Brooklyn Dressel
Assignment 2

Abstract

This project uses the encryption provided by GPG to complete both symmetric encryption using a password when prompted as well as the asymmetric encryption using a provided public key and using GPG to generate a new key with a prompted password. The file is then signed using a private key. These methods are used to encrypt a simple text file and send it through Netcat, intercepting it with WireShark the data is encrypted and therefore not accessible without knowing the key or encryption method. This proved to be a much safer method to send data across networks without losing data confidentiality and since the file is signed it also ensures authentication and non-repudiation. After this, Steghide is used to embed the original text file into an image file. Furthermore, md5 is used to find the size of both the original and the embedded image files. Upon comparison it is evident that the image integrity is compromised as the embedded file size is larger than the original. Additionally, renaming the original image and comparing the file size shows that the file data has not changed and the file is still identical to the original, renaming had no impact.

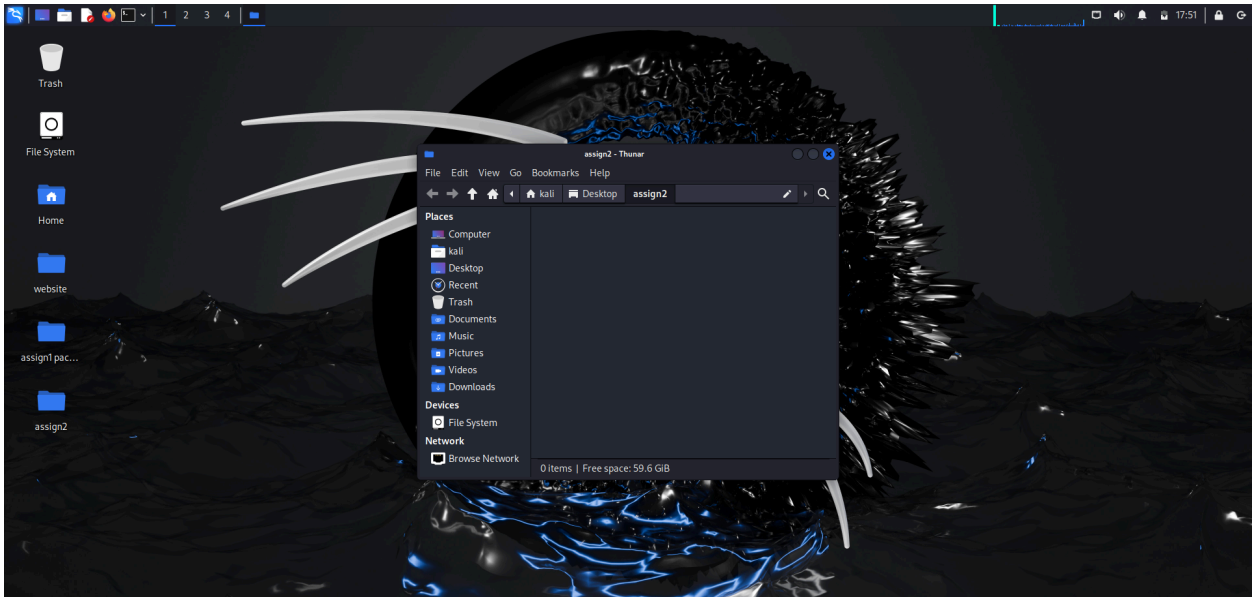
Introduction

Used Kali Linux virtual machine and the applications provided by it. GPG (GNU Privacy Guard) is used for symmetric encryption using passwords, as well as asymmetric encryption using a public key for encryption and a private key for decryption and signing. Netcat listener and sender commands are used to send the encrypted files across the network. WireShark is used to intercept these messages and access the encrypted data. Steganography tools, Steghide is used to embed a text file into an image file using a password, as well as extract the text file from altered jpeg. Finally, md5 tools are used to get the md5 hash of the original image, the embedded image, and the renamed original image. Additionally, all commands used during this project are provided separately.

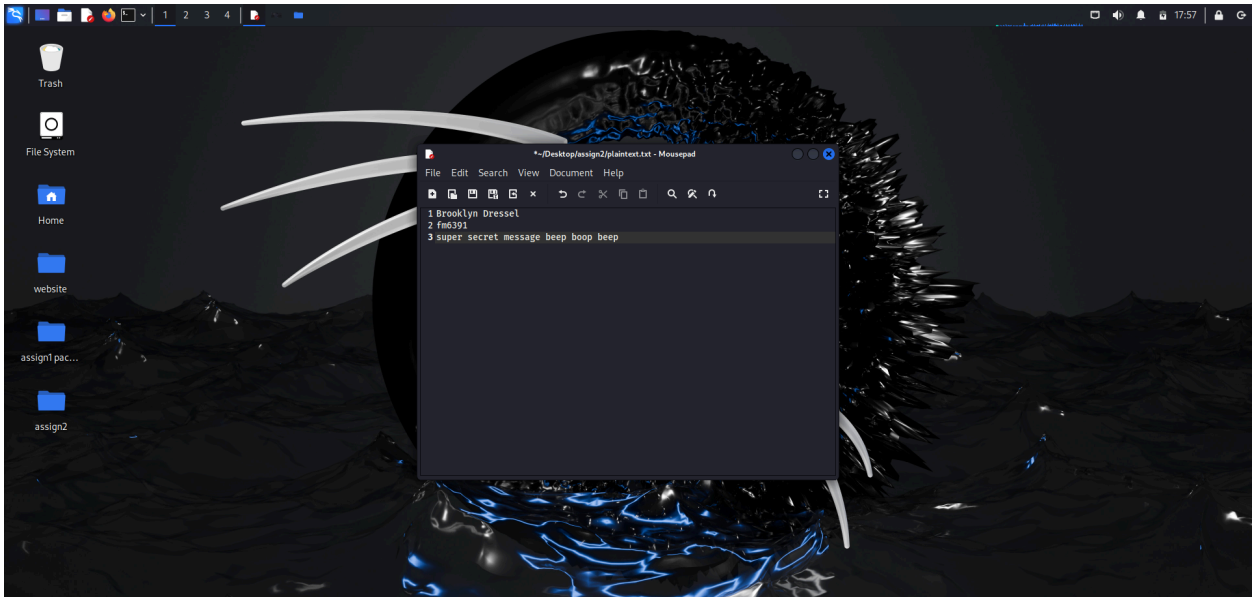
Summary of Results

The first thing I did upon starting up Kali Linux was to right click on the desktop and create a new folder for this project. This would help to ensure organization, ease-of-use, and consistency. I named the folder “assign2”. In order to help me document this

