# Network and Internetwork Security CSC4026Z: Practical

Furman, Gregory
FRMGRE001@myuct.ac.za

Dockrat, Nabeel
DCKNAB001@myuct.ac.za

Mapiti, Daniel
MPTSAN003@myuct.ac.za

Lurie, Avi
LRXAVI001@myuct.ac.za

June 2021

## Introduction

The objective of this practical was to gain experience with cryptographic functions and protocols by exchanging encrypted files and messages between two parties using a Pretty Good Privacy (PGP) cryptosystem. We achieved this by setting up a two-way communication channel using TCP where certificates, certified by a trusted simulated certificate authority, are exchanged and validated. Thereafter, messages can be securely transmitted using the princples of PGP. Namely, a combination of asymmetric & symmetric encryption, hashing, and compression.

## 1 Communications Implementation

### 1.1 Documentation on communication process

Our simulation utilised TCP, allowing for both parties to send and receive encrypted files and messages in a secure a two-way communication session. This was chosen over UDP as TCP is more reliable for such two-way communication, provides extensive error checking, and sequences its data output. For the purposes of this project, we also opted to utilise a client-server model to simulate the communication session. When communication is initiated, one party acts as a 'server' while the other party acts as a 'client'.

The server opens a TCP socket at a specific port number and waits until a client connects. Next, data input and output streams are created for two way communication. When the client module is run, it attempts to connect to the server at an IP address and port number given to the program as arguments. Once connected, the client establishes data input and output streams allowing for two-way transmissions. Both the client and server create two threads, respectively, that allow for the concurrent sending and receiving of data from the other party. Note that all client-server methods are accessed through the `Participant.java` class.

#### 1.1.1 Sending

The sending thread actively "listens" and waits for user input from the keyboard, reading it in line by line. A message $M$ can either be sent as plain text or a file with some plain text caption. In order to send a message across the network, message information is first sent to a set of supporting methods where PGP processes take place. Upon completion, the encrypted, compressed, and Base64 message is then sent to the recipient via an output stream.

#### 1.1.2 Receiving

The receiving thread actively "listens" and waits for TCP messages from the other party. When a message is received it is passed on to supporting methods which perform methods to ensure the authentication, integrity, and confidentiality of the PGP process was adhered to. Once checks have been completed and the message converted into a reading UTF form, the result is output to the recipients.

#### 1.1.3 Session Completion

The connection remains active until either the client or server disconnects from the session. A party disconnects manually by typing '-quit' as input into the console or simply fully closing the console window. A session can also terminate if some *fatal error* is encountered that would compromise the authenticity, integrity, or confidentiality of the communication session. The specifics of how messages are securely transmitted are explained in the sections to follow.

### 1.2 Documentation on key exchange process

In order for communication to take place between two parties, both are required exchange certificates. The certificate is created and signed by a trusted certificate authority (CA) and contains information relevant to secure communication. This includes the requestee's public key, a validity time interval, a unique serial number, the encrypting algorithm type, and more.

Our simulation of the CA works as follows. First, the main method of `CertificateAuthority.java` is run prior to the running of the client or server. This is due to the RSA key-pair of the CA needing to be generated. We did not deem it necessary for the CA to run as a third client making this process statically completed. The CA's main method works by firstly generating itself a private and public key pairing. Next, a self-signed certificate is created. The private key and the self-signed certificate is then stored in a `Java KeyStore` object and written to a file locked by a private password. Lastly, the CA's public key is converted to bytes and saved as a plain text file.

When contact is initiated between two parties, both request a certificate from the CA. This requires providing their name, location, and public key. The CA accepts this information, and in theory, does its due diligence before signing the certificate. When signing, the CA reads its private key from the `KeyStore` object, using its password as authentication. Thus, the certificate is then signed, returned, and sent to the respective receiving parties.

When a certificate is received, the receiving party reads in the CA's public key from the plain text file. This public key is then verified against the certificates signature using `Java's Certificate` class `verify` method. If verified, the public key of the sender is read and saved, otherwise an exception is thrown as the authenticity of the certificate is false.

