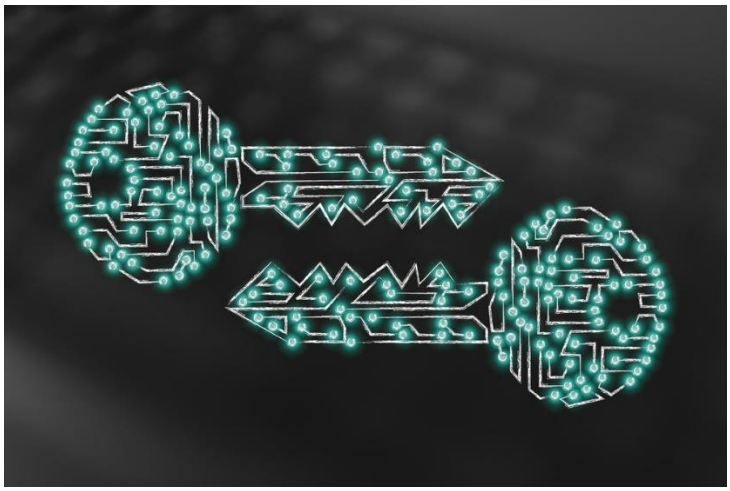
Development of a Pretty Good Privacy (PGP) Secure e-mail Protocol System Simulation and Analysis



Berke Kocadere

05170000484

Department of Electrical and Electronic Engineering

Ege University

25.06.2022

COMPUTER NETWORKS-II

Asst. Prof. Nukhet OZBEK

Abstract

Since the very early ages of humanity, communication was one of the most important skills for humans. In some fields of communication, privacy and security are the main concerns. Normally, communication messages are meant to be meaningful, but by time, humanity managed to develop science called cryptology. Cryptography or 'encryption' is all of the methods used to transform the information contained in a readable data into a form that cannot be understood by undesirable parties. Cryptography is a set of mathematical methods and is intended to provide the necessary conditions for the security of important information such as confidentiality, authenticity, authentication, and prevention of false rejection.

Since the era of Internet has begun, just as in real life communication, privacy and security are also needed in Computer Networks. So, Encryption and decryption of data is needed. There are many secure protocols are used in computer networks for safe and private data communication, and Pretty Good Privacy (PGP) is one of them.

Pretty Good Privacy (PGP), developed by Phil Zimmermann in 1991, is a computer program used to encrypt, decrypt or sign data using the OpenPGP standard, providing confidentiality and authentication of sent or received data. It is commonly used to encrypt and sign text documents, emails, files, folders and partitions. It includes encryption algorithms such as IDEA, RSA, DSA, MD5, SHA-1. The PGP protocol has nothing to do with protecting data traffic. The purpose of PGP is only the protection of files and e-mails. It can be said that PGP is the most common e-mail encryption protocol in the world. The reason for this is that in PGP, the data becomes accessible after passing mutual controls.. It is one of the most secure protocols in the world.

In this study, PGP Protocol System will be simulated using Python programming language and will be analyzed.

Table of Contents

1 Introduction

Since the very early ages of humanity, communication was one of the most important skills for humans. In some fields of communication, privacy and security are the main concerns. Normally, communication messages are meant to be meaningful, but by time, humanity managed to develop science called cryptology. Cryptography or 'encryption' is all of the methods used to transform the information contained in a readable data into a form that cannot be understood by undesirable parties. Cryptography is a set of mathematical methods and is intended to provide the necessary conditions for the security of important information such as confidentiality, authenticity, authentication, and prevention of false rejection.

Since the era of Internet has begun, just as in real life communication, privacy and security are also needed in Computer Networks. So, Encryption and decryption of data is needed. There are many secure protocols are used in computer networks for safe and private data communication, and Pretty Good Privacy (PGP) is one of them.

In this study, PGP Protocol System will be simulated using Python programming language and will be analyzed.

2 Background

In this study, we will focus on PGP Protocol system simulation. There are main concepts for briefing before method implementation.

2.1 Pretty Good Privacy (PGP)

PGP is an email and file encryption protocol developed by Phil Zimmermann. It includes encryption algorithms such as IDEA, RSA, DSA, MD5, SHA-1. The PGP protocol has nothing to do with protecting data traffic. The purpose of PGP is only the protection of files and e-mails. It can be said that

PGP is the most common e-mail encryption protocol in the world. It is one of the most secure protocols in the world. [1]

2.1.1. Release of PGP

Phil Zimmermann released the first version of PGP for free in 1991. After PGP became available, the US government reviewed PGP and launched an investigation against Zimmermann for violating export laws. This investigation continued for 3 years and Zimmermann was acquitted. Realizing that the solution of PGP is very difficult, the US government has banned PGP from going abroad for security reasons. Zimmermann explained that he wrote PGP because he wanted genuine security and respect for personal rights, in his article "Why I wrote PGP".

2.1.2. Principles of PGP

PGP is an asymmetric/public encryption protocol. PGP uses public and private keys instead of a single key. These two keys are each other's recognizers. As we know, data encrypted with a public key can only be opened with the private key of that key. Conversely, data encrypted with a private key can only be opened with the private key of that key. Adapting this to PGP is as follows; If someone wants to send us an encrypted mail, they must have our private key. He cannot send us an encrypted message without the private key. Another feature that PGP provides is: Even if the sender has our Private key, he cannot read the messages sent to us. This includes the message he sent us. The only way to open an e-mail encrypted with our public key is to know our private key.

E-mail security primarily ensures the secure transmission of e-mail to the destination. What is meant by security here; is to guarantee confidentiality, integrity, authentication and non-repudiation. In other words, people do not want to use an e-mail system where they do not trust that their e-mail will be transmitted securely. When an order is received by e-mail, how can we be sure that the e-mail is an order from the right source, the content of which cannot be read or changed by any third party, or that it is not a fake? If the e-mail system used does not use secure transmission techniques such as

encryption, authentication and electronic signature, we cannot be absolutely sure of this. It should not be forgotten that any unencrypted data can be obtained by hackers or malicious people, since e-mails that are not sent in a secure and encrypted manner pass through the public internet environment. The method used to eliminate these security problems is Cryptography (Encryption).

