

Cristian Cortez

ID: if2482

CS 471: Security & Info Assurance

Assignment 2

Abstract

In this assignment, we experiment with encryption, hashing, packet capture and analysis using the tools gpg, netcat, wireshark, steghide, and md5sum. The purpose of this assignment is to understand the fundamentals of encryption and hashes and how each provide the X.800 Security Services of authentication, access control, data confidentiality, data-integrity and non-repudiation. As a result; gpg provides Confidentiality, Authentication and non-repudiation; netcat provides weak non-repudiation; steghide provides confidentiality and data-integrity; md5sum provides data-integrity.

Introduction

Within a kali linux vm, gpg will be used to symmetrically and asymmetrically encrypt a plaintextfile. Gpg will also be used to sign the encrypted message with a private key to authenticate us as the encryptor. netcat will be used to send the ciphertext file and receive and save it within another file. This connection will be monitored with wireshark where a packet analysis will later prove how that data is transferred over a network connection. steghide will be used to embed the plaintext file within an image downloaded from the internet. These image files (the original and the one with the embedded message), will be checked with md5sum. md5sum will calculate these hash values for these image files to prove that data was modified, breaking the data-integrity between them.

Commands used:

NIX GENERAL

```
// create a file with a message
$ echo "[MESSAGE]" >> [PLAINTEXT FILE].txt

// create a file with nano
$ nano [PLAINTEXT FILE].txt

//change permission on a file
$ chmod 600 [FILE].txt

// rename a f1 to f2
mv [FILE1].txt [FILE2].txt

// Display contents of a file
$ cat [FILE].txt
```

```
//Display manual for a command
$ man [COMMAND NAME]

// Download an image jpeg from a URL
$ wget [IMAGE URL].jpeg && cp [IMAGE URL NAME].jpeg image.jpg
```

GPG

```
// ENCRYPTION:

// Symmetric encryption: binary output
$ gpg --symmetric [PLAINTEXT FILE]

// Symmetric encryption: ASCII output (armored)
$ gpg --symmetric -a [PLAINTEXT FILE]

//Decrypt
$ gpg --decrypt [CIPHERTEXT FILE].txt.asc

//Encrypt signed file to the recipient's key
$ gpg -e -a -u "[YOUR NAME]" -r "Christopher" plaintext.txt.asc.sig

// GPG KEYRING:

// Import public key to key ring
$ gpg --import [PUBLIC KEY].pub.key

// List imported keys in local keyring
$ gpg --list-keys

// Export public key in ASCII format (armored)
$ gpg --export -a > [public].key

// GPG KEYGEN

// Create a public/private key pair
$ gpg --gen-key

// GPG SIGNING

// Sign the plaintext with private key
$ gpg -a --output [PLAINTEXT FILE].txt.asc.sig --sign plaintext.txt

//
```

NETCAT

```
// Setup prearranged listener
$ nc -l -p 31337 -q 1 > [CIPHERTEXT FILE].txt.asc < /dev/null

// Send encrypted file to listener
$ cat [CIPHERTEXT FILE].txt.asc | netcat 192.168.86.220 31337
```

STEGHIDE

```
// Install
$ sudo apt-get install steghide

// Embed a plaintext message in an image file
$ steghide embed -cf image.jpg -ef plaintext -sf steg_image.jpg

// Extract plaintext message from an image file
$ steghide extract -sf steg_image.jpg
```

MD5

```
// get md5sum of a file with extension jpg/jpeg
$ md5sum [FILE].jpg
$ md5sum [FILE].jpeg
```

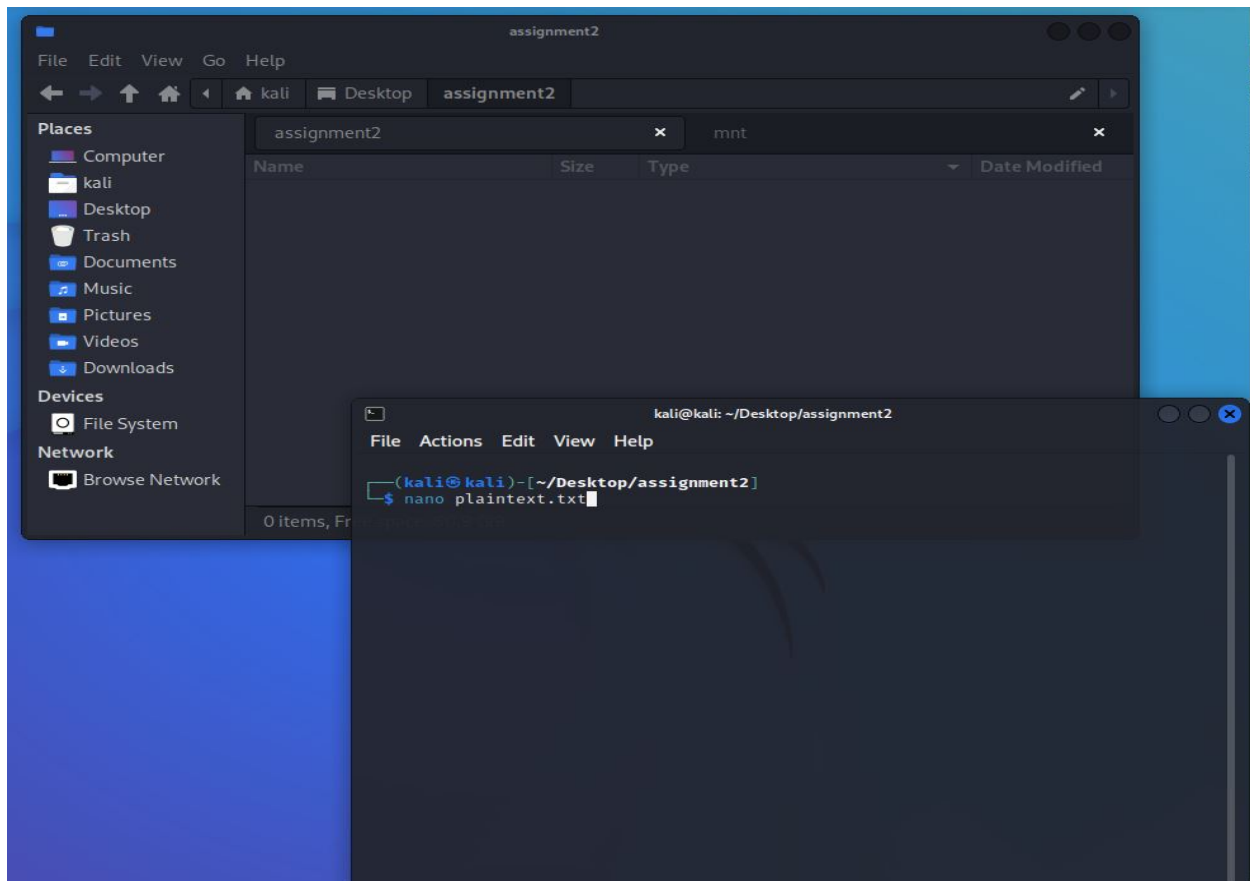
Summary of Results

A: Create a plaintext file

1. **Create a plain text name "*plaintext.txt*" file with the following text:**

```
[YOUR NAME]
[NETID]
[A MESSAGE]
```

Use either **echo** or **nano** to create the file.



Mine looks like:

"plaintext.txt"

Cristian Cortez

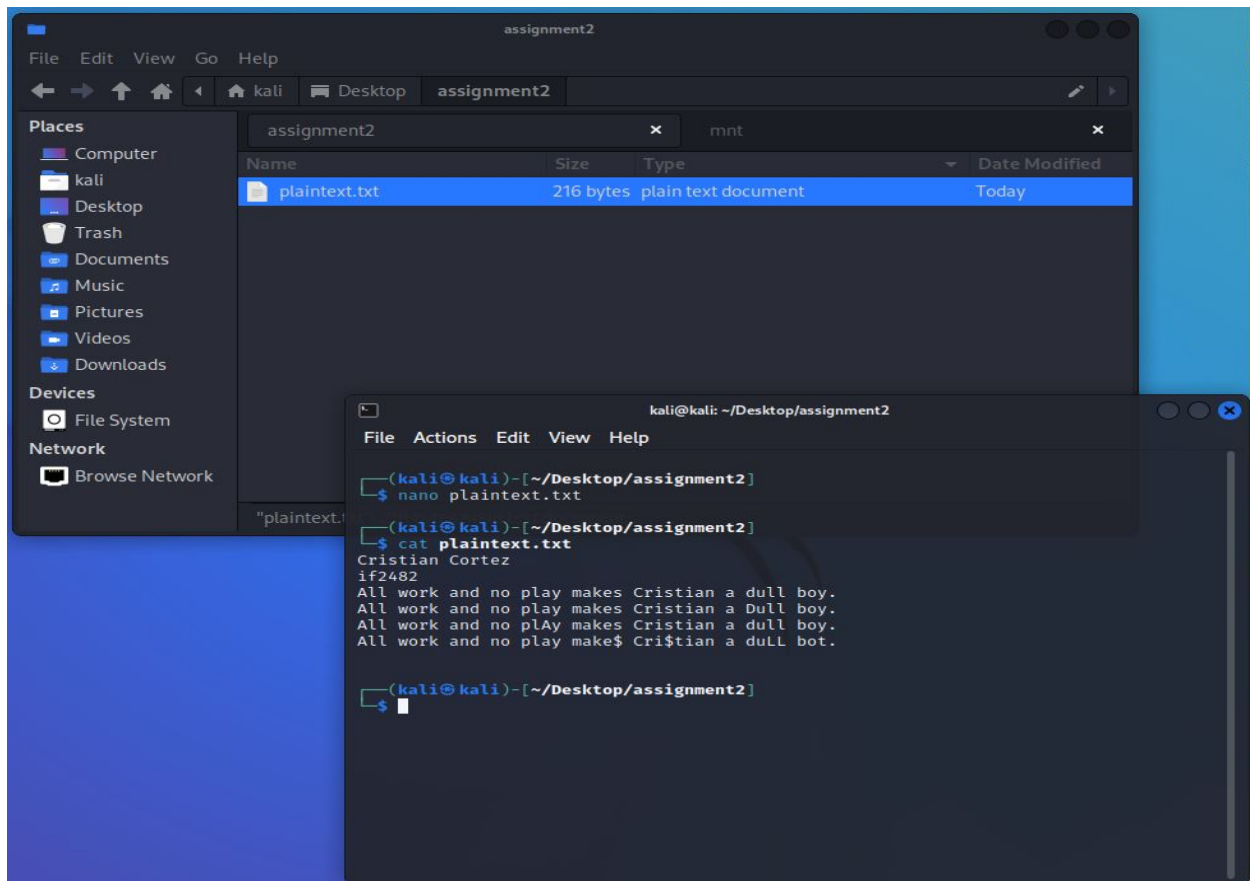
if2482

All work and no play makes Cristian a dull boy.

All work and no play makes Cristian a Dull boy.

All work and no plAy makes Cristian a dull boy.

All work and no play make\$ Cri\$tian a duLL bot.



