(n+1)sec high level API draft

Ruud Koolen

June 30, 2016

1 Introduction

The (n+1)sec library defines and implements a system of secure synchronous group communication, allowing a group of people to perform text chats while enjoying communication security guarantees similar to those granted by the OTR or *off-the-record* system. Similarly to OTR, (n+1)sec ensures that chat contents remain secret to anyone not part of the conversation; provides cryptographic authentication of the identities of the conversation participants; and provides deniability of the entire conversation exchange, making it impossible for a conversation participant to show the conversation transcript to outsiders in a verifiable way. In addition, (n+1)sec can verify that different participants in a conversation are in agreement about what has been said in this conversation, as well as ensuring that only users authorized by the existing participants of a conversation can join that chat; both of these topics are security aspects that do not play a role in a two-party chat, as implemented by OTR.

The (n+1)sec library implements its secure chat systems as a cryptographic layer on top of existing text chat technologies. For example, an (n+1)sec conversation can be implemented using an XMPP Multi User Chat (*xmpp-muc*) room as the underlying chat infrastructure, in which case the (n+1)sec conversation consists of an xmpp-muc room in which the chat participants exchange encrypted messages carrying (n+1)sec content. Similar to OTR, the (n+1)sec library is designed to be usable with any underlying text chat room system (which we call the *carrier chat system*), as long as certain requirements are met.

In this document, we describe the requirements the carrier chat system needs to satisfy in order to be able to support (n+1)sec; the way the (n+1)sec library interacts with the carrier chat system; the security properties guaranteed by the (n+1)sec system; and the interaction between the (n+1)sec library and the chat client front-end. Together, this defines the high level API of the (n+1)sec

library. In Section 2, we describe the model and properties of the carrier chat system, seen from the point of view of the (n+1)sec library; Section 3 describes the properties and behavior of (n+1)sec conversations from the point of view of a client front-end. The details of how to *join* an existing (n+1)sec conversation, and its relation to carrier chat rooms, is fraught with complication and still a work in progress; our current working approach, and its limitations, are discussed in Section 4.

2 Carrier chat model

Conversations secured by (n+1)sec use text chat systems as a carrier group communication infrastructure. These systems consist of a *room* that users can enter or leave; while in the room, they can exchange text messages, which are sent to all users in the room. Commonly used systems of this description include XMPP Multi User Chat rooms, and IRC channels. (n+1)sec uses chat *rooms* as the basic infrastructure component; it does not concern itself with the way these rooms are part of conference servers (xmpp-muc), networks (IRC), or other forms of higher levels of aggregation.

In order to function, the (n+1)sec protocol requires that the carrier chat system satisfies certain properties. If these properties are not met, this does not compromise the security of the (n+1)sec conversation; rather, the (n+1)sec library will notice the anomaly, and will not be able to establish or continue a conversation. In particular, (n+1)sec requires the carrier chat system to satisfy the following requirements:

1. The carrier chat room contains a consistent, well-defined set of users –the *members* of the room– that are considered part of the room. All users in the room can see this member set, and all members receive all messages sent to the room. Users get notified when the set of room members changes, i.e. when new members join the room, or existing members leave the room.

2. Room members have a recognizable unique identity that stays stable for at least the duration that the member remains part of the room. Changeable unique nicknames, such as used by the IRC system, are sufficient as long as users receive a notification of the nickname changes.

3. Chat events occurring in the room –which include at least chat messages, joining members, leaving members, and nickname changes (if applicable)– have a strict and consistent order enforced by the carrier chat system. Different members of the room receive the same chat events in the same order. This includes chat events originated by the receiving member; for example, if a member sends a chat message, it should be aware of the

timing at which it is received by the room, relative to other chat events in the room.

It is a noteworthy observation that not all commonly used chat systems satisfy all these requirements. In particular, the popular IRC system does not satisfy requirement 3; users connected to different servers of the same IRC network may receive near-simultaneous chat events in different orders, and the sender of a message generally cannot tell the time it is received relative to surrounding events. This means that (n+1)sec as currently described cannot use IRC as a carrier chat system. In order to broaden the applicability of the (n+1)sec system, extensions to the (n+1)sec protocol in order to lift requirement 3 are explicitly considered as a possible future improvement step.