Encryption with PGP generally has a hybrid structure consisting of a combination of cryptographic hash function, data compression, symmetric key algorithms and public key cryptography. With this method, the e-mail is encrypted until it reaches the recipient's mailbox and its content is protected against unauthorized persons.

The process of sending encrypted mail with PGP is as follows;

1. The user who will send the mail requests the public key of the receiving party.

2. Upon request, the buyer sends the public key to the receiving party.

3. The sender who has the public key encrypts the mail with the sent public key and sends it to the receiver.

4. The recipient, who receives the encrypted message, decrypts the encrypted mail with his private key.

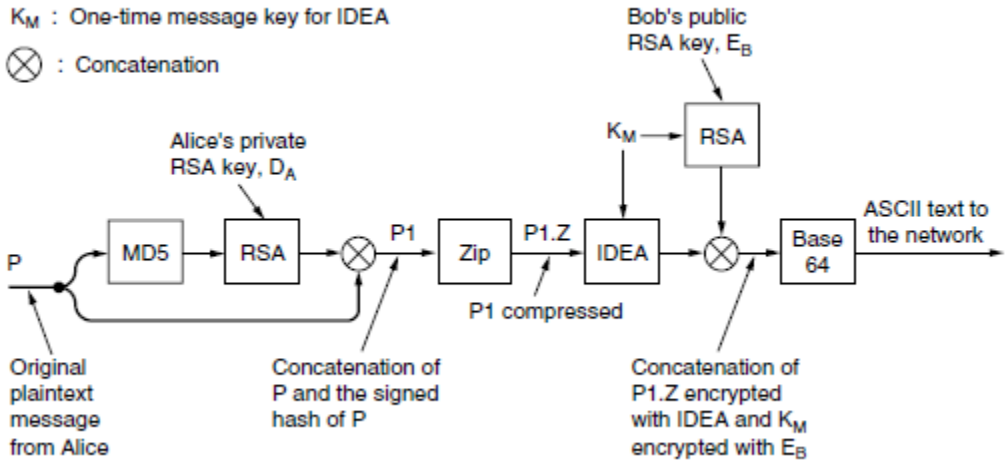


Figure 1 : Pretty Good Privacy (PGP) in operation for sending a message

As it turns out, even if the hackers get hold of the sent encrypted mail, they are trying in vain because the mail can only be opened with the private key on the recipient. Another negative condition that comes to our mind is the possibility of obtaining the public key sent to the person who will send the e-mail. The pre-important feature of PGP is already revealed in this situation. Each of the public and private keys that we have constantly mentioned above has only one task. The public key only serves to encrypt the mail, it cannot open the password. Therefore, even if the public key is captured, it is useless for decryption. This already adds to the asymmetric feature of PGP.

Encryption in PGP takes place in three steps. In step one, the sender signs the gist of the message. In the second step, the message is compressed. In the third and final step, the message is encrypted. When the receiving party wants to open the message, they should perform these steps in reverse. PGP uses RSA and IDEA algorithms as encryption algorithms. The MD5 algorithm is used when extracting the message.

2.1.3. PGP Security

PGP is known as a very secure encryption algorithm. However, some processes that could complicate PGP were found and prevented. We can list them as follows;

- Even if the message is encrypted and signed, the message can be copied by listening to the internet. Then this message is sent back to the receiver. Since the password and signature on the copied message are correct, the receiver will think that the message came from the sender. To prevent this situation, a time stamp is placed at the beginning of the message in PGP. Thus, the receiver knows when the message was sent, thus ensuring its security.
- After the sender encrypts and sends a top secret message, the plain version of the message will still be available on his computer. The sender will normally want to delete this message. But erasure operations on computers are reversible operations. PGP solves this problem by completely filling the content of the message with "0" after encryption.

We will follow the steps for PGP simulation shown in *Figure 1*.

2.2 MD5

The 128-bit hash value generated by the MD5 message-digest technique is cryptographically flawed but nonetheless frequently utilized. Despite being created with the intention of being used as a cryptographic hash function, MD5 has been shown to have numerous weaknesses. Even so, it can only be used as a checksum to ensure data integrity against accidental corruption. It continues to be useful for various non-cryptographic tasks, such as locating the partition for a specific key in a partitioned database, and might even be favored because it requires less CPU power than more contemporary Secure Hash Algorithms. [2] In order to replace the older hash function MD4,[4], Ronald Rivest created MD5, which was officially defined as RFC 1321 in 1992.

MD5 algorithm is an algorithm that provides the creation of a unique "fingerprint" of the given file or message (password, etc.) based on "hash" functions. It is a database management technique. It was developed by Professor Ron Rivest of MIT (Massachusetts Institute of Technology) in 1991. Professor Rivest designed the MD5 as an upgrade to MD4.

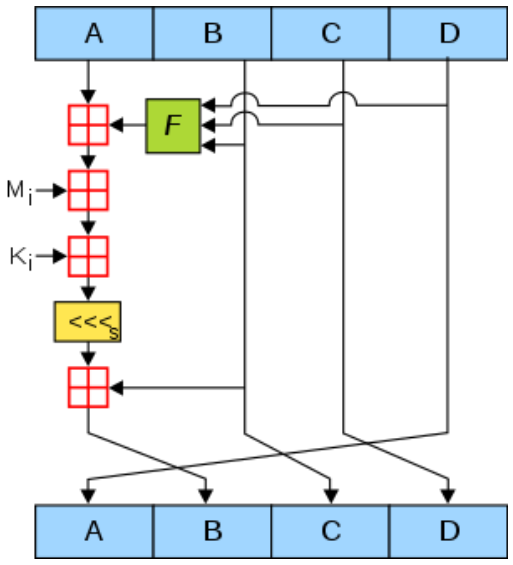


Figure 2 : One MD5 Operation

Properties

- MD5 algorithm works one way. Encryption is performed, but decryption cannot be performed.
- MD5 algorithm detects if there is any change in the processed file (transfer etc.). If a change is made, the result of passing the MD5 algorithm of the new file will be different from the MD5 result of the first file.
- The MD5 algorithm works slower than its lower version MD4, but the encryption system is much more complex and difficult to decipher.
- In general, it has a system of 4 different stages. Each stage has a different operation and consists of 16 steps. In an MD5 encryption process, 64 of the system in the picture below occur.
- Regardless of its size, a 128-bit long 32-character 16-digit string is obtained as the output of the file input to the algorithm.

