*GUI’s and Security*

In typical software applications, users access application data using parts of a GUI. Data is read, created, and changed by the use of various interface parts such as text fields, check boxes, buttons, etc. The link between GUI’s and Security policy is important, yet rarely explored. When application data (usernames, passwords, IP Addresses) is protected by any access control policy, the GUI should also be aware of, respect, and follow the control policy as strictly as possible. However, due to the vastly differing natures of GUI creation and Security policy, GUI designers are not usually aware of application security policy, their job is to design layouts and events that may trigger other specific actions within the GUI itself, not to prevent users from breaking security policy.

*Merging GUI’s with Security*

In optimal conditions, the teams designing the Security policy and the GUI will be fully aware of how each field operates and works, and in addition, will be working closely such that both will be aware of each other’s changes and developments. In a more realistic setting, GUI designers will have a concrete understanding of only what cannot be touched under any circumstances in application data, and a large grey area of what may be changed depending on each individual user’s access rights.

In the first situation, almost no problems will occur in the design and implementation of GUI security, since both the Security model and the GUI will be designed concurrently. In the second, many potential problems at the merging of GUI and the rest of the application code may be encountered. The GUI may give too much or too little access to application data depending on security policy.

A proposed solution to the second situation is the use of an automatically generated model that will take a Security Model S(X), and a GUI G(x), and combine them so that the GUI will respect the security model, under a new model G(S(X)). The current state of this automatic model generation requires the security model to be written using SecureUML**,** and is still in its infancy—as of yet not viable, but worth keeping an eye on.

The second situation may prove beneficial in different ways as well. A network admin, who prefers to remain unnamed, says the most security holes in the network he manages are not generated by any flaws in the system, but by users who manage to break the simplest things, not necessarily by stupidity, but merely by not understanding how to keep thing’s secure. If given the choice, he would let users access as little as physically possible to prevent anything from going wrong.

*Implementation*

For our GUI, we should apply the concept of as little extra access as possible. For this, our major concerns lie with two major features, private spaces and plugins.

Private Spaces: The major concern with private spaces is the actual privacy of the private spaces. (Is it really private, is it secure, is there potential man in the middle threat?) To guarantee privacy, we have several options.

Symmetric Key: Symmetric Key systems rely on a shared key between two users. To quote a highly simplified example: “With a symmetric key system, Alice first puts the secret message in a box, and locks the box using a padlock to which she has a key. She then sends the box to Bob through regular mail. When Bob receives the box, he uses an identical copy of Alice's key (which he has somehow obtained previously, maybe by a face-to-face meeting) to open the box, and reads the message. Bob can then use the same padlock to send his secret reply”. The main weakness in the Symmetric Key system are the initial creation and exchanging of the key—for one to verify both authenticity and security, some previous established contact must have been made between the two example cases, in this case, Bob somehow needs to get a copy of Alice’s key.

Pros: Significantly less computational power required.

Cons: Problem of key exchange still has not been solved

Public/Private Key aka Asymmetric Key: As a second option, we have the public/private key system for each individual user, wherein a user has two keys one public, one private. The private key is kept secret, but the public key may be widely distributed. Messages are encrypted with the recipient's public key and can only be decrypted with the corresponding private key. A significant problem with public/private keys is verification of identity, in other words, whether or not a public key is correct, and from the proper issuer. The usual solution to this problem is to host a public key infrastructure, where one or more unbiased third parties, certify key pairs. To expand on the original example, “In an asymmetric key system, Bob and Alice have separate padlocks. First, Alice asks Bob to send his open padlock to her through regular mail, keeping his key to himself. When Alice receives it she uses it to lock a box containing her message, and sends the locked box to Bob. Bob can then unlock the box with his key and read the message from Alice. To reply, Bob must similarly get Alice's open padlock to lock the box before sending it back to her.”

Pros: Better, safer, key exchange. Has many already accepted and widely used models ie. **Diffie-Hellman Key Exchange.**

Cons: Harder to implement, more costly in terms of computational power.

Private spaces do have the potential to be used maliciously, (being in multiple private spaces in order to pass information along to another source, etc) but once application and data based security is guaranteed, what each individual user does outside of our application, no longer becomes our responsibility.

Plugins: Plugins will be mostly user generated content, with a few official and/or approved plugins available from a central server. Much like the asymmetric key system, since many of the plugins will be user created, verifying the safety of the various plugins will be an issue. A central directory of previously approved plugins—not users—may prove useful in the early stages of OpenComm.