To perform secure communications, (n+1)sec needs to perform actions in the role of a member of a carrier chat room. That is to say, it needs a resource consisting of a user in the carrier chat system that is a member of a chat room, and be able to perform the following operations:

- send a chat message as the (n+1)sec user to the room;
- have the (n+1)sec user leave the room.

Moreover, (n+1)sec needs to be notified of the following events happening in the chatroom:

- a new member joins the room;
- a member leaves the room;
- a member sends a message to the room. This includes messages sent by the (n+1)sec user.

Finally, (n+1)sec needs to know the following piece of information regarding the (n+1)sec user:

- the stable unique identifier of the (n+1)sec user.

A secure chat client using the (n+1)sec library needs to implement these operations, and give (n+1)sec access to the carrier chat member; when the chat events described above happen, it needs to notify (n+1)sec of this fact.

3 (n+1)sec chat model

Secure communications implemented by (n+1)sec are organized into conversations that behave similar to chat rooms; to avoid ambiguity, we shall refer to these conversations as *(n+1)sec channels*. An (n+1)sec channel is a construction similar to a chat room in most chat systems: it consists of a set of chatting users (called *participants*, again for unambiguity reasons) that can send messages to each other, and like many chat systems these chat messages are delivered only to the chat participants. But whereas most chat systems rely on server-side access control to implement security measures such as communications privacy and authentication of participants, an (n+1)sec channel guarantees these properties relying only on end-to-end cryptography.

Using this cryptography, (n+1)sec channels have the following security properties:

- The contents of messages sent to a channel are secret. The only entities able to access the message contents are the participants of the channel.
- All channel participants can verify the public-key-based cryptographic identity of all other participants.
- Verification of identities of participants is deniable: anyone can forge a transcript containing arbitrary contents and participants, which makes the contents of saved transcripts of little use as evidence of what happened during the chat.
- New participants cannot join a channel without approval of all existing participants. Participants know the exact set of participants in the channel at all times.
- Participants can verify that they are all in agreement about the events happening in a channel, a procedure we call *transcript consistency verification*. In particular, it is not possible for different participants to receive different versions of messages, or otherwise have a different view of the chat transcript, without triggering a verification alert.
- The above properties satisfy *forward secrecy*: compromise of long-term private keys defining participant identities does not compromise the security properties of historic chats, even with access to a full transcript.
- The short-term keys used to ensure forward secrecy are only used for a short amount of time, after which they are refreshed. This ensures that the compromise of a short-term key compromises only a small fragment of long-running conversations.

In order to implement these properties, the process of a participant joining or leaving an (n+1)sec channel is a relatively complex one. Whereas most chat systems model the joining and leaving of members as atomic events, both processes when applied to an (n+1)sec channel are multi-step processes. To model this, participants of (n+1)sec channels can be in one of four distinct states:

- *authenticating*: A participant is in the *authenticating* state when they have announced their intention to join the channel, but their identity has not yet been confirmed or accepted by all *active* participants of the channel. An authenticating participant has not been established to be the person they claim they are. Authenticating participants can neither send messages to, nor decrypt messages sent to the channel. When authentication completes, the participant moves to the *joining* state.
- *joining*: A participant is in the *joining* state when they have been authenticated and approved by all *active* members, but the key exchange process that would enable the participant to decrypt channel messages is still ongoing. Joining participants cannot decrypt messages sent to the channel. They can send messages to the channel, but no guarantees of transcript consistency apply to these messages. Once the key exchange process finishes, a joining participant becomes *active*.
- *active*: A participant is in the *active* state when they have completely finished joining the channel. Active members can both send messages to the channel and decrypt messages sent to the channel, and transcript consistency is active. When an active participant wants to leave the channel, they enter the *leaving* state.
- *leaving*: A participant is in the *leaving* state when they have announced their intention to leave the channel, but they have yet to verify the transcript consistency of recent chat. A leaving participant can decrypt an unpredictable subset of messages sent to the chat; they cannot send messages. When the transcript consistency status of all chat before the announcement to leave the channel has become clear, the leaving procedure is completed and the leaving participant is removed from the channel.