Where Used

- In internet traffic. Like "SSL (Secure Sockets Layer)"
- In private computer networks. Like "VPN (Virtual Private Network)"
- In secure remote transportation applications. Like "SSH (Secure Shell)"
- In identification applications

Disadvantages

- In sites that are logged in with a user name and password, if the user forgets his password, the system cannot give the old password. The password cannot be decrypted because it is stored in the MD5 algorithm. The system assigns a new password to the user
- It takes longer to run because it generates a longer password than MD4

2.3 ZIP

Phillip Katz created the lossless-compression binary file format known as ZIP in 1989. Compression programs like WinZip and PKZIP use it. ZIP format is also an archive file format. It may aggregate and compress many files into a single file. The original files can then be recreated by first decompressing the generated ZIP file. A ZIP file has the.ZIP or.zip file extension. The size of the final ZIP file may be much less than the total size of the original files, depending on the sorts of files that were archived and compressed. [3]

2.4 Advanced Encryption Standard (AES)

Modern cryptography is generally divided into two, symmetric and asymmetric cryptography. Asymmetric encryption is based on the public key principle. The text to be hidden is encrypted with a key known to everyone and can only be decrypted with the secret key. Symmetric algorithms have a single secret key; This key is needed for encryption and decryption. Symmetric cipher can be divided into two main headings as block cipher and string cipher. In this study, Advanced Encryption Standard (AES), which is a type of block cipher, was used.

In January 1997, NIST initiated a study to develop a new encryption standard. It is thought that the new encryption standard to be developed will replace the existing standard DES. Because DES's 64-bit key space began to lose its reliability in the face of developing technology and increasing processor speeds.

A competition was held by NIST to determine the new encryption standard, and an official call for algorithms was made in September 1997. In August 1998, it was announced that 15 candidate algorithms (Round 1) were evaluated, and the number of algorithms was reduced to 5 in the 2nd Round elections held in April 1999.

After four years of evaluation and elimination, the result was announced in October 2000 and NIST announced that the Rijndael algorithm, designed by Joan Daemen and Vincent Rijmen, would be used as the Advanced Encryption Standard (AES). [4]

General Structure of the Algorithm

In the AES algorithm, input, output and matrices are 128 bits. The matrix consists of 4 rows, 4 columns (4×4), 16 partitions. This matrix is called 'state'. One byte of data falls into each partition of the state. Creates a 32-bit word per line.

The AES algorithm is a block cipher algorithm that encrypts 128-bit data blocks with a choice of 128, 192, or 256-bit keys. The difference in key length bits changes the number of AES round cycles.

In each round, 4 different sub-processes are performed. These are byte swapping, line wrapping, column shuffling, and round key aggregation, respectively. After 10 rounds, the entered data comes out encrypted. In the first round, the key is entered in its original form, and in the other rounds, newly produced keys are inserted.

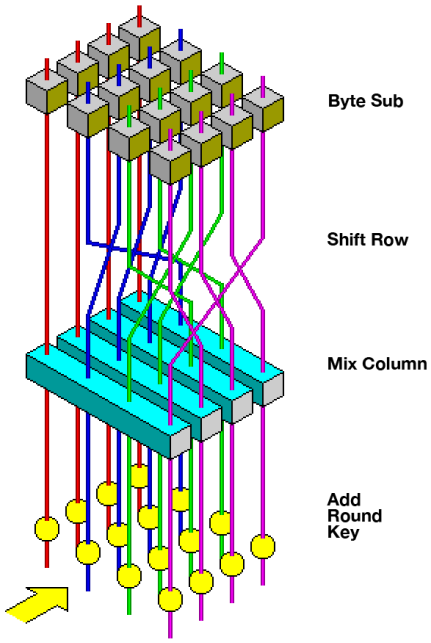


Figure 3 : Visualization of AES round function

2.5 Base64

Base64 is a collection of binary-to-text encoding techniques used in computer programming that converts binary data (more precisely, a series of 8-bit bytes) into sequences of 24 bits that can be represented by four 6-bit Base64 digits.

Base64 is a standard method for converting binary data to text and is used to transfer data between channels that can only reliably support text content. The World Wide Web[1] is where Base64 is most widely used, and one of its applications is the ability to embed picture files or other binary assets inside text assets like HTML and CSS files. [5]

Email attachments are frequently sent using Base64. This is necessary since the original version of SMTP was only intended to deliver 7-bit ASCII letters. Overhead is created by this encoding.

2.5.1. Design

Different implementations use a different set of 64 characters to represent the 64 digit values for the base. The typical approach is to select 64 printable characters that are similar to most encodings. Combining these two factors makes it difficult for the data to be altered while being transmitted across 8-bit-unclean information channels like email. [6] For the first 62 values, A-Z, a-z, and 0-9 are used in MIME's Base64 implementation. Other versions, such as UTF-7, also have this feature but have other choices for the last two values. The early examples of this form of encoding, such as uuencode for UNIX and BinHex for the TRS-80 (later modified for the Macintosh), were developed for dial-up connection between systems running the same OS. As a result, they could make more assumptions about which characters were acceptable to use. For instance, uuencode does not utilize lowercase letters and only employs uppercase letters, numbers, and numerous punctuation marks.

Base64 Encoding Table

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Figure 4 : Base64 Encoding Table

3 Methodology

Python programming language was used to implement PGP System simulation as shown in *Figure 1* between Server and Client, with TCP/IP Protocol.

In *client.py*, at first, sender inputs a plain text message. Then this plain text message is hashed with MD5 hashing algorithm. This hashed value is then encrypted with RSA using receiver public key. Encrypted MD5 Hashed value and original message is concatenated in a string variable. This string variable then compressed in ZIP format. Zipped data is encrypted with AES (instead of IDEA as shown in *Figure 1*) with a one-time message key, which we will call “session key”. Session key is then encrypted with RSA using Receiver public key.

At the end, RSA encrypted session key and AES encrypted zipped data is concatenated in a string variable. Concatenated data then Base 64 encoded and sen to receiver.

