Client connector design specification

David Stainton

Abstract

This document describes the design of the Katzenpost mix network client connector. In particular we discuss its multiplexing and privilege separation design elements as well as the protocol used by the connector library, which is incorporated into any application client that supports the connector.

Table of Contents

1. Overview

An appliance for Katzenpost client connections has several responsibilities at a minimum:

- compose Sphinx packets
- decrypt SURB replies
- send and receive Noise protocol messages
- keep up to date with the latest PKI document

The client connector is essentially a long running daemon process that listens on an abstract Unix domain socket for incoming connector library connections. Many client applications can use the same connector daemon. Those connections are in a sense being multiplexed into the daemon's single connection to the mix network.

Therefore applications will be integrated with Katzenpost using the connector library which gives them the capability to talk with the connector daemon and in that way interact with the mix network. The library itself does not do any mixnet-related cryptography since that is already handled by the connector daemon. In particular, the PKI document is stripped by the daemon before it's passed on to clients using the connector library. Likewise, the library doesn't decrypt SURB replies or compose Sphinx packets, with Noise, Sphinx, and PKI related cryptography being handled by the daemon.

2. Connector library and daemon protocol

Note that the connector daemon protocol uses abstract Unix domain sockets in datagram packet mode. The socket is of type SOCK_SEQPACKET which is defined as:

- **SOCK_SEQPACKET** (since Linux 2.6.4), is a connection-oriented socket that preserves message boundaries and delivers messages in the order that they were sent.

In golang this is referred to by the “unixpacket” network string.

2.1 Client socket naming convention

Clients using the connector library MUST randomize their abstract Unix domain socket names. Otherwise, the static name will prevent multiplexing because the kernel requires that the connection be between uniquely named socket pairs. The Katzenpost reference implementation of the connector library selects a socket name with four random hex digits appended to the end of the name like so:

```
@katzenpost_golang_thin_client_DEADBEEF
```

2.2 Daemon socket naming convention

The connector daemon listens on an abstract Unix domain socket with the following name:

```
@katzenpost
```

2.3 Protocol messages

The protocol includes two message types, `Response` and `Request`. We do not use prefix-length encoding for either because the socket type preserves message boundaries for us, permitting us to use pure CBOR encoding.

The daemon sends the `Response` message, which is defined as follows in golang as a struct containing an app ID and one of four possible events.

```
type Response struct {
// AppID must be a unique identity for the client application
// that is receiving this Response.
AppID *[AppIDLength]byte `cbor:app_id`

ConnectionStatusEvent *ConnectionStatusEvent `cbor:connection_status_event`

NewPKIDocumentEvent *NewPKIDocumentEvent `cbor:new_pki_document_event`

MessageSentEvent *MessageSentEvent `cbor:message_sent_event`

MessageReplyEvent *MessageReplyEvent `cbor:message_reply_event`
}

type ConnectionStatusEvent struct {
IsConnected bool `cbor:is_connected`
Err error `cbor:err`
}

type NewPKIDocumentEvent struct {
Payload []byte `cbor:payload`
}

type MessageReplyEvent struct {
MessageID *[MessageIDLength]byte `cbor:message_id`
SURBID *[sConstants.SURBIDLength]byte `cbor:surbid`
Payload []byte `cbor:payload`
Err error `cbor:err`
}

type MessageSentEvent struct {
MessageID *[MessageIDLength]byte `cbor:message_id`
SURBID *[sConstants.SURBIDLength]byte `cbor:surbid`
SentAt time.Time `cbor:sent_at`
ReplyETA time.Duration `cbor:reply_eta`
Err error `cbor:err`
}
```

