



SECURE EMAIL WITH KEY DISTRIBUTION

Final Project

ABSTRACT

Secure email exchange with an end-to-end encryption using a key distribution center

CSE451 – Computer and Network Security

Team Members

REDA MOHSEN REDA	18P5141
NOOR EL DIN KHALED	18P5722

Problem Statement

To Achieve a secure email exchange with an end-to-end encryption using a key distribution center.

- The sender needs to send an encrypted message (M) as an attachment (.txt file) to the receiver.
- The receiver should decrypt the message (M) (.txt file) obtained from the sender via email.
- The sender and receiver should use the same secret key (session key) for encryption and decryption of the message.

Solution

The key distribution server (KDS) has the secret key of the sender and the receiver and when it gets a request from the sender it can generate a random key (K_s) of 128 bits. And make two encrypted keys, the first encrypted key is encrypted by using the secret key of the sender using $AES-128_Encryption_Function(K_{sender}, K_s)$ where k_{sender} is the secret key of the sender and the other encrypted key is for the receiver, it is encrypted using the secret key of the receiver using $AES-128_Encryption_Function(K_{receiver}, K_s)$. Then, the key distribution server (KDS) sends as a response to the sender request the two encrypted keys.

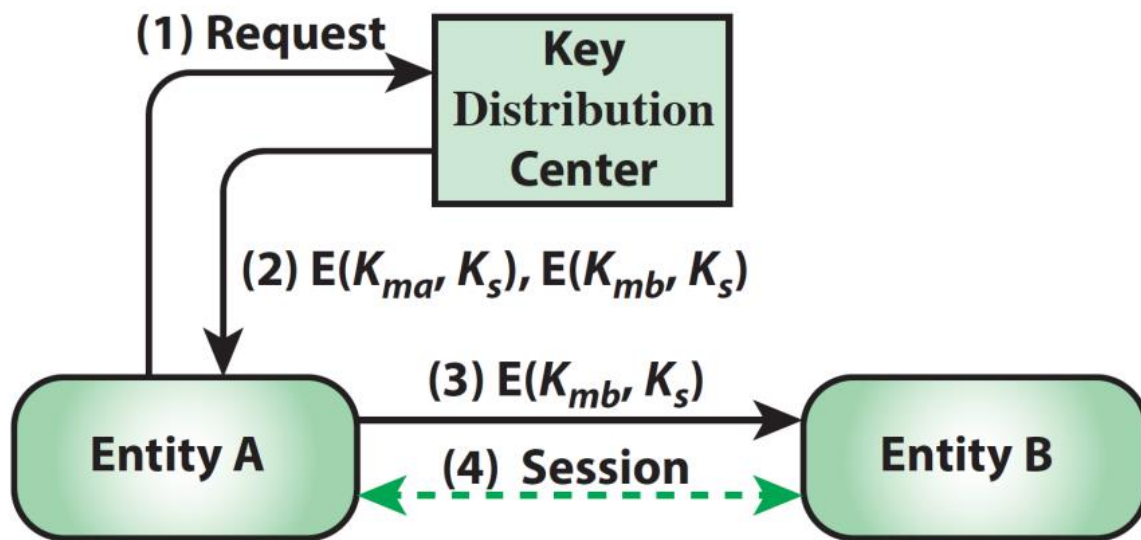
- The sender decrypts the encrypted key obtained from the KDS with its secret key and uses this decrypted key for encryption of the message body M.
- The sender sends the encrypted message body and the other encrypted key (secret key of receiver, session key) obtained from the KDS to the receiver via email attachment.
- The receiver can receive the email and decrypts the session key with its secret key and now the sender and the receiver are having the same session key.
- The receiver can decrypt now the message body using the session key.

Methodology and Design

To achieve the above solution, the following methodology needs to be followed:

File Storage:

1. A separate server is implemented to work as follows:
 - a. Each user is registered on the server with his email.
 - b. Each user has a master 128-bit secret key only known to the user and the KDS.
 - c. The KDS when receives a request, it generates a key and sends 2 different copies of the key to the requesting end, each copy is encrypted by the master key of the requester and the recipient respectively.



2. When the sending client is done with the key, it just puts it as the second attachment (that is message 3 in the above figure).

Implementation

Encrypt_and_Decrypt.py

Import libraries.

```
Encrypt_and_Decrypt.py x Sender_App.py x KDS.py x Receiver_Decryption.py x
1 import os
2 from Cryptodome.Random import get_random_bytes
3 from Cryptodome.Cipher import AES
```

Encryption function used to encrypt .txt files.

```

def encrypt_file(key, in_filename, out_filename=None, chunksize=16):
    """
    Encrypts a file using AES (CBC mode) with the given key.

    key: The encryption key - a string that must be either 16, 24, or 32 bytes long. Longer keys are more secure.
    in_filename: Name of the input file.
    out_filename: If None, '<in_filename>.enc' will be used.
    chunksize: Sets the size of the chunk which the function uses to read and encrypt the file.
    Larger chunk sizes can be faster for some files and machines. chunksize must be divisible by 16.
    """
    if not out_filename:
        out_filename = in_filename + '.enc'

    encryptor = AES.new(key, AES.MODE_ECB)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)
                outfile.write(encryptor.encrypt(chunk))

```

Decryption function used to decrypt .txt files.

```

33 def decrypt_file(key, in_filename, out_filename=None, chunksize=16):
34     """
35     Decrypts a file using AES (CBC mode) with the given key.
36
37     Parameters are similar to encrypt_file, with one difference: out_filename, if not supplied,
38     will be in_filename without its last extension
39     (i.e. if in_filename is 'aaa.zip.enc' then out_filename will be 'aaa.zip').
40     """
41     if not out_filename:
42         out_filename = os.path.splitext(in_filename)[0]
43
44     with open(in_filename, 'rb') as infile:
45         decryptor = AES.new(key, AES.MODE_ECB)
46         with open(out_filename, 'wb') as outfile:
47             while True:
48                 chunk = infile.read(chunksize)
49                 if len(chunk) == 0:
50                     break
51                 outfile.write(decryptor.decrypt(chunk))
52

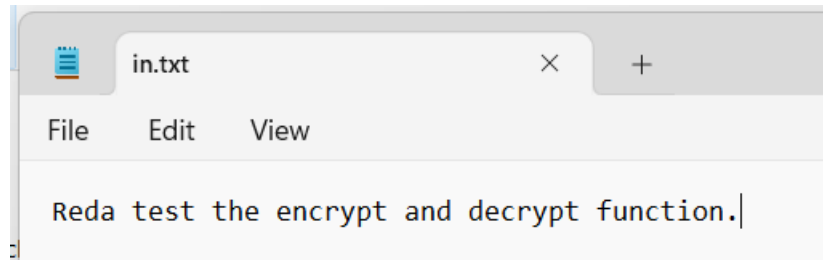
```

Test cases of Encrypt and Decrypt functions

Test the two functions with a dumb in.txt file.