In *server.py*, receiver receives data from sender. Data is base64 encoded, so then it base 64 decoded. Decoded data is a concatenated string, so it is split to encrypted session key and encrypted zipped data. Encrypted session key is decrypted with RSA using Receiver public key. Encrypted zipped data is decrypted with AES using session key. Zipped data then decompressed and concatenated string is received. This concatenated string then split to encrypted MD5 hashed value and original message plain text. Encrypted MD5 hashed value is decrypted with RSA using Receiver private key.

Finally, receiver uses the same MD5 algorithm to hash received original message plain text. This hashed value is then compared with MD5 hashed value which was received from the concatenated data. If these two value are equal, PGP operation is successful and message content is acknowledged.

We will summarize these process for simplicity by listing steps as shown below, for both client and server side.

Sender Side

1. Sender inputs a message.
2. Message is hashed with MD5.
3. MD5 Hashed value is encrypted with RSA using Receiver Public Key.
4. Encrypted MD5 Hashed value and Original message is concatenated and zipped.
5. Zipped data is encrypted with AES with a one-time message key.
6. One-time message key is encrypted with RSA using Receiver Public Key.
7. Encrypted one-time message key and Encrypted zipped data is concatenated.
8. Final data is BASE64 Encoded and sent to Receiver.

Receiver side

1. Receiver receives data from Sender.
2. Data is BASE64 decoded.

3. Decoded data is splitted to encrypted one-time message key and encrypted zipped data.

4. Encrypted one-time message key is decrypted with RSA using Receiver Private Key.

5. Encrypted zipped data is decrypted with AES using one-time message key.

6. Zipped data is unzipped and splitted to encrypted MD5 Hashed value and Original message plain text.

7. Encrypted MD5 Hashed Value is decrypted with RSA using Receiver Private Key.

8. Original message Plain text is hashed with same MD5 algorithm and compared with MD5 Hashed Value. If it's the same, PGP Protocol was succesful.

4 Results and discussion

Simulation results are shown below.

Both Client and Server generates their own RSA private and public key pairs. PyCryptodome module has been used for both AES and RSA operations. [7]

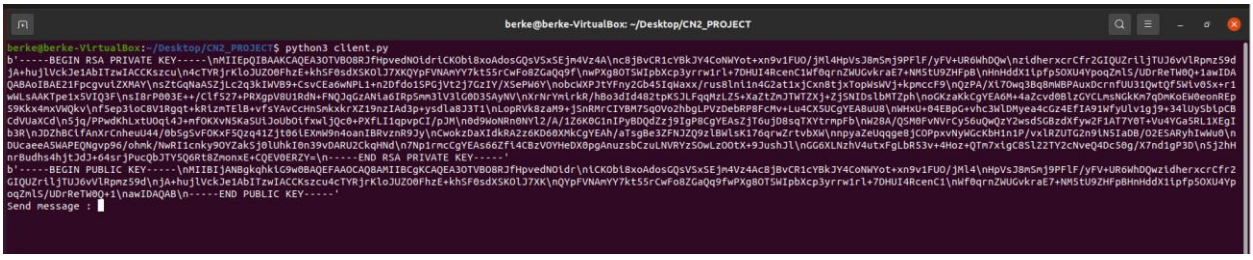


Figure 5 : Client RSA Private Key and Public Key generation

Figure 6 : Client RSA Private Key and Public Key generation

4.1 Client

At first, server waited for a connection from a client. Client is connected as shown in *Figure 6*. Socket module has been used. [8] Then user inputs a message (*message*) (Step 1) .This message is hashed with MD5 the encoded in bytes (*data_md5_bytes*) as shown in *Figure 7*. Hashlib module has been used. [9] (Step 2)

Figure 7 : Hashed value of message

MD5 hashed value (*data_md5_bytes*) then encrypted with RSA using Server's public key. [7] Server's public key is imported from file 'server_public.pem'. This encrypted value is kept in 'data_RSA'. This bytes object then converted to a string object ('data_RSA_text') for concatenation process as shown below. (Step 3)

Figure 8 : RSA Encrypted Hash value

RSA Encrypted hash value and original message plain text is concatenated in a string object (*concat_data*). This object then converted to bytes object for ZIP compression. (*concat_data_bytes*) Then it is compressed in ZIP format. (*zip_data*) (Step 4) Zlib module has been used. [10]

Figure 9: Concatenated Data

Figure 10 : Zipped data

Then, zipped data is encrypted with AES (*ciphertext*) using one-time session key (*session_key*) (Step 5) [7]. Then, this session key is encrypted with RSA using receiver public key (*enc_session_key*) (Step 6) [7]. Finally, AES encrypted zipped data (*ciphertext*), RSA encrypted session key

(*enc_session_key*) and AES nonce value (*nonce*) (This will be used for decryption on the receiver side) concatenated in a single string object (concat_data_v2). This is then converted to a bytes object for later use (*concat_data_v2_bytes*) (Step 7).