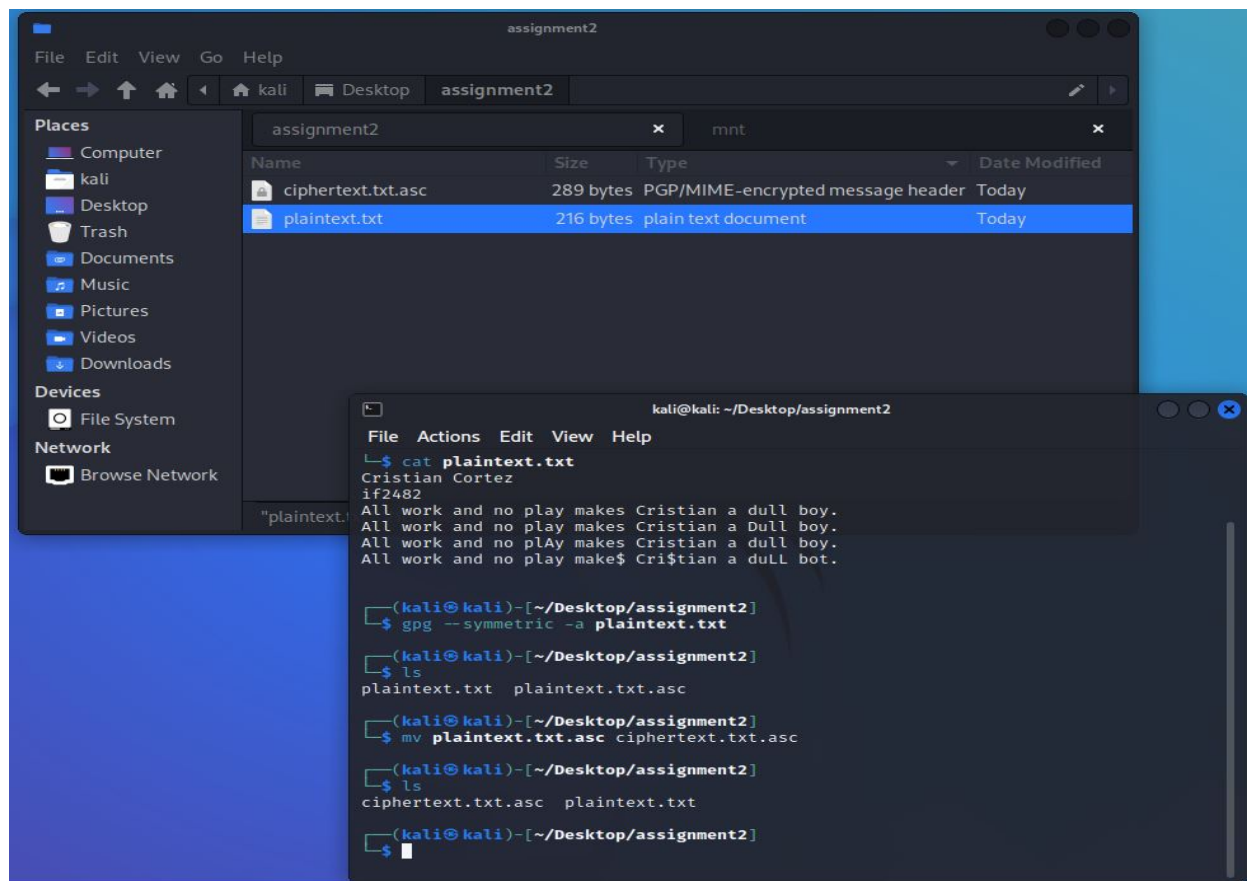
B: GPG symmetric/asymmetric encryption

1. **Encrypt the plaintext file with symmetric encryption in ASCII armored format using password "Letmein". Rename the encrypted file "ciphertext.txt.asc" and print contents.**

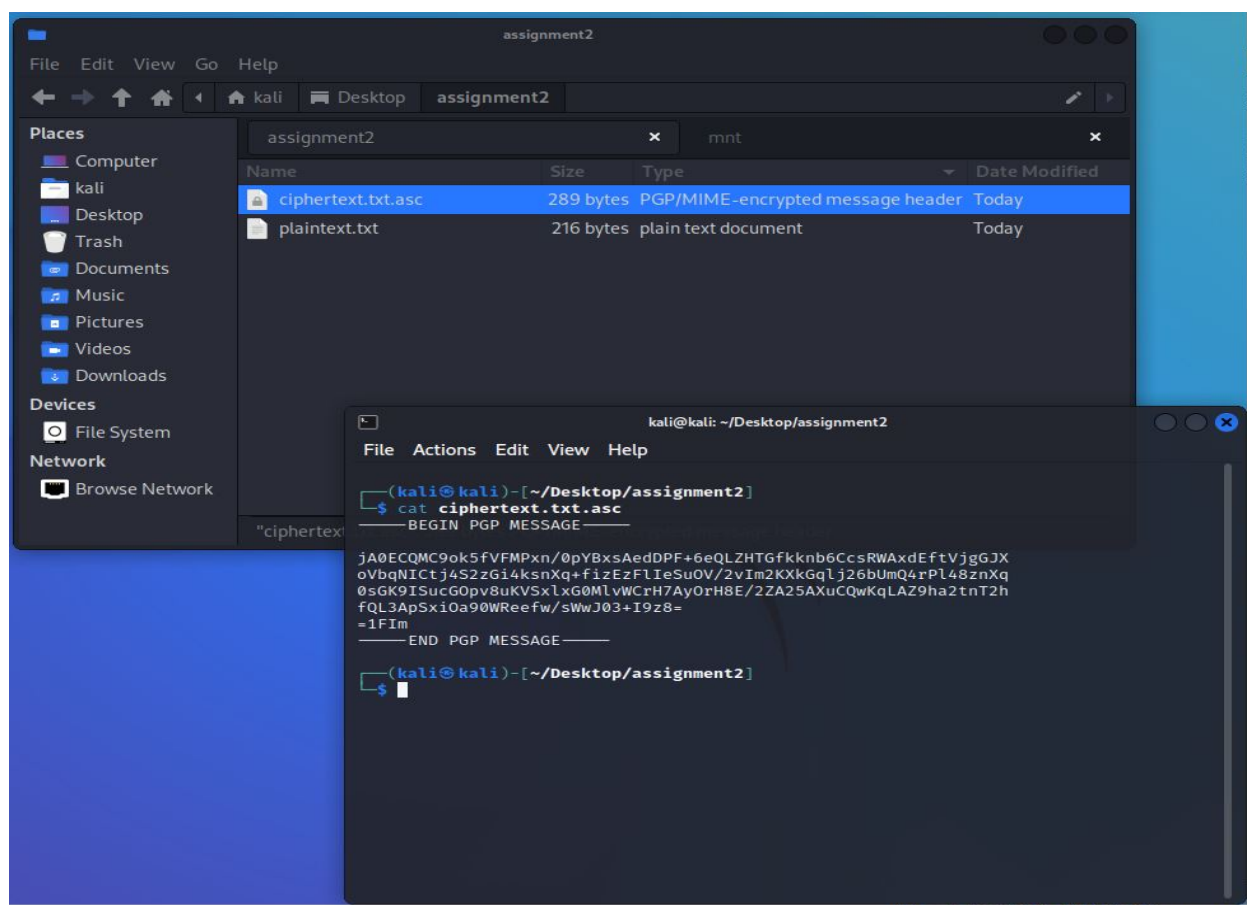
Use commands:

```
$ gpg --symmetric -a plaintext.txt
$ mv plaintext.txt.asc ciphertext.txt.asc
$ cat ciphertext.txt.asc
```

Create the cipher text



Print the contents of the ciphertext file.



Mine looks like:

"ciphertext.txt.asc"

```
-----BEGIN PGP MESSAGE-----
```

```
jA0ECQMC9ok5fVFMPxn/0pYBxsAedDPF+6eQLZHTGfkkn6CcsRWAxEftVjgGJX
oVbqNICtj4S2zGi4ksnXq+fizEzFlIeSuOV/2vIm2KXkGqlj26bUmQ4rPl48znXq
0sGK9ISucG0pv8uKVSxlxG0MlvWCrH7AyOrH8E/2ZA25AXuCQwKqLAZ9ha2tnT2h
fQL3ApSxi0a90WReefw/sWwJ03+I9z8=
=1FIIm
-----END PGP MESSAGE-----
```

2. **Import provided public key `csmith.pub.key` into the gpg keyring. Display the list of keys in the gpg keyring.**

Use commands:

```
$ gpg --import csmith.pub.key
$ gpg --list-keys
```

If the key is not already present, navigate to the folder with the public key.

List the current keys within the key ring. If you do not have a key ring, a key ring will be generated for you (as in my case).

```
kali@kali: ~/Desktop/assignment2
File Actions Edit View Help

(kali@kali)-[~/Desktop/assignment2]
$ cat ciphertext.txt.asc
-----BEGIN PGP MESSAGE-----

jA0ECQMC9ok5fVFMPxn/0pYBxsAedDPF+6eQLZHTGfkkn6CcsRWAxEftVjgGJX
oVbqNICtj4S2zGi4ksnXq+fizEzFlIeSuOV/2vIm2KXkGqlj26bUmQ4rPl48znXq
0sGK9ISucG0pv8uKVSxlxG0MlvWCrH7AyOrH8E/2ZA25AXuCQwKqLAZ9ha2tnT2h
fQL3ApSxi0a90WReefw/sWwJ03+I9z8=
=1FIIm
-----END PGP MESSAGE-----

(kali@kali)-[~/Desktop/assignment2]
$ gpg --list-keys
gpg: /home/kali/.gnupg/trustdb.gpg: trustdb created

(kali@kali)-[~/Desktop/assignment2]
$ gpg --list-keys

(kali@kali)-[~/Desktop/assignment2]
$
```

Import the provided public key.

```
kali@kali: ~/Desktop/assignment2
File Actions Edit View Help

(kali@kali)-[~/Desktop/assignment2]
$ ls
ciphertext.txt.asc  csmith.pub.key  plaintext.txt

(kali@kali)-[~/Desktop/assignment2]
$ gpg --import csmith.pub.key
gpg: key 15EACF261D1B1D51: public key "Christopher Smith <christopher.smith@csueastbay.edu>" imported
gpg: Total number processed: 1
gpg:             imported: 1

(kali@kali)-[~/Desktop/assignment2]
$
```

Now list the keys in the gpg keyring. You should see a new entry for the public key added.

```
kali@kali: ~/Desktop/assignment2
File Actions Edit View Help

(kali@kali)-[~/Desktop/assignment2]
$ ls
ciphertext.txt.asc  csmith.pub.key  plaintext.txt

(kali@kali)-[~/Desktop/assignment2]
$ gpg --import csmith.pub.key
gpg: key 15EACF261D1B1D51: public key "Christopher Smith <christopher.smith@csueastbay.edu>" imported
gpg: Total number processed: 1
gpg:             imported: 1

(kali@kali)-[~/Desktop/assignment2]
$ gpg --list-keys
/home/kali/.gnupg/pubring.kbx

pub   rsa3072 2021-09-02 [SC] [expires: 2023-09-02]
      7F7946C5B66DB850D3ACDB2F15EACF261D1B1D51
uid           [ unknown] Christopher Smith <christopher.smith@csueastbay.edu>
sub   rsa3072 2021-09-02 [E] [expires: 2023-09-02]

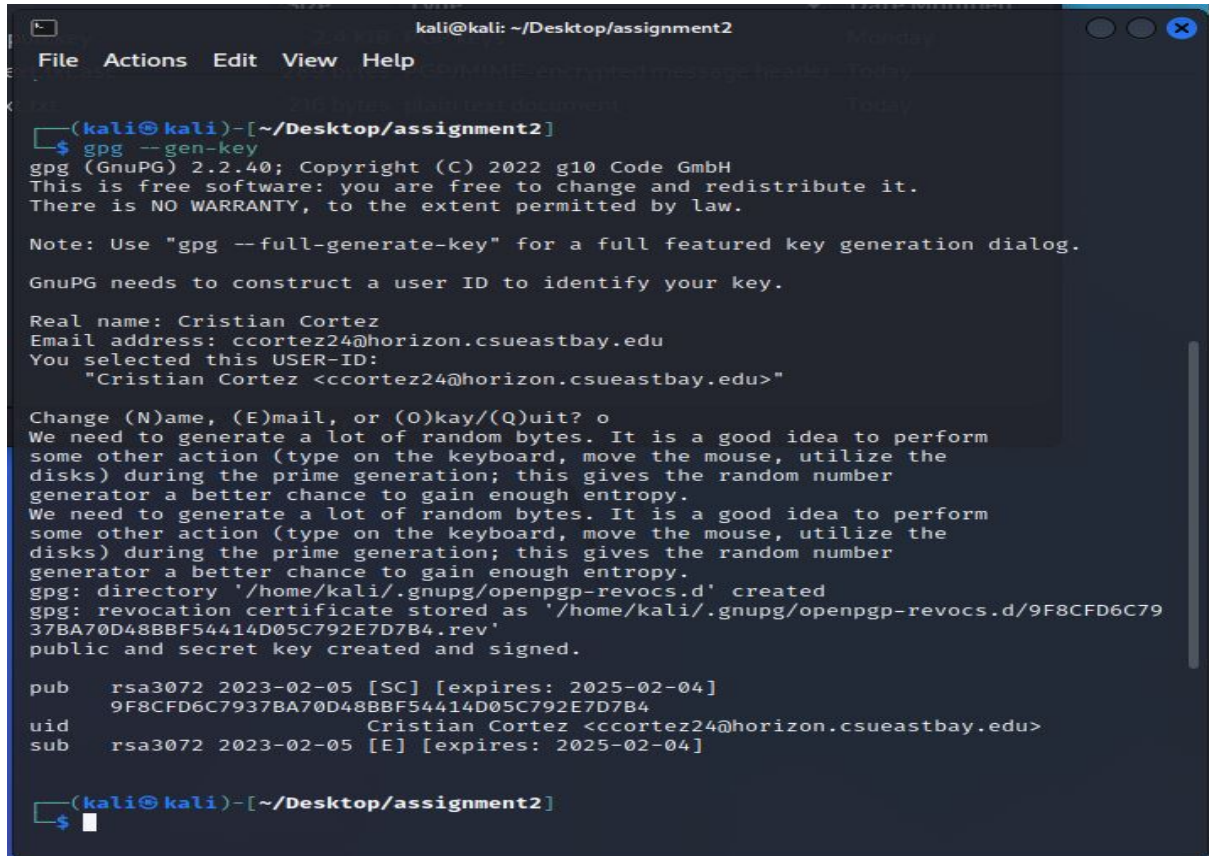
(kali@kali)-[~/Desktop/assignment2]
$
```