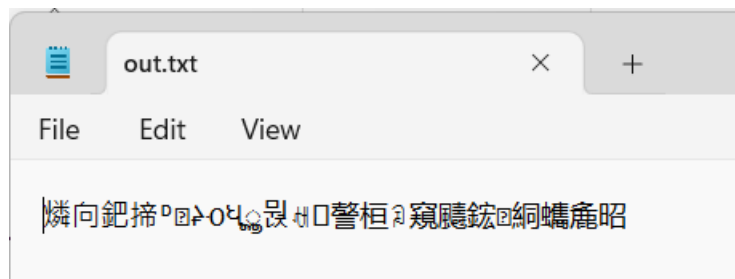
```
54 kk = b'\xce\xcd~@\x10\xee3:JW\x99\xda\xe2\x02'  
55 encrypt_file(kk, 'in.txt', 'out.txt')  
56 decrypt_file(kk, 'out.txt', 'dec.txt')
```

in.txt file (real message body).



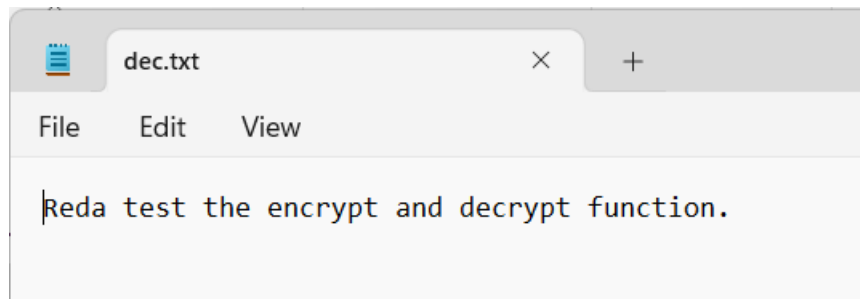
Testing the encrypt_file() function

out.txt file (encrypted message).



Testing the decrypt_file() function

dec.txt file (decrypted message).



Generate two random keys to be used as a secret keys in the sender and receiver.

```

58 # Generate two random keys to be used as a secret keys in sender and receiver sides.
59 for i in range(2):
60     Random_Key = get_random_bytes(16)
61     print(Random_Key)
62

```

The secret key of sender is: `b'\xcek\xacd~@\x10\xee3:JW\x99\xda\xe2\x02'`

The secret key of receiver is: `b'\xa1\x1c\xa5%\xe3?[n\xa3J\x9d\xa3z(\x06\x98'`

```

Run: Encrypt_and_Decrypt (1) x
D:\Privacy\ASU_Spring23\Semester10\Security\Project\security\Scripts\python.exe
b'\xcek\xacd~@\x10\xee3:JW\x99\xda\xe2\x02'
b'\xa1\x1c\xa5%\xe3?[n\xa3J\x9d\xa3z(\x06\x98'
Process finished with exit code 0

```

The keys are 128-bit (16 bytes) and are generated randomly using `Cryptodome.Random.get_random_bytes`.

KDS.py

The code imports the necessary modules and libraries: `socket` for network communication, `threading` for creating multiple threads, `get_random_bytes` from `Cryptodome.Random` for generating random bytes, and `AES` from `Cryptodome.Cipher` for encryption and decryption using the AES algorithm.

```

1 import socket, threading
2 from Cryptodome.Random import get_random_bytes
3 from Cryptodome.Cipher import AES

```

A `ClientThread` class is defined, which inherits from the `threading.Thread` class. It represents a client connection thread. The class has a `Client_Master_Keys` dictionary that stores email keys as keys and their corresponding secret keys as values.

```

5 class ClientThread(threading.Thread):
6     Client_Master_Keys = {"18P5141@eng.asu.edu.eg": "C\xa9\x8foQw/\xac\xeb\xfc\x9c\xdf%n\xcc",
7                          "18P5722@eng.asu.edu.eg": "x\xeb(\x02vV\xa6\xef\xb9\x00\x08\xa16\x8f\xe8\x82",
8                          "ai.reda.mohsen@gmail.com": "\x0e\x130\x9b\xack\x89\x13s\x02\xaeS\x8f\xf0\x91\x81"
9                      }

```

The `__init__` method is the constructor of the `ClientThread` class. It initializes the instance variables `ip`, `port`, and `csocket` with the provided arguments and prints a message indicating that a new thread has started.

```

11     def __init__(self, ip, port, clientsocket):
12         threading.Thread.__init__(self)
13         self.ip = ip
14         self.port = port
15         self.csocket = clientsocket
16         print("[+] New thread started for ", ip, ":", str(port))

```

The run method is called when the thread starts running. It prints a message indicating the connection details and sends a welcome message to the client. It also initializes the data variable with the string "dummydata".

This part of the code represents the main functionality of the thread. It runs in a loop as long as there is data to process. The iterator variable keeps track of the progress of the communication. In the first condition (iterator == 0), the server receives the sender's ID, retrieves the sender's secret key using Get_Secret_Key(), generates a session key using Generate_Session_Key(), encrypts the session key with the sender's secret key using Encrypt_Key(), and sends the encrypted session key to the client. In the second condition (iterator == 1), the server receives the receiver's ID, retrieves the receiver's secret key using Get_Secret_Key(), encrypts the session key with the receiver's secret key using Encrypt_Key(), and sends the encrypted session key to the client. In the third condition (iterator == 2), the server closes the client socket, indicating that the client has disconnected, and resets the data variable to an empty string to exit the loop.

```

18     def run(self):
19         print("Connection from : ", ip, ":", str(port))
20         clientsock.send("Welcome to the multi-threaded server".encode())
21         data = "dummydata"
22
23         iterator = 0
24         while len(data):
25             if iterator == 0:
26                 Sender_ID = self.csocket.recv(2048).decode()
27                 Sender_Secret_Key = self.Get_Secret_Key(Sender_ID)
28                 Encrypted_Session_Key = self.Generate_Session_Key()
29                 Encrypted_Session_Key = self.Encrypt_Key(Sender_Secret_Key.encode(), Encrypted_Session_Key)
30                 self.csocket.send(Encrypted_Session_Key)
31                 print("Encrypted Sender Session Key Sent:", Encrypted_Session_Key)
32                 iterator += 1
33             elif iterator == 1:
34                 Receiver_ID = self.csocket.recv(2048).decode()
35                 Receiver_Secret_Key = self.Get_Secret_Key(Receiver_ID)
36                 Encrypted_Session_Key = self.Encrypt_Key(Receiver_Secret_Key.encode(), Encrypted_Session_Key)
37                 self.csocket.send(Encrypted_Session_Key)
38                 print("Encrypted Receiver Session Key Sent:", Encrypted_Session_Key)
39                 iterator += 1
40             elif iterator == 2:
41                 self.csocket.close()
42                 print("Client at ", self.ip, " disconnected...")
43                 data = ''

```

These three methods are utility functions used within the ClientThread class. Get_Secret_Key() retrieves the secret key corresponding to the provided email from the Client_Master_Keys dictionary.