Figure 11 : Session key and RSA Encrypted session key

Figure 12 : AES Encrypted Zipped data and AES nonce value

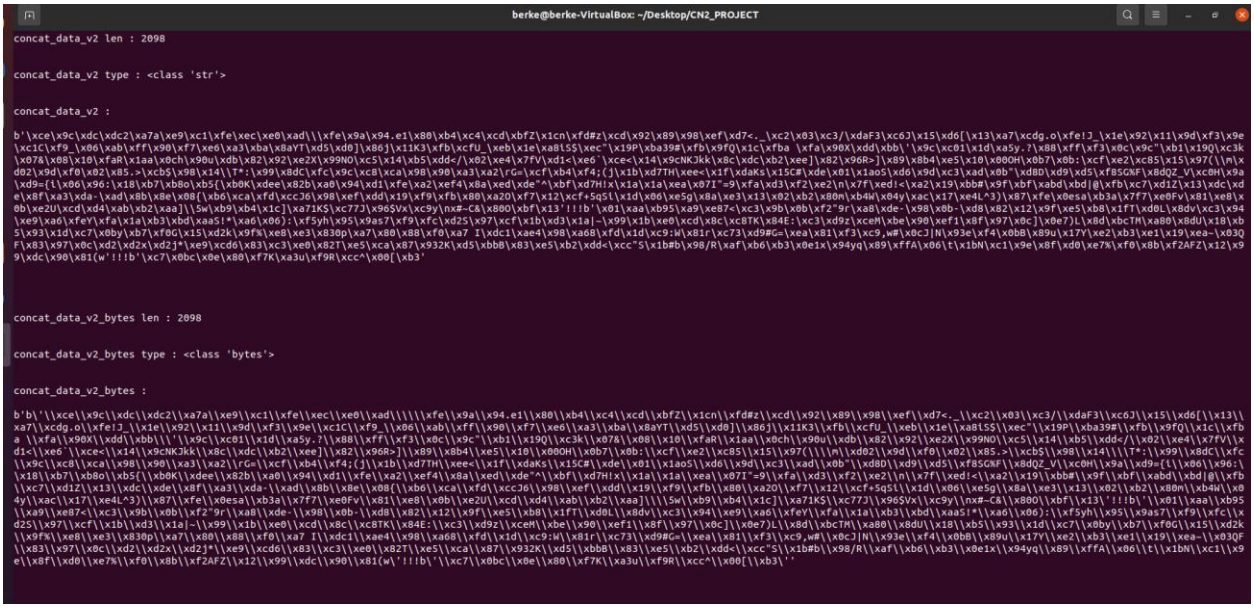


Figure 13 : Concatenated Data (Second)

At the end, this concatenated data (concat_data_v2_bytes) is BASE64 encoded and sent to receiver (Step 8) (base64_bytes). Base64 module has been used. [11]

Figure 14 : Encoding with BASE64 and Sending operation

4.2 Server

Server side steps are reversed version of client side steps. First, server is created. Then, data from client is received using recv() (Step 1) (*dataFromClient*) [8]. This is a string value, so that it is converted to bytes object (*data_bytes*). The value is same as in *Figure 14*.

Figure 15 : Data from Client

data_bytes then BASE64 decoded and we get the second concatenated data as a bytes object from *Figure 13 (concat_data_v2_bytes).* [11] Then it is converted to a string object *(concat_data_v2)*. We can observe that the values are same. (Step 2)

Figure 16 : Second Concatenated data

Then *concat_data_v2* is split into RSA encrypted session key and AES encrypted zipped data (Step 3).

Figure 17 : Ciphertext

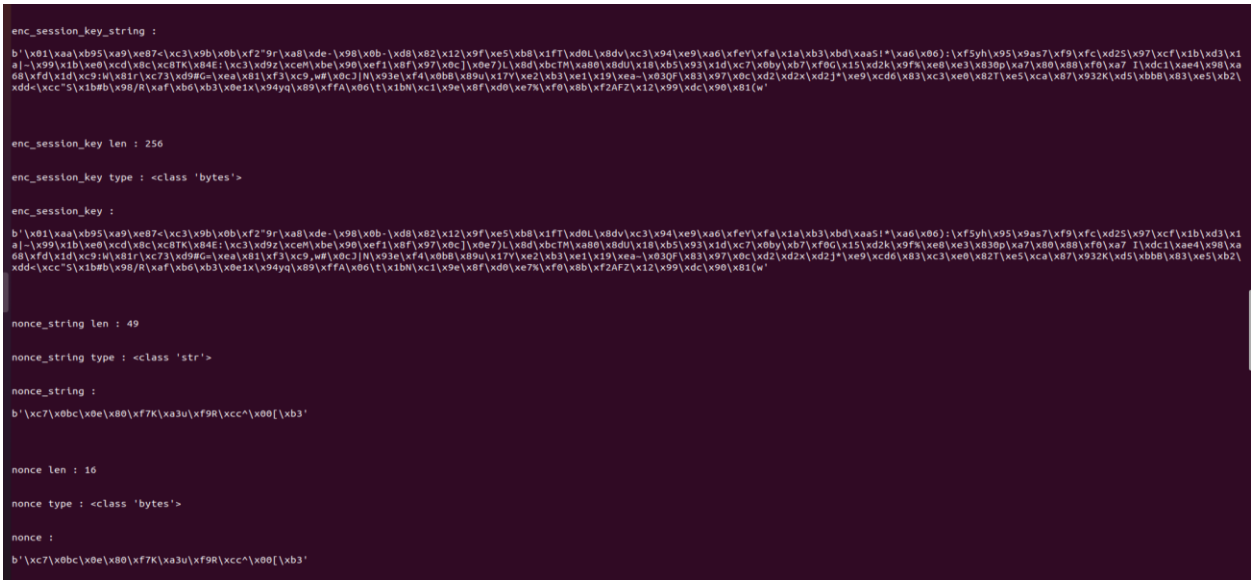


Figure 18 : RSA Encrypted session key and nonce value

In the next step (Step 4), encrypted session key is decrypted with RSA using server public key (*session_key*). [7] In Step 5, Encrypted zipped data (*ciphertext*) is decrypted with AES using *session_key (zip_data).* [7] We can observe that these values are the same as in *Figure 10* and *Figure 11.*

Figure 19 : Session Key and Zipped data

Zipped data is then decompressed and original concatenated data is received in bytes (concat_data_bytes). [10] We can observe that this is the same value as in *Figure 9*. Then, it is converted to string for splitting operation of plain text and signed hash. (Step 6)

Figure 20 : Concatenated data

Figure 21 : Split operation (plain_text and data_RSA_text)

data_RSA_text is a string object and in order to decrypt RSA encrypted data, we convert this string object to bytes object. After, we decrypt the this value with RSA using server private key [7]. After all, we can observe that we have obtained the same hash value as in *Figure 7*. (data_md5_bytes) (Step 7)

Figure 22 : Decryption of RSA Encrypted Data and Obtained MD5 hashed value in bytes

Now, we have obtained hashed value which is the same as in client. We also have plain text of the message. Server uses the same MD5 hashing algorithm on plain text and obtains another hash value. [9] Finally, it compares this hash value with the other hash value which we have decrypted. These are the same values as shown below, so PGP operation was successful. (Step 8)