3. **Create a public/private key pair. Remember to SAVE YOUR PASSWORD! Display the gpg key ring keylist.**

Use commands:

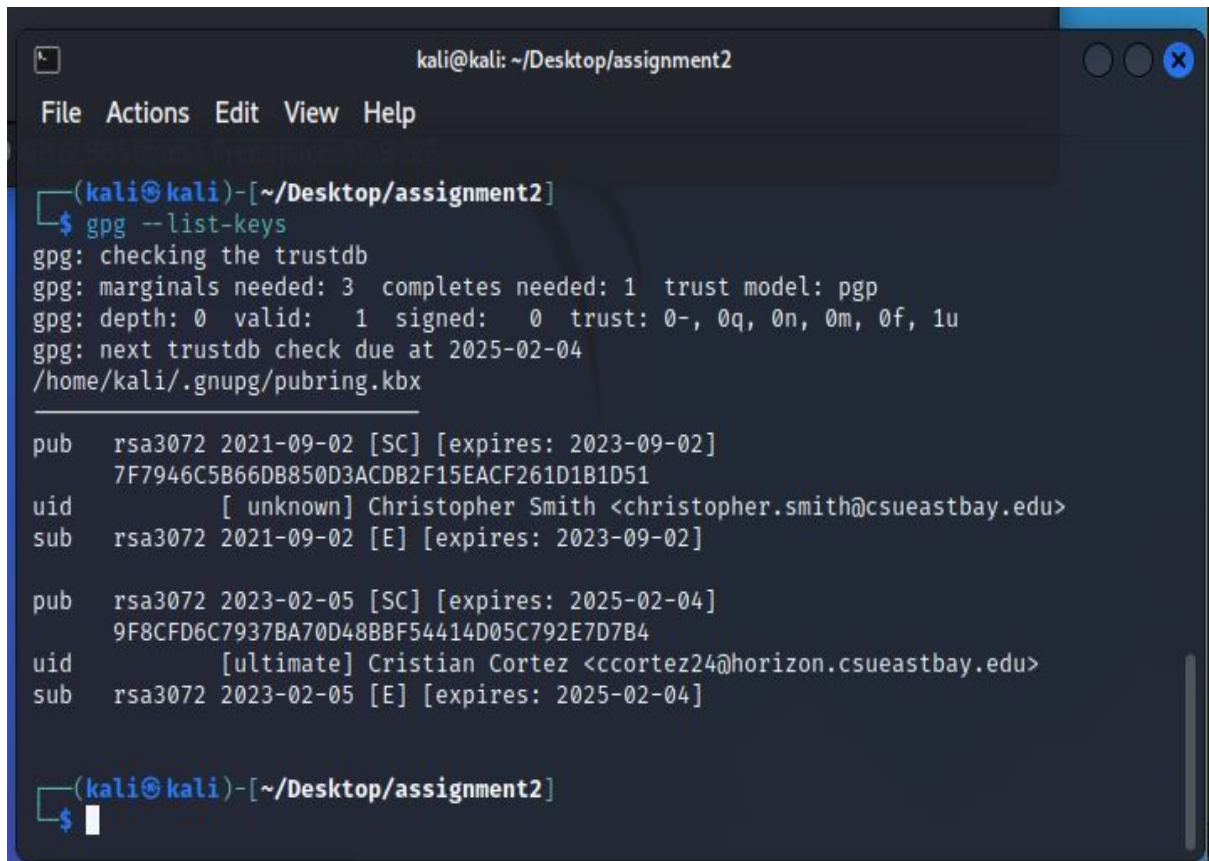
```
$ gpg --gen-key  
$ gpg --list-keys
```

Generate the private/public key pair. A prompt will ask you to enter A name, an email, and a password. A summary will then be provided (as shown below).



```
kali@kali: ~/Desktop/assignment2  
File Actions Edit View Help  
  
(kali@kali)-[~/Desktop/assignment2]  
$ gpg --gen-key  
gpg (GnuPG) 2.2.40; Copyright (C) 2022 g10 Code GmbH  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
  
Note: Use "gpg --full-generate-key" for a full featured key generation dialog.  
  
GnuPG needs to construct a user ID to identify your key.  
  
Real name: Cristian Cortez  
Email address: ccortez24@horizon.csueastbay.edu  
You selected this USER-ID:  
"Cristian Cortez <ccortez24@horizon.csueastbay.edu>"  
  
Change (N)ame, (E)mail, or (O)key/(Q)uit? o  
We need to generate a lot of random bytes. It is a good idea to perform  
some other action (type on the keyboard, move the mouse, utilize the  
disks) during the prime generation; this gives the random number  
generator a better chance to gain enough entropy.  
We need to generate a lot of random bytes. It is a good idea to perform  
some other action (type on the keyboard, move the mouse, utilize the  
disks) during the prime generation; this gives the random number  
generator a better chance to gain enough entropy.  
gpg: directory '/home/kali/.gnupg/openpgp-revocs.d' created  
gpg: revocation certificate stored as '/home/kali/.gnupg/openpgp-revocs.d/9F8CFD6C79  
37BA70D48BBF54414D05C792E7D7B4.rev'  
public and secret key created and signed.  
  
pub  rsa3072 2023-02-05 [SC] [expires: 2025-02-04]  
    9F8CFD6C7937BA70D48BBF54414D05C792E7D7B4  
uid                               Cristian Cortez <ccortez24@horizon.csueastbay.edu>  
sub  rsa3072 2023-02-05 [E] [expires: 2025-02-04]  
  
(kali@kali)-[~/Desktop/assignment2]  
$
```

List the keys in the key ring. Notice the new public key added to the key ring.

A terminal window titled 'kali@kali: ~/Desktop/assignment2' with a menu bar (File, Actions, Edit, View, Help). The terminal shows the command 'gpg --list-keys' and its output. The output lists two public keys: one for Christopher Smith (expired 2023-09-02) and one for Cristian Cortez (expires 2025-02-04).

```
kali@kali: ~/Desktop/assignment2
File Actions Edit View Help

(kali@kali)-[~/Desktop/assignment2]
$ gpg --list-keys
gpg: checking the trustdb
gpg: marginals needed: 3 completes needed: 1 trust model: pgp
gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2025-02-04
/home/kali/.gnupg/pubring.kbx

pub  rsa3072 2021-09-02 [SC] [expires: 2023-09-02]
     7F7946C5B66DB850D3ACDB2F15EACF261D1B1D51
uid      [ unknown] Christopher Smith <christopher.smith@csueastbay.edu>
sub  rsa3072 2021-09-02 [E] [expires: 2023-09-02]

pub  rsa3072 2023-02-05 [SC] [expires: 2025-02-04]
     9F8CFD6C7937BA70D48BBF54414D05C792E7D7B4
uid      [ultimate] Cristian Cortez <ccortez24@horizon.csueastbay.edu>
sub  rsa3072 2023-02-05 [E] [expires: 2025-02-04]

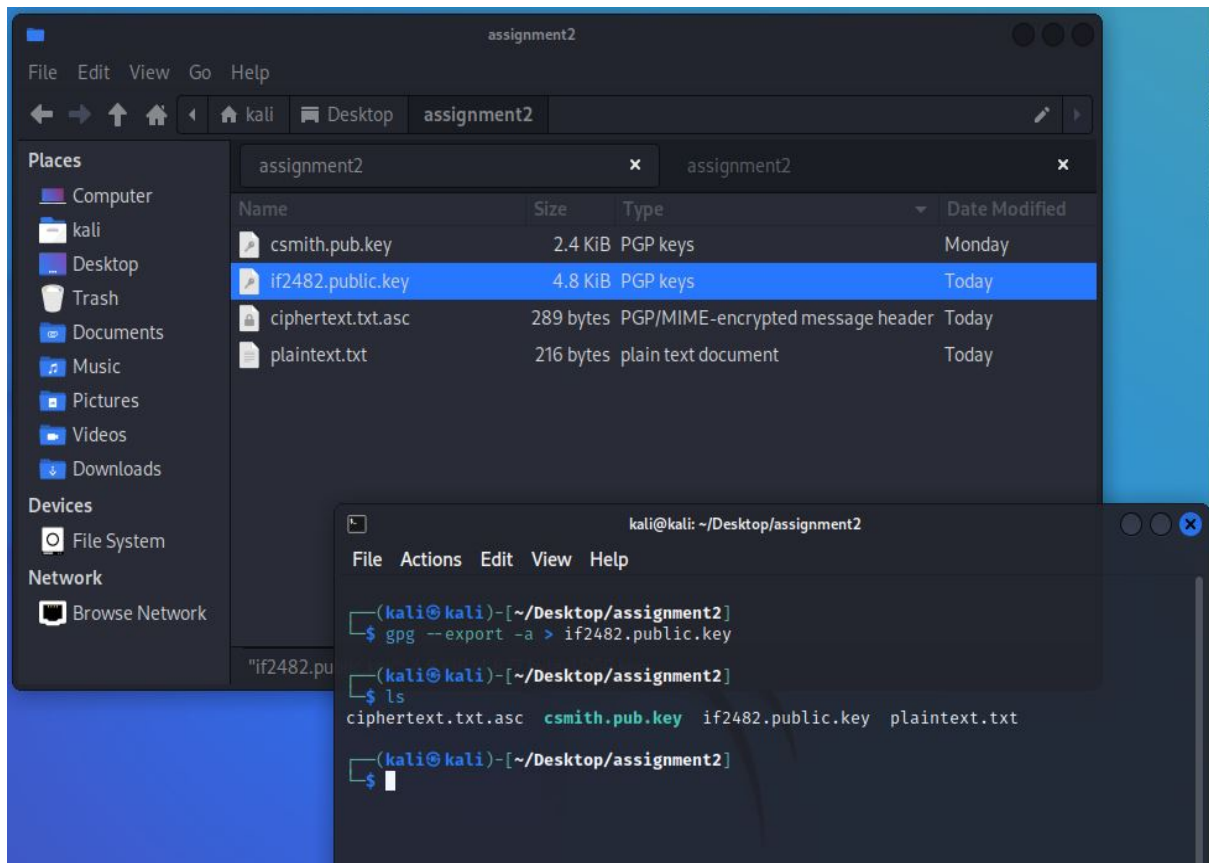
(kali@kali)-[~/Desktop/assignment2]
$
```

4. **Export the public key generated as ASCII armored format. Name the file [YOUR NETID].public.key. Display the contents of the exported key.**

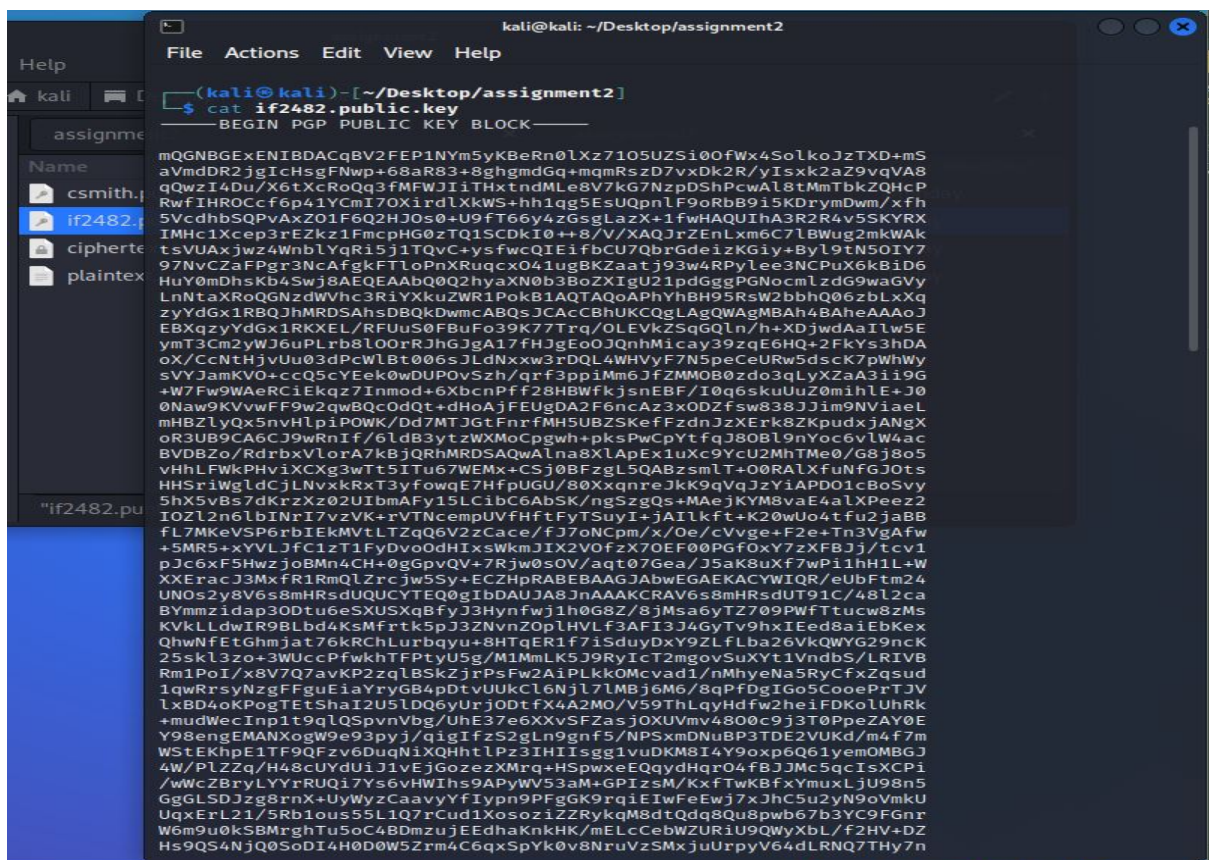
Use commands:

```
$ gpg --export -a > if2482.public.key
$ cat if2482.public.key
```

Export the public key



Display the contents within the public key.