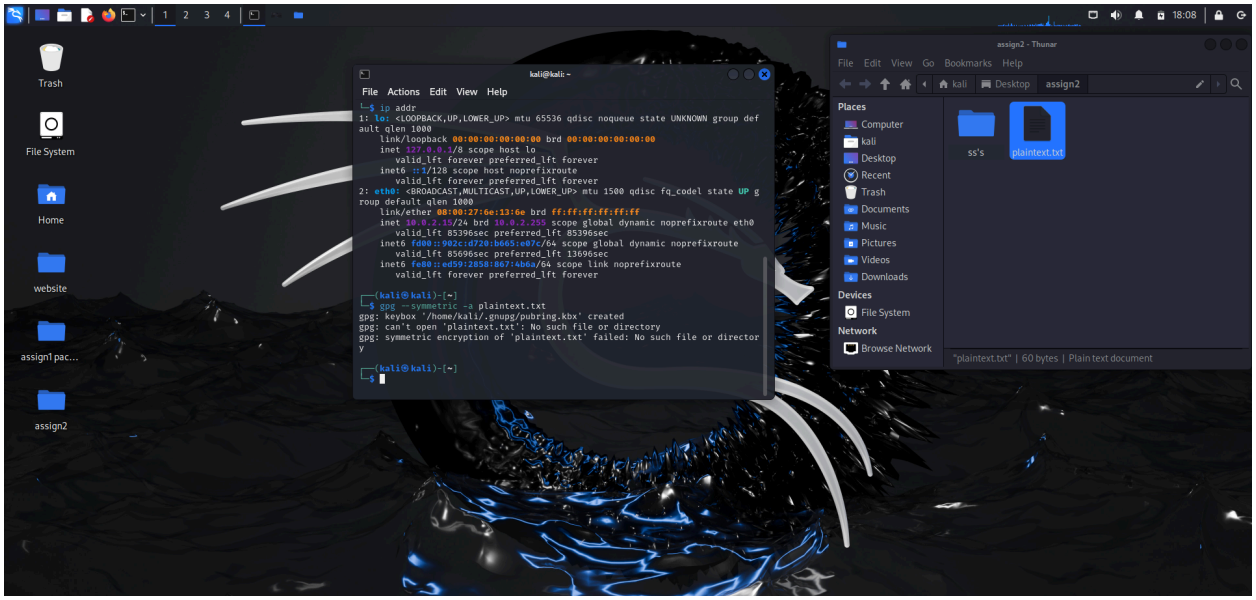
process efficiently I created a folder inside “assign2” called “ss’s” so I had a place to keep all of my screenshots as I went along.



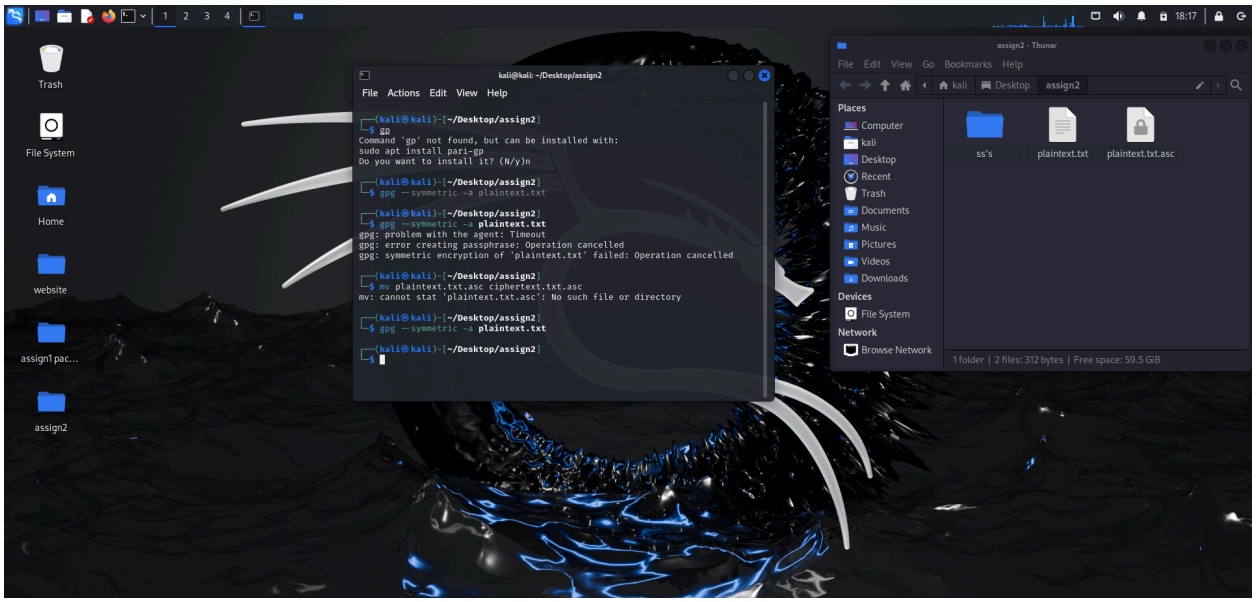
Next, I right clicked inside the folder to create a new file. I named this file “plaintext.txt”. Inside the file I wrote three lines of text. The first was my name “Brooklyn Dressel”, the next my net id “fm6391” and the last was my *secret* message “super secret message beep boop beep”. I saved the text file and closed the window.



I opened a terminal and used command `1` to encrypt “plaintext.txt” using symmetric encryption. However, the feedback from the console said that the file was not found. I immediately realized that I needed to open the console from within the folder I had created, “assign2”, as that was where my file was located.