As the CA is not a third client, its key pairings are localised to the machine that it has been run on i.e. if the program were to be tested on two different machines, both parties would not have access to the same key pairing. To avoid this issue, we have included a pre-generated public key file and private key `KeyStore` object upon submission.

## 2 Security Implementation

For the creation of the ephemeral *secret key* ($K_s$), we used AES encryption in *Cipher Block Chaining* (CBC) mode. AES encryption requires input to be a multiple of the *block length* $B$ in bytes. In the case of our system, a constant `AES_BLOCK_SIZE` was set to 32 bytes which motivates *padding* the data to be encrypted. We used Java's implementation of *PKCS5 padding*. This ensured data being was the correct length (in bytes) for encryption to take place. CBC mode also requires a securely and randomly generated *initialisation vector* to seed the AES encryption and decryption which was created in tandem with $K_s$.

The RSA algorithm in Electronic Code Book (ECB) mode was used for asymmetric key encryption and decryption. Each public ($KU$) and private ($KR$) key were set to a constant `KEY_LENGTH` of 2048 bytes. Moreover, *PKCS1* padding was used prior to encryption to pad data with (at least) 8 random bytes thus ensuring greater security.

All code for the cryptographic methods can be found in the `Cryptography` class.

### 2.1 Documentation on message integrity

When transferring data over a network, a message's integrity can be compromised leading to modification or corruption (due to an attacker or network malfunction). To maintain the integrity of the data being transferred, a secure hashing algorithm, namely **SHA-512**, is used to convert the variable-length message object into a fixed-size string of 512 bits (64 bytes).

Upon a message being sent from sender $A$ to recipient $B$, $A$ is required to use the SHA-512 algorithm to calculate a secure hash $H$ of the message being transferred. $H$ is referred to as the message digest and is sent along with the message to $B$. Upon receiving the message, $B$ independently calculates another hash $H_c$ of the data, using the same algorithm as $A$. Henceforth, $B$ compares the received hash $H$ with that of its own calculated hash $H_c$. A

discrepancy between $H$ and $H_c$ would indicate the integrity of the message has been compromised. Otherwise, we can assume that the message has not been altered and integrity has been maintained.

We also established a time to live (TTL) window in which to additionally establish message integrity. Should the time between the message being sent and the the message being received exceed the TTL, determined using the signature timestamp, the message cannot be authenticated. For the purposes of our system, the TTL of a message was determined to be 60 seconds. This small time frame was established as the transmission of a secure message or image over a network is not expected to exceed a few seconds (see Section 4.1.2). Thus, taking longer than what is expected is cause for concern and raises concerns of message interception.

### 2.2 Documentation on message authentication

The authentication of each party's identity is determined in a two-fold process. First, the simulated certificate authority generates certificates, assumed to be secure, that are exchanged upon the TCP connection being established (see Section 1.2). Second, authentication of an interaction is validated whenever a message is received.

Message authenticity is validated using a digital signature attached to each message. The signature contains a timestamp as well as the sender $A$'s public key $KU_a$ and the encrypted message digest $H$, having been encrypted using the **sender's** private key $KR_a$. Thus, as $EKR_a[H]$ is only able to be decrypted using the sender's public key $KU_a$, the receiver's calculated message digest $H_c$ being equivalent to the decrypted message digest $DKU_a[EKR_a[H]]$ allows for us to safely assume the identity of the sender has not been compromised.

### 2.3 Documentation on message confidentiality

The confidentiality of messages was achieved using a combination of the ephemeral shared session key $K_s$.

