

## Encryption and Steganography Project

## Abstract

This process uses a variety of security-based applications and tools in Kali Linux to perform symmetric and asymmetric encryption tasks. Encrypting and decrypting plaintext files, using steganography tools, md5 tools, Netcat, and Wireshark. All to showcase the use cases and limitations of these applications and to determine if they provide authentication, access control, data confidentiality, data integrity, and non-repudiation.

## Introduction

This example will be using VirtualBox version 7.0.10 with an image of Kali Linux installed on it. GPG will be used to perform encryption tasks, Netcat to transmit ASCII encrypted text, Steghide as the steganography tool to embed a plaintext file into a jpeg image, and md5 to gather a md5 hash of the image before and after performing steganography.

## Summary of Results

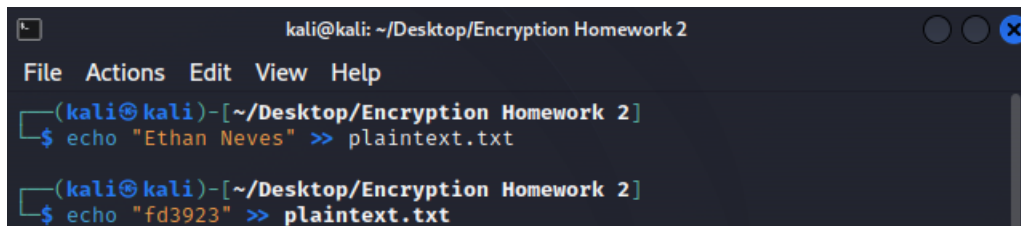
Create a plaintext file, named plaintext.txt, with the following text:

Your name

NetID (some letter number identifier)

Some secret message of your choosing.

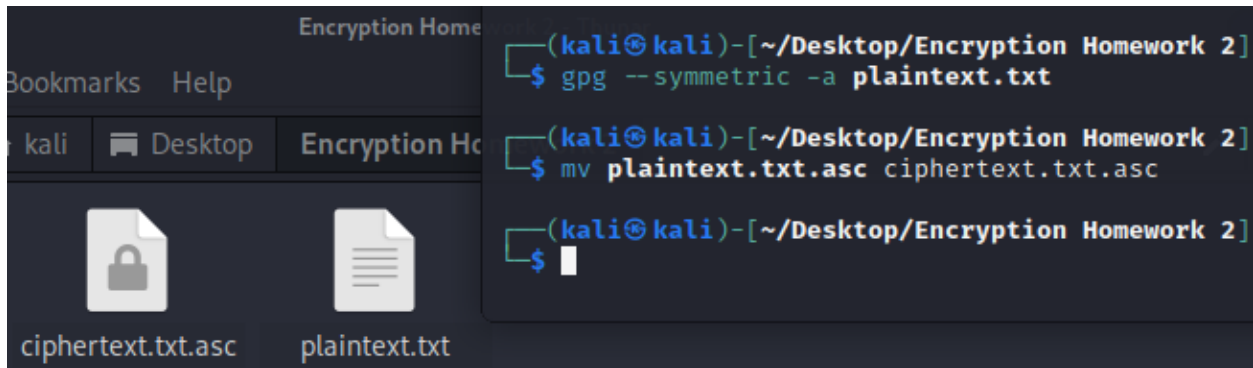
This can be done with the 'echo' command in the terminal window, or by simply creating a text file: `echo "Ethan Neves" >> plaintext.txt`

A screenshot of a terminal window titled 'kali@kali: ~/Desktop/Encryption Homework 2'. The window has a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. The terminal shows two commands being executed: first, `echo "Ethan Neves" >> plaintext.txt`, and second, `echo "fd3923" >> plaintext.txt`. The prompt is `(kali@kali)-[~/Desktop/Encryption Homework 2]`.

