegotiationDataRequest1 {
  string server_name = 1;
  string initial_protocol = 2;
  repeated string switch_protocol = 3;
  repeated string retry_protocol = 4;
}

message NoiseLinkHandshakePayloadRequest1 {
  repeated string evidence_request_type = 1;
}

message NoiseLinkNegotiationDataResponse1 {
  oneof response {
    string switch_protocol = 3;
    string retry_protocol = 4;
    bool rejected = 5;
  }
}

message NoiseLinkHandshakePayloadResponse1 {
  repeated string evidence_request_type = 1;
  repeated string evidence_blob_type = 2;
  repeated bytes evidence_blob = 3;
}

message NoiseLinkHandshakePayloadRequest2 {
  repeated string evidence_blob_type = 2;
  repeated bytes evidence_blob = 3;
}
```

5.2. NoiseZeroLink

NoiseZeroLink is a superset of NoiseLink which adds a 0-RTT capability. NoiseZeroLink can deliver encrypted and authenticated data in the first handshake payload by using the server's static public key. If the client has an incorrect view of the server's static key, or doesn't know the server's static key, then NoiseZeroLink uses the same 1-RTT `XX` handshake as NoiseLink.

NoiseZeroLink uses the same messages and Noise protocols as NoiseLink, plus two additional protocols:

- `Noise_IK_25519_AESGCM_SHA256`
- `Noise_IK_25519_ChaChaPoly_SHA256`

The IK protocols can be chosen as an initial protocol, in which case the client offers the corresponding fallback protocol as a switch protocol in case the server has changed its static public key.

5.3. NoiseTinyLink

NoiseTinyLink is a variant of NoiseLink designed for small messages and constrained devices.

NoiseTinyLink defines aliases "1", "2", and "3" as the following Noise protocols:

- "1" = `Noise_XX_25519_AESGCM_SHA256`
- "2" = `Noise_XX_25519_ChaChaPoly_SHA256`
- "3" = `Noise_XX_25519_ChaChaPoly_BLAKE2s`

The initiator indicates their choice using the alias only. Future versions of this specification may assign values to the aliases "4" through "100".

It is assumed the responder supports whichever aliases the initiator is configured to use. To keep things simple, NoiseTinyLink does not support `switch_protocol` or `retry_protocol`. To upgrade to a new protocol, either all responders need to be upgraded before any initiators, or initiators must be upgraded to perform "rejected-retry". Responders are required to send a `rejected` field when rejecting a client's `initial_protocol`, to enable rejected-retry.

NoiseTinyLink assumes the parties are configured with a single evidence blob at most, and does not negotiate its type.

Finally, NoiseTinyLink supports `max_send_length` and `max_recv_length` to negotiate shorter transport messages.

```
message NoiseTinyLinkNegotiationDataRequest1 {
  string initial_protocol = 2;
  string rejected_protocol = 5;
}

message NoiseTinyLinkHandshakePayloadRequest1 {
}

message NoiseTinyLinkNegotiationDataResponse1 {
  bool rejected = 5;
}

message NoiseTinyLinkHandshakePayloadResponse1 {
  string evidence_blob_type = 2;
  bytes evidence_blob = 3;
  NoiseTinyLinkTransportOptions transport_options = 4;
}

message NoiseTinyLinkHandshakePayloadRequest2 {
  string evidence_blob_type = 2;
  bytes evidence_blob = 3;
  NoiseTinyLinkTransportOptions transport_options = 4;
}

message NoiseTinyLinkTransportOptions {
  uint32 max_send_length = 1;
  uint32 max_recv_length = 2;
}
```

5.4. NoiseAnonBox

NoiseAnonBox provides public-key encryption to the recipient's static public key. The following Noise protocols are supported:

- `Noise_N_25519_AESGCM_SHA256`
- `Noise_N_25519_ChaChaPoly_SHA256`

Recipients must support both protocols, but senders can choose which to use.

The stream of transport messages is "short-terminated" so the recipient can recognize the final transport message.

```
message NoiseAnonBoxNegotiationDataRequest1 {
  string initial_protocol = 2;
}

message NoiseAnonBoxHandshakePayloadRequest1 {
  NoiseAnonBoxTransportOptions transport_options = 4;
}

message NoiseAnonBoxTransportOptions {
  uint32 max_send_length = 1;
  bool short_terminated = 4;
}
```

5.5. NoiseAuthBox

NoiseAuthBox provides authenticated public-key encryption to the recipient's static public key. The following Noise protocols are supported:

- `Noise_X_25519_AESGCM_SHA256`
- `Noise_X_25519_ChaChaPoly_SHA256`

Recipients must support both protocols, but senders can choose which to use.

The messages are the same as NoiseBox except the sender can transmit evidence blobs for their public key.

```
message NoiseAuthBoxNegotiationDataRequest1 {
  string initial_protocol = 2;
}

message NoiseAuthBoxHandshakePayloadRequest1 {
  repeated string evidence_blob_type = 2;
  repeated bytes evidence_blob = 3;
  NoiseAuthBoxTransportOptions transport_options = 4;
}

message NoiseAuthBoxTransportOptions {
  uint32 max_send_length = 1;
  bool short_terminated = 4;
}
```

6. IPR

This document is hereby placed in the public domain.

7. Acknowledgements

TBD

8. References