

PGP Chat

Written by Varick Erickson (varick.erickson@csueastbay.edu)

PGP was originally designed to be used with SMTP to provide email security, but that doesn't mean that parts of it can't be repurposed for other applications. In this lab, you will be creating a secure chat application that uses a python PGP library. The assignment description uses python version 3.5 and has been verified to run in Windows 10 and Debian.

Chat Client and Server

Two starter files are provided that use sockets to create a clear text based chat client/server:

chat_client_threading.py
chat_server_threading.py

Your job is to modify these programs to use features of a PGP library to provide security.

Installing gnupg

This assignment will use GnuPg, which is an open source implementation of PGP. Python has a "wrapper" written for GnuPg called gnupg that will need to be installed.

gnupg documentation: <https://pythonhosted.org/python-gnupg/>

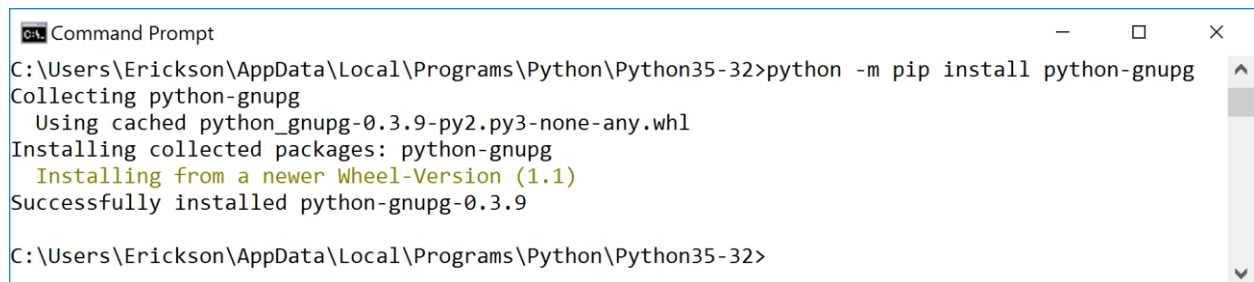
Here are a several different ways to install the gnupg wrapper package:

pip method (Linux): *pip install python-gnupg*

apt-get method (Debian only): *sudo apt-get install python-gnupg*

pip method (Windows see below): *python -m pip install python-gnupg*

Open cmd, and navigate to where python was installed (or add it to your path).



```
Command Prompt
C:\Users\Erickson\AppData\Local\Programs\Python\Python35-32>python -m pip install python-gnupg
Collecting python-gnupg
  Using cached python_gnupg-0.3.9-py2.py3-none-any.whl
Installing collected packages: python-gnupg
  Installing from a newer Wheel-Version (1.1)
Successfully installed python-gnupg-0.3.9
C:\Users\Erickson\AppData\Local\Programs\Python\Python35-32>
```

You will also need to install the following if using windows:

http://ftp.gnupg.org/gcrypt/binary/gnupg-w32-2.1.4_20150512.exe

Creating a gnupg object

```
# Creates a gnupg object using a directory. If there is a keyring in  
# the directory, it will import the keyring. If there is nothing in  
# directory, then it will just create gnupg object with an empty  
# keyring.
```

```
import gnupg  
key_home = 'WhereYouWantYourKeyDirectory'  
gpg = gnupg.GPG(gnupghome=key_home)
```

```
# Creates an gnupg object in current directory.
```

```
import gnupg  
gpg = gnupg.GPG()
```

Generating a Public/Private Key Pair

```
# NOTE: The gen_key will seem to hang on linux, but it will eventually  
# finish. It uses keyboard and clicks to initialize so feel free to  
# surf the web while you wait to help it along.
```

```
key_name = 'YourNameHere'  
key_email = 'SomeEmail@SomeDomain.com'  
rsa_default = 'RSA'  
key_type = '2048'  
key_information = gpg.gen_key_input(name_real=key_name,  
name_email=key_email, key_type=rsa_default, key_length=key_type)  
gpg.gen_key(key_information)
```

Getting keyid and fingerprint Information

The `list_keys()` method returns a list of all the keys available (public and private). Each element of the list has attributes that can be accessed (see output of `"print(gpg.list_keys()[0])"`)

```
print(gpg.list_keys())           # Shows the public keys  
print(gpg.list_keys(True))      # Shows the private keys  
  
print(gpg.list_keys()[0])       # Details of first public key  
keyid = gpg.list_keys()[0]['keyid'] # Shows first keyid  
fingerprint = gpg.list_keys()[0]['fingerprint'] # First fingerprint  
  
for key in gpg.list_keys():      # Loops through a key list  
    print(key['keyid'])
```