Repeating an echo command into the same file as before will add the echoed line onto a new line in the file. Now change the permissions on the file to read only using the command: `chmod 600 plaintext.txt`

Now, encrypt the plaintext file using gpg in symmetric ascii armored format, using the password 'letmein' and with the command: `gpg --symmetric -a plaintext.txt`

Additionally use the command: '`mv plaintext.txt.asc ciphertext.txt.asc`' to rename your new encrypted file.



The screenshot shows a Kali Linux desktop. On the left, a file manager window titled 'Encryption Home' displays two files: 'ciphertext.txt.asc' (represented by a lock icon) and 'plaintext.txt' (represented by a document icon). On the right, a terminal window titled '(kali@kali)-[~/Desktop/Encryption Homework 2]' shows the following commands and their outputs:

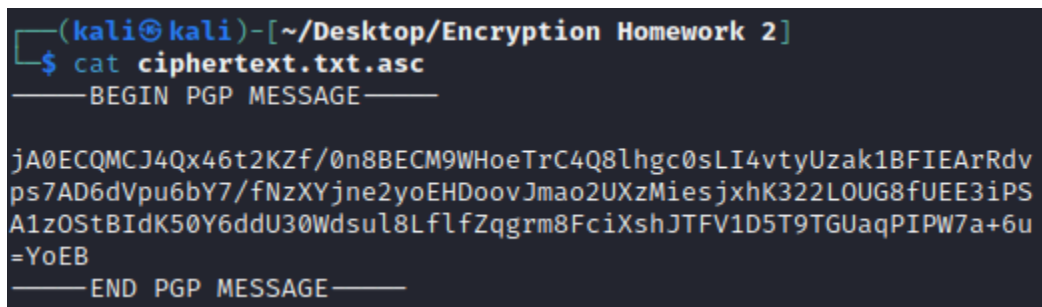
```
(kali@kali)-[~/Desktop/Encryption Homework 2]
$ gpg --symmetric -a plaintext.txt

(kali@kali)-[~/Desktop/Encryption Homework 2]
$ mv plaintext.txt.asc ciphertext.txt.asc

(kali@kali)-[~/Desktop/Encryption Homework 2]
$
```

To simplify things, I have the terminal open in a folder so all my created files will be in the same location, and I will not have to specify the path in any of my terminal commands.

Type the command: `cat ciphertext.txt.asc`



The screenshot shows a terminal window titled '(kali@kali)-[~/Desktop/Encryption Homework 2]' with the following output:

```
(kali@kali)-[~/Desktop/Encryption Homework 2]
$ cat ciphertext.txt.asc
-----BEGIN PGP MESSAGE-----

jA0ECQMCJ4Qx46t2KZf/0n8BECM9WHoeTrC4Q8lhgc0sLI4vtyUzak1BFIEArDv
ps7AD6dVpu6bY7/fNzXYjne2yoEHDoovJmao2UXzMiesjxhK322LOUG8fUEE3iPS
A1z0StBIdK50Y6ddU30Wdsul8LflfZqgrm8FciXshJTFV1D5T9TGUaqPIPW7a+6u
=YoEB
-----END PGP MESSAGE-----
```

This now displays the contents of the encrypted armored ascii file, because we specified '-a' when encrypting it, it encrypts it with ascii output. Without this specification it will encrypt it with a binary output, and then if we try to view the contents of the file, the reader will attempt to convert whatever the binary output is to ascii. This can generate some very weird and random ascii characters and symbols.

Next, we will import a public key into our gpg keyring using the command:  
`gpg --import keyname.key`

Then use the command: '`gpg --list-keys`' To list the current keys in your terminal.