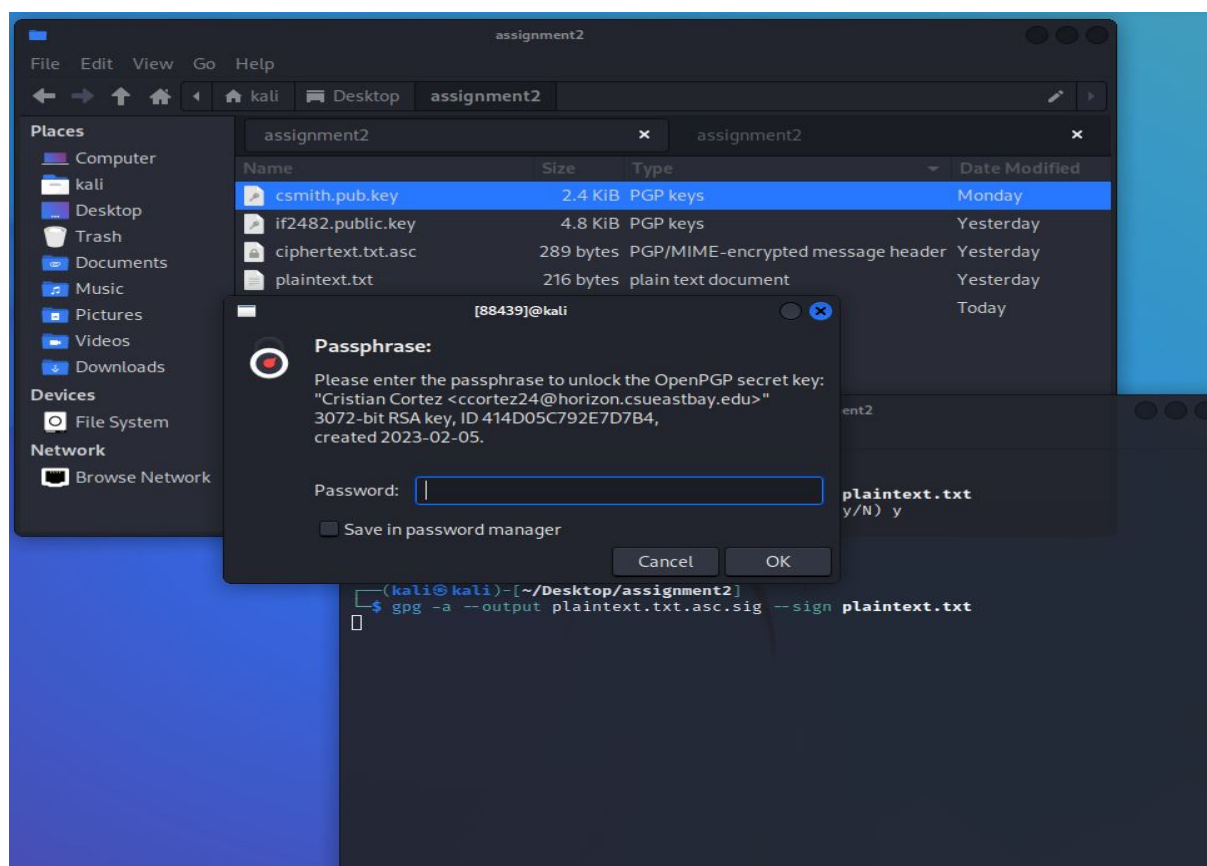


5. Sign the plaintext file created earlier with the generated private key. Remember to NOT SHARE YOUR PRIVATE KEY!

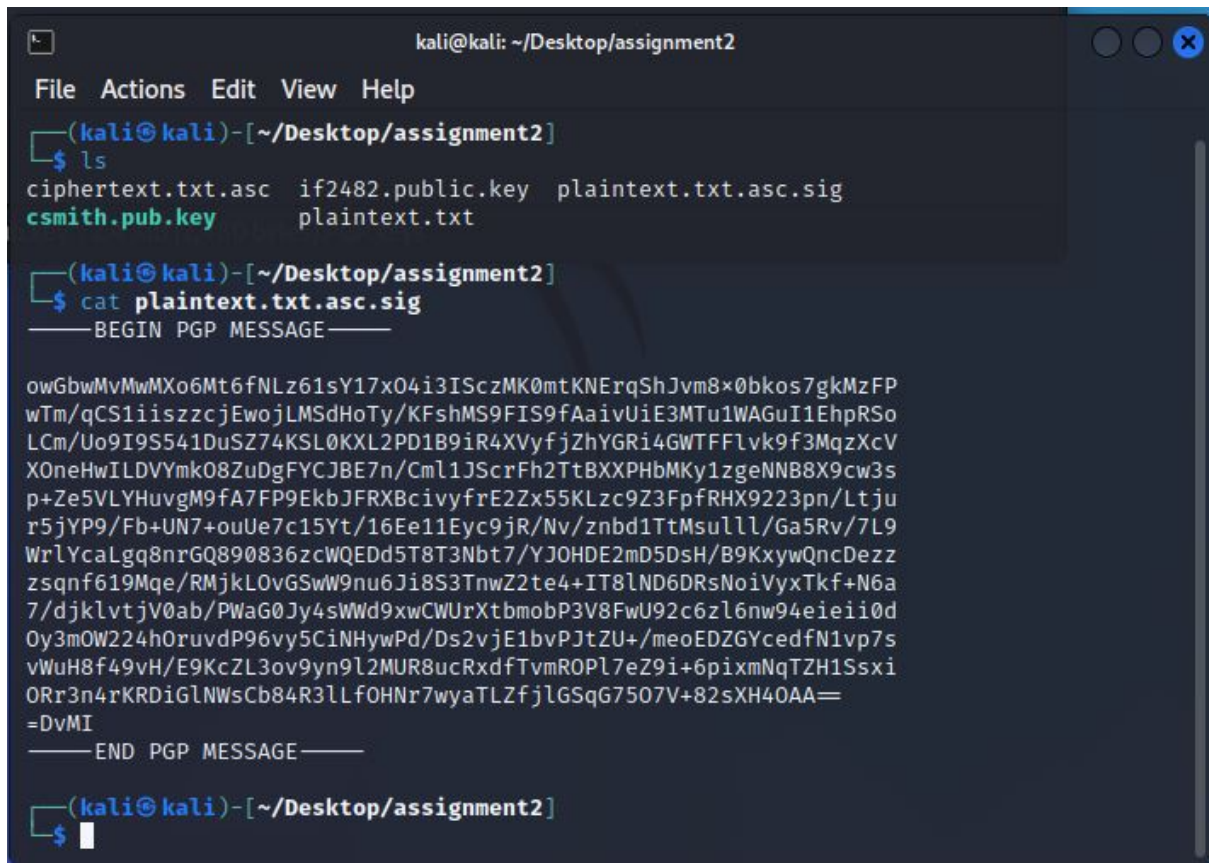
Use commands:


```
$ gpg -a --output plaintext.txt.asc.sig --sign plaintext.txt
```

Sign the plaintext file. A prompt will ask you for the password created earlier (shown below).



After, you can display the signed file with "cat". My result is below.



```
kali@kali: ~/Desktop/assignment2
File Actions Edit View Help
(kali@kali)-[~/Desktop/assignment2]
$ ls
ciphertext.txt.asc  if2482.public.key  plaintext.txt.asc.sig
csmith.pub.key      plaintext.txt
(kali@kali)-[~/Desktop/assignment2]
$ cat plaintext.txt.asc.sig
-----BEGIN PGP MESSAGE-----

owGbwMvMwMXo6Mt6fNLz61sY17x04i3ISczMK0mtKNerqShJvm8x0bkos7gkMzFP
wTm/qCS1iiszzcjEwojLMSdHoTy/KFshMS9FIS9fAaivUiE3MTu1WAGuI1EhpRSo
LCm/Uo9I9S541DuSZ74KSL0KXL2PD1B9iR4XVyfjZhYGRI4GWTFflvk9f3MqzXcV
X0neHwILDVYmk08ZuDgFYCJBE7n/Cml1JScrFh2TtBXXPHbMKy1zgeNNB8X9cw3s
p+Ze5VLYHuvGM9fA7FP9EkBJFRXBcivyfrE2Zx55KLzc9Z3FpfRHX9223pn/Ltju
r5jYP9/Fb+UN7+ouUe7c15Yt/16Ee11Eyc9jR/Nv/znbd1TtMsulll/Ga5Rv/7L9
WrlycaLgq8nrGQ890836zcWQEDd5T8T3Nbt7/YJOHDE2mD5DsH/B9KxywQncDezz
zsqnf619Mqe/RMjkLOvGSWw9nu6Ji8S3TnwZ2te4+IT8lND6DRsNoiVyxTkf+N6a
7/djklvtjV0ab/PWaG0Jy4sWWd9xwCWUrXtbmobP3V8FwU92c6zl6nw94eiei10d
Oy3mOW224hOruvdP96vy5CiNHywPd/Ds2vjE1bvPJtZU+/meoEDZGYcedfN1vp7s
vWuH8f49vH/E9KcZL3ov9yn9l2MUR8ucRxdfTvmROPl7eZ9i+6pixmapNqTZH1Ssxi
ORr3n4rKRDiGLNwsCb84R3LLfOHNR7wyaTLZfjlgSqG7507V+82sXH40AA=
=DvMI
-----END PGP MESSAGE-----

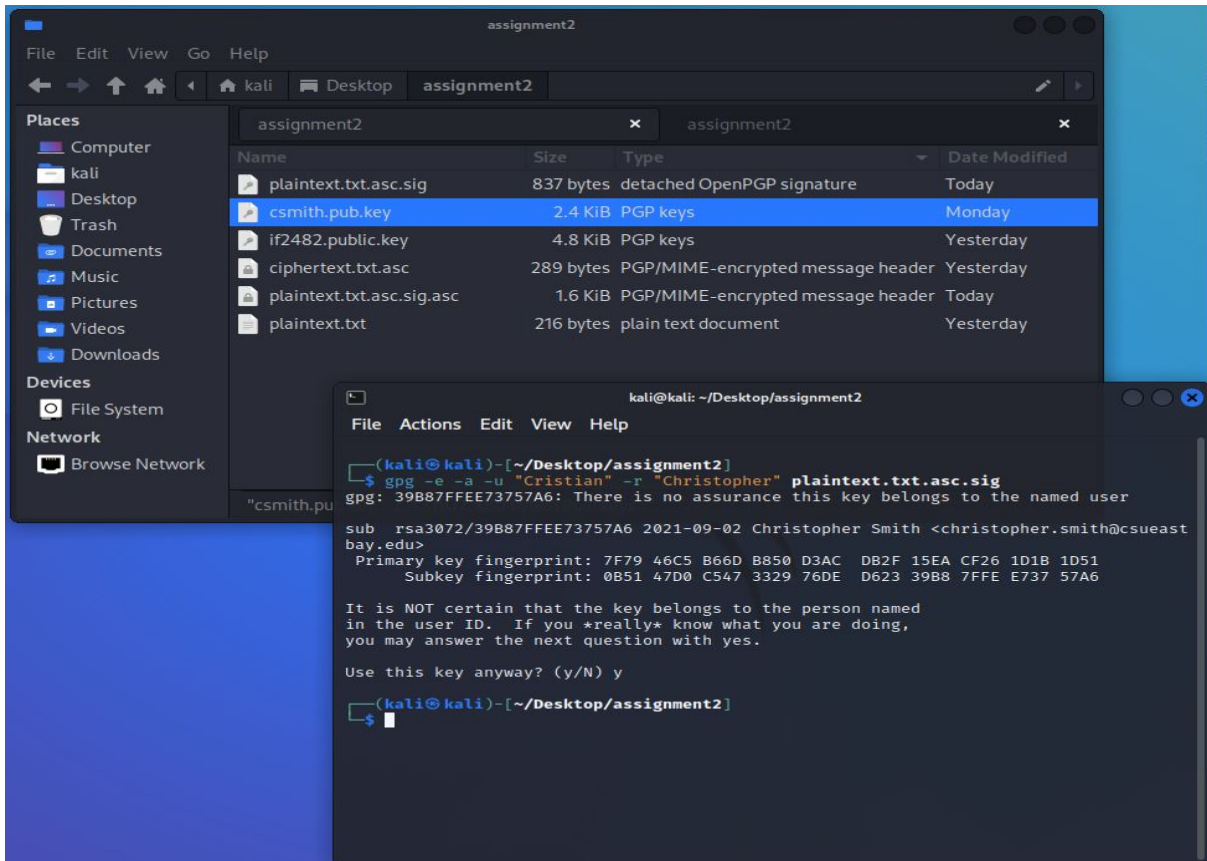
(kali@kali)-[~/Desktop/assignment2]
$
```