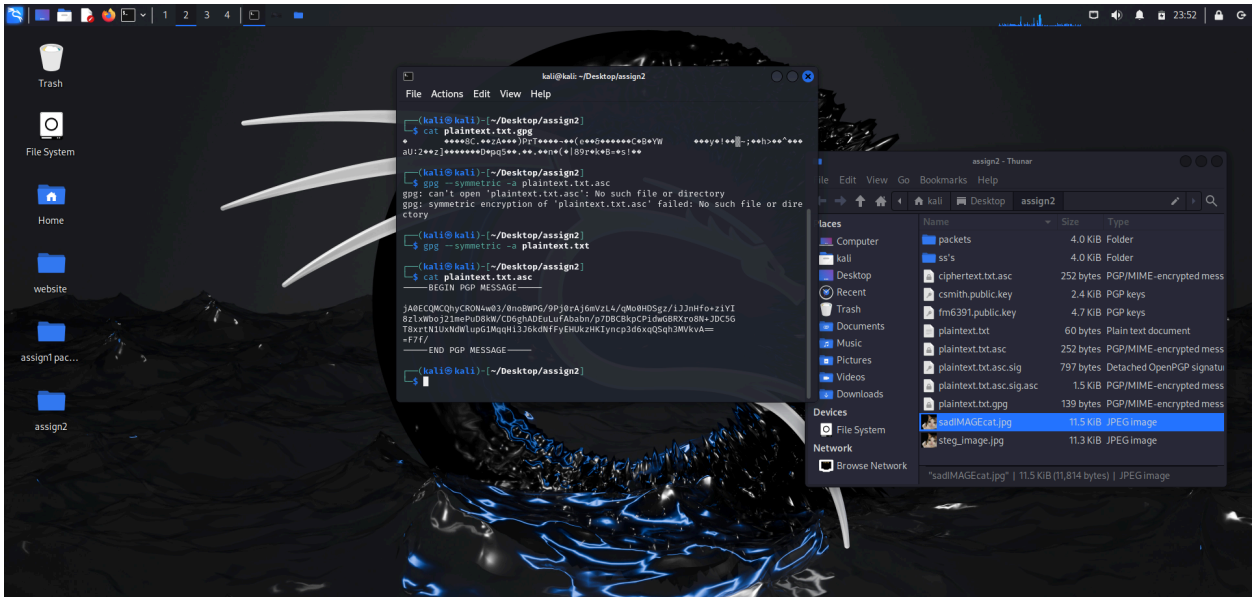


Once the terminal was opened in the correct location I was able to run command `1`. After a slight error putting in the password, I was able to successfully use symmetric encryption to encrypt “plaintext.txt” using the password “letmein”.



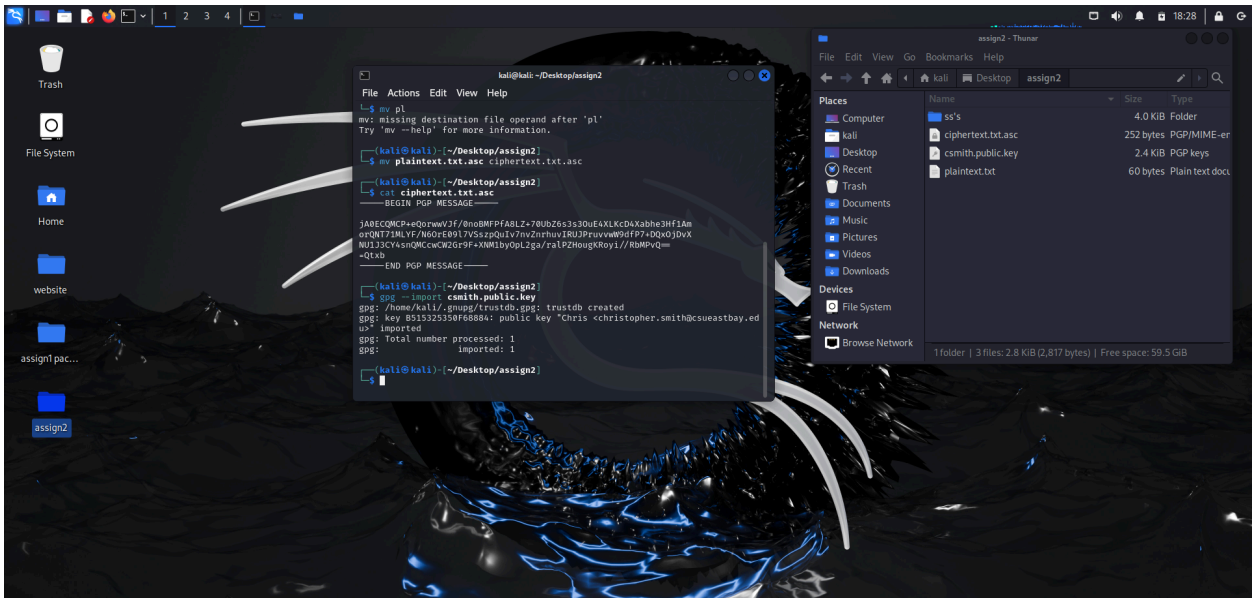
Using command `2` to display the result of this symmetric encryption, it is evident that the encryption performed on the bits were done in such a way that many of the characters can no longer be converted to ascii to be read. This is why the output of this file has many characters that are not recognized.

Next command `3` is used to perform symmetric encryption in such a way that the bits would still be recognizable ASCII characters. And so, as shown by running command `4` the result of this encryption is all recognizable ASCII characters, however they are all still completely scrambled such that you cannot tell what the contents of the text file were without decrypting it.

