Eli Meckler
Sophie Mori

Proposal Design Changes

Our final implementation deviates from our proposed design in two ways. Firstly, we implemented an RSA-encryption symmetric key transfer (RSAT) instead of using two-party Diffie-Hellman key exchange (DH). Secondly, we attach counters implicitly to message signatures. While we are aware that it is not ideal to deviate from the specification, we believe these changes to be well-advised for the reasons outlined below.

RSA Key Transfer

Problems with Diffie-Hellman Key Establishment

The DH protocol introduces additional complications for relatively little payoff. For every individual in the chat (B) besides the creator (A), there must be at minimum three messages sent: an initial message from A to B with A's component of the key, a response from B to A containing B's component of the key, and A's acknowledgement and inclusion of the chat secret key. Due to the implementation of the chat application, users must be in the chatroom at the same time. Otherwise, B may be stuck without a way to decrypt chat messages until A returns. If A and B are not in the chatroom concurrently, the protocol fails to be effective.

Since more initial messages are communicated, we must also implement additional states for the chat client and message types for the messages themselves. This becomes unwieldy for the client, as they must not only keep track of their own state but filter incoming messages in a more complex manner.

Using RSA key transfer

Switching to RSA key transfer was quite easy, as all users already had RSA key pairs in order to sign messages, with all public keys being accessible to all users. Now, immediately after founding and joining the chat[1], creator A simply sends the following message to the chat for all other users B:

$$\text{“Key”} \mid A \mid B \mid \text{RSA}_{\text{B-public}}(K_e) \mid \text{RSA}_{\text{A-private}}(\text{RSA}_{\text{B-public}}(K_e) \mid \text{ctr})$$

The first part is just a message type indicator, specifying that the message is meant to establish the chat room key K_e (and is not a normal chat message). A encrypts K_e with user B's public RSA key, so only B can recover the key. Additionally, A signs the encrypted key with their private RSA key, to be verified by B. The counter in the signature is discussed in the next section.

User B knows that they should be able to receive the key if their name is in the recipient field and easily filters out all other key transfer messages. B verifies the signature and acquires the key, allowing them to see all messages with the “Message” flag (normal chat messages).

[1] We also changed the chat room implementation so A immediately joins the chat after creating it, insuring that the key transfer messages are the first messages in the chat log. See the README for more on this.

Pros and Cons of RSAT

The downsides of RSAT is that we lose the perfect forward secrecy promised by the DH protocol. If an attacker M compromises B's computer and gains access to B's secret key, she can decrypt the key transfer message. Then, having access to K_e, M can read all chat messages. In DH, once the protocol has terminated and the keys are lost, there is no way to recover the chat room key from the existing messages stored by the server, even if a client is compromised. Note that DH would not actually ensure chat key freshness; only the DH channel key is ensured to be fresh. This downside, however, is not realistically a large problem. We generally assume that a client will not lose secrecy of their private key when discussing RSA-based encryption.

The upsides to RSAT over DH are three-fold. No clients miss out on part of the conversation due to non-overlapping group membership. When processing the chat log, the first message addressed to them will contain the chat key (which predates the "chat" messages). After they process the key transfer, they will be able to understand all following messages for the duration of the chat. Secondly, the chat client code and message format are simplified. The client does not have to track multiple states during the key establishment, and messages do not need to specify which step in the protocol they refer to. Thirdly, if a user exits their chat client, reopens it, and rejoins the chat, they will not need to re-establish key protocol, as it will be contained in the beginning of the chat log in a form that the user can decrypt.

Implicit Counters

Using implicit message counter management does not contradict our proposal, but we did not specify exactly the implementation. All users maintain a list of counters L representing how many messages have been received from every other user as well as a counter C_{sent} of how many messages they have sent. When A sends a message to B, they increment $C_{A\text{-}sent}$ by one and append it to the message only for signature production. So messages take the general form

$$[\text{header info}] \mid msg \mid RSA_{A\text{-}private}(msg \mid C_{A\text{-}sent})$$

User B, upon seeing a message from A, verifies the signature by checking it against m | L(A) + 1. If the signature is verified, they increment L[A] by one.

Using implicit counters simplifies message parsing and maintains the same level of replay protection as explicit counter management.