Generate_Session_Key() generates a random 16-byte session key using get_random_bytes(). Encrypt_Key() encrypts the provided message using the provided key and AES encryption in ECB mode.

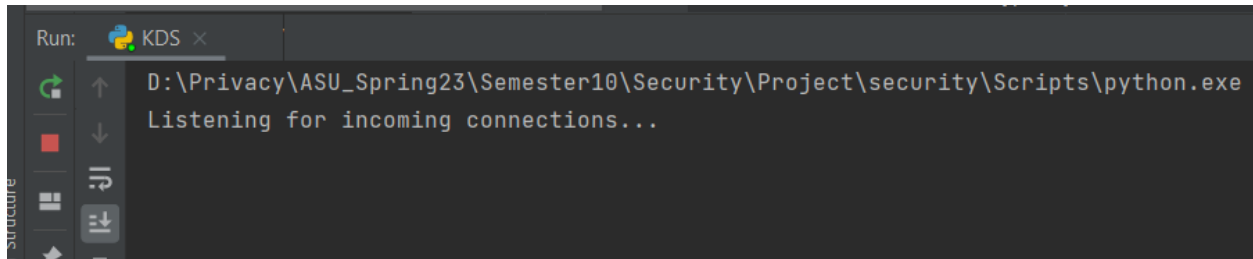
```
44
45     def Get_Secret_Key(self, email):
46         return self.Client_Master_Keys[email]
47
48     def Generate_Session_Key(self):
49         return get_random_bytes(16)
50
51     def Encrypt_Key(self, key, message):
52         AES_Cipher = AES.new(key, AES.MODE_ECB)
53         encrypted_message = AES_Cipher.encrypt(message)
54         return encrypted_message
55
```

This part of the code represents the main server code. It defines the server's host ("0.0.0.0") and port (10000). It creates a TCP socket using socket.socket() and sets a socket option to reuse the address (SO_REUSEADDR) to avoid errors when restarting the server quickly. The socket is then bound to the host and port. The server enters an infinite loop where it listens for incoming connections using tcpsock.listen(). When a client connects, it accepts the connection, creating a client socket and retrieving the client's IP and port. A new ClientThread object is created, passing the client's IP, port, and socket as arguments. Finally, the thread's run() method is called to start the thread's execution. However, note that in the last line of the code (newthread.run()), the run() method is called directly instead of using newthread.start(). This means that the code runs in the same thread instead of creating a new thread. To start the thread properly, newthread.start() should be used instead. Overall, this code represents a server that accepts client connections, communicates with clients to exchange session keys, and performs encryption using AES in ECB mode.

```
57     host = "0.0.0.0"
58     port = 10000
59     tcpsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
60     tcpsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
61     tcpsock.bind((host, port))
62     while True:
63         tcpsock.listen(4)
64         print("Listening for incoming connections...")
65         (clientsock, (ip, port)) = tcpsock.accept()
66         # pass clientsock to the ClientThread thread object being created
67         newthread = ClientThread(ip, port, clientsock)
68         newthread.run()
```

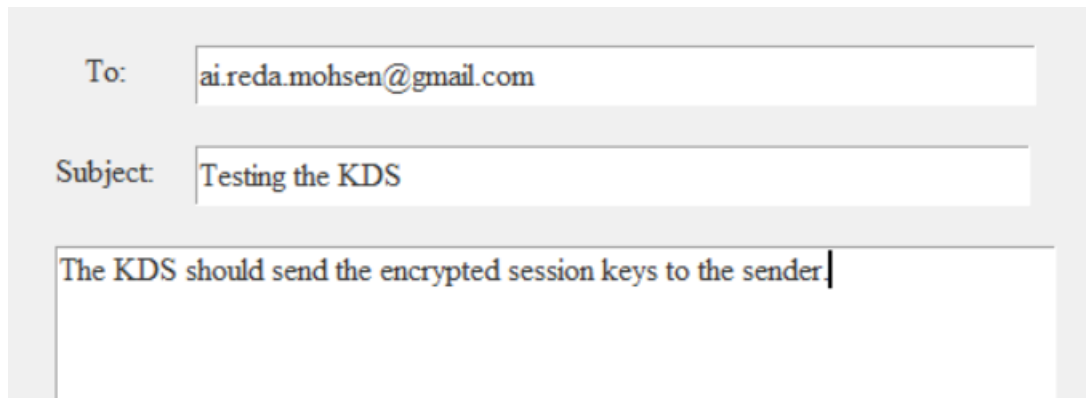

Test cases of the KDS

The KDS before gets any connection, it can listen up to 4 connections.



```
Run: KDS x
D:\Privacy\ASU_Spring23\Semester10\Security\Project\security\Scripts\python.exe
Listening for incoming connections...
```

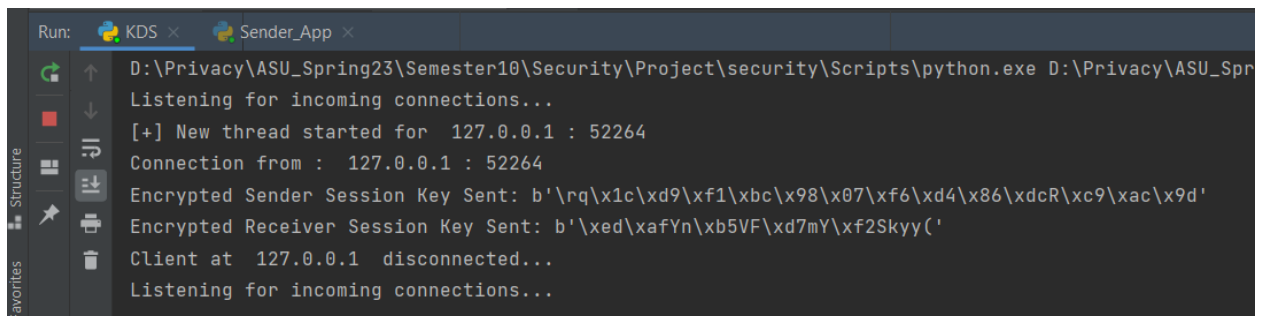
Now, if the sender clicks on send, establishes a connection with the KDS.