Note: For this step ensure that the keyname.key file is in the same path as which you have the terminal open, otherwise the path will need to be specified in the command. Here, I have all the files within the same folder, and the terminal open in that folder.

```
Bookmarks Help
kali Desktop
christopher.smith.p
ublic.key

(kali@kali)-[~/Desktop/Encryption Homework 2]
$ gpg --import christopher.smith.public.key
gpg: /home/kali/.gnupg/trustdb.gpg: trustdb created
gpg: key C6751FE8C5217CE0: public key "Christopher Smith <christopher.smith@csueb.edu>" imported
gpg: Total number processed: 1
gpg:             imported: 1

(kali@kali)-[~/Desktop/Encryption Homework 2]
$ gpg --list-keys
/home/kali/.gnupg/pubring.kbx
-----
pub   rsa3072 2023-09-10 [SC] [expires: 2025-09-09]
      55EE3E461B3A3C478D3585BAC6751FE8C5217CE0
uid   [ unknown] Christopher Smith <christopher.smith@csueb.edu>
sub   rsa3072 2023-09-10 [E] [expires: 2025-09-09]
```

Now, a public key has been processed and imported into our keyring, and we can also view the key in our terminal.

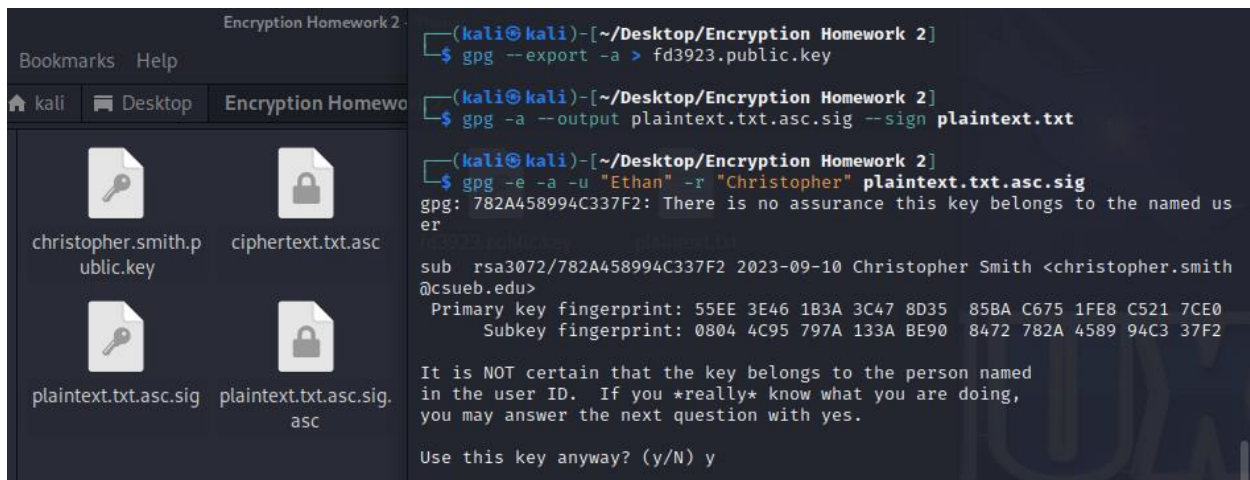
Now we will create our own keypair, you will need to specify your name, email, as well as a password. Use the command: `gpg --gen-key`

```
(kali@kali)-[~/Desktop/Encryption Homework 2]
$ 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 2025-09-12
/home/kali/.gnupg/pubring.kbx
-----
pub   rsa3072 2023-09-10 [SC] [expires: 2025-09-09]
      55EE3E461B3A3C478D3585BAC6751FE8C5217CE0
uid   [ unknown] Christopher Smith <christopher.smith@csueb.edu>
sub   rsa3072 2023-09-10 [E] [expires: 2025-09-09]

pub   rsa3072 2023-09-13 [SC] [expires: 2025-09-12]
      A76C1CFBC39B137A31345726F86634842FC06683
uid   [ultimate] Ethan Neves <ethan4083@gmail.com>
sub   rsa3072 2023-09-13 [E] [expires: 2025-09-12]
```

We can use the list command again to see that it has changed, and we now have two public keys in our gpg keyring, the one imported previously, and the new one we just created.