Importing a Public Key from a public PGP Key Server

The following shows how to import a public key from a public PGP key server and add it to the keyring.

```
# Looks up a public key from pgp.mit.edu using a keyid. If it exists,  
# then it will be added to the gpg.list_keys()  
result = gpg.recv_keys('pgp.mit.edu','64301BD8')
```

Publishing a Public Key to a public PGP Key Server

The following code shows how you can use `gnupg` to create a key that can be published on a public PGP key server.

```
keyids = gpg.list_keys()[0]['keyid'] # First Public key on ring
ascii_armored_public_keys = gpg.export_keys(keyids)
print(ascii_armored_public_keys)
```

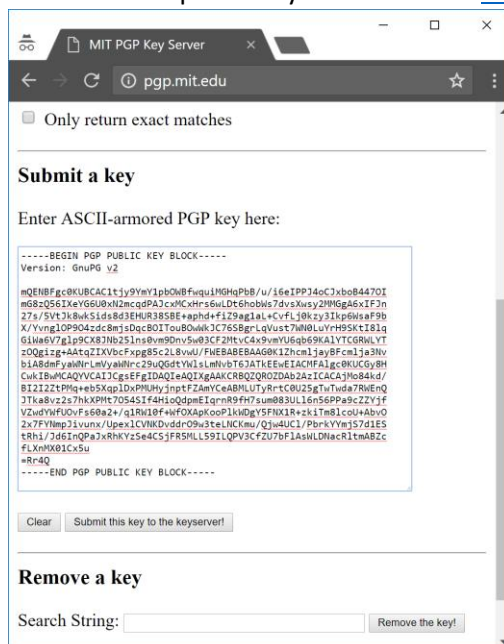
This will print something that looks like the following:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQENBFgc0KUBCAC1tjy9YmY1pbOWBfwquIMGHqPbB/u/i6eIPPJ4oCJxboB447OI
mG8zQ56IXeYG6U0xN2mcqdPAJcxMCxHrs6wLdt6hobWs7dvsXwsy2MMGgA6xIFJn
27s/5VtJk8wkSids8d3EHUR38SBE+aphd+fiZ9ag1aL+CvflJ0kzy3Ikp6WsaF9b
X/YvnglOP904zdc8mjsDqcBOITouBOwWkJC76SBgrLqVust7WN0LuYrH9SKtI8lq
GiWa6V7glp9CX8JNb25lms0vm9Dnv5w03CF2MtvC4x9vmYU6qb69KAlYTCGRWLYT
zOQgizg+AAAtqZIXVbcFxp85c2L8vwU/FWEBAEBEBAAG0K1Zhcm1jayBFcm1ja3Nv
biA8dmFyaWNrLmVyaWNrc29uQGdtYWlsLmNvbT6JATkEEWEIACMFAlgc0KUCGy8H
CwkIBwMCAQYVCAIJCgsEFgIDAQIEAQIXgAAKCRBQZQROZDab2AzICACajMo84kd/
BI2I2ZtPMq+eb5XqplDxPMUHyjnpTFZAmYCeABMLUTyRrtC0U25gTwTwda7RWEHQ
JTka8vz2s7hkXPmt7054SI4HioQdpmEIqrnR9fH7sum083UL16n56PPa9cZYZjf
VZwdYWFuOvFs60a2+/q1RW10f+WfOXApKooPlkWDgY5FNX1R+zkiTm8lcoU+AbvO
2x7FYNmpJivunx/UpexlCVNKDvddrO9w3teLNCKmu/Qjw4UC1/PbrkYYmjS7d1ES
tRhi/Jd6InQPaJxRhKYzSe4CSjFR5MLL59ILQPv3CfZU7bFlAsWLDNacRltmABZc
fLXnMX01Cx5u
=Rr4Q
-----END PGP PUBLIC KEY BLOCK-----
```

This key can be entered into a public key server such as <http://pgp.mit.edu>

If using windows, be mindful that windows uses `\r\n` as a newline rather than just `\n` in certain text editors. If you try “pasting” in a key with `\r\n` into the site, it will not accept it as a valid key.



MIT PGP Key Server

pgp.mit.edu

☐ Only return exact matches

Submit a key

Enter ASCII-armored PGP key here:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQENBFgc0KUBCAC1tjy9YmY1pbOWBfwquIMGHqPbB/u/i6eIPPJ4oCJxboB447OI
mG8zQ56IXeYG6U0xN2mcqdPAJcxMCxHrs6wLdt6hobWs7dvsXwsy2MMGgA6xIFJn
27s/5VtJk8wkSids8d3EHUR38SBE+aphd+fiZ9ag1aL+CvflJ0kzy3Ikp6WsaF9b
X/YvnglOP904zdc8mjsDqcBOITouBOwWkJC76SBgrLqVust7WN0LuYrH9SKtI8lq
GiWa6V7glp9CX8JNb25lms0vm9Dnv5w03CF2MtvC4x9vmYU6qb69KAlYTCGRWLYT
zOQgizg+AAAtqZIXVbcFxp85c2L8vwU/FWEBAEBEBAAG0K1Zhcm1jayBFcm1ja3Nv
biA8dmFyaWNrLmVyaWNrc29uQGdtYWlsLmNvbT6JATkEEWEIACMFAlgc0KUCGy8H
CwkIBwMCAQYVCAIJCgsEFgIDAQIEAQIXgAAKCRBQZQROZDab2AzICACajMo84kd/
BI2I2ZtPMq+eb5XqplDxPMUHyjnpTFZAmYCeABMLUTyRrtC0U25gTwTwda7RWEHQ
JTka8vz2s7hkXPmt7054SI4HioQdpmEIqrnR9fH7sum083UL16n56PPa9cZYZjf
VZwdYWFuOvFs60a2+/q1RW10f+WfOXApKooPlkWDgY5FNX1R+zkiTm8lcoU+AbvO
2x7FYNmpJivunx/UpexlCVNKDvddrO9w3teLNCKmu/Qjw4UC1/PbrkYYmjS7d1ES
tRhi/Jd6InQPaJxRhKYzSe4CSjFR5MLL59ILQPv3CfZU7bFlAsWLDNacRltmABZc
fLXnMX01Cx5u
=Rr4Q
-----END PGP PUBLIC KEY BLOCK-----
```

