**Hands-On Encryption**

***1. Encryption Math***

In this section, you will demonstrate your understanding of the complexities of various hashing algorithms and the capabilities of current hardware.

***Comparing Two Hacking Machines – Then and Now.***

**Part 1**: **Speed of the Bombe**

Calculate the performance statistics of the Bombe machine used to crack Enigma messages.

| Number of combinations that the Bombe needed to test in order to break the Enigma algorithm | Hacks per second\* | Average time to perform a complete decryption of the daily Nazi codes |
| --- | --- | --- |
| 67 bits of combinations | 15 | under 20 minutes |

**Part 2**: **Speed of the Modern Cracking Machine**

Compare the relative complexity of Enigma with modern encryption methods.

Assume that you plan to build a budget password hacking rig, as described in the “How to Build a Password Cracking Rig” article, located in the Topic Materials.

The statistics used in this report are measured in the following scale:

* GH/sec (Giga Hacks Per Second or Billions of Hacks Per Second)
* MH/sec (Mega Hacks Per Second or Millions of Hacks Per Second)
* KH/sec (Thousands of Hacks Per Second)

Complete the following table to compare the amount of time it will take to find a matching hash for a given 8-letter lowercase password. Some hashing algorithms are more computationally intensive than others, so each algorithm takes a different amount of computing work to complete.

See the section on the webpage called COMPLETE HASHCAT BENCHMARKS for statistics.