To:

Subject:

Here the sender gets the two encrypted session keys from the KDS and can decrypt one encrypted session key using his secret key.



```
Run: KDS x Sender_App x
D:\Privacy\ASU_Spring23\Semester10\Security\Project\security\Scripts\python.exe D:\Privacy\ASU_Spr
Listening for incoming connections...
[+] New thread started for 127.0.0.1 : 52264
Connection from : 127.0.0.1 : 52264
Encrypted Sender Session Key Sent: b'\rq\x1c\xd9\xf1\xbc\x98\x07\xf6\xd4\x86\xdcR\xc9\xac\x9d'
Encrypted Receiver Session Key Sent: b'\xed\xafYn\xb5VF\xd7mY\xf2Skyy('
Client at 127.0.0.1 disconnected...
Listening for incoming connections...
```

Sender_App.py

The code imports the necessary modules and libraries for the application, including tkinter for the GUI, smtplib for sending emails, and Cryptodome for encryption operations.

```
Encrypt_and_Decrypt.py × Sender_App.py × KDS.py × Receiver_Decryption.py ×
1  import tkinter as tk
2      import tkinter.font as tkFont
3      import smtplib
4      from email.mime.text import MIMEText
5      from email.mime.application import MIMEApplication
6      from email.mime.multipart import MIMEMultipart
7      import socket
8      import time
9      from Cryptodome.Cipher import AES
10     import os, random, struct
11     from Encrypt_and_Decrypt import encrypt_file
12
```

Defines a class called App to represent the application. Initializes some class-level variables, such as the email sender address, password, and other variables related to encryption and session keys.

```
13
14     class App:
15         sender = "18P5141@eng.asu.edu.eg"
16         password = "XXXXXXXX" # Enter password before run
17         tovar = ""
18         Sender_Secret_Key = ""
19         Receiver_Secret_Key = ""
20         Session_Key = ""
21
```

The `__init__` method is the constructor of the App class. It sets up the main window of the application with a specified title, size, and position. It also configures the font and other GUI-related properties.

```
21
22     def __init__(self, root):
23         # setting title
24         self.to_var = tk.StringVar()
25         root.title("Secure Mail Composer")
26         # setting window size
27         width = 600
28         height = 500
29         screenwidth = root.winfo_screenwidth()
30         screenheight = root.winfo_screenheight()
```

The `send_email` method is responsible for sending the email with an encrypted body and attachment to the recipient. It establishes a connection with the Key Distribution Server (KDS) to retrieve the encrypted session keys. The `Connect_and_Get_Session_Key_From_KDS` method is called to establish the connection with the KDS and retrieve the session keys. The `attach` parameter is used as a secret key to decrypt the session key. The decrypted session key is stored in `self.Session_Key`. The receiver's encrypted session key is written to a file named `"wrappedkey.txt"`. The receiver's session key is read from the file. The email body is written to a file named `"body.txt"` and encrypted using the received session key. The encrypted email body is read from the file `"EncryptedBodyMessage.txt"`. The email message is created and attached with the encrypted body and session key. The email is sent using an SMTP server.

```
83     def send_email(self, subject, body, attach, receiver):
84         """
85         Sends the email with encrypted body and attachment to the recipient.
86         """
87
88         # Establish connection with the KDS
89         # to get the encrypted session keys
90         self.Connect_and_Get_Session_Key_From_KDS(10000, self.sender, receiver) # localhost:10000
91         decryptor = AES.new(attach.encode('utf-8'), AES.MODE_ECB)
92         # Decrypt the encrypted session key using the secret key of the sender
93         self.Session_Key = decryptor.decrypt(self.Sender_Encrypted_Session_Key)
94         print("Session Key:")
95         print(self.Session_Key)
96
97         # Write the receiver's secret key to a file
98         with open("wrappedkey.txt", "wb") as f:
99             f.write(self.Receiver_Encrypted_Session_Key)
100
101         # Read the receiver's session key from a file
102         with open("wrappedkey.txt", "rb") as f:
103             key = f.read()
104
105         # Write the email body to a file
106         # Encrypt it with the received session key obtained from the KDS
107         with open("body.txt", "wb") as f:
108             f.write(body.encode("utf-8"))
109         encrypt_file(self.Session_Key, "body.txt", "EncryptedBodyMessage.txt")
110         with open("EncryptedBodyMessage.txt", "rb") as f:
111             file_contents = f.read()
112         os.remove("body.txt")
113
114         # Create and attach the email message
115         msg = MIME multipart()
116         msg['Subject'] = subject
117         msg['From'] = self.sender
118         msg['To'] = receiver
119         msg.attach(MIMEText(
120             "The message content is in the [EncryptedBodyMessage.txt]."
121             "You can use [wrappedkey.txt] to Decrypt and read the real message body!",
122             'plain'))
123         part = MIMEApplication(file_contents)
124         part['Content-Disposition'] = f'attachment; filename={os.path.basename("EncryptedBodyMessage.txt")}'
125         msg.attach(part)
126         part = MIMEApplication(key)
127         part['Content-Disposition'] = f'attachment; filename={os.path.basename("wrappedkey.txt")}'
128         part['Content-Disposition'] = 'attachment; filename=wrappedkey.txt'
129         msg.attach(part)
```

```

130
131     # Connect to the SMTP server and send the email
132     smtp_server = smtplib.SMTP("smtp-mail.outlook.com", port=587)
133     print("Connected")
134     smtp_server.starttls()
135     print("TLS OK")
136     smtp_server.login(self.sender, self.password)
137     print("login OK")
138     smtp_server.sendmail(self.sender, receiver, msg.as_string())
139     print("mail sent")
140     smtp_server.quit()

```

The `button_Send_command` method is a callback function for the "Send" button. It retrieves the email details such as the recipient, subject, and body from the GUI fields. The sender's secret key is set to 'C\xa9\x8foQw/\xac\xeb\xfc\x9c\xdf%\$n\xcc'. The `send_email` method is called with the retrieved details.

```

142     def button_Send_command(self):
143         """
144         Callback function for the "Send" button.
145         Retrieves email details and calls send_email method.
146         """
147
148         tovar = self.email_To.get()
149         print(tovar)
150         subject = self.email_Subject.get()
151         body = self.email_Body.get("1.0", "end")
152         Sender_Secret_Key = 'C\xa9\x8foQw/\xac\xeb\xfc\x9c\xdf%$n\xcc'
153         self.send_email(subject, body, Sender_Secret_Key, tovar)

