CSCI 3403 - Project 2

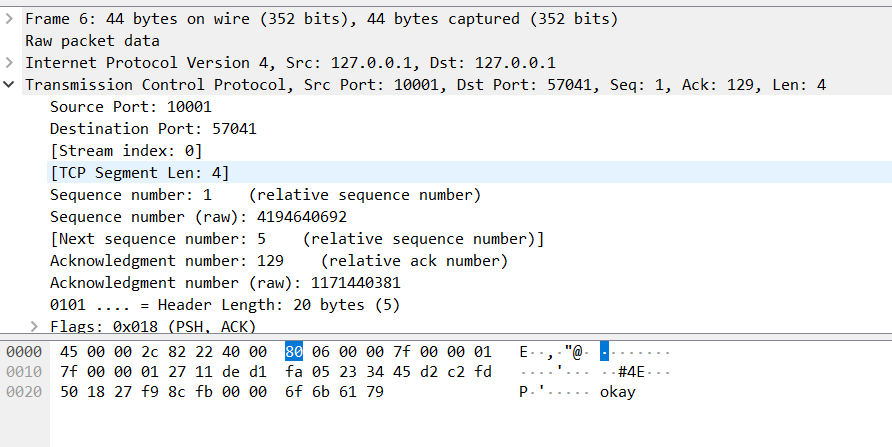
Sam Goulding

Alex Palo

1. In order to salt and hash the passwords input by the user, we imported two libraries: hashlib and uuid. The goal of salting the hashed user passwords is to prevent attackers from performing a rainbow attack on the table of passwords. When hashing the password input by the user, we used the SHA512 algorithm which is included in the library hashlib. Before hashing the password, we used the library uuid to create a salt that we then appended to the front of the password, preventing the same passwords having the same hash, and hashed the password. With the combination of a secure hashing algorithm (SHA512) and the introduction of salting, the users passwords are significantly harder for an attacker to gain access to.
2. In order to establish a secure connection between the client and server, we decided to use a RSA public/private key pair relationship between the client and server. In order to generate the key pair, we installed the Crypto package and imported the RSA library from it. By using the RSA library, we can simply generate a private key and its corresponding public key at the same time and export them to their own files for later use. The two files created (one for public, one for private key) were then relocated to their appropriate directories - the public key is included in the client directory so they can encrypt their AES key, while the private key is included in the server directory so it can decrypt the encrypted AES key sent by the client. When encrypting messages, we used the AES and PKCS1\_OAEP libraries to help us accomplish encryption. AES was necessary for encrypting messages, while PKCS1\_OAEP was necessary for encrypting the session key.
3. In order to accomplish symmetric encryption, we utilized the libraries mentioned in 2) : AES and PKCS1\_OAEP. The two functions encrypt\_message and decrypt\_message are identical in both client.py and server.py. This alone doesn’t achieve symmetric encryption, but when those functions are always being passed the same session key, we have established a method in which the client and server can encrypt messages knowing the other can decrypt that message with the established session key. When encrypting/decrypting messages, we used AES.MODE\_EAX encryption mode. When using this encryption mode, the cipher created will return not only the encrypted message, but also a unique tag for identifying the encrypted message. This tag can be passed along with the encrypted message to the other party and when they decrypt the message, they can check if the tag given to them matches the tag output by the decryption. If the tag differs, you learn that someone has interfered with the encrypted message somewhere along the line (adding an extra layer of verification). The major trade off with this encryption mode is that along with the encrypted message, you need to send over the generated tag and the ‘nonce’ (padding) created by the cipher encryption - achieving this requires that you send multiple packets at the same time (we got around this by serializing the message and nonce before sending, and deserializing once received).
4. Our program is completely safe from eavesdroppers because decryption of any messages requires a private key and all encrypted messages include a secret tag. The combination of these two elements ensure that eavesdroppers cannot read nor write new messages. Without a private key, the messages cannot be decrypted, and any message the client or server receives without the proper tag would be known to be tampered with. When viewing the traffic produced during the client-server connection, you can see that the only message that is in visible text is the servers ‘okay’, the rest are all sent via encrypted bytes
5. Our program is not secure from replay attacks. Theoretically, an attacker could mimic the actions of the client. There are multiple approaches we could take to prevent a replay attack. Firstly, we could implement a system that only allows a session key to be used once, requiring every session to have a new and unique session key. Another approach we could take to fixing this vulnerability would be to implement a timestamp system on the messages. Although this doesn’t entirely fix the vulnerability, it restricts the amount of time an attacker would have to penetrate the system.
6. Overall, we found this project extremely informative. The functions we had to implement were extremely intuitive, and also helped us visualize the various steps in an authentication process. We had to do some research on how to encrypt an AES key and how to generate RSA public-private key pairs, but once we got past that it was smooth sailing. Learning about the various AES encryption modes was extremely interesting as it showed us that there is more to symmetric key encryption than meets the eye!

Wireshark Capture:

This first image shows the packet sent from server to client which contained the ‘okay’ we specified it to send (unencrypted).



This second image shows a packet being sent from client to server (after handshake). As we can see, we can’t see what the packet contained as it is encrypted.