6. **Encrypt the signed file with ASCII output using the key.**

Use commands:

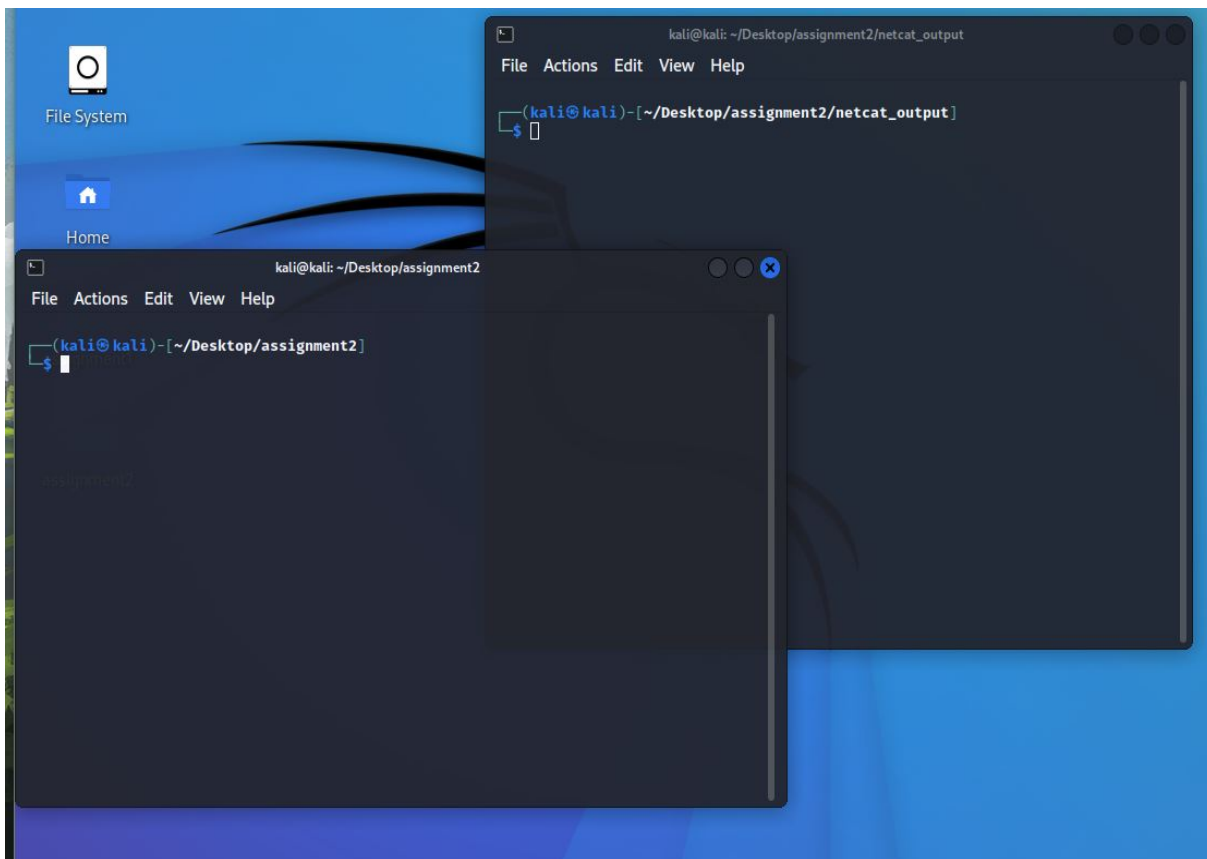
```
$ gpg -e -a -u "Cristian" -r "Christopher" plaintext.txt.asc.sig
```

Encrypt the signed file with the key. A prompt will ask you if are certain that you know the userID is certainly associated with the key. Answer yes.

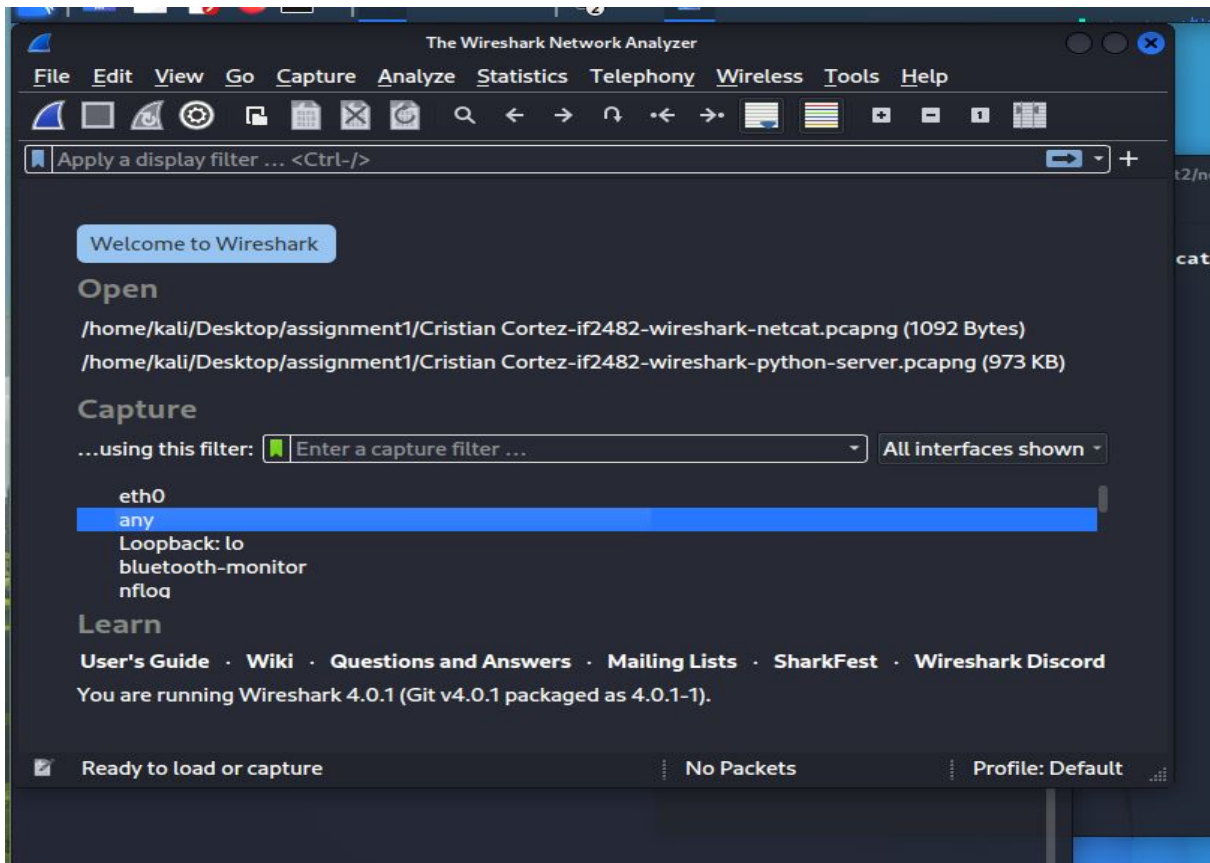


C: NETCAT encrypted text transmission

1. **Start 2 instances of the terminal (one for listener, one for sender)**



2. **Start wireshark and select "any" adapter.**



3. In one terminal, create a listener. In the other terminal create a sender.

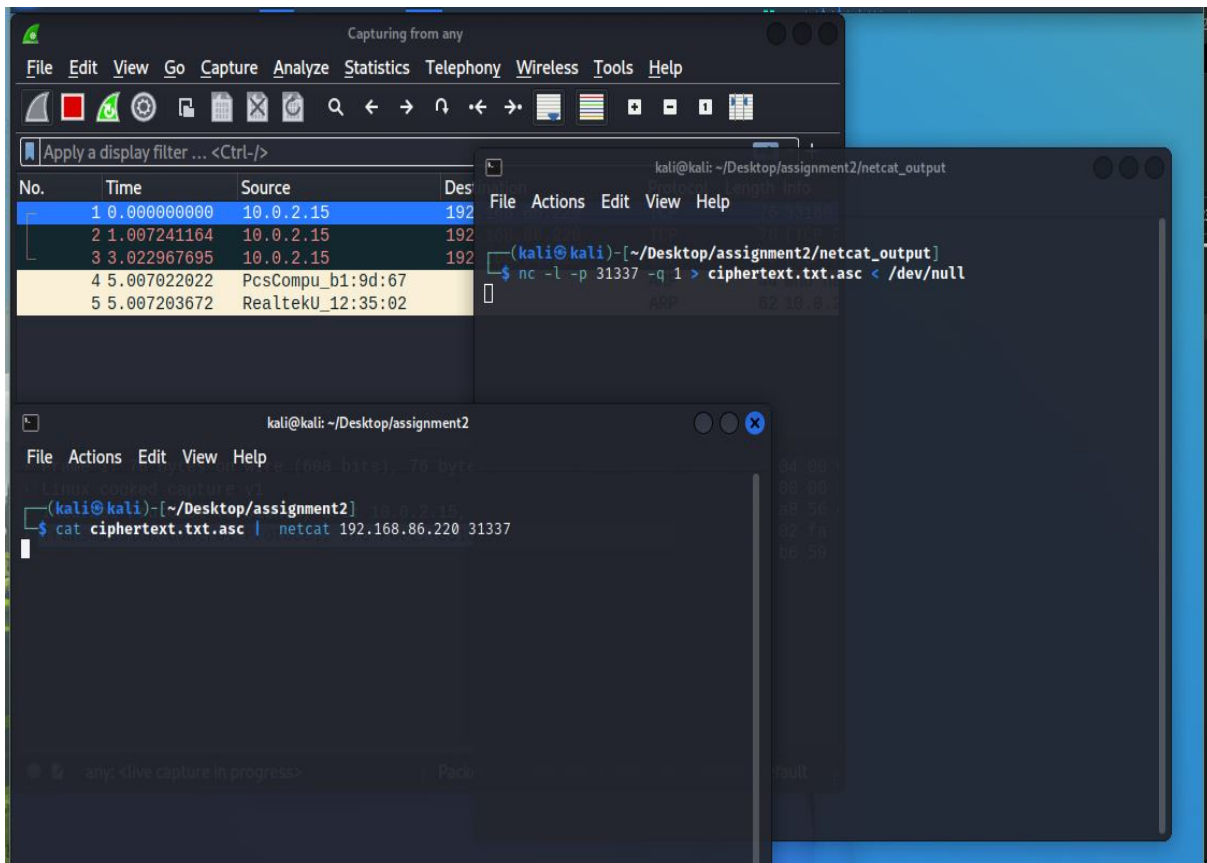
The listener will capture the data and save within a file named "**ciphertext.txt.asc**". The sender will send the output of **cat** to the listener via the TCP channel.