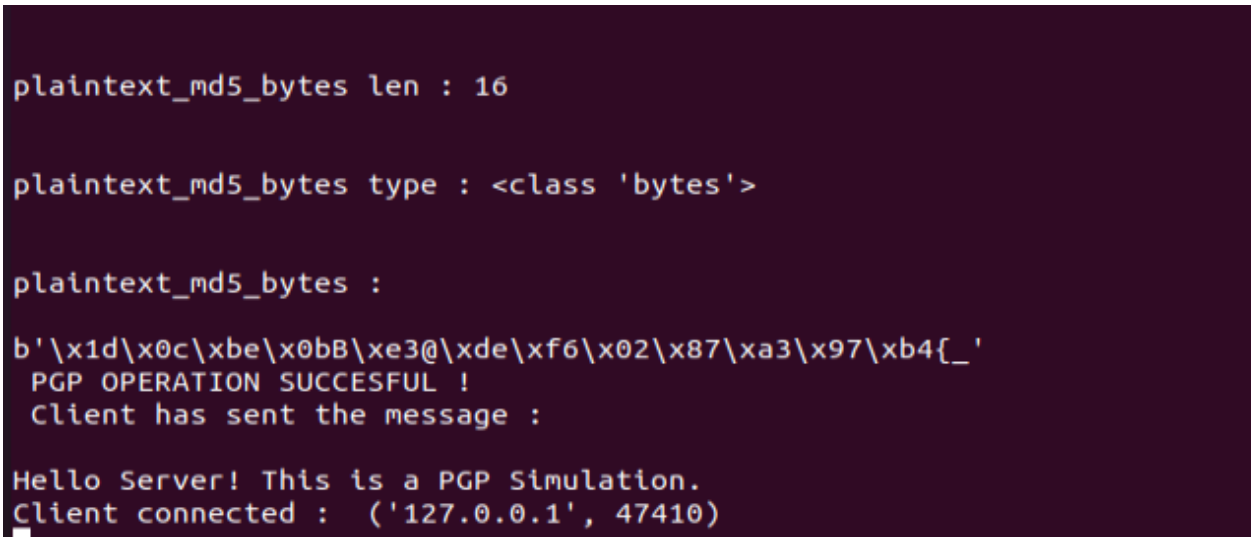


Figure 23 : Comparing plain text hash value and decrypted hash value for Successful PGP Operation

5 Conclusions and recommendations

In this work, we have simulated a simple Pretty Good Privacy (PGP) Secure Protocol in operation by using Python programming language. Program work flow in sender side was based on *Figure 1,* only with a difference which we used AES instead of IDEA. Receiver side follow the same steps in backwards.

With PGP, encryption of data is secure. Since the encrypted data can be only decrypted with a private key, it is almost impossible to decrypt a message which was captured by hackers; unless they have the private key.

This Python implementation of PGP is only a representation of PGP in a simpler version, which can not be as safe as it is used in real life. There may be still some security invulnerabilities in operation. Code can be improved in the future.

7 References

[1] https://bilgisayarkavramlari.com/2012/03/22/pgp-pretty-good-privacy/
[2] Kleppmann, Martin (2 April 2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (1 ed.). O'Reilly Media. p. 203. ISBN 978-1449373320.
[3] "Phillip Katz, Computer Software Pioneer, 37". The New York Times'. 1 May 2000. Retrieved 14 June 2009.
[4] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)" (PDF). Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001. Archived (PDF) from the original on March 12, 2017. Retrieved October 2, 2012.
[5] "When to base64 encode images (and when not to)". 28 August 2011.
[6] The Base16,Base32,and Base64 Data Encodings. IETF. October 2006. doi:10.17487/RFC4648. RFC 4648. Retrieved March 18, 2010.
[7] https://pycryptodome.readthedocs.io/en/latest/src/examples.html
[8] https://docs.python.org/3/library/socket.html
[9] https://docs.python.org/3/library/hashlib.html
[10] https://docs.python.org/3/library/zlib.html
[11] https://docs.python.org/3/library/base64.html