Now we will do a series of three commands to export our public key, sign our plaintext document, then encrypt the signed version of our plaintext document with someone else's public key.



```
(kali@kali)-[~/Desktop/Encryption Homework 2]
$ gpg --export -a > fd3923.public.key

(kali@kali)-[~/Desktop/Encryption Homework 2]
$ gpg -a --output plaintext.txt.asc.sig --sign plaintext.txt

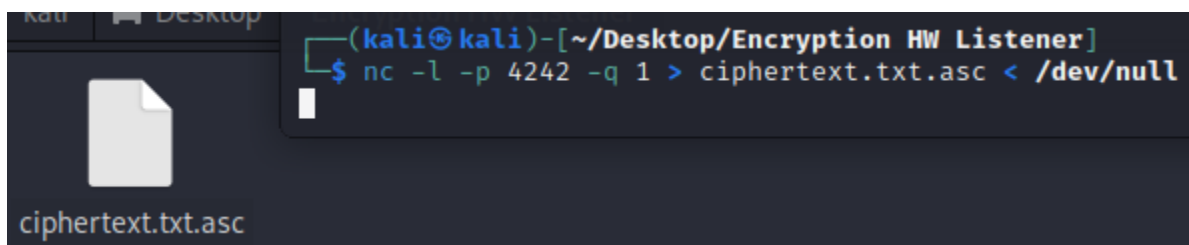
(kali@kali)-[~/Desktop/Encryption Homework 2]
$ gpg -e -a -u "Ethan" -r "Christopher" plaintext.txt.asc.sig
gpg: 782A458994C337F2: There is no assurance this key belongs to the named user
sub rsa3072/782A458994C337F2 2023-09-10 Christopher Smith <christopher.smith@csueb.edu>
Primary key fingerprint: 55EE 3E46 1B3A 3C47 8D35 85BA C675 1FE8 C521 7CE0
Subkey fingerprint: 0804 4C95 797A 133A BE90 8472 782A 4589 94C3 37F2

It is NOT certain that the key belongs to the person named
in the user ID. If you *really* know what you are doing,
you may answer the next question with yes.

Use this key anyway? (y/N) y
```

Here we have exported our public key as fd3923.public.key with the command ‘gpg --export -a > nameofkey.key’, you are free to name yours whatever you may like. Now you can share this public key with anyone, hence the name *public* key. I assure you that this is okay to do, sharing your public key will not do any harm to you, as the only way to decrypt messages is with the corresponding private key, which you should always keep secure and never share with anyone else. Your public key is essentially a safe way for others to send you encrypted messages or verify your digital signature. Actually, using the next command ‘gpg -a -output plaintext.txt.asc.sig --sign plaintext.txt’ will sign our file named plaintext.txt with our signature in ascii format. Essentially, this encrypts the file with our private key, meaning that it can ONLY be decrypted with our public key. Confirming that it was you that signed the file or message. The final command ‘gpg -e -a -u “Ethan” -r “Christopher” plaintext.txt.asc.sig’ encrypts the signed file to the recipient’s public key. Meaning it can only be decrypted with the recipient’s private key.

Now setup a prearranged listener in a separate folder using Netcat and the command: nc -l -p <PORT> -q 1 > ciphertext.txt.asc < /dev/null (remember to replace <PORT> with a port number of your choosing)



```
(kali@kali)-[~/Desktop/Encryption HW Listener]
$ nc -l -p 4242 -q 1 > ciphertext.txt.asc < /dev/null
```

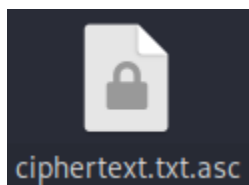
Here we can see that a file was created in our listener’s folder, however this file will not contain anything until a file is sent.

Now, let's open Wireshark and listen using the filter 'any'. Make sure not to have any browsers open, or anything else that might generate network traffic, as we only want to capture this single file transfer on Wireshark.

Now, go back to the original terminal and send the ciphertext file using the command:  
`nc 10.0.2.15 <PORT> < ciphertext.txt.asc`

Replacing <PORT> with the port number your prearranged listener is listening on, and 10.0.2.15 being the IP address of your Linux machine, you can do the command 'ip addr' to confirm it is correct.

