

# CSG 1105 / 5130 - Applied Communications

## Week 9 Tutorial

### Objectives

- To learn about encryption
- To learn about hashing
- To learn about steganography

### By the end of this workshop you should be able to

- Discuss the differences between symmetric and asymmetric encryption
- Discuss the differences between encryption and hashing
- Be able to encrypt and decrypt a file
- Be able to generate a hash file to verify a file
- Be able to perform steganography

### Required Downloads

*All downloads this week are available in Module 9 of the Unit Schedule. You will need to use your own computer to do this or visit the Tutorial and watch the Tutor show this as these software packages are not available in the Lab.*

#### - **GPG4USB**

- A free encryption tool under the GNU public license to mirror the capabilities of PGP (Pretty Good Privacy).

#### - **OpenPuff**

- A free application which will allow you to hide a file inside another.

#### - **HashCalc**

- A free application which will generate various different hash codes with your input.

### Optional Downloads

- A recording of this Tutorial from blackboard; discussions are held in class and clarifications are made too.

## **Learning Module 1 - Symmetric and Asymmetric Algorithms**

Symmetric Algorithms are cryptography algorithms that use the same cryptographic key for both encryption (ie, plaintext to ciphertext) and for decryption (ie, ciphertext to plaintext). The keys can be identical, or have a simple method to translate from one key to another.

The keys are generated from a shared secret which all parties involved know. The shared secret can be a word, series of characters or some other combination of data used to create a key to then encrypt or decrypt the message.

An example of this method is that I generate a key to encrypt my information. I could then provide that key to someone else via a USB or other physical media (it can be sent online however this is generally considered to be a bad idea). They can then decrypt any ciphertext I send them with ease.

The need for each party to have access to the shared key is the biggest drawback of symmetric algorithms, as sharing that key over the internet poses a risk in itself. This is where asymmetric algorithms come in, however, asymmetric algorithms are significantly slower than symmetric.

Some popular forms of symmetric encryption algorithms are AES, 3DES and Twofish.

Asymmetric Algorithms use the same concept of having keys for encrypt and decryption although it has two distinct keys for the two methods. There is a public key and private keys are different however they are linked through mathematical functions which allow them to encrypt and decrypt data respectively.

The public key is the tool used to encrypt plaintext into ciphertext. This key is provided to anyone that wishes to be able to communicate with whomever holds the private key. As such, the public key, can be safely distributed to other people.

The private key is the only one that can decrypt the ciphertext back to plaintext. The security of the private key is what determines the security of the encrypted communication. The private key should never be distributed.

An example of this is, I generate myself a public key and private key pair. I can then give anyone else I know the public key so they can communicate to me. I keep my private key safe on my personal belongings at all times. Anyone can then encrypt plaintext and send it to me, where I can then decrypt that ciphertext. If I wanted to be able to encrypt and send a message or information back to them, they would need to provide me with their own public key, so I can encrypt it to a form their private key can decrypt.

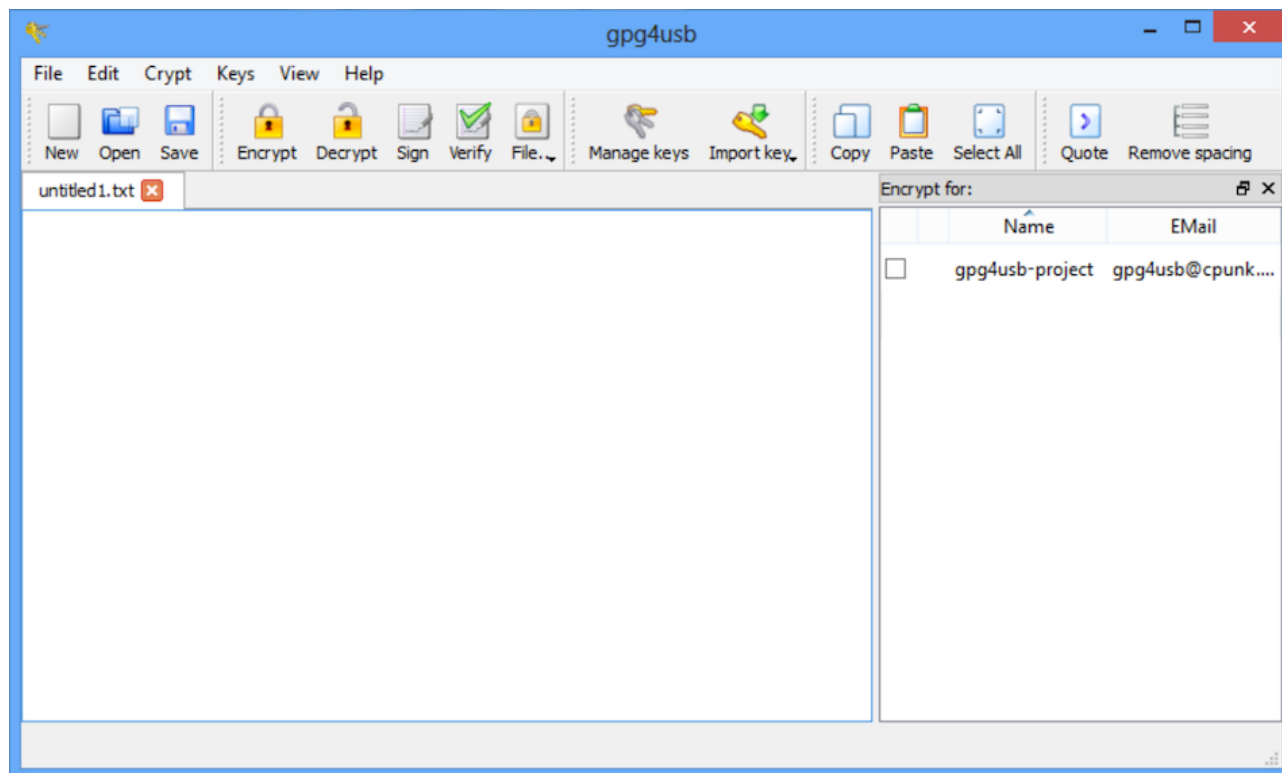
Some popular forms of asymmetric encryption algorithms are GPG, SSL, and SSH.