```

The `Connect_and_Get_Session_Key_From_KDS` method establishes a connection with the Key Distribution Server (KDS) on the specified PORT. It sends the sender and receiver IDs to the KDS. It receives the encrypted session keys from the KDS. The received session keys are stored in `self.Sender_Encrypted_Session_Key` and `self.Receiver_Encrypted_Session_Key`. The connection is closed.

```

155 def Connect_and_Get_Session_Key_From_KDS(self, PORT, Sender_ID, Receiver_ID):
156     """
157     Establishes a connection with the Key Distribution Server (KDS)
158     and retrieves encrypted session keys for sender and receiver.
159     """
160
161     # Create a socket object
162     Client_Socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
163
164     # Define the server address
165     Server_Address = ('localhost', PORT)
166
167     Key_Received_Status = False
168
169     try:
170         # Start Connection With Server
171         Client_Socket.connect(Server_Address)
172         print("Connected to PORT", PORT)
173         data = Client_Socket.recv(2048)
174
175         # Send the Emails to the KDS
176         Client_Socket.send(Sender_ID.encode('utf-8')) # Send sender id to KDS
177         time.sleep(1)
178         Client_Socket.send(Receiver_ID.encode('utf-8')) # Send receiver id to KDS
179

```

```

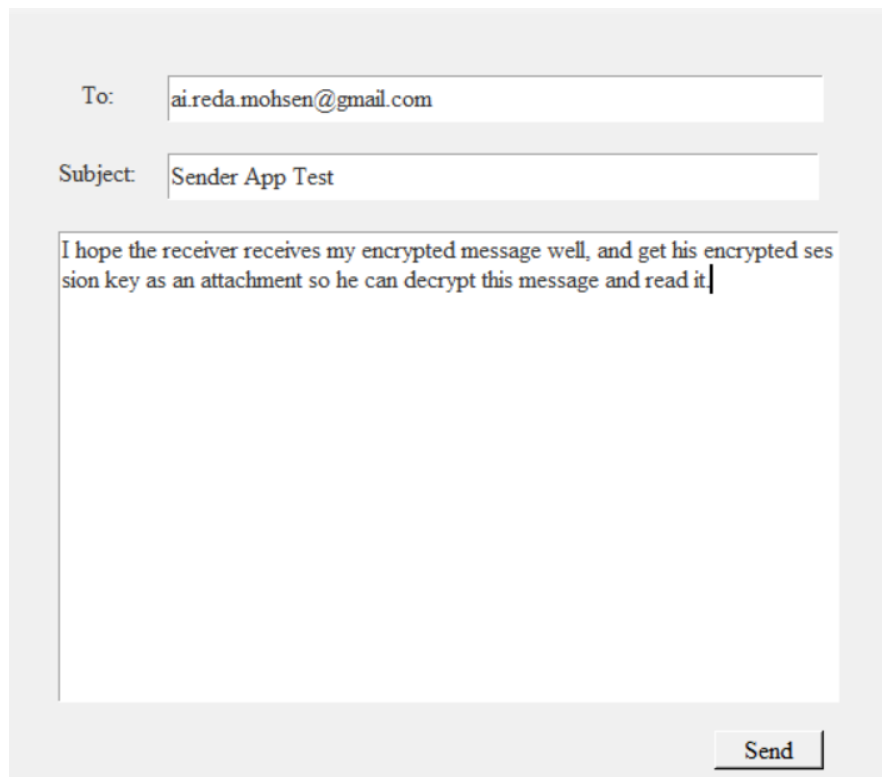
179
180     counter = 0
181     # Receive the encrypted session keys from the KDS
182     while not Key_Received_Status:
183         data = Client_Socket.recv(2048)
184         if counter == 0:
185             print(data)
186             self.Sender_Encrypted_Session_Key = data # Receive sender session key
187             counter += 1
188         elif counter == 1:
189             print(data)
190             self.Receiver_Encrypted_Session_Key = data # Receive receiver session key
191             Key_Received_Status = True
192
193     # Close the Connection
194     Client_Socket.close()
195     print("Connection Closed")
196 except ConnectionRefusedError:
197     print("Connection Error!")
198

```

This code represents a GUI application for composing and sending secure emails. It includes encryption and decryption functionalities using session keys obtained from a Key Distribution Server (KDS). The email body and attachments are encrypted before sending, and the recipient can decrypt and read the messages using the received session key.

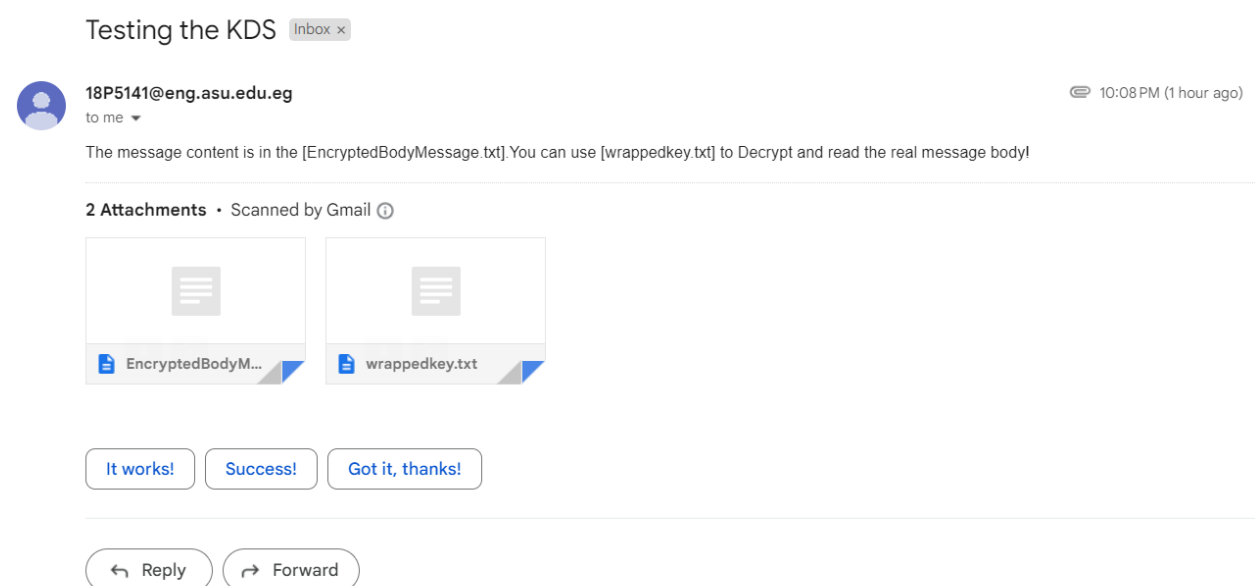
Test Cases of Sender App

Firstly, this GUI appears, the client can write in the To field an email registered in the KDS, any subject, and the message body.



