Key Agreement in Flocking

Forward secrecy is the principle that compromise of keying material used in the past cannot be leveraged to break messages sent later. In particular, private or shared keys should not be sent over a channel (hashes, public keys and nonces are the remaining candidates for transmission). In addition, parties should not communicate indefinitely using the same session keys. These notes document the forward-secrecy preserving key-agreement protocol used in flocking.

Messages between flock nodes are encrypted with 256-bit AES-GCM (*Galois counter mode*). "Galois" refers to a message authentication code (MAC) that is computed (and subsequently checked) in parallel with encryption (or decryption). The MAC may include data that is not part of the encrypted message. Every encrypted message begins with its own random 12-byte nonce that initializes the GCM internal state machine (so multiple encryptions of the same plaintext will be different).

Flock nodes are initialized with a *shared secret key* (sec0) that is the effective flock identity, i.e., the ability to sign messages using sec0 is necessary and sufficient to join the emergent flock. Nodes communicate with each other using limited-lifetime *session keys* that are established pairwise in signed Diffie-Hellman (DH) exchanges. Session-key management takes place below the flocking layer, but a leader may cause fresh session keys to be established by sending a new Info.EpochID (a nonce) to flock members.

When a node is initialized, it generates a random Curve25519 key pair (secrKey, pubKey) that it will use in DH exchanges to establish session keys with other nodes. A node stores data about its state of communication with another node in a struct Session, viz., the peer's public key, the current session key (sessKey), the previous session key (prevKey), a new (proposed) session key (respKey) and nonce material of any active DH exchange. All these data slots are empty when a node is initialized.

The flock layer calls methods Send() and Recv() to transfer Info structs. Send() will transmit a message using an existing session key if possible; otherwise it will start a DH exchange. Recv() will decrypt a message using an existing session key if possible; otherwise it will discard the message.

A node must distinguish ordinary flock Info messages from DH messages and it must be able to identify the peer Session before decryption is attempted. To this end all messages have the form

$$\text{MsgType} \,||\, \text{SrcAddr} \,||\, \text{Payload}$$

Only the payload is encrypted but the MAC covers (MsgType || SrcAddr || DstAddr) in addition to the payload (here MsgType is a small integer, SrcAddr is the address of the sending node and DstAddr is the address of the intended receiver node).

The detailed Send() behavior is: find the Session belonging to the destination address contained in the Info to be sent; if there is no Session, create an empty one for the new peer. If there is a sessKey, choose it; otherwise choose the respKey if any; now if a key has been chosen, send the message

$$\text{SessData} \,||\, \text{SrcAddr} \,||\, E_{\text{pKey}}[\text{Info}]$$

where $E_{\text{pKey}}[]$ represents encryption using the chosen key. If there was no available key, start a DH exchange, as follows. Generate a random nonce key (offerKey). Send the message

$$\text{PubkeyOffer} \,||\, \text{SrcAddr} \,||\, E_{\text{sec0}}[\text{pubKey} \,||\, E_{\text{offerKey}}[\text{Info}]]$$

(Here pubKey is the Curve255519 public key of the sending node.) In the Session, save offerKey and save the first 12 bytes of $E_{offerKey}$[Info] as offerNonce. Here E_{sec0}[] (encryption with the flock shared secret) is used for its signing property only. The network temporarily stores the Info packet; the sending node needs to remember the nonce prefix to process the DH response.

This completes the description of Send(), except for one corner case: if the node is sending a message to itself, generate a sessKey, store it in the Session and use it from now on.

The detailed behavior of Recv() depends on the message type, but in all cases the first step is: find the Session belonging to the message's SrcAddr; if there is no Session, create an empty one for the new peer.

If the message is a PubkeyOffer, decrypt the payload using sec0. Discard the message if this fails. Otherwise extract the first 12 bytes of $E_{offerKey}$[Info] as offerNonce; generate a new nonce respNonce. Compute and store in Session.respKey

$$\text{SHA256[Curve25519}(secrKey_{resp}, pubKey) \text{ || offerNonce || respNonce]}$$

where $secrKey_{resp}$ is from the local node, pubKey is from the message and Curve25519 is the elliptic-curve scalar multiplication function. Send (to the node at SrcAddr) the message

$$\text{PubkeyResp || RespAddr || } E_{sec0}[pubKey_{resp} \text{ || respNonce || } E_{respKey}[E_{offerKey}[\text{Info}]]]$$

where RespAddr and $pubKey_{resp}$ are both from the local node.

If the message is a PubkeyResp, decrypt the payload using sec0. Discard the message if this fails or if there is no Session.offerNonce. Otherwise compute sessKey

$$\text{SHA256[Curve25519}(secrKey, pubKey_{resp}) \text{ || offerNonce || respNonce]}$$

where secrKey is from the local node. Attempt to recover the original Info as

$$D_{offerKey}[D_{sessKey}[E_{respKey}[E_{offerKey}[\text{Info}]]]]$$

(where D_{key}[] represents decryption with the given key). Discard the message if any step fails. Otherwise store sessKey in the Session and send to the node at RespAddr

$$\text{SessData || SrcAddr || } E_{sessKey}[\text{Info}]$$

This delivers the data that was entrusted to Send() when the DH exchange was initiated.

Finally, if the message is SessData, decrypt it with Session.respKey if there is one, and, on success, promote Session.respKey to Session.sessKey. Otherwise try Session.sessKey and Session.prevKey (in that order). Return the Info struct if one of the keys worked, else discard the message.

There is one Recv() corner case to consider – two nodes might simultaneously send each other PubkeyOffer messages (a situation known in the telecom world as *glare*). The solution is for the node with the lexicographically larger offerNonce to drop the PubkeyOffer that it receives and wait for a PubkeyResp from the other node.

Session key freshening is triggered by receipt of an Info.EpochID different from the one stored in the node. Then for each existing Session, the node copies sessKey (if not nil) to prevKey and sets sessKey to nil. Recv() can still process incoming messages but Send() will initiate a new DH exchange.

This key-agreement protocol assumes that the underlying message rate is not too high, so that a node is unlikely to send a second message to the same peer before completion of the DH exchange triggered by the first message. There is also the assumption that the message rate is not too low so that, in the event of packet loss, recovery will be automatic (i.e., after a couple of heartbeat periods). These assumptions obviate the need for DH retransmissions and timing-related state in the Session.

Since the flock shared secret sec0 is secret, there is no (secure) way to change it. If flock identity were instead determined by a certificate chain, this problem would go away in principle (since issuing new certificates involves transmitting only public keys). Whether or not the increased bandwidth is justified depends on the expected lifetime of the flock.