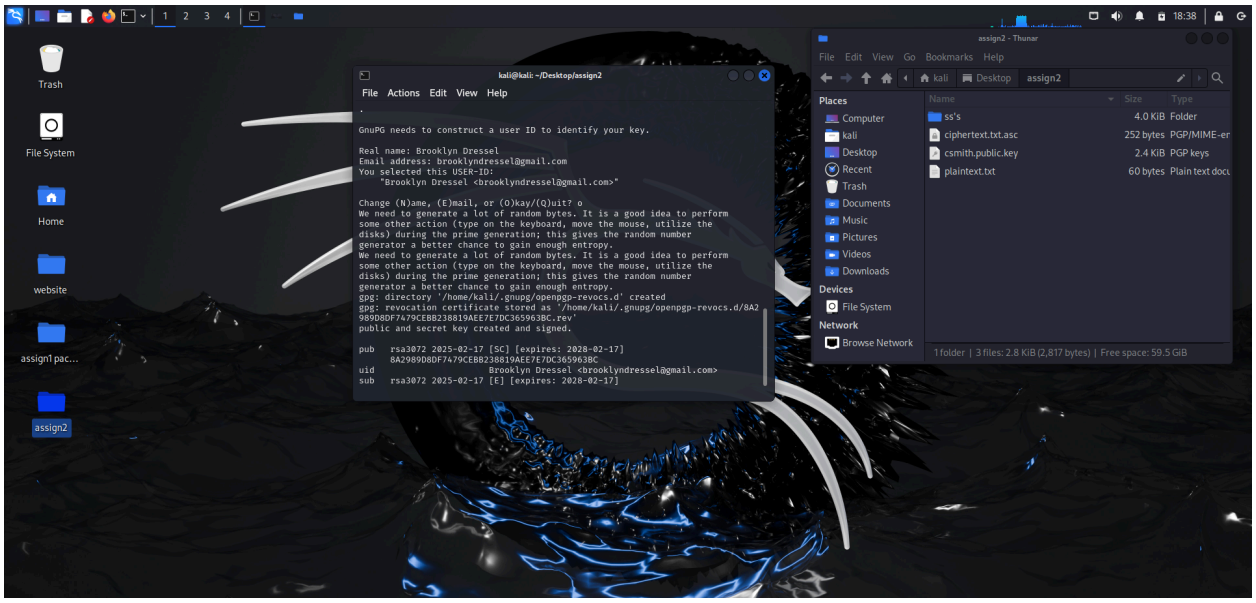


The next step is asymmetric encryption. To start this I uploaded the public key “csmith.pub.key” to Google Drive and downloaded it on my Kali Linux virtual machine. Using command `5` to import the public key “csmith.pub.key”, the following feedback was returned, showing the key id, as well as the name and email address of the key's owner.

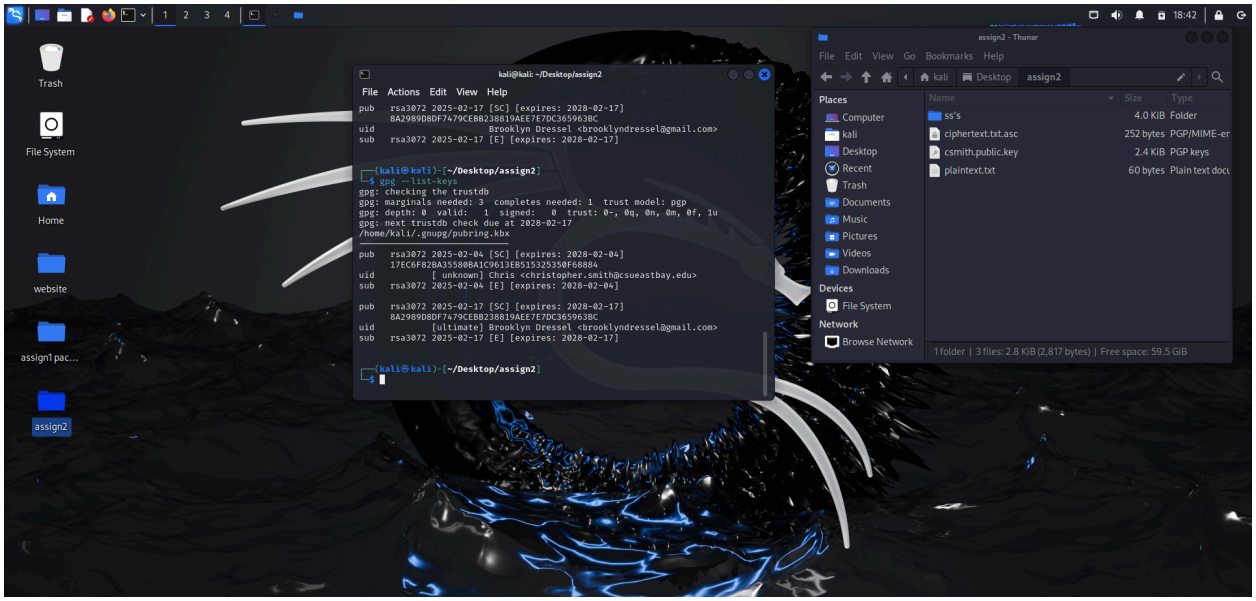
After this I used command `6` to display the keys I had saved, I will address the result of this command in a few moments.



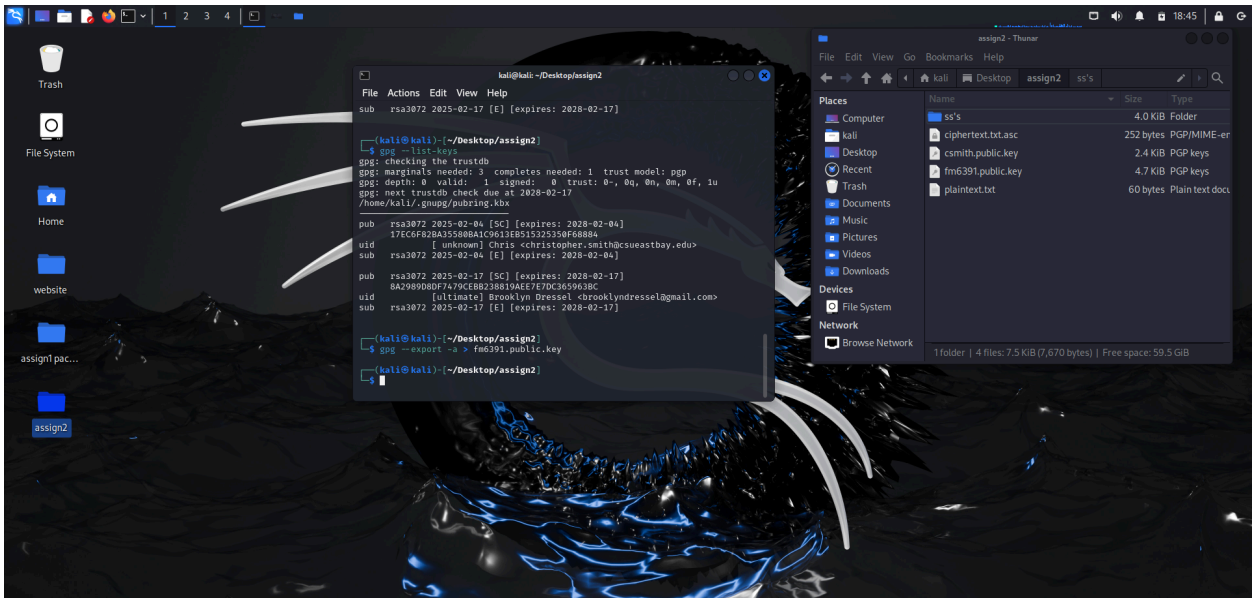
In order to successfully complete asymmetric encryption I would need a private key of my own, thus I used command `7` to generate my own key. I set my name to “Brooklyn Dressel” and my email address to “brooklyndressel@gmail.com”. After confirming that this information was correct the console gave feedback as it generated bytes for the keys, and finally gave the output “public and secret key created and signed.” Alongside this message it displayed the information for the key.



