CS 291 Network Security
Project Final Report

Ari Wilson, Shams Quayyum, John Breinig, John Chang, Erin Lang, Haley Proctor

December 5, 2006

Project Title: **Enhancing *gaim-encryption* With Secure File Transfers**.

Goal

Our goal is to modify the existing gaim-encryption plugin to include secure file transfers using the existing RSA guarantees for normal messages, utilizing gaim's modular architecture. The gaim-encryption plugin currently can provide authentication and confidential transmission for individual instant messages but not for transferring files over various protocols.

As far as we have researched, there are no other *viable* secure file transfer protocols for popular AOL Instant Messenger clients. We chose to modify gaim, one of the most common multi-protocol clients for *NIX/BSD-based operating systems. This project could prove useful for a significant number of people.

The choice of gaim-encryption as the security plugin to modify is motivated by the fact that one project member had experience with it, that it makes understandable security guarantees (without extensive research as in the gaim-otr plugin), and that it is not a rehash of what we had covered in class (made different security choices than gpg which the gaim-e plugin is based on). It is also fairly popular, which made getting support relatively easy.

An additional consideration for our choice of project was that fact that gaim and gaim-encryption are both open-source, enabling us to learn from a real-world, tested implementation of a security system. This fact also gives us experience with a large codebase (around 300 KLOC) and allows us to give back our work to the gaim-encryption project and the larger gaim community.

Approach

Most necessary algorithms, modules, and protocols for this project have already been implemented within the gaim-encryption codebase and the NSS library. We have explored the gaim file transfer protocol for file transfers originating and terminating on a hard drive. Technical considerations for this project include performance and security considerations.

Evaluation method

We have evaluated the success of our work by sending test files between clients and examining packets found using a network packet analyzer such as Wireshark on an attacker computer. These packets can be confirmed to satisfy the necessary security guarantees (confidentiality, authentication, et cetera). We will also examine any performance metrics we find important to measure and attempt to perform basic cryptanalysis on packets intercepted.

Design

Our project is based on the gaim-encryption plug-in for gaim, which provides up to 4096 bit RSA encryption using the NSS cryptography library developed by Mozilla, making use of PKCS#1. The gaim-encryption plug-in functions as follows: keys are automatically transmitted and stored, making it very easy to use, but also resistant to man-in-the-middle attacks. More specifically, the user's private keys are stored in a file (id.priv), public keys in another file (id), and accepted public keys are stored in yet another file (known_keys). Messages are encrypted exclusively with the RSA algorithm using 2048-bit keys as instant messages are assumed to be short and the performance degradation of using RSA versus a secret-key algorithm is considered to be relatively unimportant. It also ensures that key-exchange does not have to occur every time a conversation window is closed, which is a relatively frequent event in IM conversations. A signed hash of the message is attached and the entire message is sent, along with some implementation-related headers.

In the new version of gaim-encryption, the conversation-initiation procedure is strengthened. Now, the receiver must send a nonce to the sender along with the key. The sender then includes the nonce in the message, thereby verifying that it was not recorded previously. In the case that the sender believes that he or she already has an appropriate nonce, and the receiver has changed the nonce he or she was expecting, there will be an error message as well as a garbled/encrypted message. If such an error does not occur, secure IM conversations are allowed to proceed with the encryption functionality transparent to both users.

The security model for the gaim-encryption plugin is somewhat similar to the design of GnuPG but contains some differing features for the differing security goals of IM conversation use cases. Our additions to the plugin do not modify the basis of the security model for gaim-encryption. Therefore, to compare the two security systems we refer to the gaim-encryption FAQ:

> The one sentence answer is that this plugin can be every bit as secure and every bit as insecure as PGP. The longer answer involves a few principles that guided the design of this plugin, which I'll go into below.
>
> One principle is that frequently there must be a balance between better security and the added difficulties imposed on the user by extra security. Security that is difficult to use will be bypassed, and the end result is worse than a "less" secure system. However, whenever possible, the choice between security and ease of use should be left up to the user, not imposed on the user by software. Hopefully, the easy parts are built in, and the hard parts are do-able, if the user wishes. The user can then decide how much security they want, given a flexible software solution.
>
> This plugin's approach to providing this ease of use / security is very similar to that of SSH. By default, when you first talk to a user (host) that you haven't talked with before, the keys are automatically exchanged (but, like SSH, you can change this). Then, in the future, if the software sees a different key than the one you got that first time, it informs you that something may be wrong. If you want better security, you can verify that the public key that you received is the correct public key, via a channel that you feel is more secure than the original transmission of the key. This isn't too hard to do, as the public keys are stored in a human readable file (.ssh/known_hosts, and .gaim/known_keys). You can call up your friend and ask her to read the number on her screen to you, or ask your buddy to email you his key and sign it with his GPG key, or...
>
> Another reason for not using GPG is that, fundamentally, I think that many people want (and expect) a different level of security for IMs as compared to email. If a stranger

IMs you and you "accept" his public key, does that mean that you want to trust email that this same person sends to you in the future? Keeping the keys in separate pools means that you can lean towards convenience in your IM encryption but be stricter about security for your email.

