Zexing Li
Simon Fedotov

End-to-End Cryptography Solution for a Messaging Service

Introduction:

There are various attacks that could be used against messaging applications to eavesdrop, spoof, and modify data that is in transit or where it originates. Our implementation of a message encryption, decryption, and verification aims to mitigate those attacks to make messaging secure and safe. Richard programmed all the functions for decryption, encryption, key generation, and message verification in Python. Simon worked on the report as well as the presentation.

Security Analysis:

Taking the Facebook Messenger application as an example, the chat contents are stored on some server of this company. In the case of a data leak, those stored chat contents may be vulnerable to be shared around. Such data leaks are not uncommon, so it is important to find a solution that can protect the integrity and privacy of those messages. In order to exchange private messages using the Messenger application, we need to encrypt the data as it is created, while it is in transit, and as it is being received. We need to take all of these precautions because many attacks are possible on the data which is sent out, which puts the integrity of a message at risk. Similarly, man-in-the-middle attacks can either eavesdrop on the messages that are exchanged between two parties, or they can alter the contents of a message, which puts the privacy and integrity of those messages in danger. The encryption of data as it is sent out is most likely the best solution to protect the integrity and privacy of a conversation between two parties. Using public and private keys, each message sent between the two parties can be completely encrypted and only be seen by the person they are meant for. The authentication function implemented in our application allows for the two parties to verify that the message they sent/received is authentic and has not been tampered with. This mitigates the attack by an eavesdropper that could alter the data while it is in transit, which satisfies the integrity of the message.

Design Validity:

In order to initialize a conversation between Bob and Alice, Bob generates a public key (Kb) to initialize this session and sends it to Alice. Alice then generates a secret key (Ks), and encrypts it using asymmetric encryption, and sends it back to Bob. Both parties can then communicate using symmetric encryption and their secret keys (Ks).

The use of asymmetric encryption to share the private key for a session makes it easy to initialize a conversation without previously having a shared key for symmetric encryption. This allows for effortless messaging on the sender and receiver's parts, since all the encryption is done in the back end. Users can securely send and receive messages without worrying about eavesdroppers and anyone trying to mutate the data in transit.

Implementation Validity:

The basic program provides a function to generate and share new keys with asymmetric encryption and another function to encrypt data with symmetric encryption.

Zexing Li
Simon Fedotov

We start out by generating the public and private keys for Alice by calling the Alice_RSA_start function. The start function returns a call to the function RSA_generate_keys which will generate a private key and a public key, and then exports the public key to a file named "RSA_publickey.pem" and returns the private key. The private key is generated with a length of 2048 bits, which is recommended for encryption.

Next, we generate an AES key for Bob with the function Bob_AES_key_gen_send. The AES key is assigned to a value (Bob_aes_key) in the main function. The AES key is generated as a 256-bit value that is assigned randomly from various sources in the operating system. This was one of the more secure ways to generate a random key, as random number generators in Python may be too predictable. Additionally, the key_gen_send function shares the AES key with Alice using RSA. This line of code calls the RSA_encrypt_public_key function which will read the received public key file that was generated by Alice. The AES key is then encrypted with the public key and encoded in base 64. The encoded key is then saved to another file named "Encrypted_AESkey.txt" and the encoded/encrypted key is returned. Finally, the AES key generated by Bob is returned by the Bob_AES_key_gen_send function.

A new variable is declared in the main function called Alice_AES_key. This variable calls a function (Alice_get_AES_key) that allows Alice to decrypt the AES key using her private key as the argument. This function returns a call to another function called RSA_decrypt_private_key. The file which was created earlier ("Encrypted_AESkey.txt") is opened and read. The contents of the file are decrypted and decoded to get the bit values of the AES key.

Next comes the message encryption and decryption. If Alice would like to send an encrypted message, she would encrypt it using the AES_encrypt function with her decrypted AES key as an argument. At this point, we could also pass the plaintext message as an argument, however, for the sake of testing, we decided to use a constant bit string under the variable "data". At this point, we create variables for a nonce, a tag, and a ciphertext. All of these are encrypted using AES and read into three separate files: 'AES_nonce.txt', 'AES_tag.txt', and 'AES_ciphertext.txt'. These files will later be opened back up for decryption. This function serves as an encryption function for Alice, who is trying to send a message to Bob.

Lastly, the decryption function is called so Bob can read and verify the message that Alice sent him. The function is named AES_decrypt and takes Bob's AES key as an argument. As mentioned before, we open up all the files created during encryption: 'AES_nonce.txt', 'AES_tag.txt', and 'AES_ciphertext.txt'. Using the AES key and the nonce value, we create a variable called cipher which is used to generate the plaintext from the 'AES_ciphertext.txt' file. We then verify the tag of the message and if the message is authentic and not tampered with, our program will print out a verification that the message is in fact authentic and the decrypted plaintext with it. Otherwise, the program will tell us that the key was incorrect, or the message was corrupted.

Conclusion:

Zexing Li
Simon Fedotov

Overall, we think this is a good and simple baseline implementation of a key exchange messaging system. If we had more time to work on this, we would have added a more coherent user interface that allows a person to have more control over the program. Additionally, if we were to continue with implementing it for an application such as Facebook Messenger, we could have used the Facebook API to try and integrate it with Messenger. The automated process of this program would prompt the user to create and share keys, encrypt messages, and decrypt messages. With the Facebook APIs we would enable automatic message encryption so that the users only have to worry about what they will send and receive in their messages. This would allow for a secure messaging environment that can protect against eavesdroppers, man-in-the-middle attacks and the alteration of messages in transit.