Wireshark will start to pickup the traffic.

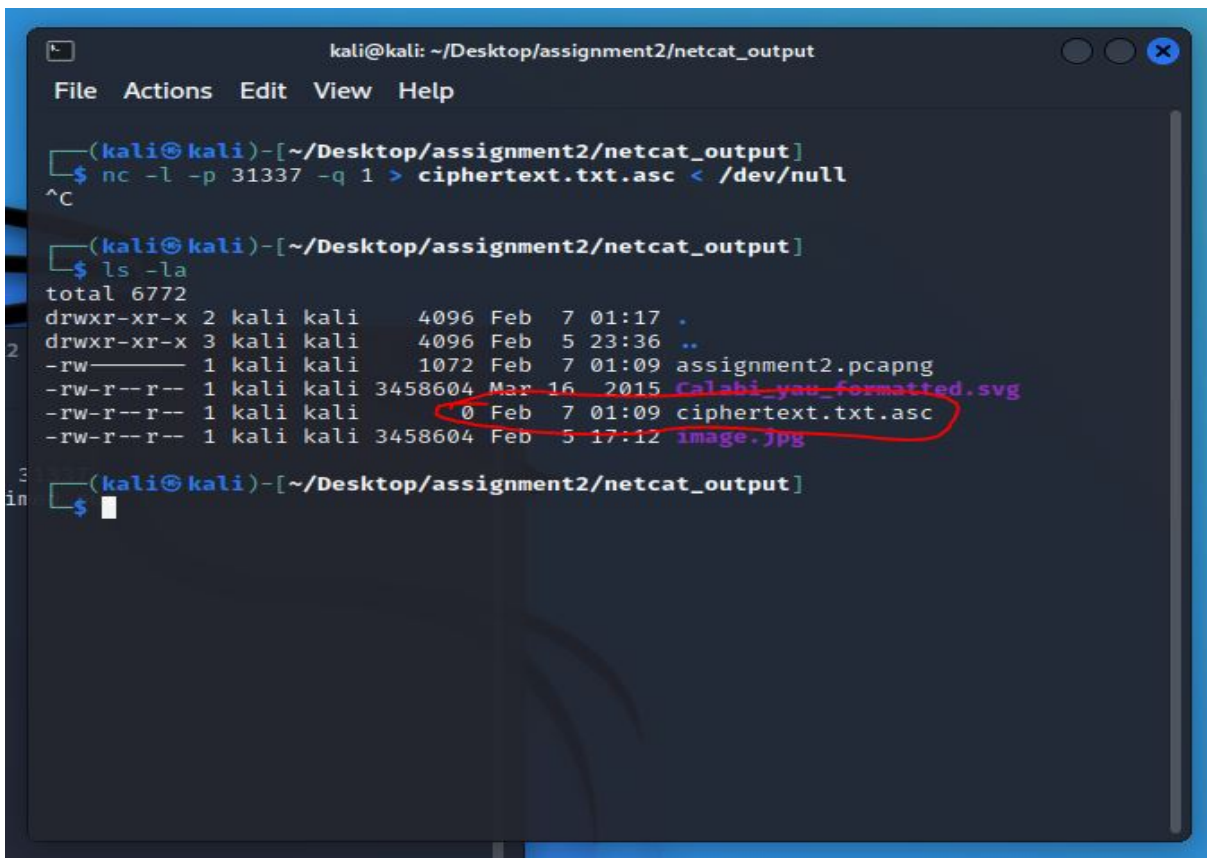
Use Commands:

```
// listener
$ nc -l -p 31337 -q 1 > ciphertext.txt.asc < /dev/null

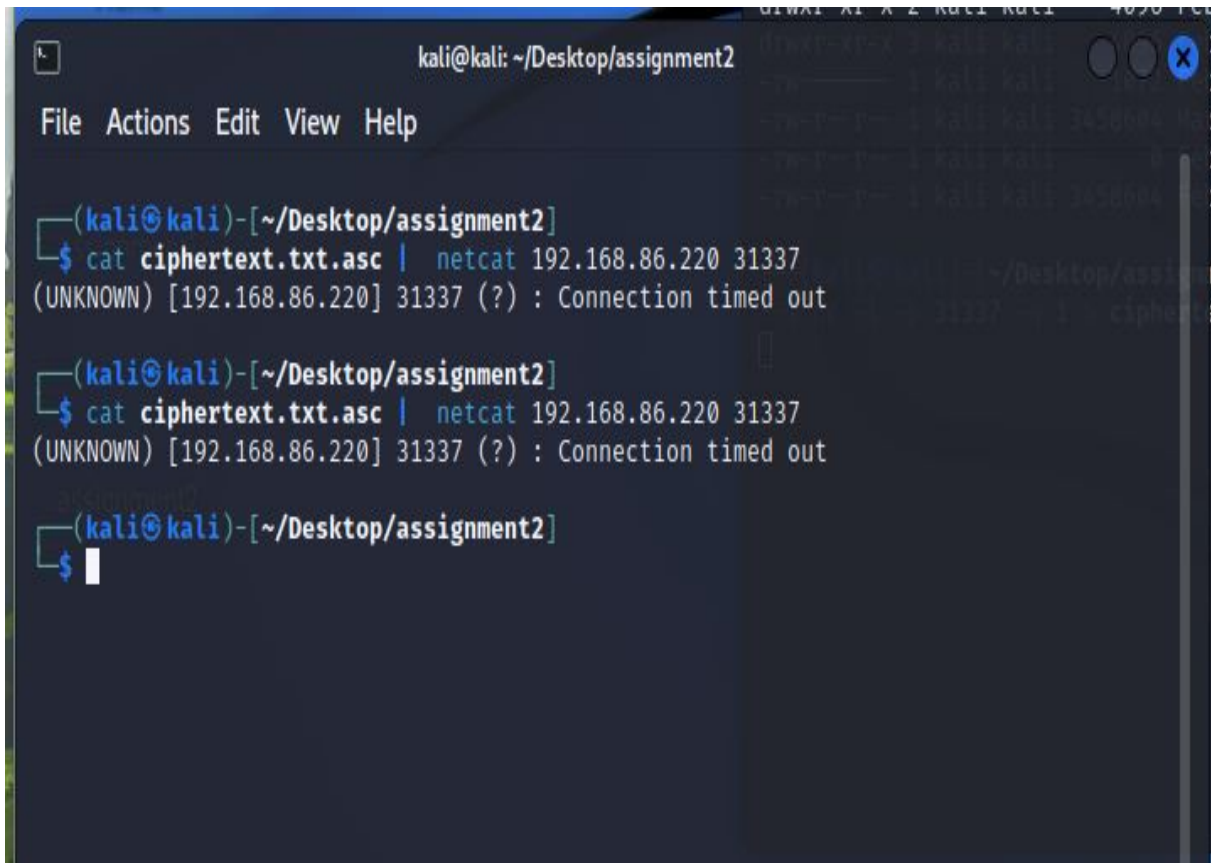
// sender
$ cat ciphertext.txt.asc | netcat 192.168.86.220 31337
```



PLEASE NOTE: I could not get my sender to send the data. Packet analysis shows that there was various TCP retransmissions and ports were reused. This indicates that a connection between the sender and the listener was never established. I looked for the TCP handshake and could not find one. This connection was never completed.



Eventually, the sender waits long enough to receive a "connection timeout" message.



```
kali@kali: ~/Desktop/assignment2
File Actions Edit View Help

(kali@kali)-[~/Desktop/assignment2]
$ cat ciphertext.txt.asc | netcat 192.168.86.220 31337
(UNKNOWN) [192.168.86.220] 31337 (?): Connection timed out

(kali@kali)-[~/Desktop/assignment2]
$ cat ciphertext.txt.asc | netcat 192.168.86.220 31337
(UNKNOWN) [192.168.86.220] 31337 (?): Connection timed out

(kali@kali)-[~/Desktop/assignment2]
$
```

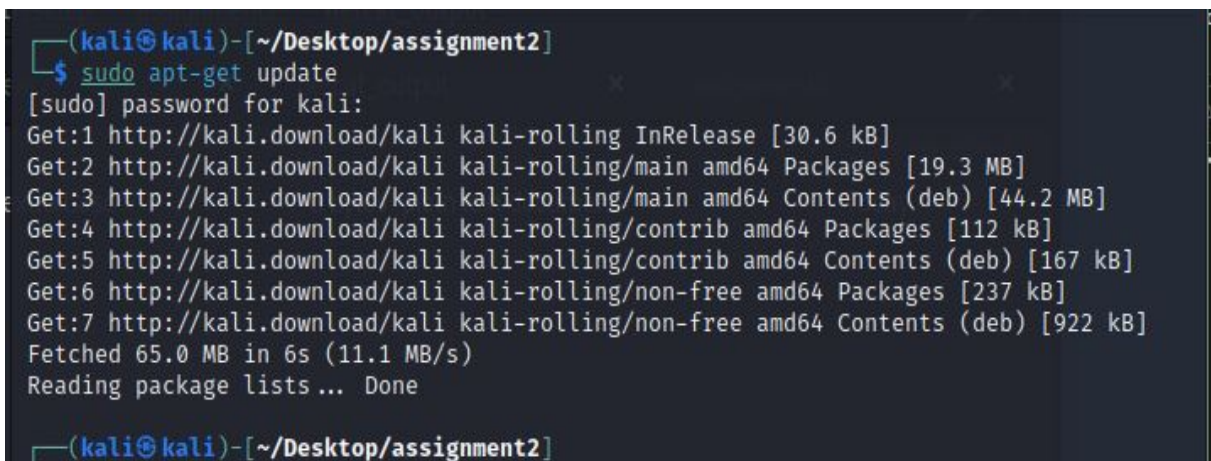
D: STEGHIDE emded plaintext.txt within jpeg

1. *Install Steghide if not already installed.*

Use Commands:

```
$ sudo apt-get update
$ sudo apt-get install steghide
$ man steghide
$ steghide
```

First, do a system update. Enter the password for system to update.



```
(kali@kali)-[~/Desktop/assignment2]
$ sudo apt-get update
[sudo] password for kali:
Get:1 http://kali.download/kali kali-rolling InRelease [30.6 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 Packages [19.3 MB]
Get:3 http://kali.download/kali kali-rolling/main amd64 Contents (deb) [44.2 MB]
Get:4 http://kali.download/kali kali-rolling/contrib amd64 Packages [112 kB]
Get:5 http://kali.download/kali kali-rolling/contrib amd64 Contents (deb) [167 kB]
Get:6 http://kali.download/kali kali-rolling/non-free amd64 Packages [237 kB]
Get:7 http://kali.download/kali kali-rolling/non-free amd64 Contents (deb) [922 kB]
Fetched 65.0 MB in 6s (11.1 MB/s)
Reading package lists ... Done

(kali@kali)-[~/Desktop/assignment2]
```

Next, install steghide. Enter "y" when prompted to continue the install.

```
(kali㉿kali)-[~/Desktop/assignment2]
$ sudo apt-get install steghide
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libmcrypt4 libmhash2
Suggested packages:
  libmcrypt-dev mcrypt
The following NEW packages will be installed:
  libmcrypt4 libmhash2 steghide
0 upgraded, 3 newly installed, 0 to remove and 1226 not upgraded.
Need to get 311 kB of archives.
After this operation, 907 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://kali.download/kali kali-rolling/main amd64 libmcrypt4 amd64 2.5.8-7 [72.6 kB]
Get:2 http://kali.download/kali kali-rolling/main amd64 libmhash2 amd64 0.9.9.9-9 [94.2 kB]
Get:3 http://kali.download/kali kali-rolling/main amd64 steghide amd64 0.5.1-15 [144 kB]
Fetched 311 kB in 1s (505 kB/s)
Selecting previously unselected package libmcrypt4.
(Reading database ... 395373 files and directories currently installed.)
Preparing to unpack .../libmcrypt4_2.5.8-7_amd64.deb ...
Unpacking libmcrypt4 (2.5.8-7) ...
Selecting previously unselected package libmhash2:amd64.
Preparing to unpack .../libmhash2_0.9.9.9-9_amd64.deb ...
Unpacking libmhash2:amd64 (0.9.9.9-9) ...
Selecting previously unselected package steghide.
Preparing to unpack .../steghide_0.5.1-15_amd64.deb ...
Unpacking steghide (0.5.1-15) ...
Setting up libmhash2:amd64 (0.9.9.9-9) ...
Setting up libmcrypt4 (2.5.8-7) ...
Setting up steghide (0.5.1-15) ...
Processing triggers for libc-bin (2.36-4) ...
Processing triggers for man-db (2.11.0-1+b1) ...
Processing triggers for kali-menu (2022.4.1) ...

(kali㉿kali)-[~/Desktop/assignment2]
$
```