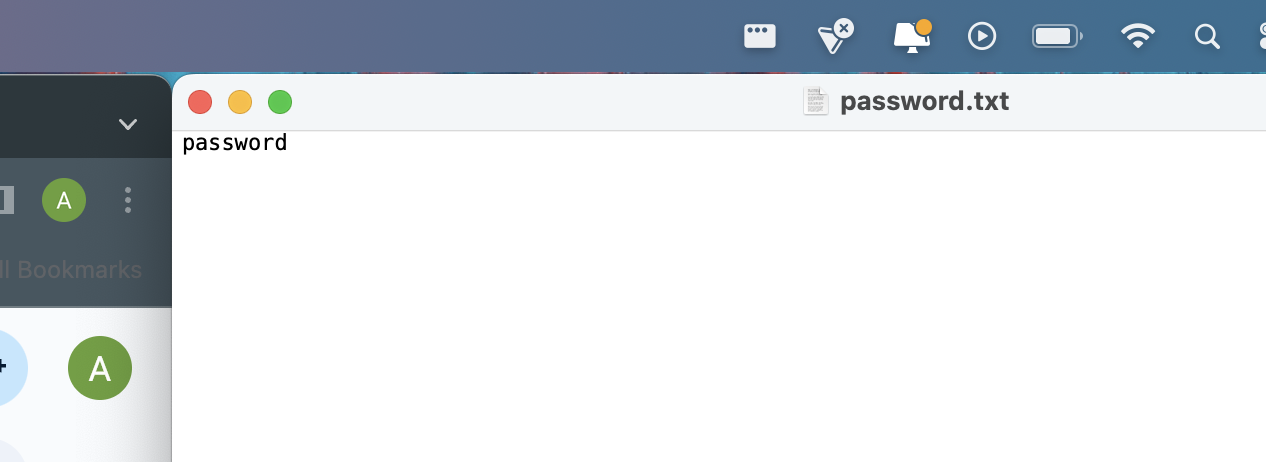
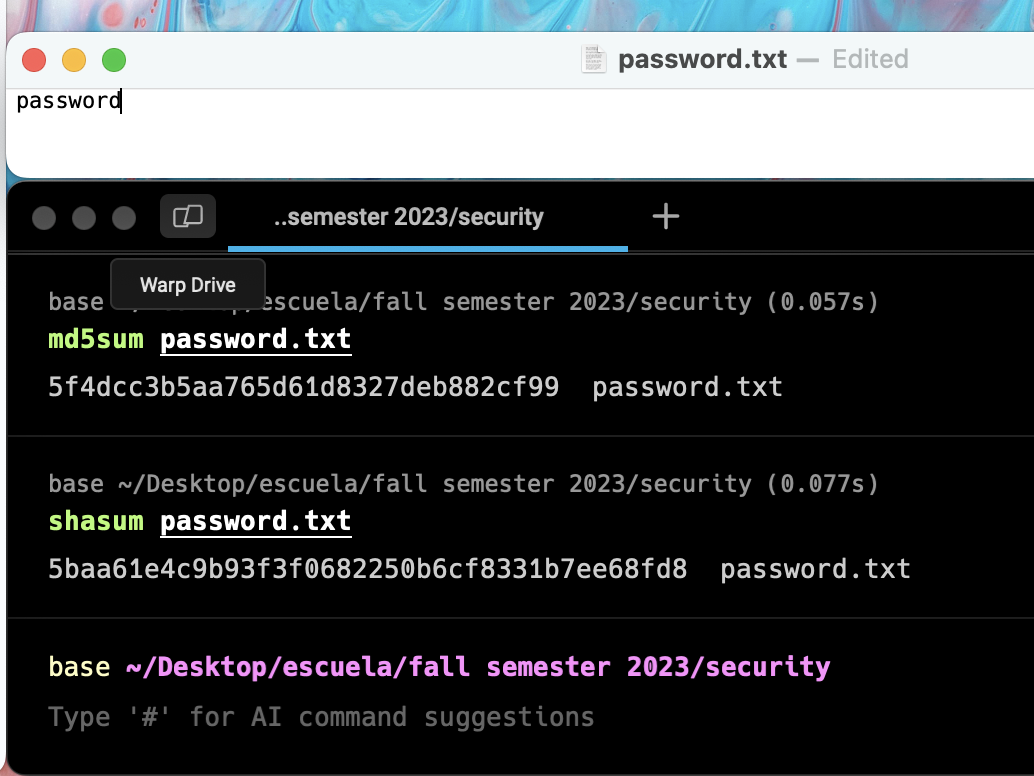
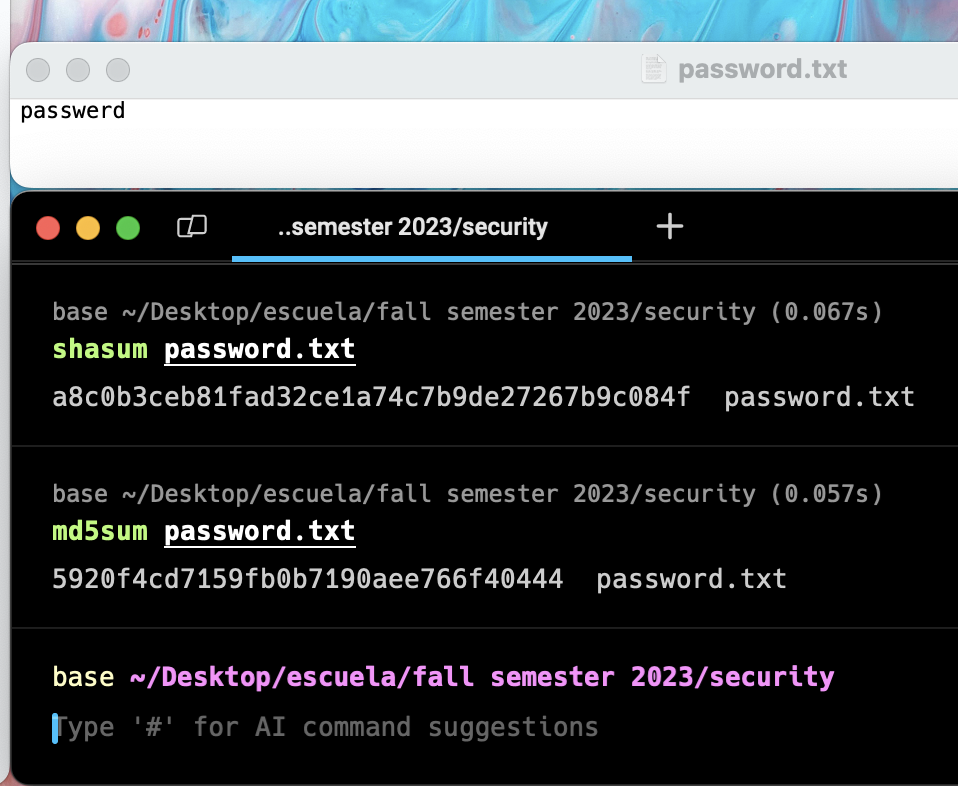
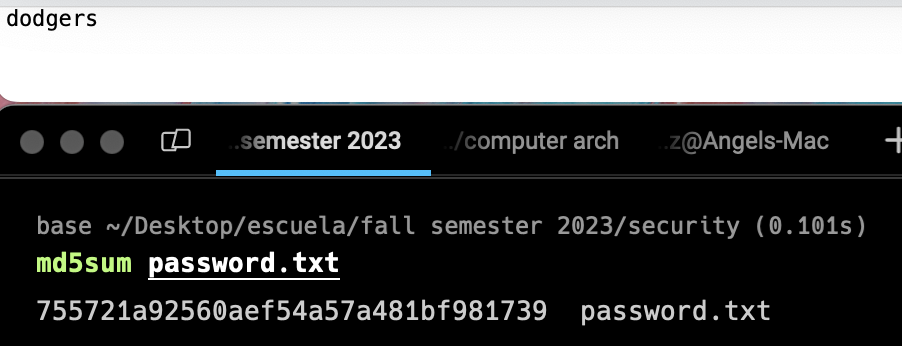
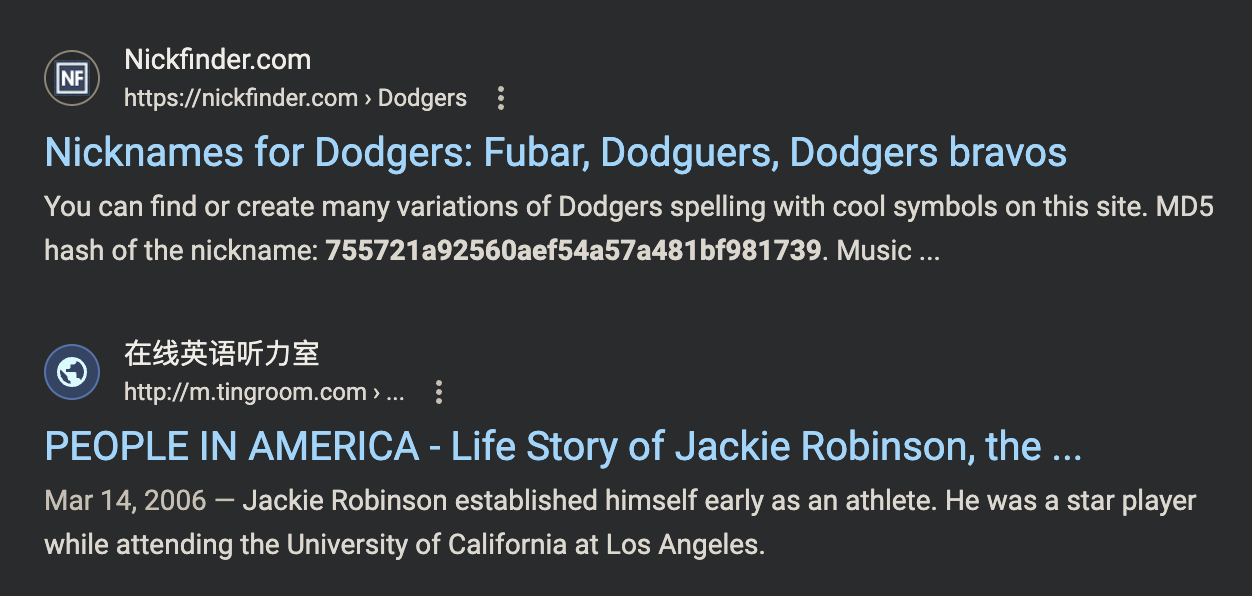
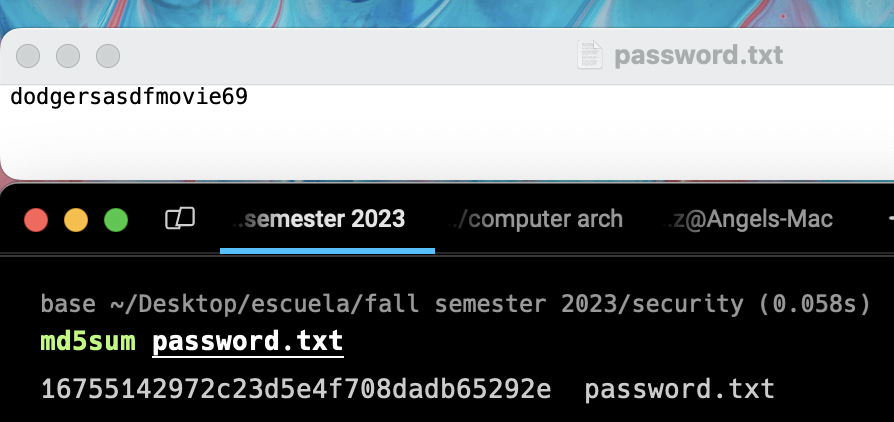
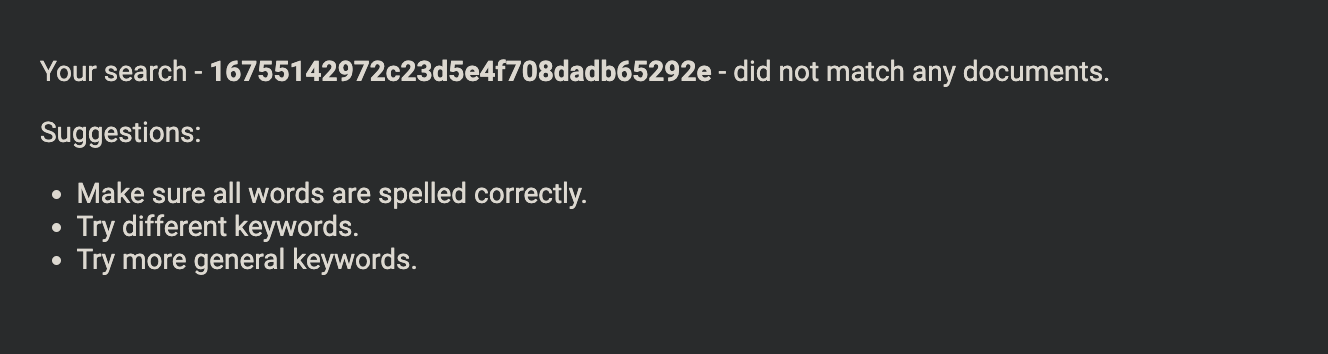
| Algorithm | What is the number of possible passwords using 8-letters, lowercase. | How many hacks per second\* this computer can perform on this algorithm. | What is the estimated time (in an understandable time scale – minutes, days, years etc) to perform a full brute force attack on an 8 letter lowercase password? |
| --- | --- | --- | --- |
| Md5 hashing | 26^8=  208,827,064,576 | 76526.9 MH/s  76,526,900,000 | 2.7 seconds |
| Sha1 | 208,827,064,576 | 25963.3 MH/s  25,963,300,000 | 8.04 seconds |
| Sha256 | 208,827,064,576 | 9392.1 MH/s  9,392,100,000 | 22.23 seconds |
| Bcrypt / blowfish | 208,827,064,576 | 43,551 H/s  43,551 | 4,795,000.45 seconds  55.5 days |