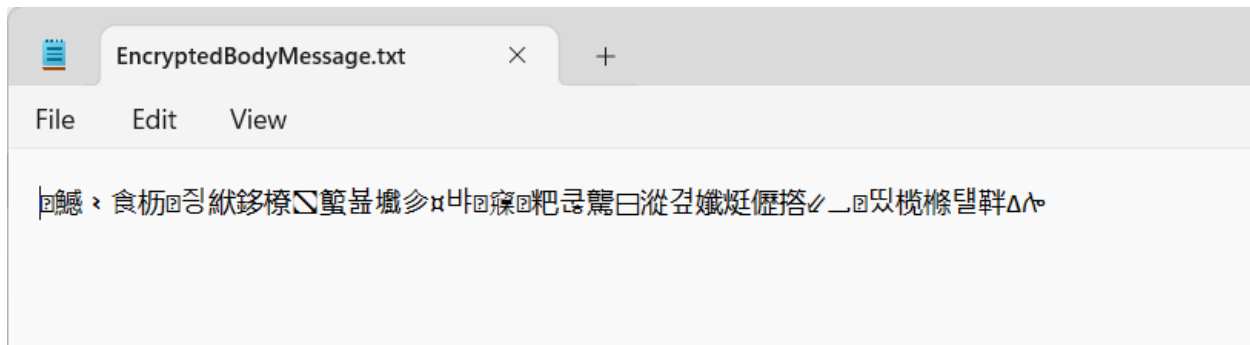
The screenshot shows a web-based form for sending an email. It has three input fields: 'To:' with the value 'ai.reda.mohsen@gmail.com', 'Subject:' with the value 'Sender App Test', and a larger 'Message:' field containing the text 'I hope the receiver receives my encrypted message well, and get his encrypted session key as an attachment so he can decrypt this message and read it'. A 'Send' button is located at the bottom right of the form.

The receiver receives the message successfully with the two attachments. Encrypted message body and encrypted session key, so he can decrypt the session key with his secret key and decrypt the message with the session key.

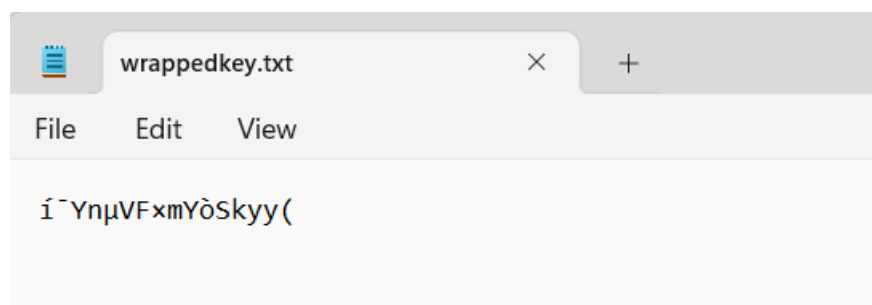


The screenshot shows an email received in a Gmail inbox. The email is titled 'Testing the KDS' and is from '18P5141@eng.asu.edu.eg' to 'me'. The message body states: 'The message content is in the [EncryptedBodyMessage.txt]. You can use [wrappedkey.txt] to Decrypt and read the real message body!'. Below the message body, there are two attachments: 'EncryptedBodyM...' and 'wrappedkey.txt'. At the bottom of the email, there are three buttons: 'It works!', 'Success!', and 'Got it, thanks!'. Below these buttons, there are two buttons: 'Reply' and 'Forward'.

The encrypted body message the receiver receives.

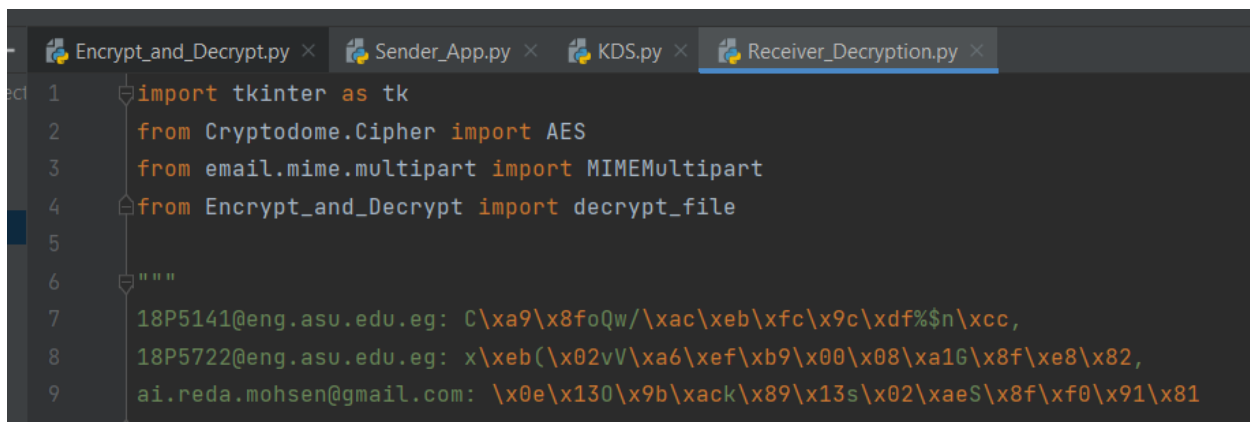


The encrypted session key the receiver receives.



Receiver_Decryption.py

This imports the required libraries for the application: tkinter for GUI, Cryptodome.Cipher for AES encryption and decryption, email.mime.multipart for creating email message objects, and decrypt_file from the custom module Encrypt_and_Decrypt.