These two methods can be combined by making use of the Diffie-Hellman Key Exchange process. By using this method two people, for example James and Stacey, both generate a public and private key each. James gives Stacey his public key and Stacey gives James her public key. They can then combine their own private key with the others public key to create a shared secret for an asymmetric encryption algorithm.

## Activity Module 1 - Encryption

For this activity we will be making use of GPG4USB, which is GNU Privacy Guard. This is an alternative to the PGP encryption which is a part of the OpenPGP system.

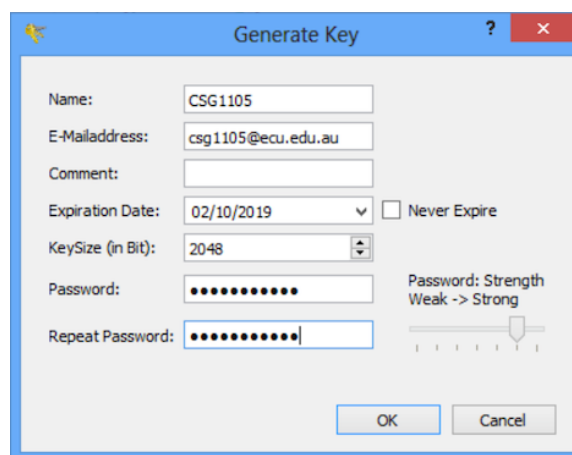
Download and open GPG4USB and run the 'start\_windows.exe' application in the folder. After the initial setup you should be greeted by the following screen:



Let's first investigate creating a key to use for our encryption. Click on **Keys > Manage Keys**. You should then see a screen with a default key and some other buttons at the top. Now, click on **Key > Generate Key**. Fill in the information so that it looks like the information to the right. The password I have used is:

CSG1105pass

Click OK and let it generate the key for you. You'll notice that this takes quite some time to do. Once you've generated the key, select the check box next to it and export it to a file. **Right Click** the file and choose **Open with** and use Notepad. Have a look at the contents, it should be similar to the screenshot on the next page.





CSG1105 csg1105@ecu.edu.au(6FD69E49E92FCE59)\_pub.asc - N...

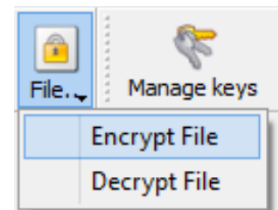
```
File Edit Format View Help
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1

mQENBFQtAs0BCAC2YcXdKr6F0xs+VoKJFHGgCW29TFxGZ1VYS2j/6/svTBRH20+0
drQgVtq8SdBekMwaxn84dstuMy70cPoOuhufyThvxmM4IamFyCKS1tzw6keI6Nhu
4QWgPW14UXq1xArhipLPcJb+zTLxNwDxr6PSqOnRJ9BCNYrCbcTtn/C+IncrUafL
DITXAey1FPBHjJtHCoB74PEkSA5zGNmCvJCWcEvczcS53mBkEApay89r8H2NxJCR
aCaQyZwDj1WcG/XXx/gn0r4T9ibjk5UxYHERLDnfRGaYWPtcrIaKcQey8wuEoS7
ufeQrGgCuyYVQRQkZA2oj+zD0kdMo51PJZDtABEBAAG0HENTRzExMDUgPGNzZzEx
MDVAZWw1LmVkdS5hdT6JATwEEwEKACYFA1QtAs0CGwMFCQ1mdTMFCwkIBwMFFQoJ
CAEEFgTBAAIeA0TXgAAKCRBy1p516S/0WbUlsR/9zTK0u1LkKeE01018TVGRTa5dg
```

