Dillon Prendergast & Tao Ma

CNT 5412/4406

Network Security Project

3 December 2019

Secure Instant Point-to-Point (P2P) Messaging

When creating a messaging service, one of the most fundamental functions should be the security of its users’ and the service as a whole. Following the CIA triad - confidentiality, integrity, and availability – should be core for this service. How effective would a service be if there was no sense of privacy between users, or if the service does not work when needed or drops messages in transmission? Even worse if anyone can impersonate a user using the messaging service without any verification needed. For a connection between users’ to be secure, the service must use proper security practices and encryptions throughout and protect the privacy and integrity of any transmissions between users.

The first step in creating this secure messaging service is using a reliable connection. In this instance the Python3 socket module is used. “Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server” (*Socket Programming in Python*). This service will create a direct connection between two users, where the host of the socket is referred to as the ‘server’ and the user connecting to the host is referred to as the ‘client’. Once the server sets up the initial socket the server is able to maintain control of the used port until socket closure, creating a stable network environment for the messaging system. By using the *socket.listen()*  and then *socket.accept()* functions the server is able to first listen for any incoming requests to the running port and then accept the connection to the server (*socket- Low-level networking interface*). The first connection to be accepted by the server will be the known as the client, and from then on the server will not listen for more incoming connection requests, preventing a third party from eavesdropping within the chatroom. There is now a chatroom that has been securely established between a server and a single client, however there must be a method in place so that both parties can verify who is on the other side of the chatroom.

In order to verify both parties, this service utilizes a challenge response authentication. “In computer security, challenge–response authentication is a family of protocols in which one party presents a question ("challenge") and another party must provide a valid answer ("response") to be authenticated” (van Tilborg); this service utilizes a symmetric challenge response in which the underlying code will provide both the server and the client with a challenge, and the responses of the server and the client must match for a valid result. This challenge takes the form of a password entry, so the prerequisite for verification in the chatroom is that the server and the client must have previously discussed their passwords via another means of communication. The server and the client receive their challenges and submit their responses synchronously with one another so that the responses cannot be used in a reflection attack. In a reflection attack “a challenge-response authentication system that uses the same protocol in both directions…is to trick the target into providing the answer to its own challenge” (Tanenbaum). However, this attack is prevented because there is not a window of opportunity for an attacker to receive a response and flip it around for the attacker’s own challenge. This is further prevented by the responses being encrypted using RSA protocols.

Due to the sensitive nature of the password being transmitted, this challenge response is completed using RSA public-key protocols. Using the Python3 Crypto module, the server and the client are able to produce their own RSA public and private keys through *Crypto.PublicKey.RSA.generate()*. This *RSA.generate()* function takes in two parameters: key size and a selected pseudo-random function. In this instance a 1024-bit key is used and the Python3 Random module’s *Random.new()* is used to produce the pseudo-random seed for *RSA.generate()*. Once the public and private keys are generated for the server and client, the underlying code with exchange the server’s and client’s public RSA key. This public key is imported using *Crypto.PublicKey.RSA.import()*, which will store the sender’s public key for future encryption use. The stored public key is then used to encrypt the challenge response on both sides of the chatroom. This prevents eavesdroppers on the conversation from using a replay attack with a sent plaintext challenge response.

While it may be possible for an eavesdropper to reproduce the response knowing the public key and the ciphertext, only the sender of the response or the recipient knowing the RSA private key will successfully decrypt the plaintext response. To provide further security, when these public keys are transmitted, they are sent using OAEP, Optimal Assymmetric Encryption Padding. This padding scheme, commonly used with RSA, is composed of a Feistel network with random oracles. OAEP serves two goals: “Add an element of randomness which can be used to convert a deterministic encryption scheme (e.g., traditional RSA) into a probabilistic scheme” and “Prevent partial decryption of ciphertexts (or other information leakage) by ensuring that an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation” (Bellare). This serves to make the transmitted challenge-response ciphertexts proven secured against a chosen-plaintext attack. The OAEP padding scheme is imported from within the Crypto.Cipher.PKCS1\_OAEP Python3 module. It is then implement using *ciphertext.encrypt()*. By combining the RSA public key exchange and OAEP, the server and client are able to submit their responses to the challenge, while the response is safely encrypted and secured from interception by an eavesdropping third party.