This class defines the ReceiverApp class that encapsulates the functionality of the receiver application. The `__init__` method initializes the class attributes: `Session_Key`: It stores the session key used for decryption. It's initially set to an empty string. `Receiver_Secret_Key`: It contains the secret key specific to the receiver. It's a hexadecimal string. `window`: It creates the main window of the application using tkinter's Tk class and sets the window title. `attachments`: It is an empty list to store any attachments associated with the received message. `Do_GUI()` method is called to create the GUI elements. `message`: It creates an instance of MIMEMultipart to handle the email message.

```

12
13 class ReceiverApp:
14     def __init__(self):
15         self.Session_Key = ""
16         self.Receiver_Secret_Key = "\x0e\x130\x9b\xack\x89\x13s\x02\xaeS\x8f\xf0\x91\x81"
17         self.window = tk.Tk()
18         self.window.title("Received Message Decryption")
19
20         self.attachments = []
21         self.Do_GUI()
22
23         self.message = MIME multipart()

```

The Do_GUI method creates the GUI elements for the application: It creates a button labeled "Read Message" and associates it with the read_message method. It creates a text area (output_text) with a height of 10 lines and width of 40 characters. It is initially disabled for editing. It creates a vertical scrollbar and associates it with the output_text area.

```

25     def Do_GUI(self):
26         read_button = tk.Button(self.window, text="Read Message", command=self.read_message)
27         read_button.pack(pady=11)
28
29         self.output_text = tk.Text(self.window, height=10, width=38, state=tk.DISABLED)
30         self.output_text.pack()
31
32         scrollbar = tk.Scrollbar(self.window)
33         scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
34         self.output_text.config(yscrollcommand=scrollbar.set)
35         scrollbar.config(command=self.output_text.yview)

```

The read_message method is called when the "Read Message" button is clicked. It starts by enabling the text area for writing and clearing any existing text. It opens the file "wrappedkey.txt" in binary mode and reads the encrypted session key from it. It creates an AES decryptor object (decryptor) using the receiver's secret key. The decryptor decrypts the encrypted session key, and the resulting session key is stored in the Session_Key attribute. The decrypt_file function is called to decrypt the message body using the session key and save the decrypted message to "DecryptedMessage.txt". The decrypted message is then read from the file and inserted into the text area for display. Finally, the text area is set back to a disabled state.


```

36
37     def read_message(self):
38         self.output_text.config(state=tk.NORMAL)
39         self.output_text.delete(1.0, tk.END)
40         with open("wrappedkey.txt", 'rb') as file:
41             Encrypted_Session_Key = file.read()
42             decryptor = AES.new(self.Receiver_Secret_Key.encode('utf-8'), AES.MODE_ECB)
43             self.Session_Key = decryptor.decrypt(Encrypted_Session_Key)
44             print(self.Session_Key)
45
46         decrypt_file(self.Session_Key, "EncryptedBodyMessage.txt", "DecryptedMessage.txt")
47
48         with open("DecryptedMessage.txt", "rb") as f:
49             decryptedMessage = f.read()
50             self.output_text.insert(tk.END, decryptedMessage)
51             self.output_text.insert(tk.END, "\n\n")
52             self.output_text.config(state=tk.DISABLED)

```

The run method starts the application's main event loop using `mainloop()` to handle user interactions. An instance of the `ReceiverApp` class is created (`app`), and the `run` method is called to start the application.

```

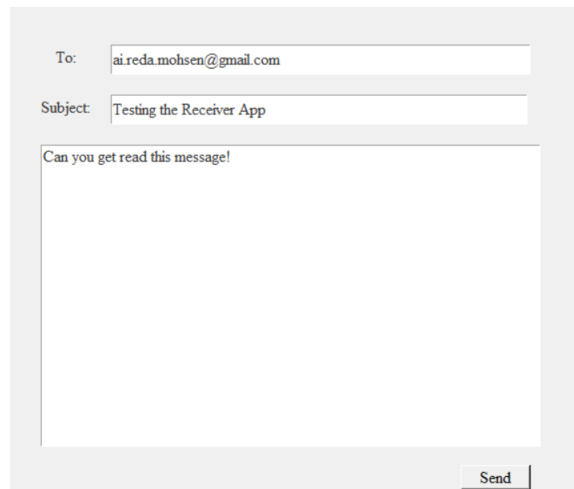
53
54     def run(self):
55         self.window.mainloop()
56
57
58     app = ReceiverApp()
59     app.run()
60

```

This code for decrypting the session key received from the sender email and decrypting the message using the session key.

Test Cases of the Receiver App

The sender sends this email taken from sender app.



Here is the session key, the encrypted sender session key, and the encrypted receiver session key printed from KDS.

Session key = b'\r\xe3\x95\xbd2\x8bvy\xf1\xcf\x97\x0f6\xb3;\x80'

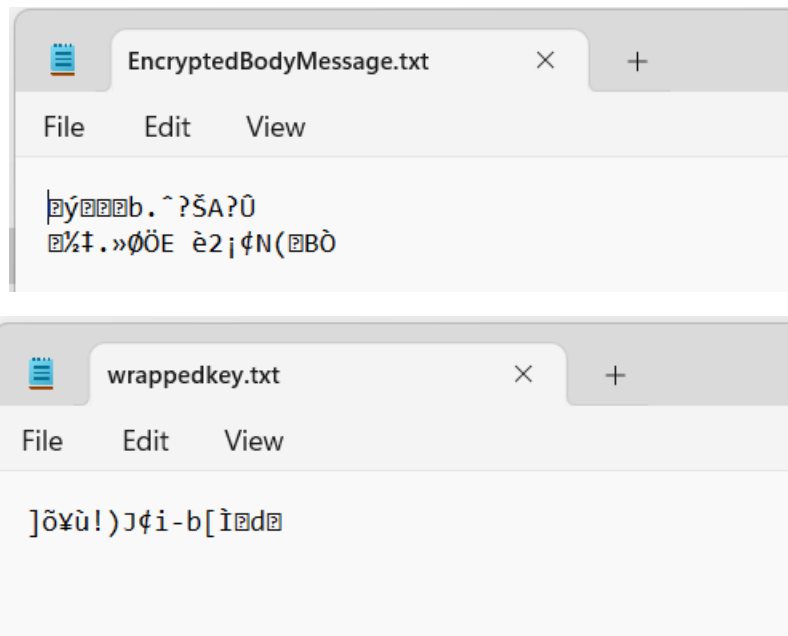
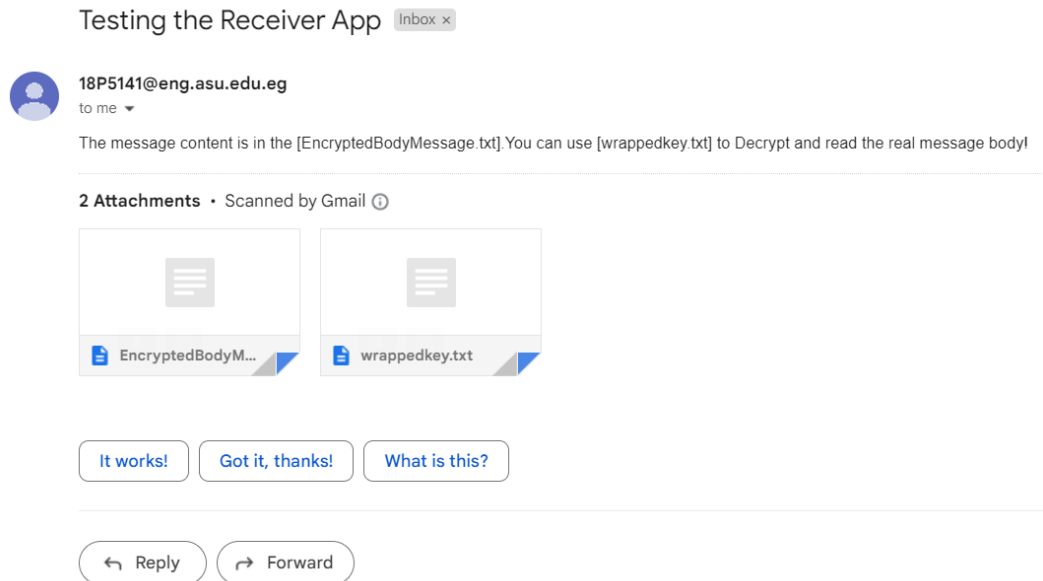
```
[*] New thread started for 127.0.0.1 : 53061
Connection from : 127.0.0.1 : 53061
b'\r\xe3\x95\xbd2\x8bvy\xf1\xcf\x97\x0f6\xb3;\x80'
Encrypted Sender Session Key Sent: b"\xefe\xeb\xcf\x91\xee\x91\xbf\x16$\x9b\x13\xb5\xe3"
Encrypted Receiver Session Key Sent: b']\xf5\xa5\xf9!)J\xa2i-b[\xcc\x1cd\x12'
Client at 127.0.0.1 disconnected...
Listening for incoming connections
```