***2. Using Hashing Tools***

**Hashing tools**

1. Install an MD5 utility on your computer. You can use a version of Linux with a virtual OS solution such as VMWare or Virtual Box. Mac OS has MD5 as part of the OS. You can install the Microsoft File Checksum Integrity Verifier which is available online.
2. Research how to use hashing algorithms in Linux by viewing “Linux Tutorial for Beginners - 9 - Verify Files Using Checksum,” located in the Topic Materials. This is a good place to start for the Shasum hashing algorithm.
3. Create a Microsoft Word document titled, “Hashing passwords with shasum and md5.” Include the following content.

**Document Content**

1. Explain what hashing does and how it is different from encryption.
   1. Encryption is a two-way function in which information is scrambled in a way that it can be decrypted. Hashing is one-way only where any length of text produces a fixed-length value.
2. Demonstrate the use of shasum or md5 or both.
   1. Create a text file that contains a password.
      1. text file with password
   2. Hash the file and record the results.
      1. hash with md5 and shasum
   3. Create screenshots and caption each picture with what is being demonstrated.
3. Change one character of a password and compare hashes. The hash results should be different even in the case of a minor change.
4. Demonstrate the weakness of MD5.
   1. Create some md5 hashed passwords for common words “steelers,” “Cardinals,” “superman,” etc. 
   2. Try to find the hash string using a simple Google search. 
   3. Hash some slightly more complex passwords using MD5. Search Google for the hashes. Determine how complex a password needs to be in order to avoid being hacked by a rainbow table result.
   4. Create screenshots and caption each picture with what is being demonstrated.
      1. In this screenshot, I simply added some salt to my dodgers password and could not find any google results. Adding salt to a simple password can increase the security of any password

***3. Gnu PGP Tutorial***

**Software Requirements:**

VMWare

Ubuntu or Kali Linux virtual machine with GPG installed. If you have an alternative GPG app on your Windows or Mac you can use that instead.