Now the file should have been sent to the listener, to confirm, visit the folder of the listener, the ciphertext file should now have a lock on it, like this:



Using the cat command in the listener terminal, you can view the contents of the received encrypted file: `cat ciphertext.txt.asc`

It should be in ASCII encrypted format.

Now, let's go back to Wireshark and see if the file transfer was captured.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.15	10.0.2.15	TCP	76	37538 → 4242
2	0.000014481	10.0.2.15	10.0.2.15	TCP	76	4242 → 37538
3	0.000029329	10.0.2.15	10.0.2.15	TCP	68	37538 → 4242
4	0.000225151	10.0.2.15	10.0.2.15	TCP	68	4242 → 37538
5	0.000783709	10.0.2.15	10.0.2.15	TCP	324	37538 → 4242
6	0.000797871	10.0.2.15	10.0.2.15	TCP	68	4242 → 37538
7	0.000811488	10.0.2.15	10.0.2.15	TCP	68	37538 → 4242
8	0.000817196	10.0.2.15	10.0.2.15	TCP	68	4242 → 37538

It seems like Wireshark was able to capture a total of 8 packets during the encrypted file transfer. Scrolling down and viewing packet number 5, we can see the contents of the encrypted file:

5	0.000783709	10.0.2.15	10.0.2.15	TCP	324
<pre>00 00 03 04 00 06 00 00 00 00 00 00 00 08 00 ..... 45 00 01 34 16 cd 40 00 40 06 0a da 0a 00 02 0f E-4 @ @ ..... 0a 00 02 0f 92 a2 10 92 75 c1 b2 38 fe e9 ab 14 ..... u 8 ..... 80 18 02 00 19 44 00 00 01 01 08 0a 1f 7c 99 d7 ..... D ..... 1f 7c 99 d6 2d 2d 2d 2d 2d 42 45 47 49 4e 20 50  ...-BEGIN P 47 50 20 4d 45 53 53 41 47 45 2d 2d 2d 2d 2d 0a GP MESSA GE----- 0a 6a 41 39 45 43 51 4d 43 4a 34 51 78 34 36 74 .jA0ECQM CJ4Qx46t 32 4b 5a 66 2f 30 6e 38 42 45 43 4d 39 57 48 6f 2KZf/0n8 BECM9WHo 65 54 72 43 34 51 38 6c 68 67 63 30 73 4c 49 34 eTrC4Q8l hgc0sLI4 76 74 79 55 7a 61 6b 31 42 46 49 45 41 72 52 64 vtyUzak1 BFIEArRd 76 0a 70 73 37 41 44 36 64 56 70 75 36 62 59 37 v-ps7A06 dVpu6bY7 2f 66 4e 7a 58 59 6a 6e 65 32 79 6f 45 48 44 6f /fNzXYjn e2yoEHDo 6f 76 4a 6d 61 6f 32 55 58 7a 4d 69 65 73 6a 78 ovJmao2U XzMiesjx 68 4b 33 32 32 4c 4f 55 47 38 66 55 45 45 33 69 hK322LOU G8fUEE3i 50 53 0a 41 31 7a 4f 53 74 42 49 64 4b 35 30 59 PS-A1z0S tBIK50Y 36 64 64 55 33 30 57 64 73 75 6c 38 4c 66 6c 66 6ddU30wd suL8Llf 5a 71 67 72 6d 38 46 63 69 58 73 68 4a 54 46 56 Zqgrm8Fc iXshJTFV 31 44 35 54 39 54 47 55 61 71 50 49 50 57 37 61 1D5T9TGU aqPIPW7a 2b 36 75 0a 3d 59 6f 45 42 0a 2d 2d 2d 2d 2d 45 +6u.=YoE B-----E 4e 44 20 50 47 50 20 4d 45 53 53 41 47 45 2d 2d ND PGP M ESSAGE-- 2d 2d 2d 0a ----</pre>					

Although, these contents would mean nothing to a potential attacker, as it seems like complete gibberish. It can be seen that it says "BEGIN PGP MESSAGE" so an attacker or listener would know that it is an encrypted file, but they would not be able to do anything with this kind of information, not without a key.