The most obvious and user-friendly way to store private keys securely would be to use the password of the associated instant messaging user account as the password to decrypt the private key, but this password is not accessible to a gaim plugin. gaim-encryption makes the compromise of assuming that the user file system protection on the system running gaim-encryption is assumed to be secure. Therefore the user's private and public keys are stored in the clear in the user's profile directory (i.e. /home/wilsona6/.gaim/). There is another instant messaging client, ScatterChat, which modifies gaim itself to encrypt the key on the hard drive, as well as making other security guarantees.

Our design is implemented in C as a patch to the gaim-encryption codebase with the eventual intent of submitting our code back to the gaim-encryption project. Our design adds several steps to the existing procedures in order to support transparent secure file transfer in the context of a secure conversation. We added in hooks to the gaim-encryption code that detect when a file transfer is about to begin or end. Once a file transfer starts, we get the handle of the file about to be sent, create an encrypted version of the file (using existing encryption functionality and the keys that are stored in various data structures after the plugin loads) that is signed, and return a pointer to that data to the gaim file-transfer subsystem which is then sent over the network. If a file is too large for the block size, we break it using ECB mode (soon to be changed to CBC) and encrypt it that way. The actual transfer is done using existing gaim functionality. After the recipient gets the file, we hook in, authenticate it, decrypt it appropriately and write it to another file.

Implementation

We developed our code with version 2.0b3.1 of gaim, version 3.0b6 of gaim-encryption, and tested on a team-distributed image of virtualized Kubuntu 6.06 LTS on VMware Workstation 5.0 using gcc 4.0.3. This common development environment allowed code changes to be quickly redistributed for immediate testing.

Our implementation began by analyzing various sample plugins that were bundled with gaim in order to discern how they were interacting with the gaim subsystems. We found out that the way plugins interacted with gaim was through the use of callback routines that execute when event flags were raised. In our modifications to the gaim-encryption plugin, we had to register the secure file handling functions with gaim using the `gaim_signal_connect()` to link our function to specific program events and dynamically exporting them to gaim. When specific events occur, those events emit a signal which is caught by the gaim signal handling functionality. This callback mechanism allows for plugin-defined functions to easily interact with gaim, allowing changes in almost every facet of gaim's operation without making changes to the core code.

Once we figured out how to interface with gaim, we had to deduce a debugging mechanism. Starting gaim on the command line with the `-d` command line option starts it up in verbose mode, which was quite helpful to see in what context our functions were being called and what messages were being sent during implementation. We also found printing functions to aid us in printing our own debugging messages and viewing our execution in context of the entire file transfer process.

In order to ease implementation with the limited time available to us after confronting our challenges, we made several simplifying assumptions about the compatibilities of two clients. We assume that the keys of the two user accounts have already been transmitted, accepted, and stored

on both ends. We assume both clients are running gaim-encryption with our modifications. We assume that any person is only signed on using one software client, which is an assumption gaim-encryption itself makes. None of these assumptions impact the security of the file transmission itself and can be resolved fairly simply later, as they have been mostly dealt with in the existing IM handling routines.

The file sending routine has the following structure: find our key using the `GE_find_key_by_name` function, find their key using the `GE_get_key` function. We then sign and encrypt the file using the `GE_encrypt_signed` function. We finish by sending a complete block of memory with message size, digests, and the file itself created using `g_malloc` and `sprintf`. The file receiving routine mirrors the file sending routine, so it will not be explained in detail here.

Challenges

Most of the challenges we encountered were related to understanding the existing gaim source code, its modular architecture, and the source for the gaim-encryption plug-in. Such inspection was necessary in order to properly understand the functionality and security guarantees provided by existing code in order to implement our own code and modifications. We had to dive into the code as well as the included documentation to get a better idea of what we were looking at. Although the code is not poorly organized or labeled, the code does use C-style naming conventions and programming patterns, with which our group does not have much experience. The only documentation available is automatically-generated doxygen HTML files, which were mostly neither informative nor thorough.

We also encountered problems in designing a method to get the handle of the file that is about to be sent or received and having our code be executed before the code path for file transfer triggers.

Evaluation results

We tested our modification to the gaim encryption plugin by executing a file transfer between 2 clients running gaim with gaim-encryption. We also used another computer running Wireshark 0.99.4 to monitor and analyze the packets that were sent between the 2 clients involved in the file transfer. To thoroughly test our modifications, we tested transfers of files of sizes 2KB, 2MB and 50MB and analyzed the packets that were intercepted. Our use of RSA encryption ensured that no information could be extracted from any of the intercepted packets and our use of authentication assured that no one could alter the message, either. When tested, our modifications to gaim-encryption performed exceptionally well and provided the same security guarantees as the secure messaging. Intercepted packets were unable to provide us with any meaningful data when we tried to extract the original file that was transmitted.

The performance of our modifications also meet our requirements in terms of speed and security. Just like the secure conversations, we wanted to make the secure file transfer as transparent as possible (the user should not really notice a difference when using the encrypted file transfer). We planned on copying the file, encrypting it, and then sending the encrypted version, which is the way it is currently implemented in our code but this may change by the project due date.

This project was also beneficial to the group. The intrinsic value of this project was the real-world implementation experience the group received. Not only did the group have to understand unfamiliar code, but we also had to extend the code to provide additional functionality. It has been a great exercise in network security and has given us valuable insight into the world of open source software development as well as a better understanding of general software engineering principles.