Remove a key

Search String:

Interacting with a Public PGP Key Server

The following code looks up a key from a public PGP key server and adds it to the gpg keyring

```
gpg.recv_keys('pgp.mit.edu',keyid)    # keyid or fingerprint will work
```

Encrypting/Decrypting using Public Private Keys

The following encrypts a string with a public key:

```
msgE = gpg.encrypt("hello world", '214AED1760AD488C', always_trust=True)

if (msgE.ok):
    print(msgE.data.decode('utf-8'))    # Prints the encrypted data
```

To decrypt the message using a private key, you use the following:

```
msgD = gpg.decrypt(msgE.data)    # Automatically figures out key to use
if (msgD.ok):
    msg = msgD.data.decode('utf-8')
    print(msgD.key_id)            # Prints out the keyid of sender
    print(msg)                    # Prints the decrypted data
```

Symmetric Encryption/Decryption

In order to use symmetric encryption/decryption, you need use a passphrase:

```
msgE = gpg.encrypt("hello world", recipients=[], symmetric="AES256",
passphrase="happy")

msgD = gpg.decrypt(msgE.data, passphrase="happy")
msg = msgD.data.decode('utf-8')
print(msg)
```

Chat Client Behavior

When the chat client is started, it should create the gpg object using the current directory (key_home = './'). If the keyring is empty, then the program should prompt the user for a name and email and display an armored public key suitable for entering into a PGP key server (see previous section for details).

It is assumed that the client already knows the public keyid of the server and can use recv_keys method to ensure the server key is in the keyring.

When the client connects to the server, the first message sent to the server should contain the desired username along with the keyid using a ':' as a delimiter:

```
username+": "+keyid
```

The server will reply back with a session passphrase encrypted with the client public key. Messages should be sent using symmetric encryption using this session passphrase. When messages are received, then the passphrase should be used to decrypt the messages.

Chat Server Behavior

- When the server is started, it should start with one public/private key pair it owns. If it does not have a public/private key pair, it should generate one in a similar way as the client.
- The server should prompt for a session passphrase at initialization
- At initialization, chat server should be given a list of keyid's that are authorized to use the chat server (this can be a list passed to the ChatServer object).
- For each of these keyid's, the server should
 - The server should look up the key from pgp.mit.edu using the recv_key method to add it to the current keyring (ie the object gpg).

When the server receives a message from a client

- The server should attempt to decrypt the message using the session passphrase
- If the message is successfully decrypted, then the message should be broadcast to all the other clients connected to the server

A Note to Window's Users

The IDLE editor does not handle threading properly when the method input("") is use. In this case, it will actually block the receive thread and no messages will be received until a message is entered. The easiest way around this is to run chat_client_threading.py in a cmd window rather than directly in IDLE.

Deliverables

- chat_client_threading.py written to the specifications described
- chat_server_threading.py written to the specifications described