Continue with the same Word document. At each point in the tutorial, capture an image of the results. Explain what each screen capture is demonstrating.

**Part 1 - Synchronous Encryption**

Synchronous Encryption, or “simple” encryption, uses the same key to lock and unlock a file.

First, you should familiarize yourself with the possible commands available to the utility. Open the linux terminal and type this command to get help.

gpg –h

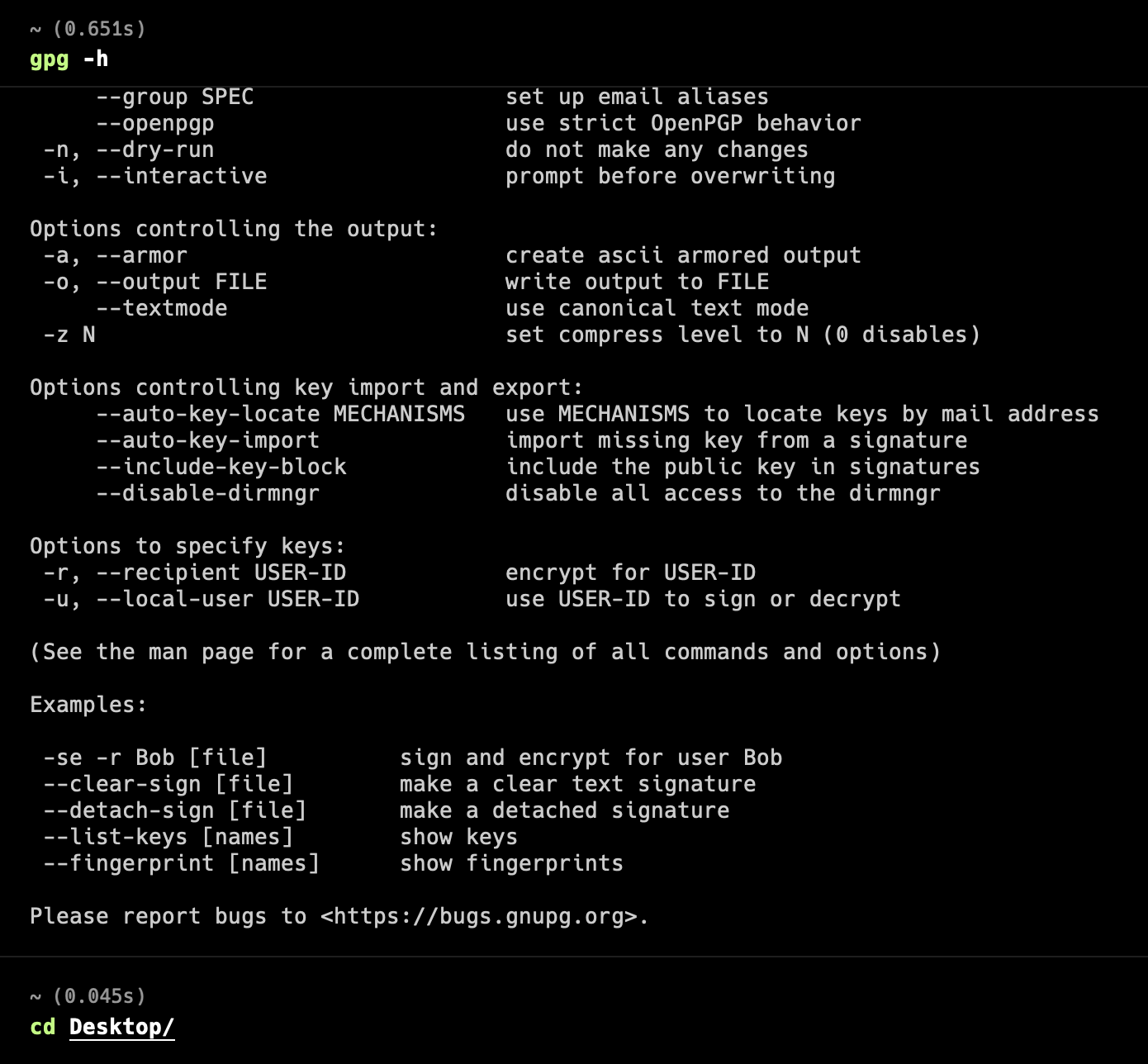
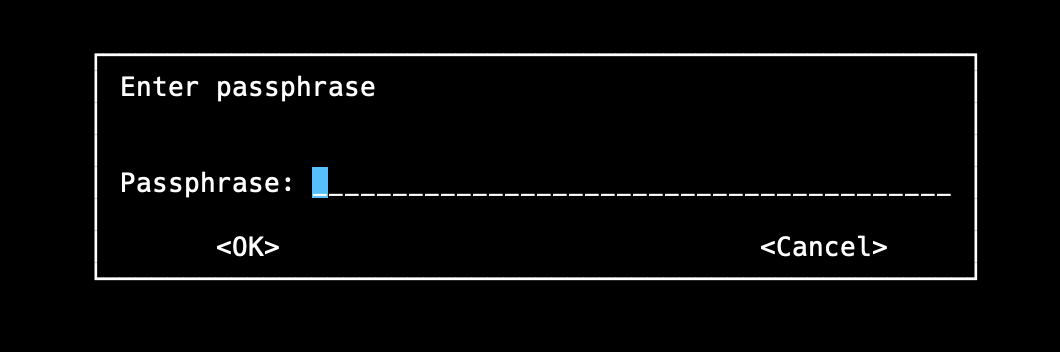
We will use some of the commands that are listed in the help instructions.