2. Download an image either from the terminal or by hand through a web browser.

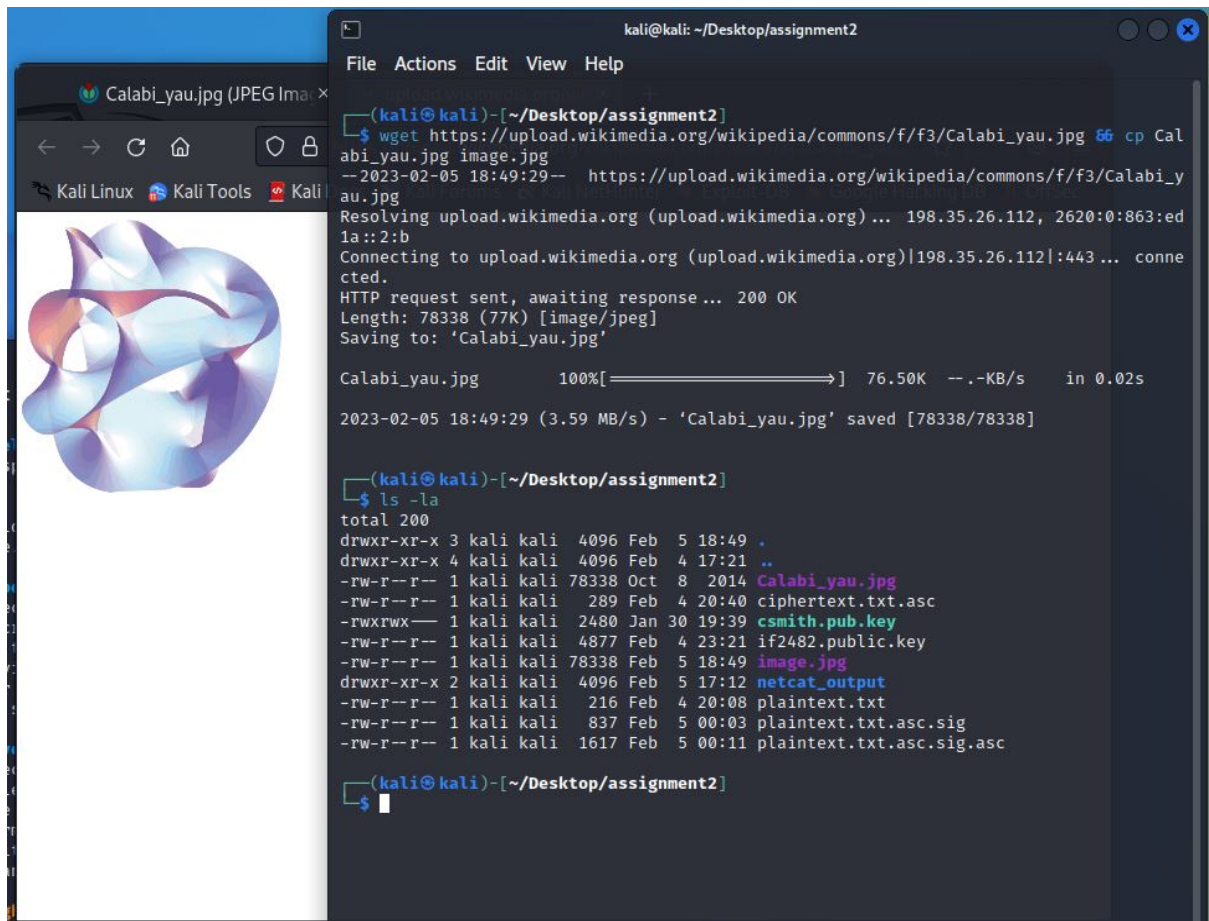
From the steghide manual: the only supported file types for embedding are: AU, BMP, JPEG or WAV. This would make BMP and JPEG the only images capable of text embeds.

PLEASE NOTE: Ensure your image is of the supported types. It will not work with anything else.

Use Commands:

```
$ wget https://upload.wikimedia.org/wikipedia/commons/f/f3/Calabi_yau.jpg && cp Calabi_yau.jpg image.jpg
```

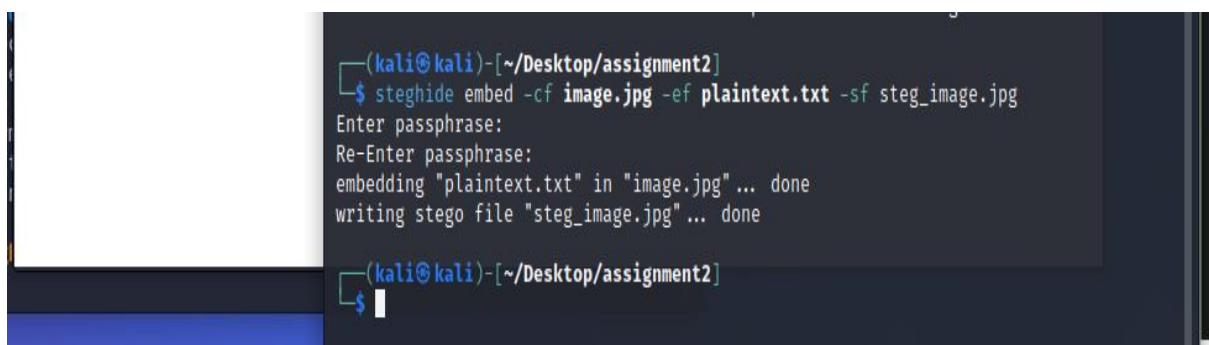
For this assignment, I decided to download from the terminal. First browse online for an image and copy that URL.



3. **Embed the plaintext file created earlier into the jpeg image. Use password "Letmein" to embed the image.**

Use Commands:

```
$ man steghide
$ steghide embed -cf image.jpg -ef plaintext.txt -sf steg_image.jpg
```



4. **Extract the plaintext message and display its contents.**

Use Commands:

```
$ steghide extract -sf steg_image.jpg
```

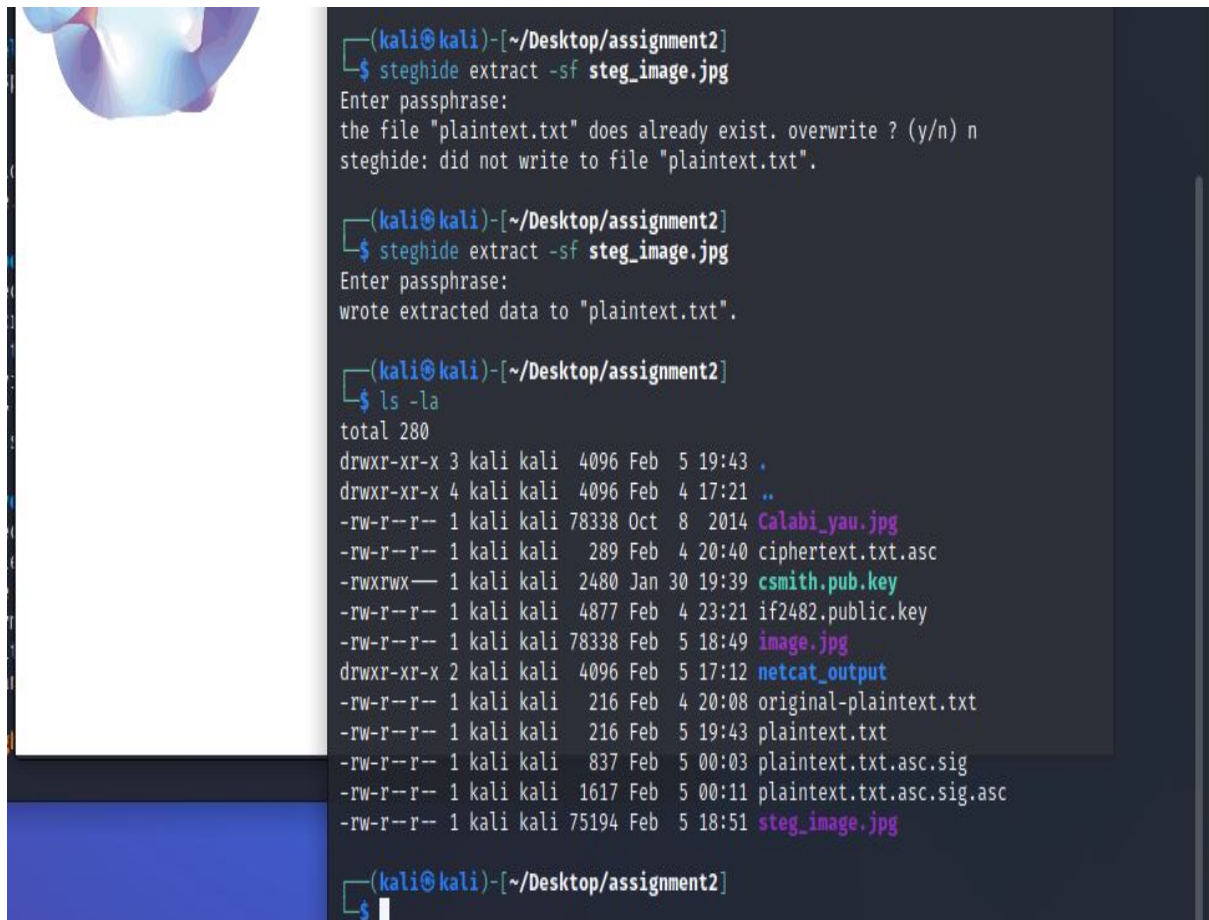

The extraction will always save the extracted data within a file name equal to the name of the file used to embed. Later we will be comparing both files for integrity; So we cannot overwrite the original file. The extraction will prompt you to "overwrite" an existing file. Answer "No".

Because of this, I recommend 2 possibilities:

1. Rename original plaintext file "**original-plaintext.txt**"
2. Mv the file to the another folder.

I choose the first option and renamed my original plaintext.

As you can see in the screengrab, the two files are the same size, which would suggest the have similar hashings.



```
(kali@kali)-[~/Desktop/assignment2]
$ steghide extract -sf steg_image.jpg
Enter passphrase:
the file "plaintext.txt" does already exist. overwrite ? (y/n) n
steghide: did not write to file "plaintext.txt".

(kali@kali)-[~/Desktop/assignment2]
$ steghide extract -sf steg_image.jpg
Enter passphrase:
wrote extracted data to "plaintext.txt".

(kali@kali)-[~/Desktop/assignment2]
$ ls -la
total 280
drwxr-xr-x 3 kali kali 4096 Feb  5 19:43 .
drwxr-xr-x 4 kali kali 4096 Feb  4 17:21 ..
-rw-r--r-- 1 kali kali 78338 Oct  8 2014 Calabi_yau.jpg
-rw-r--r-- 1 kali kali 289 Feb  4 20:40 ciphertext.txt.asc
-rwxrwx--- 1 kali kali 2480 Jan 30 19:39 csmith.pub.key
-rw-r--r-- 1 kali kali 4877 Feb  4 23:21 if2482.public.key
-rw-r--r-- 1 kali kali 78338 Feb  5 18:49 image.jpg
drwxr-xr-x 2 kali kali 4096 Feb  5 17:12 netcat_output
-rw-r--r-- 1 kali kali 216 Feb  4 20:08 original-plaintext.txt
-rw-r--r-- 1 kali kali 216 Feb  5 19:43 plaintext.txt
-rw-r--r-- 1 kali kali 837 Feb  5 00:03 plaintext.txt.asc.sig
-rw-r--r-- 1 kali kali 1617 Feb  5 00:11 plaintext.txt.asc.sig.asc
-rw-r--r-- 1 kali kali 75194 Feb  5 18:51 steg_image.jpg

(kali@kali)-[~/Desktop/assignment2]
$
```

E: MD5 hash comparison of embeded/extracted plaintext files aswell as jpeg files with, without embeded and renamed original images.