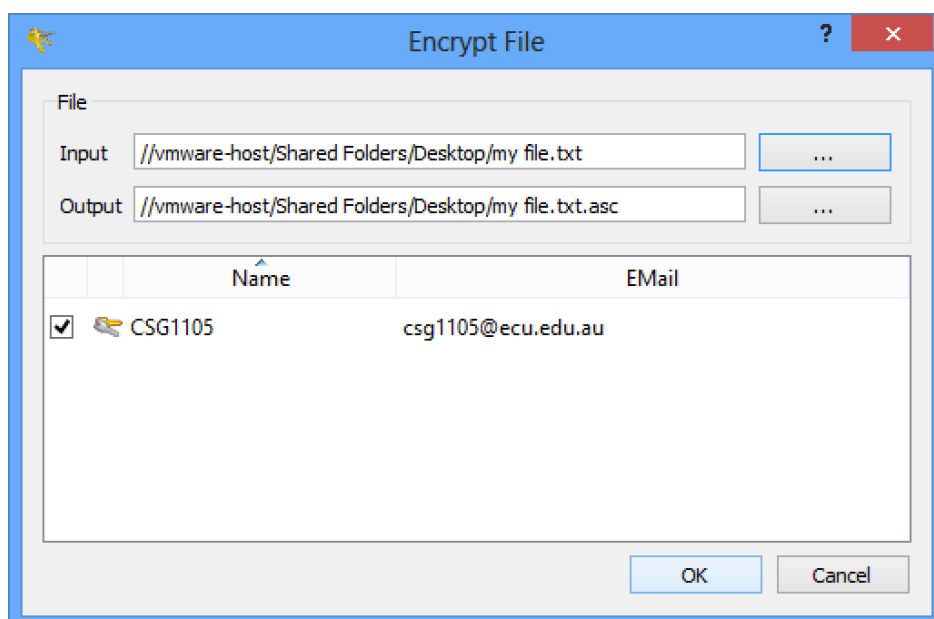
You'll notice that even though we used the exact same information, the key that has been generated is still different. This is because it also uses time and the machine identifier to aid in the generation of the key. This is part of why the key encryption system is secure, we have also used one of the highest levels of encryption with 2048-bit key size.

You'll also notice that it says PGP PUBLIC KEY BLOCK. This is because when you export the key you are exporting the **public** key which allows encryption of files and information so you can then decrypt it with your **private** key (which cannot be shared).

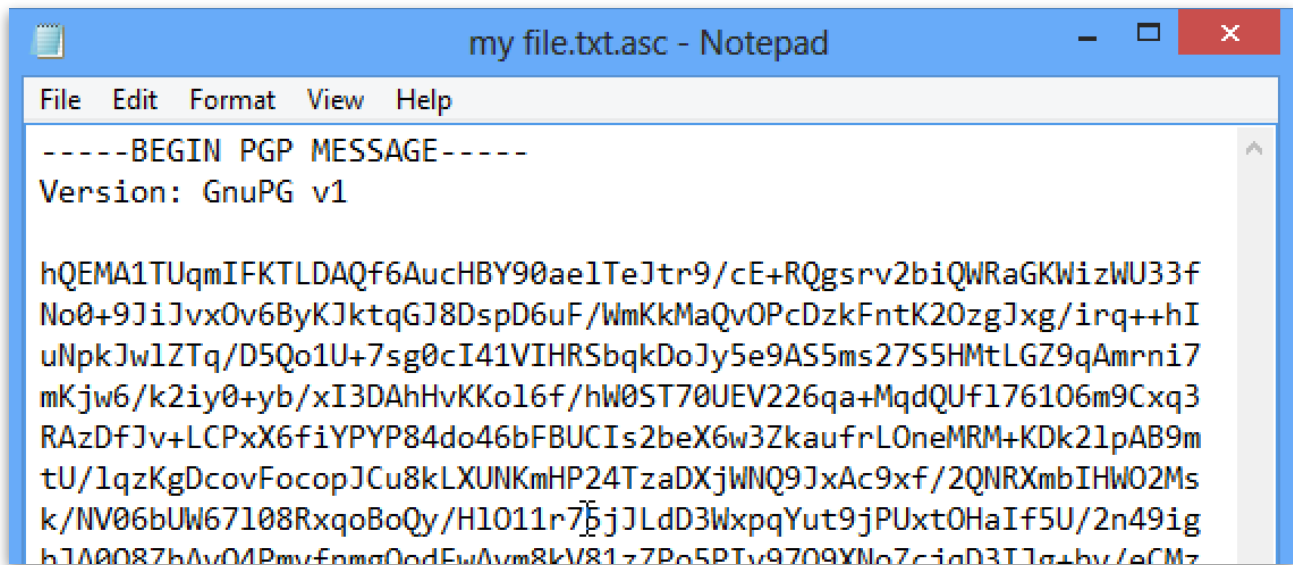
Now, let's play with encrypting and decrypting files. Start by creating a new text document on your desktop and name it whatever you wish. I've called mine 'my file.txt'. Now back in GPG4USB click on the 'File..' button and choose 'Encrypt File'. It should look like mine to the right.