Appendix A: Python codes

client.py

```
'''


   Author = Berke Kocadere
   Date = 25.06.2022
   Version = 1.0

   This program is a simple Pretty Good Privacy (PGP) Secure Protocol
simulation for Sender Side. MD5, RSA, ZIP, AES and BASE64 Encoding are used.
Work flow of the program is shown below.

   Sender Side
   1. Sender inputs a message.
   2. Message is hashed with MD5.
   3. MD5 Hashed value is encrypted with RSA using Receiver Public Key.
   4. Encrypted MD5 Hashed value and Original message is concatenated and
zipped.
   5. Zipped data is encrypted with AES with a one-time message key.
   6. One-time message key is encrypted with RSA using Receiver Public Key.
   7. Encrypted one-time message key and Encrypted zipped data is
concatenated.
   8. Final data is BASE64 Encoded and sent to Receiver.


'''

# Imports
import socket
import hashlib
import zlib
import base64
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Hash import MD5



'''
   Function : RSA Key Generator

   Description :
      This function creates RSA private and public key for Client and
      store them in file system as PEM format.

   Return : key

'''
def RSA_keyGen():
   key = RSA.generate(2048) # RSA Keypair Generation
   private_key = key.export_key() # RSA exporting private key
```

```
    file_out = open("client_private.pem", "wb")
    file_out.write(private_key) # Store private key in PEM format
    file_out.close()

    public_key = key.publickey().export_key() # RSA exporting public key
    file_out = open("client_public.pem","wb")
    file_out.write(public_key) # Store public key in PEM format
    file_out.close()

    # Print Key Values
    print(private_key)
    print(public_key)

    return key # Return Key Object



'''
    Function : TCP Client Connection

    Description :
        This function creates a TCP connection to a host server by using
socket.

     Return : client

'''
def clientConnection():
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # TCP socket

    host = "127.0.0.1"
    port = 15000

    client.connect((host, port)) # Client connection

    return client # Return Client object



'''
    Function : Print Data

    Description :
        This function is used for printing length, type and content of a
variable.

    Return : None

'''
def printData(data, dataString):
    print("\n\n")
    print("\n{} len : {}\n".format(dataString,len(data)))
    print("\n{} type : {}\n".format(dataString,type(data)))
    print("\n{} : \n".format(dataString))
    print(data)
```

```
'''
   Function : Print Hash

   Description :
      This function is used for printing type, size and content in hex of a
hash value.

   Return : None

'''
def printHash(data_md5):
   print(data_md5.name)
   print(data_md5.digest_size)
   print(data_md5.hexdigest())



# RSA Private and Public Key Generation for Client
key = RSA_keyGen()
# Client Connection
client = clientConnection()



while True:
   # Message input P
   message = input("Send message : ")

   # Encode message in 'data'
   data = message.encode("ascii")
   data_str = "data"
   printData(data, data_str)

   # Hash message with MD5 in 'data_md5'
   data_md5 = hashlib.md5(data)
   data_md5_str = "data_md5_str"
   printHash(data_md5)

   # Encode hash value of the message in 'data_md5_bytes'
   data_md5_bytes = data_md5.digest()
   data_md5_bytes_str = "data_md5_bytes"
   printData(data_md5_bytes, data_md5_bytes_str)

   # RSA Encryption Setup and Read Server's Public Key from file
   server_publicKey = RSA.import_key(open("server_public.pem").read())
   cipher_RSA = PKCS1_OAEP.new(server_publicKey)

   # Encrypt the hash value with Server's Public RSA Key in 'data_RSA'
   data_RSA = cipher_RSA.encrypt(data_md5_bytes)
   data_RSA_str = "data_RSA"
   printData(data_RSA, data_RSA_str)

   # String conversion of Encrypted hash value into 'data_RSA_text'
   data_RSA_text = str(data_RSA)
   data_RSA_text_str = "data_RSA_text"
   printData(data_RSA_text, data_RSA_text_str)
```

```
    # Concatenation of message P and the signed hash of P in 'concat_data'
    concat_data = data_RSA_text + "!!!" + data.decode("ascii")
    concat_data_str = "concat_data"
    printData(concat_data, concat_data_str)

    # Encode Concatenated message in 'concat_data_bytes'
    concat_data_bytes = concat_data.encode("ascii")
    concat_data_bytes_str = "concat_data_bytes"
    printData(concat_data_bytes, concat_data_bytes_str)

    # Comress Concatenated message with ZIP format in 'zip_data'
    zip_data = zlib.compress(concat_data_bytes)
    zip_data_str = "zip_data"
    printData(zip_data, zip_data_str)

    # Create a one-time session key in 'session_key'
    session_key = get_random_bytes(16)
    session_key_str = "session_key"
    printData(session_key, session_key_str)

    # Encrypt the session key with Server's Public RSA Key in
'enc_session_key'
    enc_session_key = cipher_RSA.encrypt(session_key)
    enc_session_key_str = "enc_session_key"
    printData(enc_session_key, enc_session_key_str)

    # AES Encryption Setup
    cipher_AES = AES.new(session_key, AES.MODE_EAX)

    # Encrpyt the zipped data with Server's Public RSA Key in 'ciphertext'
    ciphertext = cipher_AES.encrypt(zip_data)
    ciphertext_str = "ciphertext"
    printData(ciphertext, ciphertext_str)

    # Nonce value of AES Encryption - one time value - in 'nonce'
    nonce = cipher_AES.nonce
    nonce_str = "nonce"
    printData(nonce, nonce_str)

    # Concatenation of Encrypted zipped data, Encrypted session key and nonce
value of AES Encryption in 'concat_data_v2'
    concat_data_v2 = str(ciphertext) + "!!!" + str(enc_session_key) + "!!!" +
str(nonce)
    concat_data_v2_str = "concat_data_v2"
    printData(concat_data_v2, concat_data_v2_str)

    # Encode New Concatenated message in 'concat_data_v2_bytes'
    concat_data_v2_bytes = concat_data_v2.encode("ascii")
    concat_data_v2_bytes_str = "concat_data_v2_bytes"
    printData(concat_data_v2_bytes, concat_data_v2_bytes_str)

    # Encode New Concatenated message with Base 64 Encoding in 'base64_bytes'
    base64_bytes = base64.b64encode(concat_data_v2_bytes)
    base64_bytes_str = "base64_bytes"
    printData(base64_bytes, base64_bytes_str)
```

```
    # Send the final message to Server
    client.send(base64_bytes)

    print("\nPGP On Sender Side is Done.\n")
    ### PGP On Sender Side is Done ###

    if message == "Exit":
        break

client.close()
```