The client sends the `Request` message, which is defined in golang as follows.

```
type Request struct {
// ID is the unique identifier with respect to the Payload.
// This is only used by the ARQ.
ID *[MessageIDLength]byte `cbor:id`

// WithSURB indicates if the message should be sent with a SURB
// in the Sphinx payload.
WithSURB bool `cbor:with_surb`

// SURBID must be a unique identity for each request.
// This field should be nil if WithSURB is false.
SURBID *[sConstants.SURBIDLength]byte `cbor:surbid`

// AppID must be a unique identity for the client application
// that is sending this Request.
AppID *[AppIDLength]byte `cbor:app_id`

// DestinationIdHash is 32 byte hash of the destination Provider's
// identity public key.
DestinationIdHash *[32]byte `cbor:destination_id_hash`

// RecipientQueueID is the queue identity which will receive the message.
RecipientQueueID []byte `cbor:recipient_queue_id`

// Payload is the actual Sphinx packet.
Payload []byte `cbor:payload`

// IsSendOp is set to true if the intent is to send a message through
// the mix network.
IsSendOp bool `cbor:is_send_op`

// IsARQSendOp is set to true if the intent is to send a message through
// the mix network using the naive ARQ error correction scheme.
IsARQSendOp bool `cbor:is_arq_send_op`

// IsEchoOp is set to true if the intent is to merely test that the unix
// socket listener is working properly; the Response payload will be
// contain the Request payload.
IsEchoOp bool `cbor:is_echo_op`

// IsLoopDecoy is set to true to indicate that this message shall
// be a loop decoy message.
IsLoopDecoy bool `cbor:is_loop_decoy`

// IsDropDecoy is set to true to indicate that this message shall
// be a drop decoy message.
IsDropDecoy bool `cbor:is_drop_decoy`
}
```

2.4 Protocol description

Upon connecting to the daemon socket the client must wait for two messages. The first message received must have its `is_status` field set to **true** and its `is_connected` field indicating whether or not the daemon has a mixnet PQ Noise protocol connection to an entry node.

The client then awaits the second message, which contains the PKI document in its `payload` field. This marks the end of the initial connection sequence. Note that this PKI document is stripped of all cryptographic signatures.

In the next protocol phase, the client may send `Request` messages to the daemon in order to cause the daemon to encapsulate the given payload in a Sphinx packet and send it to the gateway node. Likewise the daemon my send the client `Response` messages at any time during this protocol phase. These `Response` messages may indicate a connection status change, a new PKI document, or a message-sent or reply event.

2.5 Request message fields

Several `Request` fields need further discussion.

Each `Request` message sent by a connector needs to have the `app_id` field set to an ID that is unique among the applications using the connector. The `app_id` is used by the daemon to route `Response` messages to the correct connector socket.

The remaining fields must conform to the following:

- `with_surb` is set to **true** if a Sphinx packet with a SURB in its payload should be sent.
- `surbid` is used to uniquely identify the response to a message sent with the `with_surb` field set to **true**. It should NOT be set if using the built-in automatic repeat request (ARQ) for reliability and optional retransmission.
- `is_send_op` must be set to true.
- `payload` must be set to the message payload being sent.
- `destination_id_hash` is the 32-byte hash of the destination entry node's public identity key.
- `recipient_queue_id` is the destination queue identity. This is the destination that the message will be delivered to.

If a one-way message should be sent with no SURB, then `with_surb` should be set to **false** and `surbid` may be **nil**. If, however, the thin client wishes to send a reliable message using the daemon's ARQ, then the following fields must be set:

- `id` specifies the message ID that uniquely identifies this message and its eventual reply.
- `with_surb` is set to **true**.
- `is_arq_send_op` is set to **true**.
- `payload` is set to the message payload, as usual.
- `destination_id_hash` is the 32-byte hash of the destination service node's public identity key.
- `recipient_queue_id` is the destination queue ID. This is the destination that the message will be delivered to.

2.6 Response message fields

The connector library always begins a connection to the daemon with two messages, a connection status followed by a PKI document.

After this connection sequence phase, the daemon may send the library a connection status or PKI document update at any time.

The connector library can receive four possible events inside of `Response` messages:

1. **connection status event**

   - `is_connected` indicates whether the client is connected or not.
   - `err` may contain an error indicating why connection status changed.
2. **new PKI document event**
   - `payload` is the CBOR-serialized PKI document, stripped of cryptographic signatures.
3. **message sent event**
   - `message_id` is a unique message ID.
   - `surb_id` is the SURB ID.
   - `sent_at` is the time when the message was sent.
   - `replay_eta` is the time when we expect a reply.
   - `err` is the optional error we received when attempting to send a message.
4. **message reply event**
   - `message_id` is a unique message ID.
   - `surb_id` is a the SURB ID.
   - `payload` is the replay payload .
   - `err` is the error, if any.