Now you'll see the below window appear and you can select the file you wish to encrypt by clicking on the ... button at the top, then proceed to navigate to your text file and select it. Once you have done that, ensure your key is checked below and click **Encrypt**. It should look like the picture below.



Once you have encrypted the file you should see another file with an identical name appended with **.asc** at the end. This is because it has encrypted it into an ASCII file. Try opening this with **Notepad** once again.



```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1

hQEMA1TUqmIFKTLDAQf6AuchBY90ae1TeJtr9/cE+RQgsrv2biQWRaGKWizWU33f
No0+9JiJvx0v6ByKJktqGJ8DspD6uF/WmKkMaQvOPcDzkFntK20zgJxg/irq++hI
uNpkJw1ZTq/D5Qo1U+7sg0cI41VIHRSbqkDoJy5e9AS5ms27S5HMTLGZ9qAmrni7
mKjw6/k2iy0+yb/xI3DAhHvKKo16f/hW0ST70UEV226qa+MqdQUf176106m9Cxq3
RAzDfJv+LCPxX6fiYPYP84do46bFBUCIs2beX6w3ZkaufRLOneMRM+KDK21pAB9m
tU/1qzKgDcovFocopJCu8kLXUNKmHP24TzaDXjWNQ9JxAc9xf/2QNRXmbIHW02Ms
k/NV06bUW67108RxqoBoQy/H1011r75jJLdD3WxpqYut9jPUxt0HaIf5U/2n49ig
b1A0087bAv04Pmvfomg0odEwAvm8kV81z7Po5PTy9709XNo7cjoD3T1g+by/eCMz
```

Our empty notepad file has been encrypted into this file made of random strings of ciphertext. We can then decrypt the file once again using a similar process. Go ahead and try decrypting the file back to plaintext.

Now, to properly show the use of public keys and private keys. Delete your key by clicking on **Keys > Manage Keys** and then check your key and press **Delete Checked Key(s)**. We'll now Import the key we exported earlier back into our key list. Click on the **Import Key From** button then **File**.

Close the Key Management window and once again try encrypting our original file. You may need to rename the output file as we'll have duplicates. You should be able to successfully encrypt the file again.

However, now, you'll find that you cannot decrypt the file. This is because when we deleted the key from our system, we deleted the **private** key which is used for *decryption*.

## Learning Module 2 - Hashing

Hashing is a process of generating a string of characters which represent the integrity of the file that has been hashed. It is a **one way** process, this is because the string (called a Hash or Checksum) is generated from the file, rather than the file being converted into the hash. It cannot be used to secure communication.

Ideally all hashing algorithms have four main characteristics. They are as follows:

- It is easy to generate a hash for any given file or message.

- It is infeasible to generate a file or message from a hash.

- It is infeasible to modify the file or message without modifying the hash.

- It is infeasible for two distinct files or messages to generate the same hash.

One purpose of a hash is to ensure that a file has not been intercepted and modified in transit to the receiver. It would normally be sent separately to the file in question, or notably, on a website when you download a file you may see a checksum to compare to your downloaded file. This is a method for you to be able to confirm that the installer/file is as the developer/creator intended it to be.

It is also a way for you to be able to confirm that your assignment is valid, and why we encourage you to include a MD5 hash in your Blackboard comments for your assignments!

Another method is for password verification. Rather than sending the password in plaintext across the internet, or storing a password in plaintext in a database, you could hash the password upon creation and simply store that value. This works well as it is highly infeasible that someone could guess the password (or a different value) that would result in that hash. It also cannot be reverse engineered to discover the password itself.

You can also use the hash of the file to generate a key with which you could encrypt the same file! Combining the security methods together to create a stronger key and protection of information.

There are many common hashing algorithms, most notably are MD5 and SHA (there are multiple).

Let's play with some of these algorithms now!