The session key is received by the sender app and encrypts the message body with it.

Session key = b'\r\xe3\x95\xbd2\x8bvy\xf1\xcf\x97\x0f6\xb3;\x80'

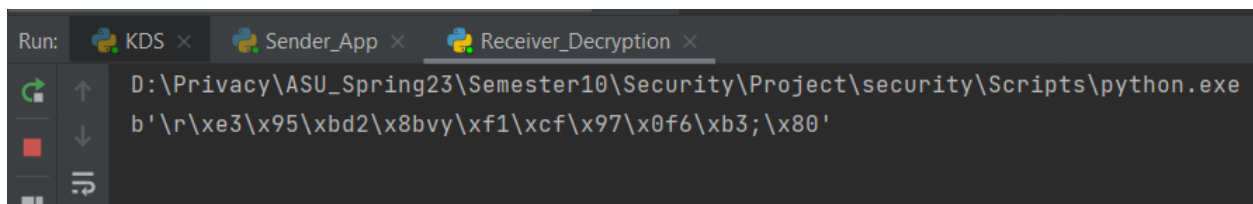
```
Connection Closed
Session Key:
b'\r\xe3\x95\xbd2\x8bvy\xf1\xcf\x97\x0f6\xb3;\x80'
Connected
TLS OK
login OK
mail sent
|
```

The received mail.

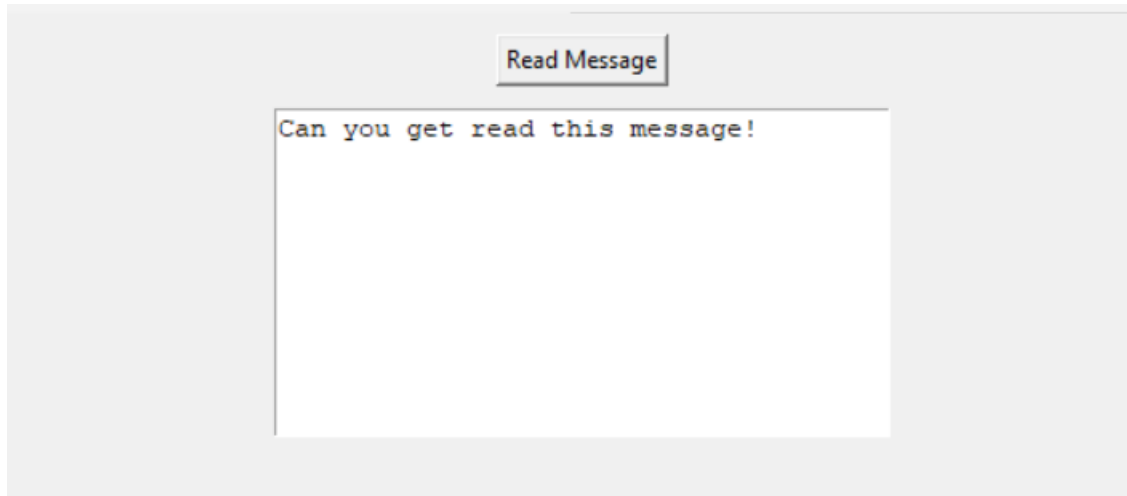


The receiver decrypts the encrypted session key and print the session key.

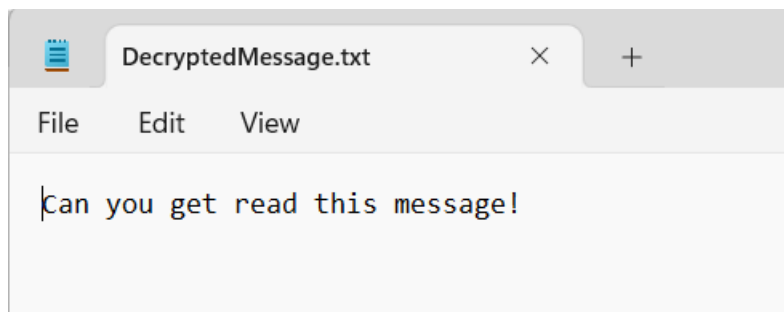
Session key = b'\r\xe3\x95\xbd2\x8bvy\xf1\xcf\x97\x0f6\xb3;\x80'



The receiver can read the message through clicking the Read Message button. The receiver decrypts the encrypted message body using the session key.



The decrypted message the receiver obtained.



GitHub Link

[reda-mohsen/Secure-Email: Achieve a secure email exchange with an end to end encryption using a key distribution \(github.com\)](https://github.com/reda-mohsen/Secure-Email)