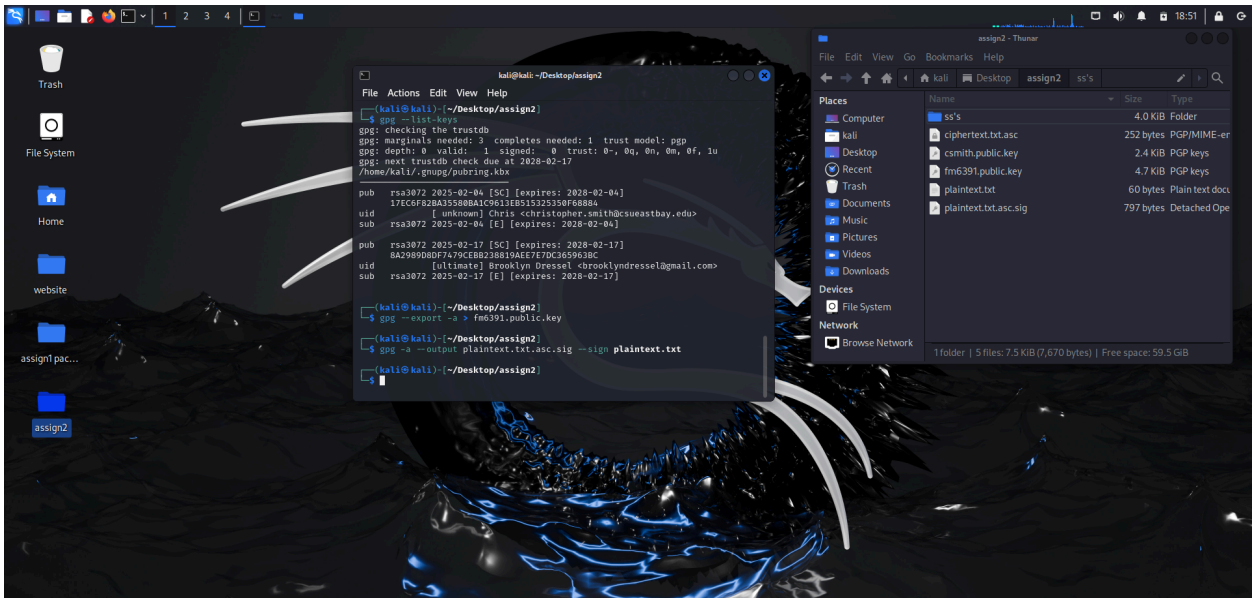
Once again I used command `6` to view all my saved keys, to ensure they had successfully generated and been imported. The result is shown below, the date created, the date they key will expire, the random bytes generated, as well as the keys' owner's name and email address.



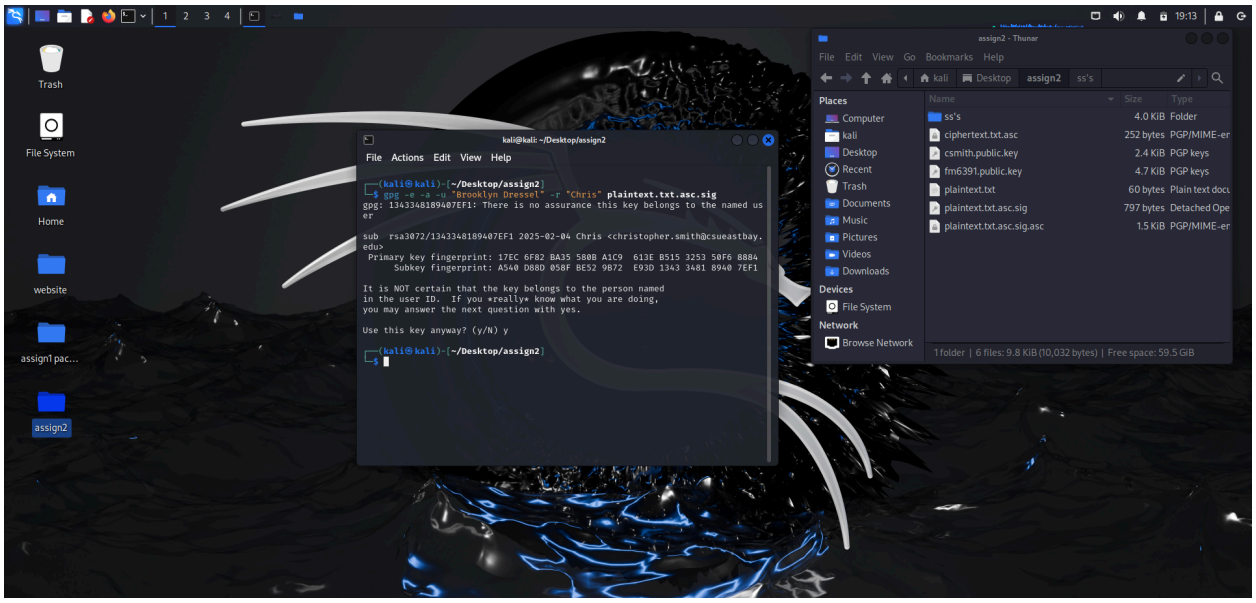
Since asymmetric encryption only works if other users have your public key, I next needed to export my key, I did so using command `8`. This resulted in a file "fm6391.public.key" being saved into the folder "assign2". This key can be used by others to encrypt files to be sent to me, which my secret key can be used to decrypt.