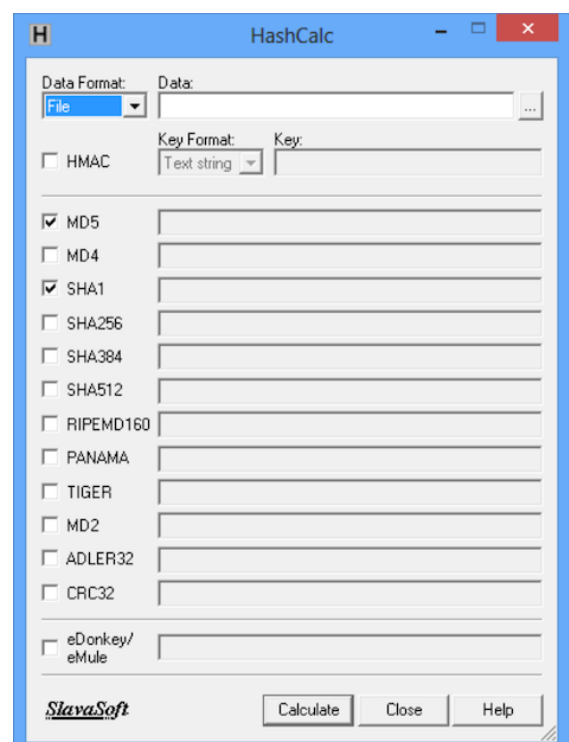
### Activity Module 2 - Hashing

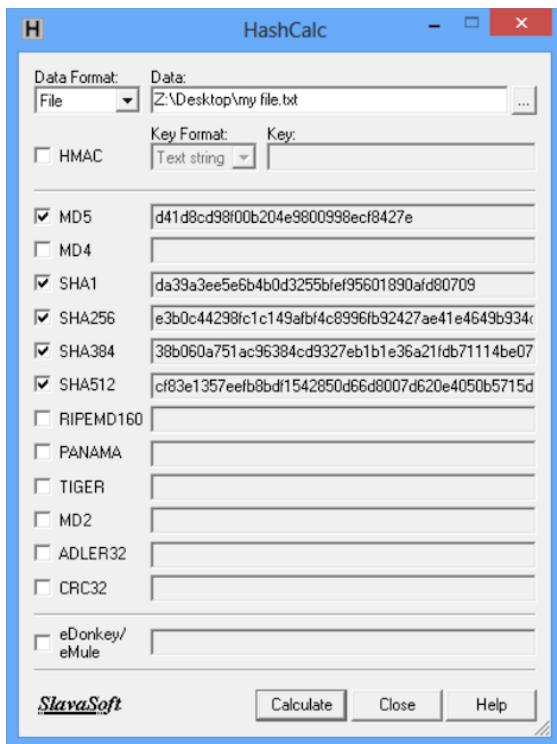
For this activity we will be making use of HashCalc. Ensure you have downloaded it from the Unit Schedule in this weeks Module.

HashCalc is a free application which can generate the hash checksums for multiple algorithms either singularly or all at once.

For now, let's use our empty text file we had created before, once again my file will be called 'my file.txt'. When you open HashCalc you should be greeted by the window to the right.

For the time being lets select MD5, SHA1, SHA256, SHA384 and SHA512. We're going to inspect how long the checksum string is for each of them! Click on the ... towards the top right and then choose your empty text document. Then click **Calculate**.





Once we click it we instantly have our checksums! But if you look at the length of the strings they get progressively longer with each of the algorithms. This is because the cipher used for each is making use of large bit keys.

If we use a larger file we would start to see the longer algorithms (SHA512, SHA384, SHA256) take progressively longer than the previous one.

Take note of the MD5 and SHA512 hashes by copying them both into a new text file (on separate lines). Now, to demonstrate how much they change with minimal changes made, we'll open the file and type in just one character then save the file.

Now, recalculate the checksums. Copy the MD5 and SHA512 underneath their previous values in your notepad file and notice how vastly they have changed. This is why it is unfeasible to reverse engineer the original file or to have two distinct files end up with the same value.

See my comparisons below:

#### MD5

Original: d41d8cd98f00b204e9800998ecf8427e

Modified: c4ca4238a0b923820dcc509a6f75849b

#### SHA512

Original: cf83e1357eeeb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36...

Modified: 4dff4ea340f0a823f15d3f4f01ab62eae0e5da579ccb851f8db9dfe84c5...

(Shortened so they fit)

We'll apply these learnings to our next Activity Module after the last Learning Module below!



### Learning Module 3 - Steganography

Steganography is a method of hiding one file or message inside another file. It is the ability to be able to secretly transfer information that could otherwise be seen to be completely normal.

