Taurus Design Notes

Purpose
A minimal system for receiving and sending short encrypted text messages using the TauNet protocol. (The details of the protocol are out of scope for this document; what's relevant is that it operates over TCP between a fixed set of individuals and is encrypted using CipherSaber2.) Some priorities are simplicity (as in deployability), usability, security, and efficiency, in that order.

Structure
Two main components: a listening daemon and an interactive reader/writer, with a separate encryption module accessible to both. The listener runs in the background and collects messages in an encrypted blob. The reader is opened on demand by the user and can read the blob, delete messages from it, or compose new ones. This trades away some of the more complex concurrency problems (thread management, I/O lag) for just file locking. It also allows the system to collect messages without the user being connected or having the UI up all the time. Finally, a config file keeps track of network information (and should thus also be encrypted, possibly with a user password separate from the network one?).

Listener
The listener runs in the background, receiving on a port specified by the TauNet protocol. When it receives a message, it decrypts it in memory; opens the blob, noting its last modification time (or hashing it?); decrypts the blob; appends the new message; encrypts the new blob; locks the blob file; checks for modifications since reading, and backs up a few steps if necessary; writes the new blob; unlocks the blob file; and resumes waiting for messages.

Writer/Reader
Haven't finished thinking this bit out yet, but in general terms, it should be able to load the blob and decrypt it into memory, allow browsing and deletion of messages (and cap the backlog at some length), notice when the blob has updated and refresh, allow the user to write new messages to people specified in the network config, and start/stop the listener. This could be a pretty basic command-line utility (which solves the blob-refreshing challenge) or a curses thing if I get fancy.

Message Blob
Message history is stored in a single encrypted file, with records separated by, at a minimum, the length of the following message (to avoid having to designate a record separator that could be present in a payload). This is preferable to storing messages in individual files for a few reasons. The main one is encryption overhead: RC4 keystreams take some time to generate because of key scheduling, but are quick to apply once they're generated, so working with one large encrypted file is faster than working with a bunch of small files. By "faster" I mean it takes a noticeable fraction of a second to run the dozen or so tests in the encryption module; I can probably optimize this some, but shouldn't rely on it when dealing with a backlog of perhaps 100

messages. Additionally, keeping the messages in one large blob prevents a lot of metadata (individual message quantity, sizes, and receipt times) from being visible in the filesystem.

There are a few potential disadvantages to this strategy. Adding messages to the blob creates some overhead when receiving a message: the whole thing has to be read and decrypted, the new message appended, and the blob re-encrypted and rewritten. However, the alternatives are worse. If messages are just stored encrypted as they're received, they have to be decrypted on the fly, which prototyping suggests would be noticeably laggy. (Decryption could be done in a separate thread from user I/O, but this raises the concurrency problems we're avoiding by separating the listener and reader in the first place.) A compromise, time-wise, would be decrypting in the listener and storing messages as cleartext, but this is an unacceptable security tradeoff. Storing all the messages together means that one breach compromises all of them, but realistically, if they're stored together and encrypted under the same key, that was already true.

Re-encrypting the blob regularly with new IVs might make it easier to infer the network key; this is worth looking into, and is potentially a good reason to encrypt it with an individual user password (which is particular to the instance of the client, and not to the network).