1. The initiator A generates an ephemeral secret key $K_s$ (called `key` in the codebase). Thereafter, some 16 byte initialisation vector `iv` is securely and randomly generated to be used as a seed for the AES CBC encryption.
2. The recipient B's public key $KU_b$ is used to encrypt $K_s$ resulting in $E_{KU_b}[K_s]$.
3. $E_{KU_b}[K_s]$, $KU_b$, and `iv` (all in byte form) are then attached to a `SessionKeyComponent` object.
4. Next, `Message` and `Signature` objects are created for the storage of information regarding a message and digital signature, respectively.
5. After having all necessary data written to them, `Message` and `Signature` are then compressed and encrypted (in that order) using a compression algorithm and $K_s$, respectively. These objects are then concatenated together, along with `SessionKeyComponent`, in a broader `PGPMessage` object.
6. `PGPMessage` is then sent to B via a TCP connection.
7. B then uses $KR_b$ to decrypt $E_{KU_b}[K_s]$ thus giving them the secret session key $K_s$.
8. B can then use the decrypted $K_s$ to decrypt the compressed & encrypted `Message`.
9. Since $K_s$ was encrypted using B's public key, B is the *only* party that should be able to decrypt $E_{KU_b}[K_s]$ using $KR_b$.

Thus, by utilising a combination of asymmetric and symmetric encryption-decryption protocols, we are able to guarantee that the confidentiality of the communication process is maintained.

## 3 Overall System Design

### 3.1 Documentation on compression

Our system uses `java.util.zip`'s `Inflater` and `Deflater` classes in order to order to perform lossless `ZLIB` compression and decompression at level 6. These classes are primarily accessed through our `compress()` and `decompress()` methods, however, slightly different approaches are taken for sending (3.1.1) messages and (3.1.2) files, respectively.

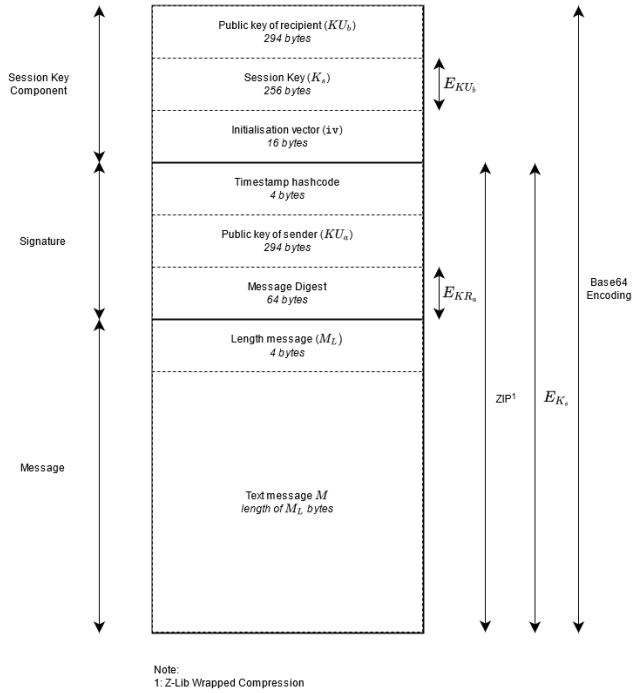


Figure 1: Format of a PGP message our system uses to securely send text messages across a TCP socket.

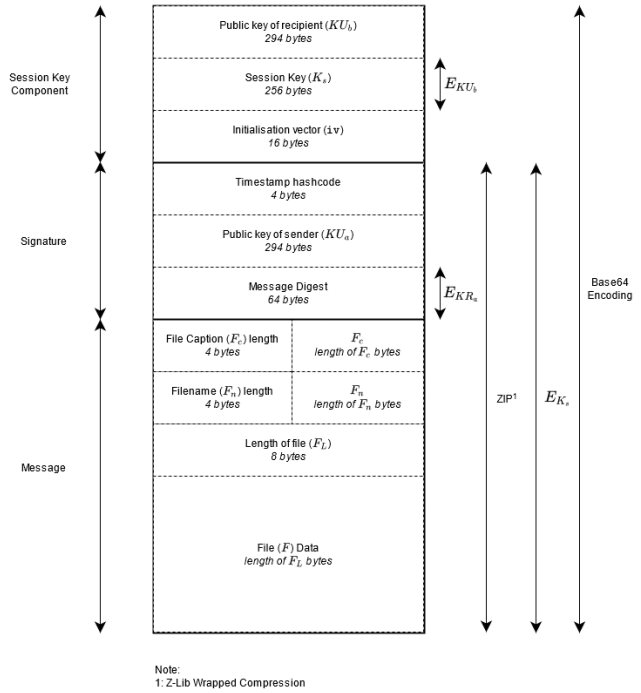


Figure 2: Format of a PGP message our system uses to securely send files with captions across a TCP socket.