In order to ensure authentication and non-repudiation, I then used command `9` to use ASCII armored output, sign the “plaintext.txt” file using my private key, and save the new file as “plaintext.txt.asc.sig”. Thus a resulting file “plaintext.txt.asc.sig” can be seen in the “assign2” folder.



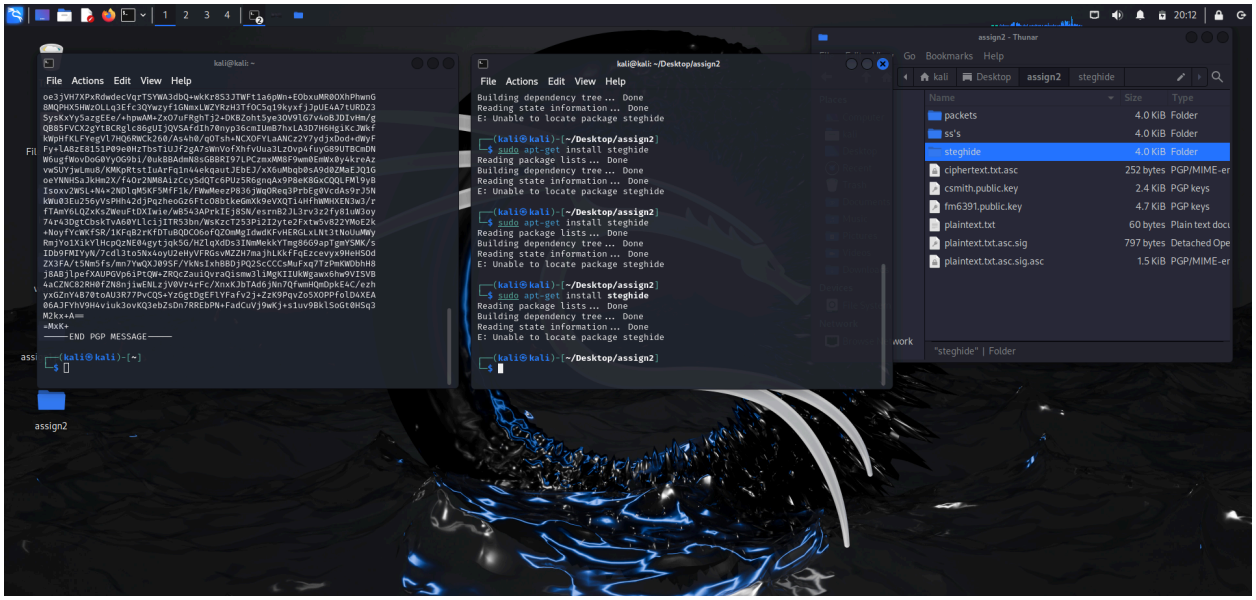
Finally, in order to send this file to “Chris” it must be encrypted using their public key. Thus the command `10` is used, after confirming to use this key, a newly encrypted file “plaintext.txt.asc.sig.asc” is generated inside the folder “assign2”.



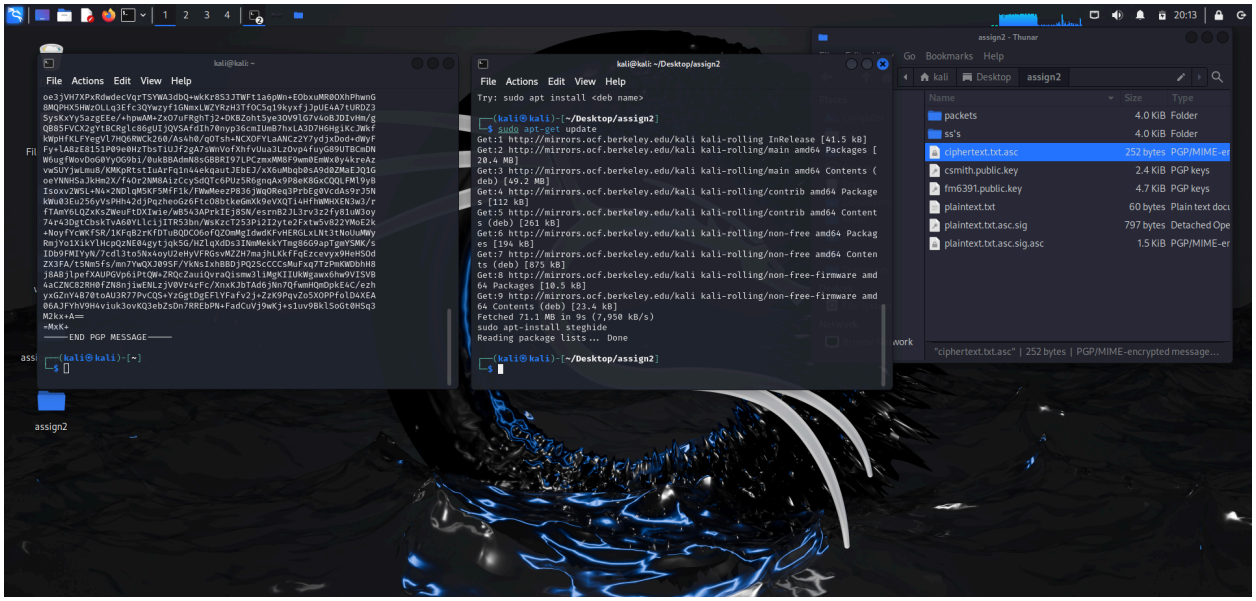
In order to send this encrypted file over the network, as well as intercept it, firstly, WireShark is opened. It is set to capture packets from any device on the network to ensure I capture my desired packets. Next two Netcat terminals are opened, one to

listen, and the other to send the encrypted file. The listener window is shown below on the left, and using command `11`, I set up a listener using my ip address, the port number “31337”, set it to close the connection after one second of inactivity, saving all sent data (nothing read from the terminal) to the file “ciphertxt.txt.asc”. Next the client terminal, or sender terminal, shown as the middle window in the image below, is set up using command `12`. This command designates the destination address and port and sends the encrypted file “plaintext.txt.asc.sig.asc” to the listening connection. One second after this file is received the connection closes and I run command `13` to read the contents of whatever file was received by the listener. The ASCII armored encrypted file contents are shown below and can only be decrypted using “Chris” ‘s private key. Inside WireShark I stopped the capture of packets and found the file with this data, and found it to be unintelligible as the files data is encrypted.