server.py

```
'''


    Author = Berke Kocadere
    Date = 25.06.2022
    Version = 1.0

    This program is a simple Pretty Good Privacy (PGP) Secure Protocol
simulation for Receiver Side. MD5, RSA, ZIP, AES and BASE64 Encoding are
used. Work flow of the program is shown below.

    Receiver side
    1. Receiver receives data from Sender.
    2. Data is BASE64 decoded.
    3. Decoded data is splitted to encrypted one-time message key and
encrypted zipped data.
    4. Encrypted one-time message key is decrypted with RSA using Receiver
Private Key.
    5. Encrypted zipped data is decrypted with AES using one-time message key.
    6. Zipped data is unzipped and splitted to encrypted MD5 Hashed value and
Original message plain text.
    7. Encrypted MD5 Hashed Value is decrypted with RSA using Receiver Private
Key.
    8. Original message Plain text is hashed with same MD5 algorithm and
compared with MD5 Hashed Value. If it's the same, PGP Protocol was succesful.


'''

# Imports
import socket
import sys
import base64
import traceback
import hashlib
import zlib
from ast import literal_eval
from Crypto.PublicKey import RSA
```

```
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.Hash import MD5



'''
   Function : RSA Key Generator

   Description :
      This function creates RSA private and public key for Client and
      store them in file system as PEM format.

   Return : key

'''
def RSA_keyGen():
   # RSA Private and Public Key Generation for Client
   key = RSA.generate(2048)

   private_key = key.export_key()
   file_out = open("server_private.pem", "wb")
   file_out.write(private_key)
   file_out.close()

   public_key = key.publickey().export_key()
   file_out = open("server_public.pem","wb")
   file_out.write(public_key)
   file_out.close()

   return key



'''
   Function : TCP Server Connection

   Description :
      This function creates a TCP Server.

    Return : server

'''
def serverConnection():
   # Client connection
   server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

   host = "127.0.0.1"
   port = 15000

   server.bind((host, port))
   server.listen(1)

   return server
```

```
'''
   Function : Print Data

   Description :
      This function is used for printing length, type and content of a
variable.

   Return : None

'''
def printData(data, dataString):
   print("\n\n")
   print("\n{} len : {}\n".format(dataString,len(data)))
   print("\n{} type : {}\n".format(dataString,type(data)))
   print("\n{} : \n".format(dataString))
   print(data)



# RSA Private and Public Key Generation for Server
key = RSA_keyGen()
# Server Connection
server = serverConnection()

# Wait for a client connection
while True:
   print("Waiting for a connection...")
   try:
      connection, client_address = server.accept() # Connection accepted
      while True:
         print("Client connected : " , str(client_address))

         # Receive data from client in 'dataFromClient'
         dataFromClient = connection.recv(3096).decode()
         dataFromClient_str = "dataFromClient"
         printData(dataFromClient, dataFromClient_str)

         # RSA Encryption Setup
         cipher_RSA = PKCS1_OAEP.new(key)

         # Encode data in 'data_bytes'
         data_bytes = dataFromClient.encode("ascii") # data_bytes =
base64_bytes
         data_bytes_str = "data_bytes_str"
         printData(data_bytes, data_bytes_str)

         # Decode data with Base 64 Decoding in 'concat_data_v2_bytes'
         concat_data_v2_bytes = base64.b64decode(data_bytes)
         concat_data_v2_bytes_str = "concat_data_v2_bytes"
         printData(concat_data_v2_bytes, concat_data_v2_bytes_str)

         # Decode concatenated data in 'concat_data_v2'
         concat_data_v2 = concat_data_v2_bytes.decode("ascii")
         concat_data_v2_str = "concat_data_v2"
         printData(concat_data_v2, concat_data_v2_str)

         # Split concatenated data into list in 'concat_data_v2_list'
```

```
        concat_data_v2_list = concat_data_v2.split("!!!")
        concat_data_v2_list_str = "concat_data_v2_list"
        printData(concat_data_v2_list, concat_data_v2_list_str)

        # Get Encrypted zipped data in 'ciphertext_string'
        ciphertext_string = concat_data_v2_list[0]
        ciphertext_string_str = "ciphertext_string"
        printData(ciphertext_string, ciphertext_string_str)

        ciphertext = literal_eval(ciphertext_string)
        ciphertext_str = "ciphertext"
        printData(ciphertext, ciphertext_str)

        # Get Encrypted session key in 'enc_session_key_string'
        enc_session_key_string = concat_data_v2_list[1]
        enc_session_key_string_str = "enc_session_key_string"
        printData(enc_session_key_string, enc_session_key_string_str)


        enc_session_key = literal_eval(enc_session_key_string)
        enc_session_key_str = "enc_session_key"
        printData(enc_session_key, enc_session_key_str)

        # Get nonce value in 'nonce_string'
        nonce_string = concat_data_v2_list[2]
        nonce_string_str = "nonce_string"
        printData(nonce_string, nonce_string_str)

        nonce = literal_eval(nonce_string)
        nonce_str = "nonce"
        printData(nonce, nonce_str)

        # Encrypted session key is decrypted with RSA using Server's Private
Key in 'session_key'
        session_key = cipher_RSA.decrypt(enc_session_key)
        session_key_str = "session_key"
        printData(session_key, session_key_str)

        # AES Encryption Setup
        cipher_AES = AES.new(session_key, AES.MODE_EAX, nonce)

        # Decrypt Encrypted zipped data with RSA using session key in
'zip_data'
        zip_data = cipher_AES.decrypt(ciphertext)
        zip_data_str = "zip_data"
        printData(zip_data, zip_data_str)

        # Decompress zipped data in 'concat_data_bytes'
        concat_data_bytes = zlib.decompress(zip_data)
        concat_data_bytes_str = "concat_data_bytes"
        printData(concat_data_bytes, concat_data_bytes_str)

        # Decode concatenated data in 'concat_data'
        concat_data = concat_data_bytes.decode("ascii")
        concat_data_str = "concat_data"
        printData(concat_data, concat_data_str)
```

```
        # Split concatenated data into list in 'concat_data_list'
        concat_data_list = concat_data.split("!!!")
        concat_data_list_str = "concat_data_list"
        printData(concat_data_list, concat_data_list_str)

        # Plain text message in 'plain_text'
        plain_text = concat_data_list[1]
        plain_text_str = "plain_text"
        printData(plain_text, plain_text_str)

        # Hashed message 'data_RSA_text'
        data_RSA_text = concat_data_list[0]
        data_RSA_text_str = "data_RSA_text"
        printData(data_RSA_text, data_RSA_text_str)

        data_RSA = literal_eval(data_RSA_text)
        data_RSA_str = "data_RSA"
        printData(data_RSA, data_RSA_str)

        # Decrpyt Encrypted hashed value with RSA using Server's Public Key
        data_md5_bytes = cipher_RSA.decrypt(data_RSA)
        data_md5_bytes_str = "data_md5_bytes"
        printData(data_md5_bytes, data_md5_bytes_str)

        # Encode plaintext for later use in 'plain_text_bytes'
        plain_text_bytes = plain_text.encode("ascii")
        plain_text_bytes_str = "plain_text_bytes"
        printData(plain_text_bytes, plain_text_bytes)

        # Hash message with MD5
        plaintext_md5 = hashlib.md5(plain_text_bytes)
        plaintext_md5_bytes = plaintext_md5.digest()
        plaintext_md5_bytes_str = "plaintext_md5_bytes"
        printData(plaintext_md5_bytes, plaintext_md5_bytes_str)

        # Compare received hashed value and received plain text hashed with
same algorithm
        if data_md5_bytes == plaintext_md5_bytes:

            ### PGP On Receiver Side is Done ####
            print(" PGP OPERATION SUCCESFUL ! ")
            print(" Client has sent the message : \n")
            print(plain_text)

        elif dataFromClient == "Exit":
           break
        else:
           ### PGP OPERATION FAILED  ###
           print(" PGP OPERATION FAILED ! ")
           continue
   except:
      traceback.print_exc()
```