A Secure Social Network

Earl Lee, Peter Nguyen, and Marvin Qian

Introduction

Our current Peerster has good functionality. It can broadcast messages across a network of nodes, allowing connected nodes to be involved in an open chat room of some sort. It also allows for direct peer-to-peer messaging using a decentralized message transfer scheme and file sharing in a similar manner. However, in order for our Peerster to meet industry standards and be a viable tool in the real world, we must incorporate some form of encryption to secure direct messages. Note that open messages spread across the network using rumor mongering do not need encryption, since they are meant for every receiving node to read. However, direct messages serve a different use case. They are meant to be private and should not be exposed to anyone but the two parties involved.

Thus, we propose to adapt a version of Pretty Good Privacy (PGP) to encrypt direct messages. The primary components of PGP include public-key and symmetric-key encryption, digital signatures, and a web of trust, each of which we will implement in our Peerster.

In addition to this basic level of security, we want to add an additional layer of security in the form of friendships. PGP will allow us to establish a base-line of trust and then distribute keys through our web of trust, but we may not want to automatically trust everyone within that web. To allow for a more exclusive form of trust, users will need to send encrypted and signed friend requests and responses to each other. Users will be able to transmit direct messages only once a friendship is established. With the level of trust that a friendship represents, users can feel more secure when sharing files if they can share it to only their friends, which is the last feature we hope to include.

Key Transfer

One of the most critical tasks in securing point-to-point messages in Peerster is to establish a secure way to advertise public keys. The system will use a simple web of trust mechanism that relies on direct connections and manually selecting trusted peers. Peerster will display all peers (IP:port pairs) that are directly connected to the user's node and the user can select those that it trusts and directly notifies those nodes that it wants to merge their respective webs. If both peers agree to the merge, they will trade information about each other's webs (a set of origin names and their public keys) and then propagate it through their web of trust via a broadcast message. Thus, everyone in the web of trust will automatically be connected by transitive, direct trust links.

To walk through this process, let's suppose that Alice wants to add Bob to her web of trust. She will first send a message to Bob with two QVariantLists, "Keys" which contains QStrings (origin names) and "Names" which contains QByteArrays (public keys). Since Alice isn't already a trusted link, Bob will inspect the lists and decide if he wants to accept the connection. If he chooses to accept, he will send Alice a message with the origin names and public keys form his web and then propagate all the new keys to the rest of his web through his other direct links (and his links, since they are part of Bob's web of trust, will pass it on to their links and so on). Alice, when she receives

Bob's reply, will pass on all of Bob's keys to her direct links which will propagate through her web. As nodes relay the keys, they only pass on the ones that they have not yet encountered to prevent loops (since if they have encountered the key before they should have passed it on to all of their links before). At the end of this process, every node in Alice's and Bob's webs will have each other's public keys and thus the two webs will be merged. If Bob chooses not to connect with Alice, he simply does not respond.

While this approach is simple, it has a few drawbacks. One is that broadcasting one node's public key and origin name pair will have to span that entire connected component of the web of trust. Furthermore, it will trigger responses from all newly trusted nodes which will also increase network traffic. Instead of doing this any time a node wants to connect to a new, untrusted node, the nodes could trigger this process periodically (e.g., once per minute) and discover new, trustable nodes that way. Another way to resolve this is to try to accomplish the transfer through direct messages instead of broadcast along the web of trust. The next hop entry for each origin is not a feasible approach since there is no guarantee that that hop is a trusted peer and, even if it were trusted, there is no guarantee that the closest trusted hop has a path to the target peer. One possible solution is to keep a list of all next hops that are trusted regardless of latency. This could significantly reduce network traffic, but may not be part of the initial implementation. The final issue is that it might still not be enough privacy. The web of trust will likely weed out any legitimately malicious nodes, but just being not-malicious may not be enough to be willing to accept private messages from them. To remedy this, we propose a simple social-network to establish friendships.

Social Network

Friendships establish reciprocal trust at the level of sending private messages and sharing specific files. A friend request will appear much like an ordinary direct message except that, instead of a "ChatText" field, there will be a "FriendRequest" field with an encrypted, randomly generated QString. To accept the friend request, the node will respond with a similar message except the "FriendRequest" will become a "FriendResponse," and it will contain the QString that was received in "FriendRequest" but encrypted for the initiator to read and verify. To deny the friend request, the node simply does not respond.

Before friendship is established, direct messages from non-friends will be automatically dropped. After two nodes become friends, then direct messages will be properly passed between them. Furthermore, nodes will be able to share files that are only visible to their friends. The user will be able to specify if a file should only be shared to friends when sharing as well as specify if a search should only accept results from friends when searching. To actually secure the search replies for a file that is shared with friends only, the user will have to encrypt the search reply before sending it to his or her friend.

Sending and Receiving Messages

Once two parties are friends, they can begin to send secure private messages. Suppose Alice is trying to send a private message to Bob. She will first generate a random key for symmetric cryptography. She will then use the random key to encrypt her message to Bob. After that, she'll encrypt the random key with Bob's public key. Next, she'll hash the original, unencrypted message to create a digest. Finally, she'll encrypt the digest with her private key to create a signature. The

encrypted digest and encrypted random key will be additional fields in the message datagram that is sent to Bob.

When Bob receives the message, he will first decrypt the random key with his private key. Then he will use the random key to decrypt the message. Following that, he'll hash the message to create a digest. Next, he'll decrypt the digest from the datagram with Alice's public key and compare it to the digest he generated from the message. If they match, then the message from Alice reached him intact. If not, he'll assume that someone tampered with the message and discard it.

Suppose a malicious third party, Mallory tries to intercept the message along the path from Alice to Bob. There is no way for her to discover or modify the contents of the message. She can't decrypt the random key since she does not have Bob's private key so she cannot read the message. If she tries to modify the message, the digest won't match. If she tries to modify the digest too, it still won't match since it needs to be signed by Alice's private key which she also does not have. Thus, she cannot create a man-in-the-middle attack except for throwing away all or some of the messages.

To support potentially encrypted messages in addition to regular messages, the project restructures the message format. Since routing information is needed regardless of whether the message is encrypted, the new message format separates actual message content from routing info. This is achieved by having the top-level QVariantMap that represents a datagram contain a "Message" key that points to a QByteArray value which is a representation of another QVariantMap that contains the actual message content relevant to the receiving node. The "Message" value is a QByteArray instead of a QVariantMap so that it can be encrypted if necessary. The top-level QVariantMap will also contain any relevant routing info (e.g., Origin, Dest, Budget, HopLimit, LastIP, and LastPort) as well as a "Type" key that will map to a QString specifying how the message should be processed (e.g., private, rumor, search, etc.). Finally, if the message is encrypted, then the top-level QVariantMap will also contain a "Key" key that maps to the encrypted random key and a "Signature" key that maps to the signature for the encrypted message.

Conclusion

In our naive implementation of PGP, we can successfully prevent third-party nodes from reading or successfully tampering with messages sent between two nodes engaging in direct messaging. Hostile nodes will not be able to make sense of encrypted data, and any tampering would be evident upon decryption—so long as the hostile node does not have access to private keys. The friend request extension on top of that provides another layer of privacy and security, and it even allows for more restricted sharing of files.

We are still struggling to find a good way to build a web of trust given our limited resources. Manually selecting trusted nodes requires our initial trusted nodes to truly be trustworthy, but there is no easy way to confirm that in our small test environment.