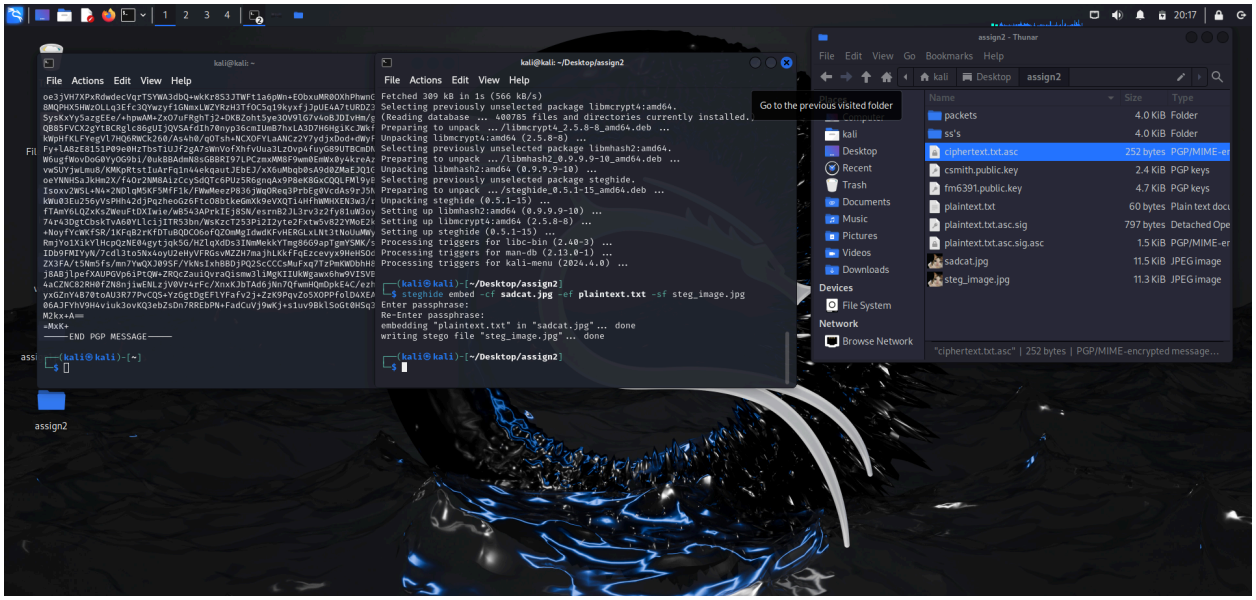
The next topic is using steganography tools to embed messages inside images. First I had to figure out how to download steghide. At first I thought I needed to download this separately and upload it inside my Kali Linux virtual machine, however this was not working.



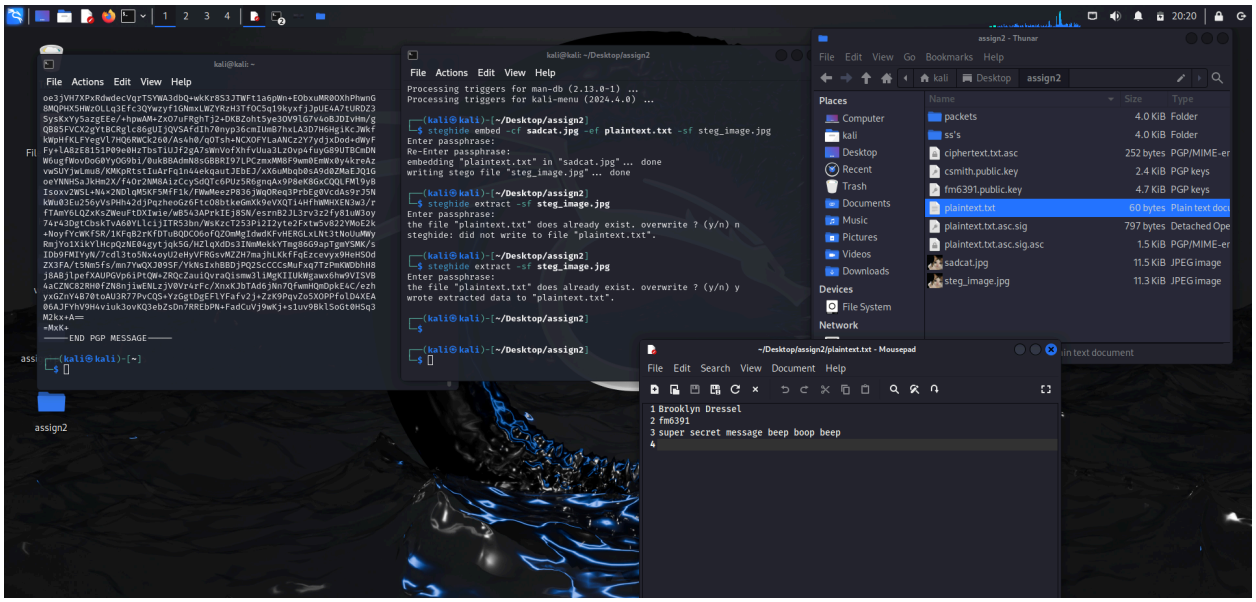
After a quick email to my professor, I was provided with commands `14` and `15` which enabled me to update my virtual machine and install steghide, respectively.



Now that I had access to steghide I downloaded an image named “sadcat.jpg” and used command `16` to embed the text file “plaintext.txt” into the image using the password “letmein” and save it as a new file named “steg_image.jpg”. This file is shown in the folder “assign2”.