{\displaystyle f}On both ends of the chatroom, once the encrypted challenge response is received it is decrypted using the server’s or client’s private key and the received plaintext is compared with the initial send plaintext on both ends. Thus, a symmetric verification process is created, and both the server and the client will can trust that the recipient of their messages is not fraudulent. This effectively prevents a user from assuming a false identity as the client, and the third party is also prevented by replaying the response of the server or the client due to simultaneous entry of the challenge responses by the original parties. Users of this messaging system can be confident in the recipient of their messages, and at this point the program will open up to message transmission.

To allow simultaneous sending and reception of messages on both ends, threading from the Python3 threading module was used to run a constant *SendMessage()* and *RecvMessage()* function. The *SendMessage()* and *RecvMessage()* transmit messages between the server and client, while both the server and client have threads of the form *threading.Thread(target = SendMessage, args = (socket, None))* and *threading.Thread(target = RecvMessage, args = (ConSock, None))*. The purpose of these threads is to appear as if *SendMessage()*, which takes user input and sends a ciphertext of the input to the other party, and *RecvMessage(),* which listens for transmissions over the socket and decrypts the sent ciphertext, are both consistently running.

In this scenario there are no race conditions present as the functions will be mostly idle, so the threads work well without having to set locks on execution. However, to maintain reliability, the socket used by the connection must be successfully closed after each use so that the connecting port is available for the next usage. For this, there must be an exit procedure so that the server-side and client-side programs know when to disconnect from the socket and free up the port. For this, there is an option to send the message *\*quit, which will end the *SendMessage()* function and will end the *RecvMessage()* function when received. From there the client and server must be able to end the opposite thread that they are running as well. The most efficient way to do this was to set the threads as daemon threads with *thread.daemon=True* so that when the initial thread is ended with *thread.join()*, the socket can be closed with *socket.close()* and the daemon will end the remaining thread upon program exit (*Anderson)*. The messaging program has ensured that the server and client are both verified, and that its services are reliable by not holding the port in use after closing the chatroom, now the program must ensure that the users’ messages are securely transmitted in order to protect their confidentiality.

If the messages exchanged between the server and client user are not encrypted, then any eavesdropper will be able to intercept plaintext messages and identify the conversation. This would negate any of the security practices in place by voiding the confidentiality of the users. For the encryption of messages between the server and client users, a 56-bit key is used. Because of this, the Data Encryption Standard (DES) is used to encrypt the transmitted messages. As messages are input in the *SendMessage()* function they are DES encrypted using the Python3 pyDes module. The function *pyDes.des()* takes in a Des\_Key, a encryption mode, an initial value(IV), and a padding scheme, and then the *pyDes.des.encrypt()* function will use the input parameters to turn the plaintext message into a ciphertext. DES encryption is symmetric so the *RecvMessage()* function utilizes *pyDes.des.decrypt()* with the same parameters and the ciphertext to reproduce the plaintext. DES alone is nowadays considered insecure because the 56-bit key is easily brute forced, as 56-bits have only 256 combinations that are possible for the encryption (Schneier*)*. However, there are some additional steps that can be used to make the most of DES.

Selecting a proper key is important for keeping the DES encryption algorithm secure. This program utilizes the shared secret password that was previously input by the server and client users. With that password the program utilizes Password Based Key Derivation Function 2 (PBKDF2) which applies “a pseudorandom function, such as hash-based message authentication code (HMAC), to the input password or passphrase along with a salt value and repeats the process many times to produce a *derived key*, which can then be used as a cryptographic key in subsequent operations” (Moriarty). By increasing the number of iterations that PBKDF2 uses, the input key will become unrecognizable and make future encryption indeterministic to an eavesdropper. The PBKDF2 function is found in the Python3 hashlib module under *hashlib.pbkdf2\_hmac()*. This function was chosen as it is a recommended usage in the DES-based Public Key Cryptography Standard #5 (PKCS#5) for creating 56-bit keys in tandem with DES. The supplied pseudo-random function is SHA256, with a shared salt between the server and client users, and the recommended 100,000 iterations. The usage of PBKDF2 limits the ability for a brute force of the Des\_Key or a dictionary attack if the plaintext password were to be used as the key.