Now, we will be using Steghide as the steganography tool to embed a plaintext file into a jpeg image. Here are a few commands that may be needed if you do not have steghide already installed:

`sudo apt-get install steghide` (to install steghide)

`sudo apt-get update` (if you run into the error: “Unable to locate the package steghide”, then use this command to update your package lists to see if it becomes available)

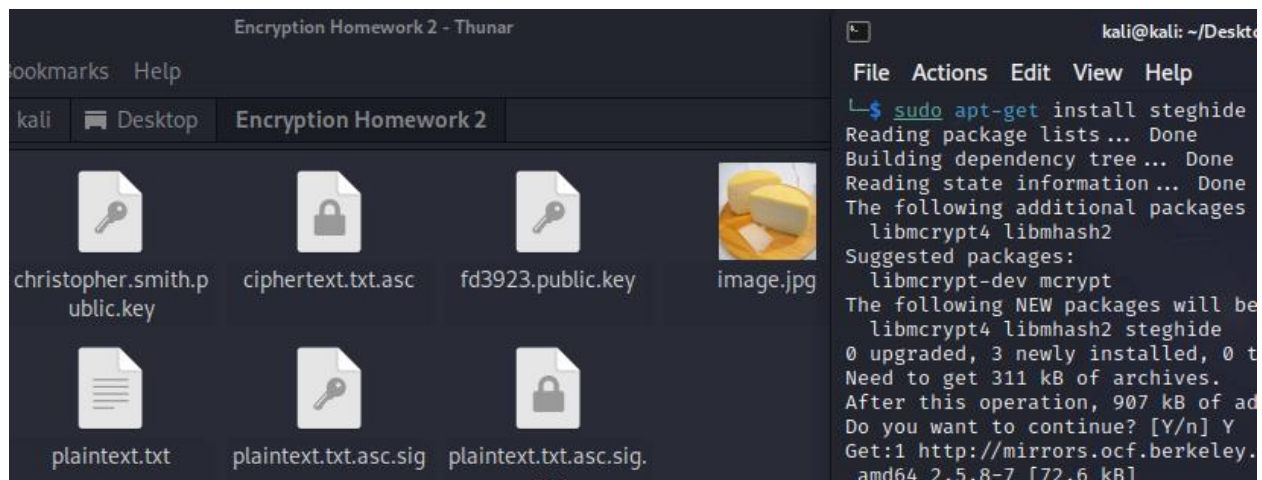
`man steghide` (view the steghide manual pages)

`steghide` (view a list of steghide commands)

Once you have steghide installed, we will now download an image file (of your choosing). You can download a jpeg found online, or use the command ‘wget’ in your terminal to download it:

wget <https://thisisanimagelink.com/1234.jpeg> && cp 1234.jpeg image.jpg

If you downloaded the image online, copy the image and rename the new one ‘image.jpg’ for simplicity: cp yourimage.jpg image.jpg