### 3.1.1 Messages

During the sending of a standard PGP message, compression is performed on two different data concatenates, the `Message` and the `Signature`.

First, a byte array containing a message type `boolean`, the message `length` and byte array of the actual message `data` is parsed to `compress()`, which uses `Deflater` to return the relevant compressed data in the form of a byte array. Second, a byte array containing the `timestamp`, the `Public Key` of the sender and byte array of the $E_{KR_a}$-encrypted message digest is parsed to `compress()`, which returns the relevant compressed data in the form of a byte array. This is demonstrated in Figure 1.

Upon reception by the other party, following the relevant byte array decryption, both the compressed `Message` and the compressed `Signature` are parsed to the `decompress()` method, which calls `Inflater` to decompress the data accordingly.

### 3.1.2 Files

When sending a file-bearing PGP message, compression of data is handled in a different manner to that of plain-text messages.

First, a byte array containing a file type `boolean`, the caption `length` and byte array, and finally the filename `length` and byte array are parsed through `compress()`, which uses `Deflater` to return the relevant compressed data in the form of a byte array. The actual `data` of the file is then parsed to it's own `Deflater` instance, which itself is wrapped by a `CipherOutputStream`. In this way, we were able to ensure that the file `data` is compressed, encrypted and streamed out without overloading the `Java memory heap`.

Second, a byte array containing the `timestamp`, the `Public Key` of the sender and byte array of the $E_{KR_a}$-encrypted message digest is parsed to `compress()`, which returns the relevant compressed data in the form of a byte array. This is demonstrated in Figure 2.

Upon reception by the other party, following the relevant byte array decryption, both the compressed `Message` and the compressed `Signature` are parsed to the `decompress()` method, which calls `Inflater` to decompress the data accordingly. Much like during the compression phase, the file `data` is assigned it's own `Inflater` object, which is wrapped by a `CipherInputStream`. In this way, we were able to ensure that the file `data` is streamed in, decrypted and decompressed without overloading the `java memory heap`.

## 3.2 Documentation on the shared key used

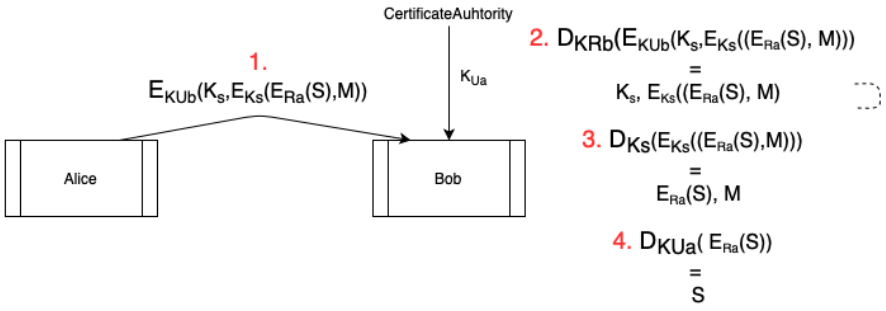


Figure 3: How $K_s$ is used when *Alice* sends a PGP message to *Bob*

In our system setup, a unique shared key, hereafter referred to as a `Secret Key`, $K_s$ is generated each time a message or file is sent. It is generated using `javax.crypto.KeyGenerator's init` method, with `AES` as its instance specfication. Naturally, this yields an AES-formatted 256 byte `Secret Key`.

As mentioned above, this `Secret Key`, $K_s$, is then used to encrypt the `PGP Message` and `Signature`, using `AES` in 32-Block `Cipher Block Chaining` (CBC) mode with `PKCS5Padding`.

The `Secret Key`, $K_s$ itself is then sent to the receiver, encrypted with the recipient's `Public Key`, $KU_b$, using `RSA` encyption in `Electronic Code Block` mode with `PKCS1Padding`. with it's own `Private Key`, $KR_b$. If the receiever is unable to decrypt $K_s$, it will also thereby be unable to decrypt the `PGP Message` and the $E_K R_a$ `Signature`. A streamlined explanation of this process, showing *Alice* sending a message to *Bob* is demonstrated in Figure 3. Note that the

It is in this way that both authentication and data integrity are ensured.