The modifications are generally undetectable to the human eye or ear on the computer and as such can only be identified through the use of other tools to detect that a file may have been hidden inside the original. Although, simply identifying that a file is hidden inside another doesn't mean you can find it.

The method used to hide a file is to modify the **Least Significant Bit** (LSB) of a bit-sequence sample. Modifying this LSB makes a change to the colour of a pixel or the sound of a tone in a song, but it is so minute that without having specialised tools to analyse the colour spectrum for a picture or a wave-form analyser for a wav file we would not be able to detect it.

If we change the LSB to fit our information only 50% of the time is it likely to have an effect on the original file, have a look at the table to the right.

For example,

*'Every X samples, the LSB of the sample is set to a 0 or 1 value according to the data being inserted into the file. Thus to insert a byte of information into a file, we adjust the LSB of only 8 samples out of the Y-byte block of samples'*  
(Mansfield Jr. et. al., p. 791)

Original Bit	Adjusted Bit	Effect
0	0	None
0	1	Change
1	0	Change
1	1	None
Mansfield Jr. et. al., p. 791		

Essentially, we choose the sample size of the bytes we that we'll be modifying the LSB, for example, 16. This means, every 16 bytes we modify the LSB. This means that in each sample of 16 we are modifying 1 out of 128 bits of information. A very tiny change.

This sample size will dictate how large the sample block will be that we are hiding it in. So, to hide 1 byte of information in a file with a sample size of 16, we would need 8 x 16-bytes = 128-byte block.

The more you spread out the modified LSB of your information, the less chance of detection it has, but it also means that the file in question must be larger as well.

To show you how hard it is to detect a message hidden inside a file, look at the two pictures below... Which one is modified? Left or right?



Can't tell? I have made them available in full resolution in the Unit Schedule for this weeks module. Inside the picture I have hidden the full file of the encrypted 'my file.txt.asc' from earlier! I actually hid the information in the left photo. See the next page for the full copy-paste of the information hidden!



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

Version: GnuPG v1

hQEMA0lexQVtsCSKAQf9ERX7AK25qngrGlu/4ws5YfkIpLLnTRPtv4HE6VWDOG9S  
CMgad3y8vm8QnKsXmLOXSrvRcBo00IQ14BV8r6rZMPL9QGg+4ZGFaE6fOfgLmTJQ  
lQVibJYrwYlADr41wx2P7aWklUmdwbjiOvDfh3fbxOFpQzJP9nh1tMb8KTM2Eotj  
sRe/SEQSZMpSH7/SK+YFLOpkrdhwAzFhaWmNeKcmDbZDW85B2u5F0TDKnKBrS2+V  
UoeDc7NXHcICBroPfmTDLhSpGSF86rSbavT4FTicuc6ZHe9sZa9SSWnt3iUB1+Hb  
GKza7NOJeWiZ5n8ZYOEZ+9oeD7b6lI0q5h4Isuk0Z9I7AfKy+YDcDiP37wzw1ohd  
kE9X7GL5gwAbUrQOpUd7KHktYTzC9MbswXqA85G4hwV5SGms/v+rCiZluGA=  
=Aa04

-----END PGP MESSAGE-----

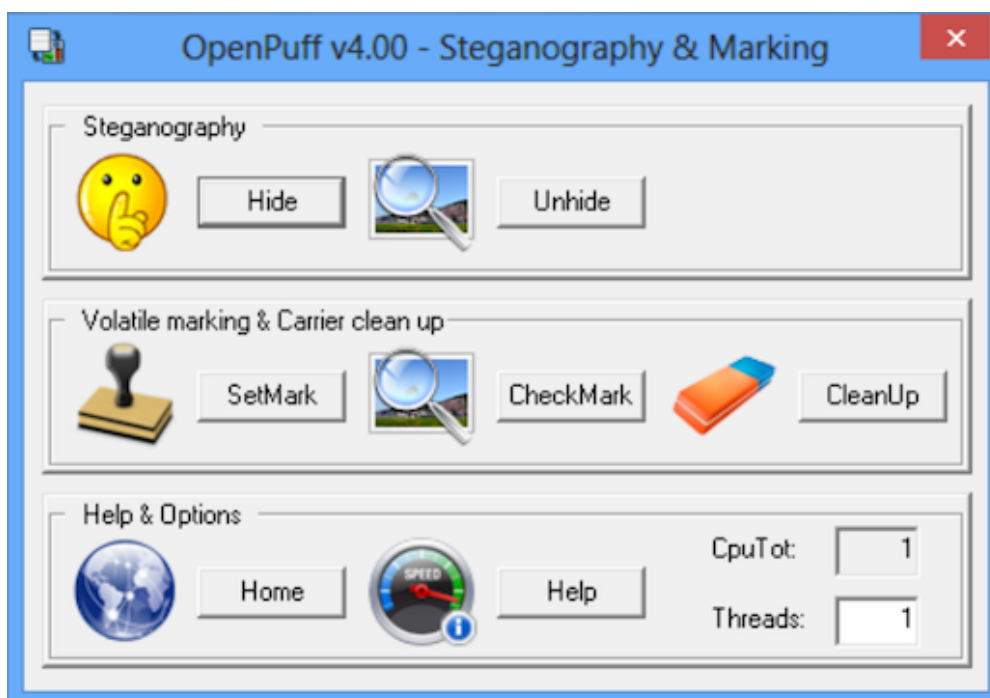
Now, onward to make use of encryption, steganography and hashing!