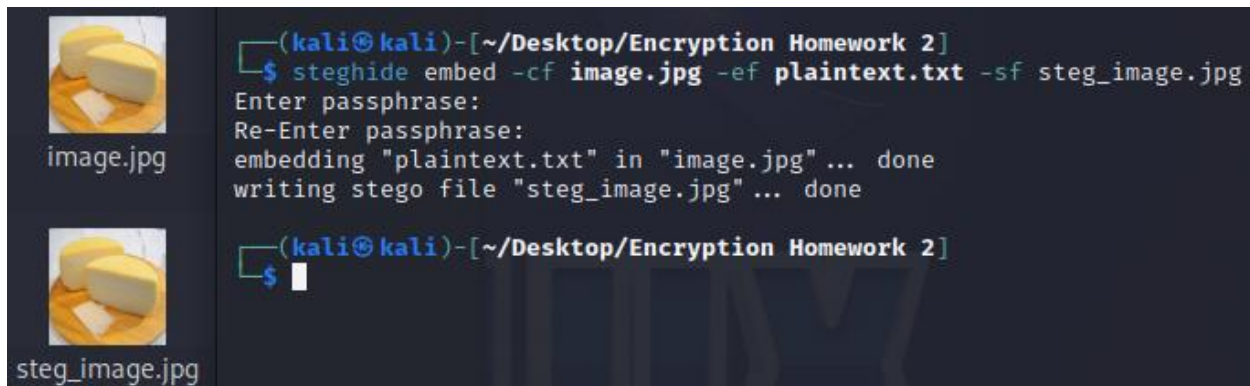


Notice, I am still in the same folder as previous, with the terminal open from the folder.

Now, we will embed a plaintext message into our beautiful image of cheese (yours will be something different), using the command:

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

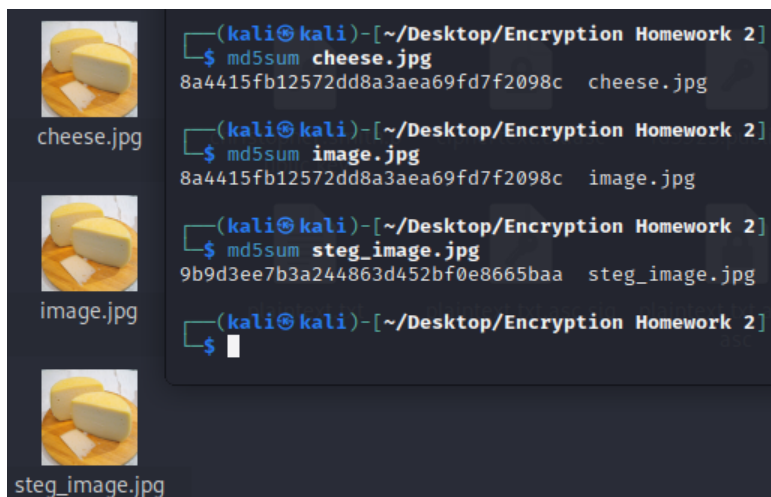




Notice, you will have to enter a password in order to embed your plaintext into the image, and our new image will be saved into the folder named: steg\_image.jpg. Use the password 'letmein' to embed your text file. Normally steganography does not use encryption, it just conceals the files within the image, however, Steghide actually encrypts your file within the image. This is why you need a password to embed, and unembed the file.

Now, although these images appear to be the same when viewing them, the plaintext file is still 'hidden' within the steg\_image.jpg. There will be no difference in the pixels of the image, so it is impossible to see any changes. However, there is still a way to confirm that these files are indeed different. By getting the md5 hash of the images, an md5 hash uses an algorithm in order to give each image a unique hash based on the contents of the file. Lets get the hash of all three of our images to see, one of the original image we downloaded, the renamed 'image.jpg' file, and the image we performed steganography on.

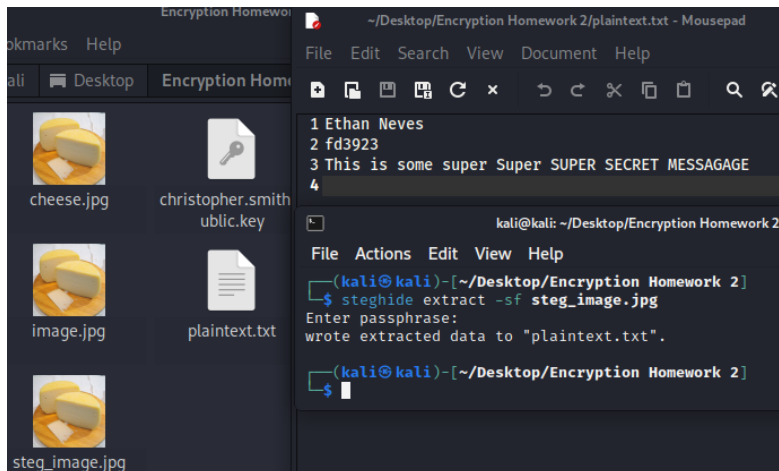
To do this we will use the command: md5sum image.jpg



We have now hashed all three images, and it can be seen that both 'cheese.jpg' and 'image.jpg' hashed exactly the same, telling us that the name of the file has no effect, just the contents. Viewing the hash for 'steg\_image.jpg' we can see that it is different. Which tells us that the file is not the same or has been changed.

