Client2 design specification

David Stainton

Abstract

This document describes the design of the new Katzenpost mix network client known as client2. In particular we discuss its multiplexing and privilege separation design elements as well as the protocol used by the thin client library.

Table of Contents

1. Overview

A Katzenpost mixnet client 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

Client2 is essentially a long running daemon process that listens on an abstract unix domain socket for incoming thin client library connections. Many client applications can use the same client2 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 thin client library which gives them the capability to talk with the client2 daemon and in that way interact with the mix network. The reason we call it a thin client library is because it does not do any mixnet related cryptography since that is already handled by the client2 daemon. In particular, the PKI document is stripped by the daemon before it's passed on to the thin clients. Likewise, thin clients don't decrypt SURB replies or compose Sphinx packets, instead all the that Noise, Sphinx and PKI related cryptography is handled by the daemon.

2. Thin client and daemon protocol

Note that the thin client 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

Thin clients MUST randomize their abstract unix domain socket name otherwise the static name will prevent multiplexing because the kernel requires that the connection be between uniquely nameed socket pairs. The Katzenpost reference implementation of the thin client 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 client2 daemon listens on an abstract unix domain socket with the following name:

```
@katzenpost
```

2.3 Protocol messages

Note that there are two protocol message types and they are always CBOR encoded. We do not make use of any prefix length encoding because the socket type preserves message boundaries for us. Therefore we simply send over pure CBOR encoded messages.

The daemon sends the `Response` message which is defined 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:

```
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 it's `is_status` field set to true. If so then it's `is_connected` field indicates whether or not the daemon has a mixnet PQ Noise protocol connection to an entry node.

Next the client awaits the second message which contains the PKI document in it's `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 entry node. Likewise the daemon my send the client `Response` messages at any time during this protocol phase. These `Response` messages may indicated a connection status change, a new PKI document or a message sent or reply event.

2.5 Request message fields

There are several `Request` fields that we need to discuss.

Firstly, each `Request` message sent by a thin client needs to have the `app_id` field set to an ID that is unique among the applications using thin clients. The `app_id` is used by the daemon to route `Response` messages to the correct thin client socket.

The rest of the fields we are concerned with are the following:

- `with_surb` is set to true if a Sphinx packet with a SURB in it's payload should be sent.
- `surbid` is used to uniquely identify the reponse to a message sent with the `with_surb` field set to true. It should NOT be set if using the built-in ARQ for reliability and optional retransmissions.
- `is_send_op` must be set to true.
- `payload` must be set to the message payload being sent.
- `destination_id_hash` is 32 byte hash of the destination entry node's identity public key.
- `recipient_queue_id` is the destination queue identity. This is the destination 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` the message id which uniquely identifies this message and it's eventual reply.
- `with_surb` set to true
- `is_arq_send_op` set to true
- `payload` set to the message payload, as usual.
- `destination_id_hash` set to the destination service node's identity public key 32 byte hash.
- `recipient_queue_id` is the destination queue identity. This is the destination the message will be delivered to.

2.6 Response message fields

A thin client connection always begins with the daemon sendings the client two messages, a connection status followed by a PKI document.

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

Thin clients recieve four possible events inside of `Response` messages:

1. connection status event
   - `is_connected` indicated 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 serialied PKI document, stripped of all the cryptographic signatures.
3. message sent event
   - `message_id` is a unique message ID
   - `surb_id` is the SURB ID
   - `sent_at` is the time the message was sent
   - `replay_eta` is the time we expect a reply
   - `err` is the optional error we received when attempting to send
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.