Katzenpost mix network wire protocol

Yawning Angel
David Stainton

Abstract

This document defines the Katzenpost Mix Network Wire Protocol for use in all network communications to, from, and within the Katzenpost Mix Network.

Table of Contents

1.1 Conventions Used in This Document

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in the section called “References” [6].

The “C” style Presentation Language as described in the section called “References” [6] Section 4 is used to represent data structures, except for cryptographic attributes, which are specified as opaque byte vectors.

`x | y` denotes the concatenation of x and y.

1. Introduction

The Katzenpost Mix Network Wire Protocol (KMNWP) is the custom wire protocol for all network communications to, from, and within the Katzenpost Mix Network. This protocol provides mutual authentication, and an additional layer of cryptographic security and forward secrecy.

1.2 Key Encapsulation Mechanism

This protocol uses ANY Key Encapsulation Mechanism. However it’s recommended that most users select a hybrid post quantum KEM such as Xwing. the section called “References” [6]

2. Core Protocol

The protocol is based on Kyber and Trevor Perrin's Noise Protocol Framework the section called "References" [6] along with "Post Quantum Noise" paper PQNOISE. Older previous versions of our transport were based on NOISEHFS.

Our transport protocol begins with a prologue, Noise handshake, followed by a stream of Noise Transport messages in a minimal framing layer, over a TCP/IP connection.

Our Noise protocol is configurable via the KEM selection in the TOML configuration files, here's an example PQ Noise protocol string:

```
Noise_pqXX_Xwing_ChaChaPoly_BLAKE2b
```

The protocol string is a very condensed description of our protocol. We use the pqXX two way Noise pattern which is described as follows:

```
pqXX: -> e <- ekem, s -> skem, s <- skem
```

The next part of the protocol string specifies the KEM, `Xwing` which is a hybrid KEM where the share secret outputs of both X25519 and MLKEM768 are combined.

Finally the `ChaChaPoly_BLAKE2b` parts of the protocol string indicate which stream cipher and hash function we are using.

As a non-standard modification to the Noise protocol, the 65535 byte message length limit is increased to 1300000 bytes. We send very large messages over our Noise protocol because of our using the Sphincs+ signature scheme which has signatures that are about 49k bytes.

It is assumed that all parties using the KMNWP protocol have a fixed long or short lived `Xwing` keypair XWING, the public component of which is known to the other party in advance. How such keys are distributed is beyond the scope of this document.

2.1 Handshake Phase

All sessions start in the Handshake Phase, in which an anonymous authenticated handshake is conducted.

The handshake is a unmodified Noise handshake, with a fixed prologue prefacing the initiator's first Noise handshake message. This prologue is also used as the `prologue` input to the Noise HandshakeState `Initialize()` operation for both the initiator and responder.

The prologue is defined to be the following structure:

```
struct {
uint8_t protocol_version; /* 0x03 */
} Prologue;
```

As all Noise handshake messages are fixed sizes, no additional framing is required for the handshake.

Implementations MUST preserve the Noise handshake hash `[h]` for the purpose of implementing authentication (Section 2.3).

Implementations MUST reject handshake attempts by terminating the session immediately upon any Noise protocol handshake failure and when, as a responder, they receive a Prologue containing an unknown protocol_version value.

Implementations SHOULD impose reasonable timeouts for the handshake process, and SHOULD terminate sessions that are taking too long to handshake.

2.1.1 Handshake Authentication

Mutual authentication is done via exchanging fixed sized payloads as part of the `pqXX` handshake consisting of the following structure:

```
struct {
uint8_t ad_len;
opaque additional_data[ad_len];
opaque padding[255 - ad_len];
uint32_t unix_time;
} AuthenticateMessage;
```

Where:

- `ad_len` - The length of the optional additional data.
- `additional_data` - Optional additional data, such as a username, if any.
- `unix_time` - 0 for the initiator, the approximate number of seconds since 1970-01-01 00:00:00 UTC for the responder.

The initiator MUST send the `AuthenticateMessage` after it has received the peer's response (so after `-> s, se` in Noise parlance).

The contents of the optional `additional_data` field is deliberately left up to the implementation, however it is RECOMMENDED that implementations pad the field to be a consistent length regardless of contents to avoid leaking information about the authenticating identity.

To authenticate the remote peer given an AuthenticateMessage, the receiving peer must validate the `s` component of the Noise handshake (the remote peer's long term public key) with the known value, along with any of the information in the `additional_data` field such as the user name, if any.

If the validation procedure succeeds, the peer is considered authenticated. If the validation procedure fails for any reason, the session MUST be terminated immediately.

Responders MAY add a slight amount (+- 10 seconds) of random noise to the unix_time value to avoid leaking precise load information via packet queueing delay.

2.2 Data Transfer Phase

Upon successfully concluding the handshake the session enters the Data Transfer Phase, where the initiator and responder can exchange KMNWP messages.

A KMNWP message is defined to be the following structure:

```
enum {
no_op(0),
disconnect(1),
send_packet(2),

(255),
} Command;

struct {
Command command;
uint8_t reserved;    /* MUST be '0x00' */
uint32_t msg_length; /* 0 <= msg_length <= 1048554) */
```

```
opaque message[msg_length];
opaque padding[];    /* length is implicit */
} Message;
```

Notes:

- The padding field, if any MUST be padded with `'0x00'` bytes.