## 3.3 Order documented and justified

The order in which the data of our messages is compressed, encrypted and sent is in accordance with that of the general PGP approach. PGP only compresses the message after applying the *signature* and before *encryption*. This is done because:

- It is preferable to sign an uncompressed message so that the signature does not depend on the compression algorithm.
- Encryption after compression strengthens the encryption, since compression reduces redundancy in the message.
- Compression algorithms work on detecting redundancies and structure in the data, and encryption is designed to hide redundancies and structure.

Most importantly, if the compression was done before the signature was applied, it would mean that the hash value in the signature is not that of the original message, but rather that of the compressed message. Thus, for checking the authenticity at the receiver's end, we would have to either maintain the compressed message or to recompress the message. However, due to nature of the PGP algorithm, recompression would yield a further issue, as the algorithm may produce a variety of results based on it's implementation, and thus we may have inconsistency in the hash values obtained from them. Hence, there would be inconsistency in the authenticity verification.

# 4 Testing

## 4.1 Documentation on testing procedure

The following was done in order to stress-test our system:

### 4.1.1 Debugging statements

Fundamentally, PGP aims to ensure integrity of data, authentication of participants, and confidentiality of messages. We therefore included in multiple debugging statements to ensure that these core principles were adhered to. Thus, whenever a message or file & caption was sent across a socket, a host of debugging statements were output to the command line. This allowed for us to visually inspect, if an error was encountered, where the system was failing. Figures figs. 4 to 7 show a textual displays of cryptographic attributes (such as public keys, private keys, sessions keys, etc) needed in order to deem a channel of communication secure.

### 4.1.2 File Types & Sizes

Despite the project brief stating that images should be securely sent between participants, in order to better stress-test our system's ability to utilise cipher and compression streams (see Section 3.1.2), files of varying sizes were tested in order to validate our system's performance and accuracy. Furthermore, this also allowed for us to gauge the efficiency of our chosen encryption protocols for the task at hand. We tested different file sizes deemed small (less than 500kb), medium (greater than 500kb and less than 10mb), and large (greater than 10mb) relative to the size of a normal JPEG. Results showed our system to both accurately and efficiently stream compressed and decrypted data across sockets. File types used were PDFs, TXTs, MP4s, and various image types.

### 4.1.3 Using different machines

We used two different machines, both on the same network, to test whether or not intra-network communication could be achieved. While the speed at which small to medium sized files were transferred was not significantly affected, the sending of larger (>10mb) files saw minor, yet noticeable, performance diminishment in comparison with a single machine running both client and server locally. The performance when sending messages was not seen to be affected.

### 4.1.4 Unmatching RSA Keys & Unauthorised CAs

In the case of unmatching asymmetric key pairs, we designed our system to throw an exception. We tested this in various ways. Firstly we created certificates using two different certificate authorities. In this case an exception was thrown as neither party could verify the authenticity of the other party. Next, we created certificates with an incorrect public key as input. An exception was thrown later in the program as received messages could not be decrypted. Lastly, we ran tests where we changed a party's personal private key. And exception was thrown again as received messages could not be decrypted.

### 4.1.5 Comparing of Message Digests

Using debugging statements we were able to visually inspect for discrepancies in MDs between communicating participants. Moreover, as unequal hashes would indicate a message had either been tampered with, incorrectly transferred, or was not sent from the correct party - we deemed a discrepancy between the calculated and decrypted message digests a fatal error. Thus testing saw certain aspects of the final message digest be purposefully miscalculated on the recievers side in order to assess whether our system was able to throw an exception when encountering incorrectly calculated MDs.