Now, we will extract the data back out of our image. Firstly, either move, delete, or rename the 'plaintext.txt' as the extracted file from steg\_image will be saved as plaintext.txt. Now, to extract the text file back out of the image,

Use the command: `steghide extract -sf steg_image.jpg`



You will have to enter your password to extract the data. We can see there is now an extracted file named 'plaintext.txt' and when opening it we can see that it is the same text file that we embedded into the image, the one containing our super secret message!

## Conclusion

Through these security-based applications and tools, we were able to perform symmetric and asymmetric encryption tasks, steganography, hashing, file transfers, and network analysis. Encrypting data with GPG, sending it over the network with Netcat and capturing the send with WireShark. Performing steganography with steghide, and checking the integrity of those images using an md5 hash.

For symmetric and asymmetric encryption we used the GPG application within Kali Linux, this is a useful application for encrypting, decrypting, and generating keys. Some use cases include, email encryption, file encryption, and digital signatures. The limitations of the GPG application fall under the same limitations of symmetric and asymmetric encryption. With symmetric encryption requiring a secure key exchange, and not being good for more than a one-to-one transmission. While asymmetric encryption generally provides good security, the main issue still falls under having a secure key exchange. The GPG application can provide users with a lot of security through the use of digital signatures; authentication, non-repudiation, and data integrity. With these signatures, you can be sure of who encrypted the information (authentication and non-repudiation), and that the information has not been altered (integrity). GPG also provides data confidentiality through symmetric and asymmetric encryption, however it does not directly provide any methods to perform access control.



The Netcat application is a useful tool for communication over the network, however it lacks in its security features, and is not suitable for data transfers of sensitive information. Because of this, Netcat does not provide authentication, access control, data confidentiality, data integrity, or non-repudiation. Similarly, the WireShark application does not provide any of these security features. However, it is a great tool for analyzing the network and capturing data sent over the network.

The Steghide application that was used for steganography is a tool used in order to embed information within images and is effective at doing so. The limitations lie in how this is the main, and only use case of Steghide. Additionally, it is detectable, as we showcased using the md5 hash, and how it is different after the file is embedded. For security, in general steganography does not directly provide any methods of encryption, although you could embed pre-encrypted files into images. Steghide does not provide authentication, access control, data integrity, or non-repudiation. However, as mentioned previously it does provide a level of data confidentiality through encryption as it hides information within compressed media files, prompting a password from the user in order to embed and unembed files from an image.

The md5 application is used in order to create a hash of data, and can be used to ensure that files downloaded online, or transferred are not corrupted during the process. The limitation of this application is the chance of a hashing collision. Although the chance is small, it still exists, which makes it possible for attackers to potentially generate a malicious file that hashes to the same value as the file you are trying to download or receive. MD5 does not provide authentication, access control, confidentiality, or non-repudiation. However, it does provide good data integrity, by comparing the hash values of the original data and the received data, the users are able to determine if the data was altered while being sent. Although, as mentioned previously, the possibility of collisions still applies.

This exemplifies how a combination of these tools can provide some strong encryption. Seen through a capture of the packet in WireShark, the data sent over Netcat was encrypted and would've been completely unreadable to an attacker. This shows the importance of keeping our network traffic encrypted, as it is able to protect our data over the network and provide us with good aspects of security. Showing how in the internet space you need a combination of; authentication, access control, data confidentiality, data integrity, and non-repudiation, in order to be somewhat confident that your data is 'secure'. Additionally, we were able to pick up on some of the limitations of these applications and tools, like hashing collisions and key exchanges. Showcasing how even though your data may seem extremely secure, there are always improvements that can be made.