### Activity Module 3 - Steganography

To do this part, ensure you have downloaded the application OpenPuff from the Unit Schedule for this weeks Module.

OpenPuff is another free application. This one will let you hide, or reveal information from within another file. In order to be able to reveal the information however, you need to know the method that it was hidden in, otherwise it is lost forever.

Let's run OpenPuff. You should be greeted by the window below.



Firstly, we'll click on the button saying **Hide**. This will then present us with a new window which can be confusing at first however if you look at the section titles they are numbered 1, 2, 3 and 4 - making it incredibly easy to step through it.

Our first step is to create 3 passwords with which it will create a Hamming Distance. Hamming Distance is the number of positions at which corresponding symbols are then different, or it measures the minimum number of substitutions required to change a string. It can also measure the number of errors that can transform a string to another.

You want to choose 3 passwords that are **as distinct from each other as possible**. This makes this process much more secure than if you simply change a single characters value. My passwords are listed on the next page.

### Cryptography Passwords:

(A) QameshAghol3832

(B) NASU37Tochigi1

### Scrambling Password:

(C) 4997Caslavice

(1) Insert 3 uncorrelated data passwords (Min: 8, Max: 32)

Cryptography (A) [scrambled] (B) [scrambled]

Scrambling (C) [scrambled] Enable (B) ☒ (C) ☒

Passwords check **H(A, B) H(A, C) H(B, C) = { 35%, 36%, 35% }**

H(X, Y) = Hamming distance {X}{Y} >= 25%

When you have entered passwords that are uncorrelated enough the password check will show up green, showing the percentage of difference between the passwords when using it's algorithm. You want these numbers high, and 25% is the minimum you should aim for.

Now, we want to choose the file to be hidden. I will once again choose to hide the encrypted 'my file.txt.asc' inside our other file. To do this go to the right of the window and click on **Browse** then navigate to your desired file and select it. You will see that it shows a quick analysis of the size of the file. This includes a static 10, the size of the name and then the size of the data inside the file.

(2) Data (Max: 256Mb)

Target \\vmware-host\Shared Folders\Desktop Browse

Size 10 + name[15] + data[542] bytes

Now, let's choose the file. I will once again hide the file in the picture of the puffin. I will be using the picture called 'puffin.jpg' (found in the Unit Schedule for this Module). This is done in the Carrier Selection pane (number 3). The carrier is the file (or files) that contain the hidden information. Click on **Add** and navigate to your desired carrier file and select it, ensure that it is larger than the Data to be hidden or at least ensure that you have multiple files to select.

(3) Carrier selection [Order sensitive]

(Name) Sort by name / (Bytes) Sort by bytes Shuffle Clear

Name	Bytes	Chain Order
puffin.jpg	784	# 00000

(-) Move up selected / (+) Move down selected / (Del) Delete selected

Add Selected / Total **784 / 567 bytes**

When you have selected a file (or files) that is large enough to contain your data, the byte allocation at the bottom will change it's colour to green.

Lastly, we can choose the Bit Selection options in section 4. This lets us spread the bits out more, or compress them together (changing the sample size). Having a larger sample size (1/8, 1/4 etc.) means we need a much larger file. So for this one I will be selecting **jpeg > 50%**. This only just covers the file size of the carrier. Once you have chosen this, click on **Hide Data!** and choose your output location (different to the folder of the carrier file). Now we wait!

Compare your original carrier file to the now modified version. Can you pick the difference?

Try generating a hash checksum for both files and see if you can tell the difference then. You should be able to very easily. This is where hashes can come in very handy if someone is sending you an important document!

