Ryan Stillings

Cryptography and Network Security I

Final Project Writeup

Functional Overview

In the first section of this writeup I'm going to discuss a high level overview of the functionality of our project. For starters, a core design choice for this project was to enable direct peer to peer messaging without the need for a dedicated server. As such, we needed a way for a single file to act as either a client or a server during the connection process. What we ultimately created is a single main file which utilizes multiple threads to simultaneously allowing for outgoing connections, incoming connections, and a responsive GUI. When a connection attempt is sent out to another instance of the program, the sender acts as the client and the receiver acts as the server for the duration of the negotiation and secure channel establishment phases. While this does result in more complicated netcode for us to reason about, it ultimately allows the application to be entirely peer to peer, and creates a very seamless feeling connection process.

To give some additional background on the threads used, there are a total of four threads in our application. The first is the main thread, which initializes the program and then runs the GUI main loop. This thread allows for a responsive event driven GUI, and acts as the master for all child threads, who are created as daemons bound to the main thread. The second thread is solely tasked with listening for incoming connections. This thread accepts incoming connections on a listener socket whenever the user is not currently engaged in an existing chat.

When an incoming connection is established, the program state is set to securingConnection, and the third thread is spun up. This thread's job is to run through the secure connection protocol, performing negotiation and establishing a secure channel with the other member of the chat. This is where the on-the-fly server/client model comes into play, as this new thread will act as the server if it was spawned due to an incoming connection, but will act as the client if it was spawned due to an outgoing connection. Finally, the fourth thread is responsible for receiving and handling incoming messages. This thread only operates when a connection is established and secured, and will handle disconnecting when the connected peer exits the chat.

Next, I'd like to briefly touch on our project's GUI. The GUI was created in wxPython, a wrapper for a tried-and-tested GUI library which has existed for over two decades. While a GUI was not a requirement for this project, my partner and I felt that it would be the most effective way to facilitate chatting. The GUI provides us with fine control over what information is shared with the user and how, such as allowing us to display labels with the user's current ip and port, and fields in which to enter the desired peer's ip and port. The GUI also frees up the terminal, which runs alongside it, to store various debug and error information. Another advantage of the GUI is the ease of use when chatting. The entire chat history is stored in a scrollable pane, with the pane automatically scrolling to the bottom when a new message is sent or received.

To finish the overview, I'd like to mention the additional files included in our project. First of all, each cipher is stored in a package with a corresponding name used for both the package and the contained python source file. Cipher packages also include an empty file stating the non-abbreviated name of the algorithm, in case any cipher abbreviations are unclear. Global variables are stored in the shared module networking.py. This is where we keep

ip and port information, open sockets, the public and private key, and preference information for the current chat. As python threads operate on shared memory, this file acts as an effective hub for storing data that is required by more than one thread. Preferences are stored in the preferences.cfg config file. This file is loaded at startup, and indicates the user's supported ciphers listed in order of preference. The file can easily be changed in any plaintext editor, and includes top level comments indicating the supported cipher names from which preferences may be selected. Next, we have primality.py, which stores functions related to primes and primality. In particular, this file contains implementations of Miller Rabin, Pollard Rho, and prime generation in a Galois field. There's also cryptoutilities.py, which stores some common utlity functions we wrote for use in the various ciphers, such as modular inverse calculation, the extended Euclid algorithm, and various type conversions. Finally, we have hashing.py and sha1.py, which store our minimalist hmac and sha1 implementations (used for message authentication) respectively.

Technical Details

In this section of the writeup I'm going to discuss the technical details of our project implementation. Firstly, I'm going to explain the negotiation algorithm. In essence, our project follows a simplified version of the TLS protocol. The first thing that happens upon initiating a connection is that the client sends the server a message "Hello {0}", where {0} contains the client's ordered key exchange and encryption cipher preferences (as a reminder, we do not use a true server, rather we treat the user who sends the initial connection request as the client and the other user as the server for the duration of this protocol). If the server receives this correctly formatted message, it selects the highest ranked key exchange and encryption ciphers

that it received from the client. If at least one key exchange and encryption cipher was able to be selected between the server and client preferences, the server then responds to the client with “Hello2 {0} {1}”, where {0} contains the agreed upon key distribution cipher, and {1} contains the agreed upon encryption cipher. Upon receipt of this message, the client sets the agreed upon preferences, and returns a final “Hello ACK”.

If the server receives “Hello ACK”, the server and client can then move on to the secure channel establishment algorithm. During this phase, the server runs either the Diffie Hellman (implemented in homework 2) or RSA (implemented by Tommy for this project) key exchange protocol with the client, in order to establish a public and private key. The private key is then used by both the server and the client to seed Python’s random function, so that additional random values required by the agreed upon encryption cipher (such as primes) can be generated the same way by both parties. Any unexpected errors encountered during the secure channel phase or during the negotiation phase result in immediate termination of the connection process by both parties.

With a secure channel established, the client and server are finally able to move on to the message passing phase. It is at this time that the two users can send and receive encrypted messages with each other freely using the chat box implemented in the GUI. Four encryption ciphers are supported for message passing at this time: AES, RSA, Blum-Goldwasser, and Paillier (technically Toy DES is supported as well, but its use is only for testing/development purposes). AES and RSA were implemented by Tommy, Blum-Goldwasser was implemented during homework 3, and Paillier was implemented by me for this project. The Paillier implementation is pretty simple and by the books, generating matching primes and storing the resulting public

and private key in simple structs, with encryption and decryption made easy thanks to python's built-in modular exponentiation pow function. Whenever a message is sent, it is first encrypted via the agreed upon cipher. Different ciphers will convert the message into different sized chunks of their supported data type, but the message contents are ultimately encoded as a string, with the assistance of JSON for non-trivial string conversions. Once the encrypted string has been calculated, a mac is calculated using our custom hmac and sha1 implementations, and the encrypted message and mac are sent together to the other user. The other user, upon receiving a message, first splits the encrypted string and mac, and calculates their own mac. If the calculated mac does not match the sent mac, the receiver displays an error and disconnects from the chat, as this means someone is attempting to compromise the chat. Otherwise, the receiver proceeds to decrypt the message chunks by the agreed upon cipher, ultimately revealing the original unencrypted message. During this whole process, the sent and received message are added to each user's GUI in order to establish a message history for each user.

Closing remarks

I'd like to conclude the writeup by sharing a few remarks. First of all, the serverless peer to peer chat proved considerably more complex than initially expected. Allowing any user to act as either a client or a server at any time meant that parts of our application needed to be carefully constructed to avoid state errors and race conditions. Coming off of our white hat presentation, the biggest difference was our decision not to use a generic cipher class. As we discovered while implementing the various ciphers, Python's weak dynamic typing meant that we could use generic code that just worked despite differences between ciphers, as long as we had generated correct public/private keys for the agreed upon cipher. So while some sort of

generic cipher class may have allowed us to abstract some functionality, we decided the minimal benefit would not justify the increase in overall design complexity. For future releases, it would be a fun challenge to pursue full SSL compatibility, as well as to implement additional supported ciphers.