All outgoing Message(s) are encrypted and authenticated into a pair of Noise Transport messages, each containing one of the following structures:

```
struct {
uint32_t message_length;
} CiphertextHeader;

struct {
uint32_t message[ciphertext_length-16];
} Ciphertext;
```

Notes:

- The `ciphertext_length` field includes the Noise protocol overhead of 16 bytes, for the Noise Transport message containing the Ciphertext.

All outgoing Message(s) are preceded by a Noise Transport Message containing a `CiphertextHeader`, indicating the size of the Noise Transport Message transporting the Message Ciphertext. After generating both Noise Transport Messages, the sender MUST call the Noise CipherState `Rekey()` operation.

To receive incoming Ciphertext messages, first the Noise Transport Message containing the CiphertextHeader is consumed off the network, authenticated and decrypted, giving the receiver the length of the Noise Transport Message containing the actual message itself. The second Noise Transport Message is consumed off the network, authenticated and decrypted, with the resulting message being returned to the caller for processing. After receiving both Noise Transport Messages, the receiver MUST call the Noise CipherState `Rekey()` operation.

Implementations MUST immediately terminate the session any of the `DecryptWithAd()` operations fails.

Implementations MUST immediately terminate the session if an unknown command is received in a Message, or if the Message is otherwise malformed in any way.

Implementations MAY impose a reasonable idle timeout, and terminate the session if it expires.

3. Predefined Commands

3.1 The no_op Command

The `no_op` command is a command that explicitly is a No Operation, to be used to implement functionality such as keep-alives and or application layer padding.

Implementations MUST NOT send any message payload accompanying this command, and all received command data MUST be discarded without interpretation.

3.2 The disconnect Command

The `disconnect` command is a command that is used to signal explicit session termination. Upon receiving a disconnect command, implementations MUST interpret the command as a signal from the

peer that no additional commands will be sent, and destroy the cryptographic material in the receive CipherState.

While most implementations will likely wish to terminate the session upon receiving this command, any additional behavior is explicitly left up to the implementation and application.

Implementations MUST NOT send any message payload accompanying this command, and MUST not send any further traffic after sending a disconnect command.

3.3 The send_packet Command

The `send_packet` command is the command that is used by the initiator to transmit a Sphinx Packet over the network. The command's message is the Sphinx Packet destined for the responder.

Initiators MUST terminate the session immediately upon reception of a `send_packet` command.

4. Command Padding

We use traffic padding to hide from a passive network observer which command has been sent or received.

Among the set of padded commands we exclude the `Consensus` command because it's contents are a very large payload which is usually many times larger than our Sphinx packets. Therefore we only pad these commands:

GetConsensus NoOp Disconnect SendPacket RetrieveMessage MessageACK Message MessageEmpty

However we split them up into two directions, client to server and server to client because they differ in size due to the difference in size between `SendPacket` and `Message`:

Client to Server commands:

NoOp SendPacket Disconnect RetrieveMessage GetConsensus

Server to client commands:

Message MessageACK MessageEmpty

The `GetConsensus` command is a special case because we only want to pad it when it's sent over the mixnet. We don't want to pad it when sending to the dirauths. Although it would not be so terrible if it's padded when sent to the dirauths… it would just needlessly take up bandwidth without providing any privacy benefits.

5. Anonymity Considerations

Adversaries being able to determine that two parties are communicating via KMNWP is beyond the threat model of this protocol. At a minimum, it is trivial to determine that a KMNWP handshake is being performed, due to the length of each handshake message, and the fixed positions of the various public keys.

6. Security Considerations

It is imperative that implementations use ephemeral keys for every handshake as the security properties of the Kyber KEM are totally lost if keys are ever reused.

Kyber was chosen as the KEM algorithm due to it's conservative parameterization, simplicity of implementation, and high performance in software. It is hoped that the addition of a quantum resistant

algorithm will provide forward secrecy even in the event that large scale quantum computers are applied to historical intercepts.

Acknowledgments

I would like to thank Trevor Perrin for providing feedback during the design of this protocol, and answering questions regarding Noise.

References

XWING

Manuel Barbosa, Deirdre Connolly, João Diogo Duarte, Aaron Kaiser, Peter Schwabe, Karoline Varner, Bas Westerbaan, “X-Wing: The Hybrid KEM You’ve Been Looking For”, https://eprint.iacr.org/2024/039.pdf.

NOISE

Perrin, T., “The Noise Protocol Framework”, May 2017, https://noiseprotocol.org/noise.pdf.

NOISEHFS

Weatherley, R., “Noise Extension: Hybrid Forward Secrecy”, https://github.com/noiseprotocol/noise_hfs_spec/blob/master/output/noise_hfs.pdf.

PQNOISE

Yawning Angel, Benjamin Dowling, Andreas Hülsing, Peter Schwabe and Florian Weber, “Post Quantum Noise”, September 2023, https://eprint.iacr.org/2022/539.pdf.

RFC2119

Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, http://www.rfc-editor.org/info/rfc2119.

RFC5246

Dierks, T. and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246, DOI 10.17487/RFC5246, August 2008, http://www.rfc-editor.org/info/rfc5246.

RFC7748

Langley, A., Hamburg, M., and S. Turner, “Elliptic Curves for Security”, RFC 7748, January 2016, https://www.rfc-editor.org/info/rfc7748.