(4) Bit selection options

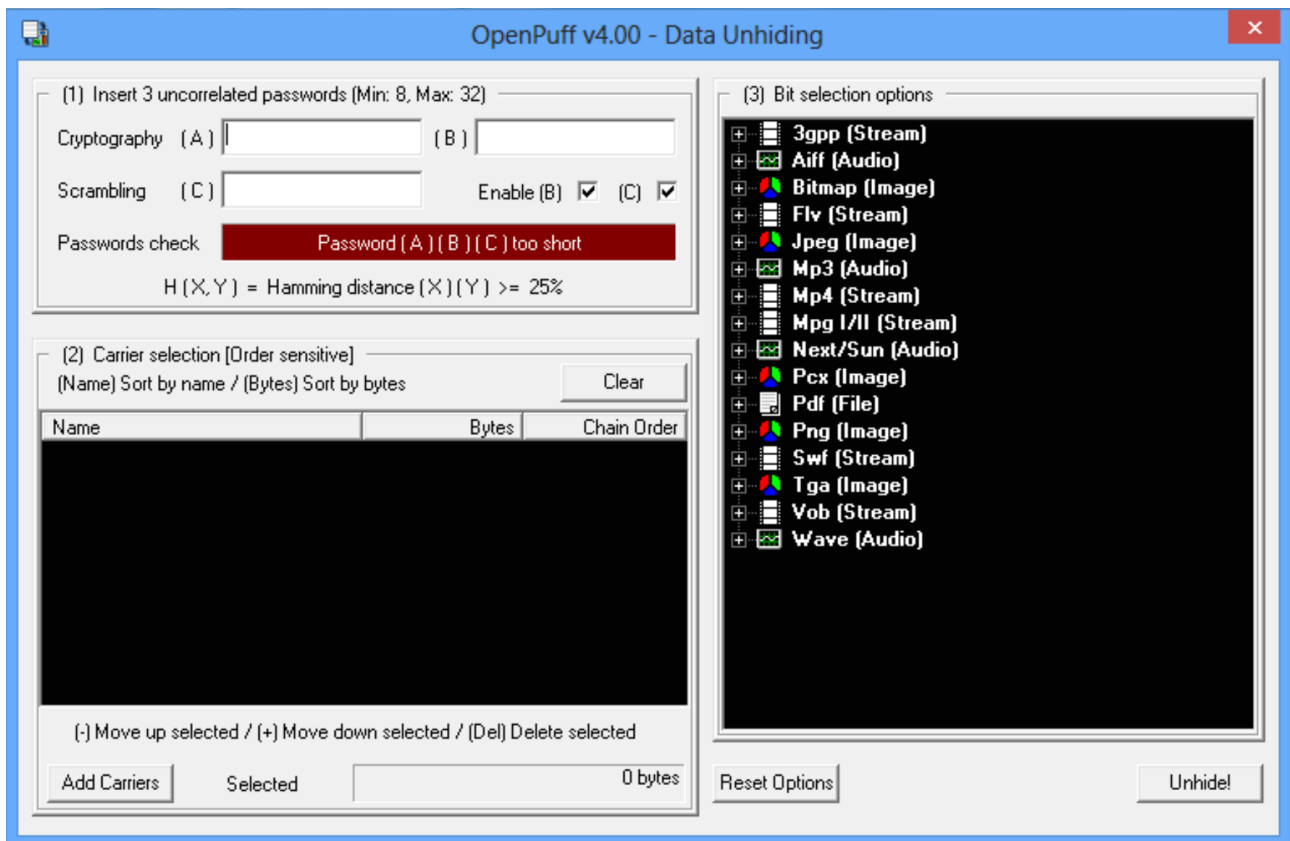
- 3gpp (Stream)
- Aiff (Audio)
- Bitmap (Image)
- Flv (Stream)
- Jpeg (Image)
  - ☒ 1/8 (12%) - Minimum
  - ☐ 1/7 (14%) - Very Low
  - ☐ 1/6 (17%) - Low
  - ☐ 1/5 (20%) - Medium
  - ☐ 1/4 (25%) - High
  - ☐ 1/3 (33%) - Very High
  - ☒ 1/2 (50%) - Maximum
- Mp3 (Audio)
- Mp4 (Stream)
- Mpg 1/II (Stream)
- Next/Sun (Audio)

Reset Options Add Decoy! Hide Data!

Other ways you can secure this would be to try making your passwords, then hashing them and using the first 32 characters of each of the hashed passwords for the Hamming Distance calculation.

You could even encrypt the carrier file again! So the carrier is encrypted and then the hidden data is also encrypted! Now, we'll extract that information back out again, see how on the next page.

Firstly, close our hiding data window and it should revert back to the initial window we saw when we launched OpenPuff. Now click on **Unhide** and we'll be greeted by another different (but very similar) window, like below.



Now, simply put in all the same information as before, however choose our now hidden data file for section 2, you should then see the hidden information come back out exactly as it was before! (Choose a different target directory).

You can even use a hash to check it's integrity against the original hidden data.