When joining an (n+1)sec channel, participants start in the *authenticating* state, and barring complications eventually become *active* participants. Participants do not necessarily pass through the *leaving* state before leaving a channel; participants can leave without warning at any stage if they are not interested in verifying transcript consistency, and this is also what will generally happen in case of connectivity problems.

Internally, (n+1)sec channels are constructed out of a multitude of cryptographic constructions which we call (n+1)sec *sessions*. A session is an agreement

between the active participants of a channel to use a particular shared key for encrypting chat messages. Unlike channels, sessions cannot be joined or left; when participants join or leave a channel, the channel spawns a new session to accomodate the changed set of participants, and the old session is eventually replaced. New sessions are also created periodically in order to refresh short-term keys, as part of the effort to limit the damage done by the compromise of a short-term private key.

As the interface to (n+1)sec channels, the (n+1)sec library provides an object representing a channel of which the user is a participant. This *channel* object contains at least the following properties:

- A set of participants, each containing a carrier-chat identifier, a public key, and a participant state;
- The participant representing the user;
- The private key of the user;
- Several settings configuring the timeouts used in several places of the (n+1)sec protocol.

The chat client can perform the following two operations on the channel object:

- Send a message to the channel, assuming the user is in the *joining* or *active* state;
- Leave the channel, waiting for transcript consistency to complete;
- Leave the channel immediately.

The channel object notifies the chat client of the following events:

- A participant sends a message to the channel. This includes messages sent by the participant representing the user.
- A participant joins the channel.
- A participant leaves the channel.
- A participant changes its join state.
- An authenticating participant needs to be authenticated and authorized to join the chat. When this event happens, the chat client needs to make a decision whether or not to grant access to the authenticating participant; it needs to notify the channel object of this decision asynchronously within a certain time limit.

- A message sent earlier to the channel has been inspected by the transcript consistency verification system. This event either tells the chat client that the past message is properly consistent with the channel view of the rest of the participants in the channel; or it tells the chat client that the status of this message is disputed by the participants, indicating an attack.

A secure chat client can implement (n+1)sec by allocating channel objects as the user tries to join channels, and responding to the event notifications as desired. Different configurations of the timeout settings can configure the (n+1)sec protocol for different levels of network reliability; different implementations of the authentication callback can be used to define different authentication models.

4 Joining and constructing (n+1)sec channels

The previous section describes how (n+1)sec channels behave once one has started the procedure of joining one. What this section does not describe is how one can join a channel in the first place; or, alternatively, how to create an empty one from scratch.

Ideally, the concept of (n+1)sec channels is nearly identified with the concept of carrier chat rooms. Each carrier chat room may contain an (n+1)sec channel; if one wants to hold a secure chat inside a particular carrier chat room, one queries whether one exists; sends a join request if it does; or starts a one-man channel if it does not. This yields a simple interaction model that is easy to understand by end users.

Unfortunately, it is not clear whether this protocol is feasible. Several different sets of members of a carrier chat room might try to hold separate (n+1)sec channels inside the room; while it would be rare for this situation to arise spontaneously, it can certainly be crafted deliberately as a denial of service attack. When this happens, the simple interaction model described above breaks down.

This problem can be avoided in fairly simple ways by adjusting the channel-joining protocol to allow multiple (n+1)sec channels in a given carrier chat room, giving the user the choice of selecting what channel to join. However, we fear the resulting design is both cumbersome to work with from a chat client perspective, and be problematically complex for the end user. What is more, this design is still vulnerable to certain denial of service attacks of a more subtle form; for example, one could try constructing an overwhelming amount of channels inside a room, making it very inconvenient for the user to join the correct channel. For this reason, this part of the high-level (n+1)sec API is still a work in progress.