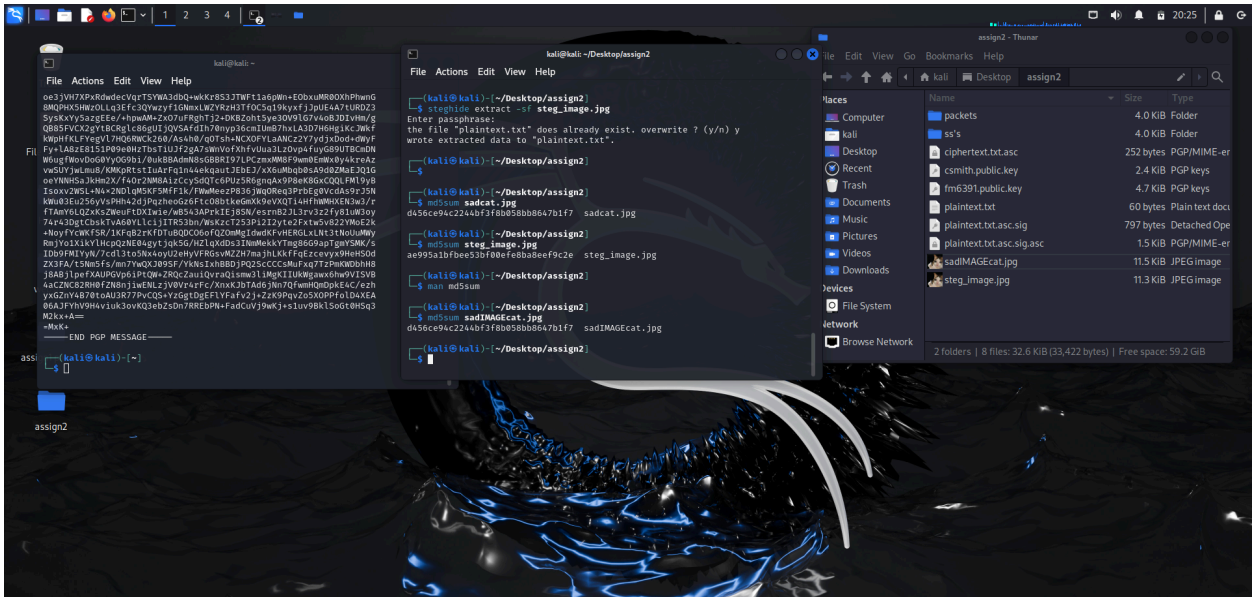


In order to extract the embedded text file command `17` was used and the password “letmein” was once again imputed. The terminal prompted me for permission to overwrite the file “plaintext.txt” since it already existed. I gave permission and opened the newly overwritten file, which was shown to have the exact same data as when it was originally created. This means that the embedding and extracting were successful.



Finally, in order to check the data integrity of the image files I used commands `18`, `19`, and `20` to compare the hashes of the images, the results are shown below. “md5sum” computes and prints the MD5 hash (checksum) of the images. This can be used to tell if the two files are identical, as it is rare to end up with the exact same hash map. From

this hashing we can see that the images “sadcat.jpg” and “steg_image.jpg” are indeed different, as the result of the hashes produces different results. However, after renaming the original image file “sadcat.jpg” to “sadIMAGEcat.jpg” and comparing the new hash value, it is evident that changing the name did not affect the hash value. This means that there is a fundamental difference between the images “sadcat.jpg” and “steg_image.jpg” apart from just the two files names. This is because the text file “plaintext.txt” was embedded into the original image, resulting in “steg_image.jpg” having many more bits than it’s original counterpart.



Conclusion

In conclusion, through this project we have learned that there are many different ways to encrypt data. Starting with symmetric encryption, where the same key is used both to encrypt and decrypt and can be done using a simple password. This process of encryption ensures a certain level of access control, authentication, and data confidentiality as only those who know the password can decrypt the data to be able to read it. Next, asymmetric encryption is possible where users have access to others’ public keys, allowing them to encrypt data meant for that user. And each user has access to their own private, or secret, key which allows them to decrypt this data. This process of encryption has a much higher level of access control, authentication, and data confidentiality because only the intended recipient can decrypt the data. Whereas with symmetric encryption anyone who gained access to the password would also be able to gain access to the unencrypted data.

Furthermore, encryption can be completely random, resulting in random 1's and 0's that no longer translate into ASCII characters. Or, this encryption can be done as ASCII armored encryption, where the bytes are still randomized, but done so in a particular way so that they are still able to be translated to ASCII characters. Using ASCII armored encryption is useful when the encrypted data needs to be sent as a text file.

In addition to asymmetric encryption allowing users to specify their intended recipient, private and public keys can also be used to sign the files and data being encrypted and sent. This ensures authentication and non-repudiation as only the individual with the private key can sign it that exact way. Thus you can tell who exactly sent the data. Additionally, this can ensure data integrity, as if the data was scrambled the signature would be unintelligible and would not decrypt correctly. Additionally, unencrypted messages can be sent using encrypted signatures for a quick, moderately secure method to send data. This is often useful in cases where mass messages need to be sent.

Both forms of encryption ensure that data sent over networks is not easily readable by attackers. However, neither of these methods entirely prevent attackers from seeing the encrypted data. The fact that data is sent is still able to be seen by attackers, as shown through WireShark.

Next, we saw that by using steganography tools, such as Steghide, text files were able to be hidden in images. This allows for a greater level of data confidentiality, as without knowing that there is a message hidden in the image, it may go undetected by attackers. Without access to the original image to compare the hash map to, it would be hard to tell if the file was simply an image or had data embedded into it. This method relies on keys as well so it also ensures authentication and access control. This also ensures data integrity to a degree, as if the data was modified, the extracted data would be corrupted and the hash map would be modified. Since this method relies on an image file, it is impractical for large files to be embedded as it would not be possible to properly embed it into the file.

Using md5 hashes (checksum) is fairly practical for ensuring data integrity. In the case you know the hash of the original file, it is highly unlikely that any corrupted file would produce the same hash, however not impossible.

Finally, GPG has complex key management, but is slow on large files, and doesn't provide a large enough key size to be secure against brute force attacks. Steghide can be fairly easily detected and is only useful for short length data embedding. And md5sum is weak as a security measure as it does not ensure authentication, access

control, data confidentiality, or non-repudiation, only data integrity. Additionally, the size of md5sum is much too small for the current technology of this time and is prone to collisions, meaning that there are too many identical hash map combinations to be considered trustworthy.