In addition to PBKDF2, this program utilizes CBC (Cipher Block Chaining) as the cipher mode for its DES encryption. CBC is a sequential cipher mode, meaning that each subsequent block is dependent on the block before it or the IV. In this scenario there IV supplied to DES is shared, so that there is no way for it to become corrupted. In the event that a CBC ciphertext becomes corrupted in transmission then only two consecutive blocks are affected in the decrypted plaintext, making it acceptable for a message transmission without losing the message (Dierks). CBC modes primary weakness in that the message must be padded correctly to align with the size of the CBC blocks being used. This program utilizes the PKCS#5 padding scheme for this. In PKCS#5 padding, “Padding is in whole bytes. The value of each added byte is the number of bytes that are added, i.e. *N* bytes, each of value *N* are added. The number of bytes added will depend on the block boundary to which the message needs to be extended” (Housley). This padding scheme was created as a PKCS#7 variant for 64-bit block sizes, which fits the CBC block sizes that it will be used for. PKCS#5 is also error resistant in that if a byte is lost is exchange, each padding byte reveals how many bytes of padding there should be. This makes it efficient in messaging where a trailing byte may be lost in transmission; the missing byte will be easily detected. These supplements to DES make it efficient in both the speed and accuracy of transmitting messages, as PBKDF2 is a quick method of creating a secure key and the CBC mode and PKCS#5 padding give the program strong error detection and recovery.

Through the use of DES, the server and client users’ messages will be secured from the eyes of eavesdroppers as they are in transmission, and through RSA key exchange both users will be verified as their proper identities. This creates an ideal environment for both users to establish a connection and exchange information on. The weakness is this program is the use of 56-bit keys in DES as they as trivial to brute force given the state of current adversaries. If it were to be improved on, two-key triple DES could be used to mitigate the likelihood of a brute force on the encrypted messages, as triple DES provides security equivalent to 2112 combinations in a brute force attack. The second key could be derived from the PBKDF2 function as well using a different IV to further prevent an adversary from breaking the encryption.

Further improvements focus on the RSA public key exchange needed in order verify the server and client user. Having both users required previous contact for the password exchange is not always an ideal situation, but it also only provides one-time verification. After the initial verification, an adversary may step in and use an active messaging environment to masquerade as a verified user. Perhaps a better verification technique would be using the RSA keys for a signature on each message in addition to the original verification. Through this, each message must be signed by the private RSA key that is created locally on the server’s or client’s side. If a user’s session were to be hijacked by an adversary, they would not have to private key needed to make a message signature, so the message they send would not be verified.

One final improvement that can be made to the program is accepting multiple clients to the server, in the same chatroom or different chatrooms. The *socket.accept()* function will accept any incoming request on the same IP:PORT, so multiple active connections can be made. However, in the same chatroom there will have to be extra security precautions made to verify all of the participants so the RSA key exchange and challenge-response procedure would need to be modified. After the verification stage, the encryption and decryption of messages should work the same if the *SendMessage()* function is modified to send a copy of the message to all participants. It is important that these modifications would still be able to support the security of users.

Anderson, Jim. *An Intro to Threading in Python,* Real Python. https://realpython.com/intro-to-python-threading/

Bellare, M., P. Rogaway. *Optimal Asymmetric Encryption -- How to encrypt with RSA*. Extended abstract in Advances in Cryptology - Eurocrypt '94 Proceedings, Lecture Notes in Computer Science Vol. 950.

Dierks, T., *The Transport Layer Security (TLS) Protocol Version 1.1*. p. 20.

*hashlib-Secure hashes and message digests*, Python. https://docs.python.org/3/library/hashlib.html

Housley, R., *Cryptographic Message Syntax (CMS)****.*** https://tools.ietf.org/html/rfc5652#section-

*Module RSA*, Dlitz. https://www.dlitz.net/software/pycrypto/api/current/Crypto.PublicKey.RSA-module.html

Moriarty K*.; et al. "RFC 8018 - PKCS #5: Password-Based Cryptography Specification, Version 2.1".*tools.ietf.org.

*PKCS1\_OAEP,* Dlitz. *https*://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.PKCS1\_OAEP-module.html

Schneier, Bruce. *The Legacy of DES - Schneier on Security*. www.schneier.com.

*Socket-Low-level networking interface, Python. https://docs.python.org/3/library/socket.html*

*Socket Programming in Python,* GeeksforGeeks. https://www.geeksforgeeks.org/socket-programming-python/

Tanenbaum, Andrew S. *Computer Networks* , 4th edition.

*Threading-Thread-based parallelism,* Python. https://docs.python.org/3/library/threading.html

van Tilborg , Henk C.A.; Sushil Jajodia (8 July 2014). *Encyclopedia of Cryptography and Security*. Springer Science & Business Media.

Whiteman, Todd. *pyDes*, Sourceforge. http://pydes.sourceforge.net/