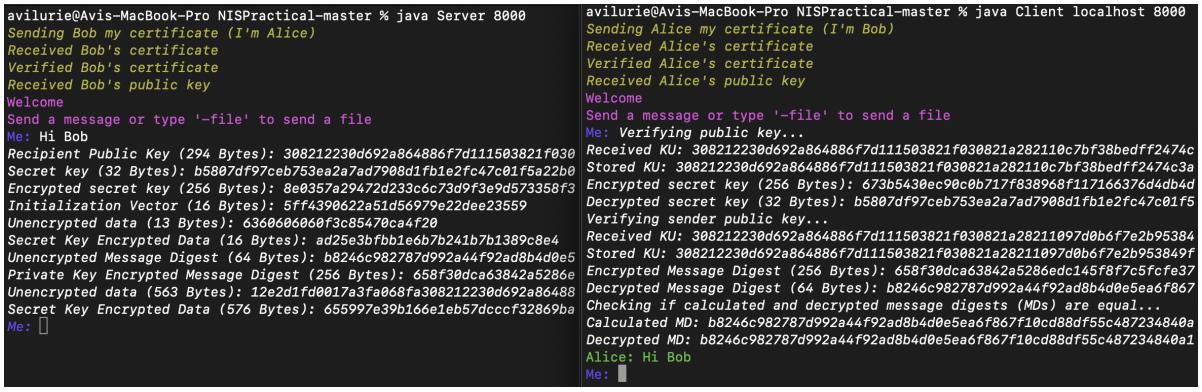


Figure 4: *Alice*(*right*) seen sending a message to *Bob*(*left*)

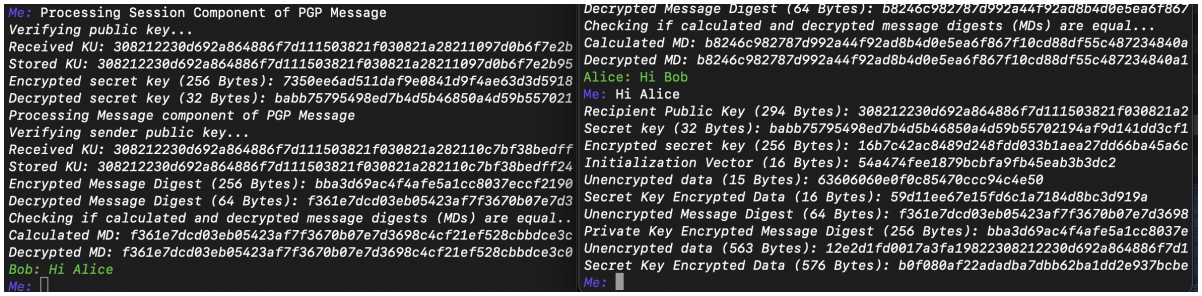


Figure 5: *Bob*(*left*) replying to *Alice*(*right*)

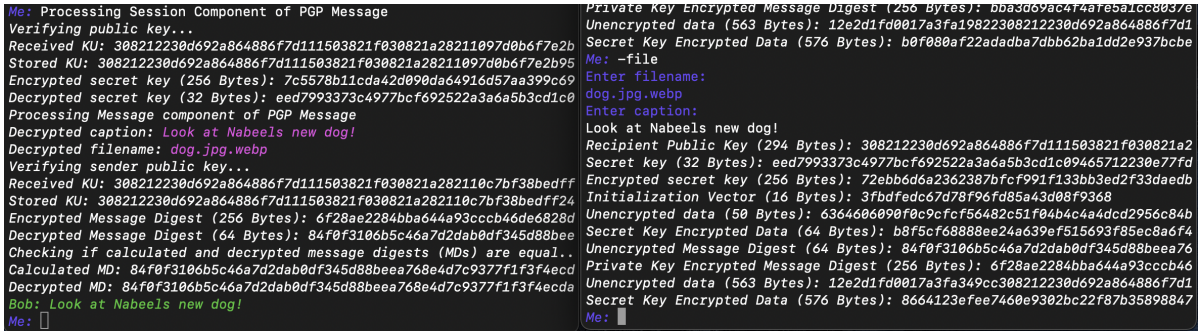


Figure 6: *Bob*(*right*) seen sending a picture to *Alice*(*left*)


Figure 7: *Alice*(*left*) acknowledges the picture and quits