Plaintext Files

1. *Use md5 tool to get the checksum for both the embeded and the extracted plaintext file versions.*

Use Commands:

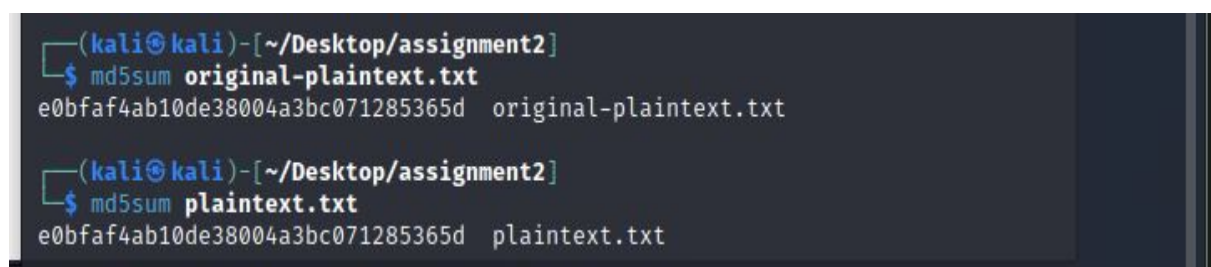
```
$ md5sum original-plaintext.txt
$ md5sum plaintext.txt
```

Compare the computed hashes. Mine look something like this:

```
$ md5sum original-plaintext.txt
e0bfaf4ab10de38004a3bc071285365d  original-plaintext.txt

$ md5sum plaintext.txt
e0bfaf4ab10de38004a3bc071285365d  plaintext.txt
```

In my case, the hashes are identical. When an image is embedded with a message, that message is encrypted using Rijndael AES 128 bit encryption. Encryption has a two-way characteristic in that a ciphertext can be decrypted to preserve the original message. It would make sense then that these messages produce the same hash.



```
(kali㉿kali)-[~/Desktop/assignment2]
$ md5sum original-plaintext.txt
e0bfaf4ab10de38004a3bc071285365d  original-plaintext.txt

(kali㉿kali)-[~/Desktop/assignment2]
$ md5sum plaintext.txt
e0bfaf4ab10de38004a3bc071285365d  plaintext.txt
```

Original/Embedded/Renamed Image Files

2. ***Use md5 tool to compare the checksums for both the original image as well as the image with the embedded message.***

Use commands:

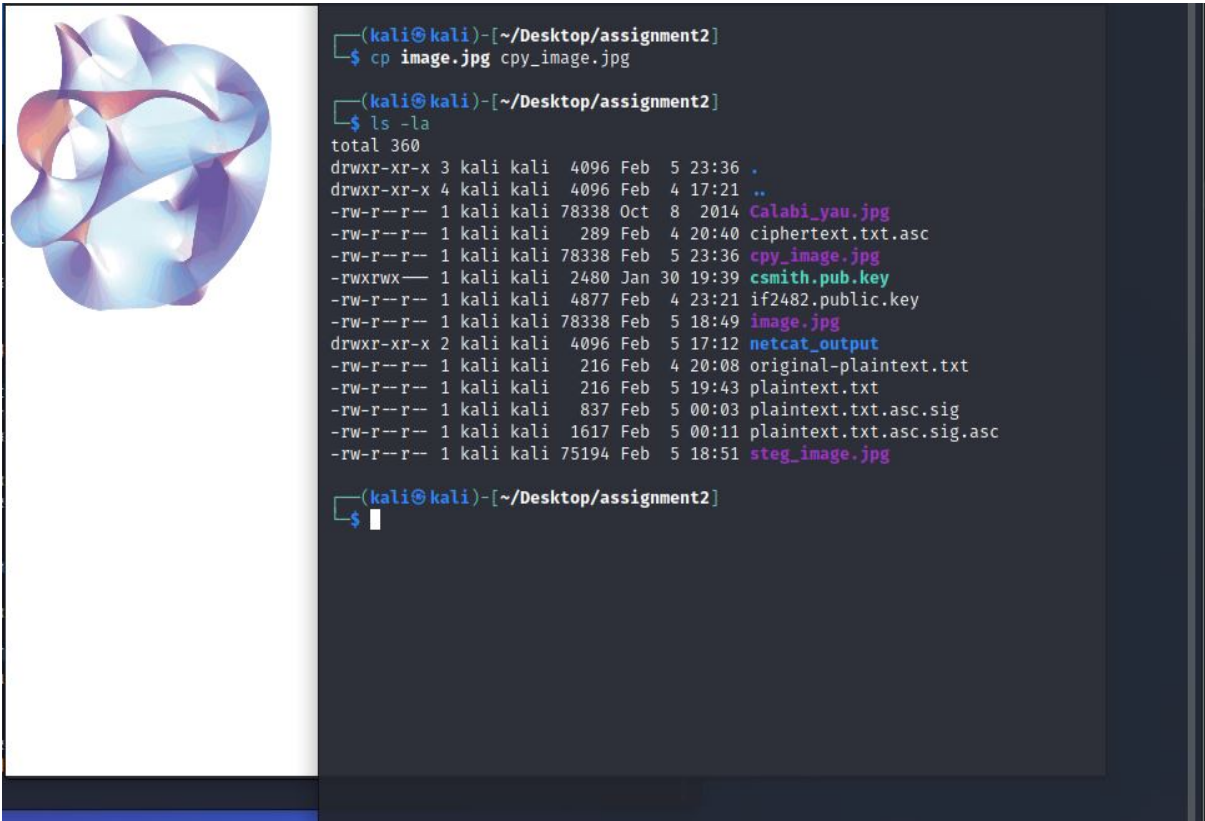
```
// make a copy of original
$ cp image.jpg cpy_image.jpg

// original without embedded message
$ md5sum image.jpg

// original renamed
$ md5sum cpy_image.jpg

// embedded with message
$ md5sum steg_image.jpg
```

Make copy of the original image file.



Compare the hash values for the files. Mine look like:

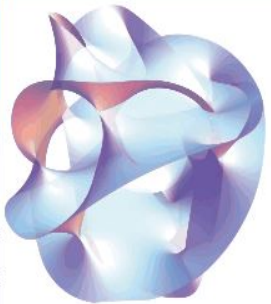
```

$ md5sum image.jpg
f9acc2780db1ac277098247f86620156  image.jpg

$ md5sum cpy_image.jpg
f9acc2780db1ac277098247f86620156  cpy_image.jpg

$ md5sum steg_image.jpg
4584fe89327cbb614f9a4ba48a2f1cb9  steg_image.jpg

```



```

(kali@kali)-[~/Desktop/assignment2]
$ cp image.jpg cpy_image.jpg

(kali@kali)-[~/Desktop/assignment2]
$ ls -la
total 360
drwxr-xr-x 3 kali kali 4096 Feb  5 23:36 .
drwxr-xr-x 4 kali kali 4096 Feb  4 17:21 ..
-rw-r--r-- 1 kali kali 78338 Oct  8  2014 Calabi_yau.jpg
-rw-r--r-- 1 kali kali  289 Feb  4 20:40 ciphertext.txt.asc
-rw-r--r-- 1 kali kali 78338 Feb  5 23:36 cpy_image.jpg
-rwxrwx--- 1 kali kali 2480 Jan 30 19:39 csmith.pub.key
-rw-r--r-- 1 kali kali 4877 Feb  4 23:21 if2482.public.key
-rw-r--r-- 1 kali kali 78338 Feb  5 18:49 image.jpg
drwxr-xr-x 2 kali kali 4096 Feb  5 17:12 netcat_output
-rw-r--r-- 1 kali kali  216 Feb  4 20:08 original-plaintext.txt
-rw-r--r-- 1 kali kali  216 Feb  5 19:43 plaintext.txt
-rw-r--r-- 1 kali kali  837 Feb  5 00:03 plaintext.txt.asc.sig
-rw-r--r-- 1 kali kali 1617 Feb  5 00:11 plaintext.txt.asc.sig.asc
-rw-r--r-- 1 kali kali 75194 Feb  5 18:51 steg_image.jpg

(kali@kali)-[~/Desktop/assignment2]
$ md5sum image.jpg
f9acc2780db1ac277098247f86620156  image.jpg

(kali@kali)-[~/Desktop/assignment2]
$ md5sum cpy_image.jpg
f9acc2780db1ac277098247f86620156  cpy_image.jpg

(kali@kali)-[~/Desktop/assignment2]
$ md5sum steg_image.jpg
4584fe89327cbb614f9a4ba48a2f1cb9  steg_image.jpg

(kali@kali)-[~/Desktop/assignment2]
$

```

As we can see, the original image shares its hash value with only the copy image and not with the embedded image.

This implies two things:

1. File names do not contribute to the file hash value. This would allow for files to be copied but **still hold integrity** in that data is not modified.
2. Embedding an image modifies the image data. This means that **image data integrity is not held** when embedding a message. This could be usefull in detecting images that have been maliciously modified.

Conclusion

In this assignment, we expiemented with various security tools including: gpg, netcat, wireshark, steghide and md5sum. Gpg was used to encrypt and decrypt a plaintext file. Both symmetric and asymmetric encryption was used. Also, we practiced signing an ecrypted file with a private key, which when decrypted with a public key would validate us as the encryptor. Netcat and Wireshark were then used to send the encrypted text, capture it and analyze the packets for data contents. Next, steghide was used to embed plaintext data within an image file. This was used to demonstrate how data can seemly be included in files by going unnoticed. Lastly, md5sum was used to check hash values of various files. These files were then checked to verify data had been embeded within. If the hash had been the same, then data held its intrgity. Since they were not (image vs image with embeded message), data integrity was not held.

Before describing how each tool used in this assignment provides or does not provide the X.800 Secuirty Services, lets take a brief moment to define them.

1. *Authentication*: ensures that all parties involved in a data access or connection are who they say they are.
2. *Access Control*: the ability to limit and control access to system resouces through security policies and mechanisms.

3. *Data Confidentiality*: prevents unauthorized data access.
4. *Data-Integrity*: provides assurance that total data streams remain unchanged by unauthorized entities.
5. *Non-repudiation*: protects against denial of involvement within a connection.

gpg: Confidentiality, Authentication, Non-repudiation

The OpenPGP tool provides encryption. Encryption typically supports *confidentiality* because only authorized entities that have the key to decrypt would be able to do so. However, encryption would not provide authentication alone. As we can see within symmetric encryption exercise, ciphertexts are enc/dec with the same key. Therefore, it is impossible to prove authentication since authentication requires the signature of a private key (one of the keys within asymmetric key pairs).

Authentication is provided by signatures, or a hash value. Hash values also provide data integrity, so an encryption tool like gpg would not support it alone. In this assignment, we signed a plaintext file with our private key. We then submitted our public key so that our message could get decrypted. Decrypting the message with our public key will produce our plaintext message. By doing that, it provides *authentication* that we are the ones who encrypted it. If after decrypting, the plaintext message was something entirely different, then the private/public key pair would not match and therefore break authentication of our identity. In this way, since private keys are only known to the owner, a successful decryption verifies it was signed by the private key, which is a form of *non-repudiation*.

gpg encryption does not provide access control or data-integrity. The tool would need:

1. access control: A way to create a security policy
2. data-integrity: A way to create file hashes

netcat: Non-repudiation (weak)

Netcat is a tool that "reads and writes data across network connections" (netcat manual). It creates a TCP connection between a sender and a receiver.

In this lab, netcat does not provide any authentication. So long as someone knows the host address and port number, a connection can be made without entity authentication. Similarly, netcat does not provide access control beyond that of a firewall.

The TCP connection made by netcat is easily intercepted and captured. In the assignment, we are meant to do this using Wireshark. So, netcat does not provide data confidentiality. Nor data-integrity, since messages can be easily intercepted, and modified.

Netcat provides a weak form of non-repudiation in the form of output file (network session log). Network connections made by netcat would then be recorded within a file that could be used against a claim of denial of involvement.

steghide: Confidentiality, Data-integrity

Steghide, a tool that embeds data within image and audio files, includes features such as embedded data compression, encryption and integrity checking via a checksum.

Compression does not provide any X.800 Security Service, since its only use is to reduce the amount of data of a file needed for storage and transmission. Compression is not a security tool on its own.

As described above, encryption provides *confidentiality* and when paired with a signature authenticity. Steghide offers no way to sign embeded data, so it does not provide authentication.

It does provide data *integrity* through its ability to automatically compare a cheksum.

Steghide does not provide access control or non-repudiation.

md5sum: Data-integrity

Md5sum is tool that computes the md5 hash value of a file. Computed hash values are unique to its input. So, comparing hash values across multiple files can prove wether data within the files are identitical (regardless of filename).

In this assignment, we were tasked with comparing the hash values of an image and the same image with an embded message. These hashes were different, which proves that data within them are different (modified). This exemplifies *data-integrity* in that two files with seemly the same image are actually different files since one file contained extra, embeded data.

Md5sum does not provide authentication, access control, data confidentiality or non-repudiation. All it does is compute hash values. However, hashing is an important component within authentication and data confidentiality. So, while md5sum alone does not provide anything more than data-intgrity, it can be used along side other tools to expand its security coverage.