Here is the plan for what we will do in the following steps: (1) create a doc (2) create an encrypted version (3) decrypt the file to another (hopefully identical) text file.



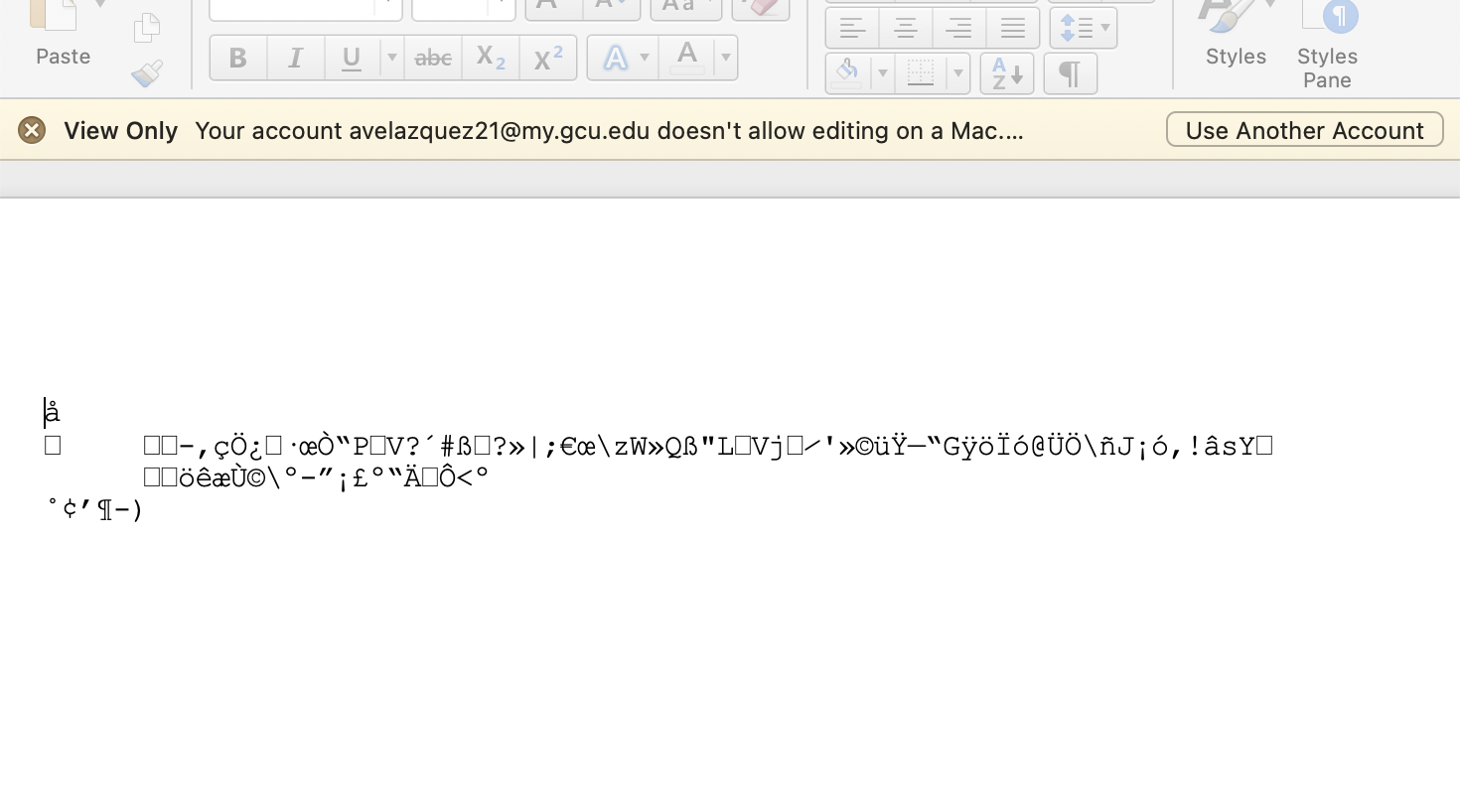


1. Encrypt a file using default settings.

1. Open Linux.
2. Create a text document called message.txt and save it.
3. Run gpg to encrypt it.

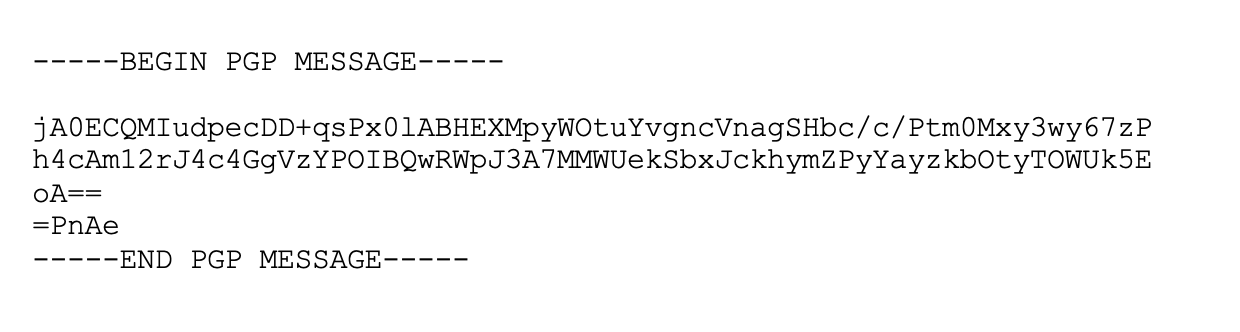
**gpg -c message.txt**

The –c option is for symmetric encryption instead of the public key option.

This will create an encrypted file named message.txt.gpg A text editor probably cannot display the contents properly. Try opening the encrypted file and you should see a binary file with gibberish characters.

2. Encrypt a file that can be copied into an email message. The armor option creates an output file that contains only ASCII standard characters instead of the default (and unreadable by a text editor) binary format.

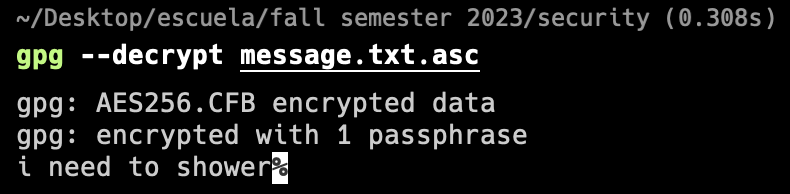
**gpg -c --armor message.txt**

This will create an encrypted file named message.txt.asc. The asc stands for ASCII. You should be able to open it with a text editor.

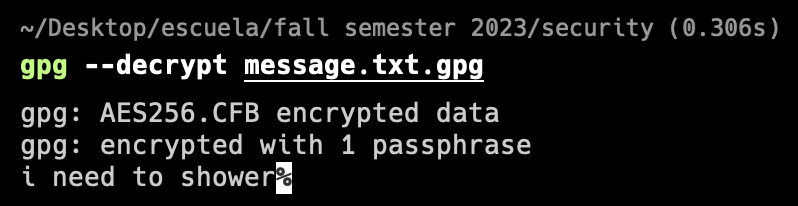
3. Decrypt the two files.

Rename the message.txt file to message-original.txt

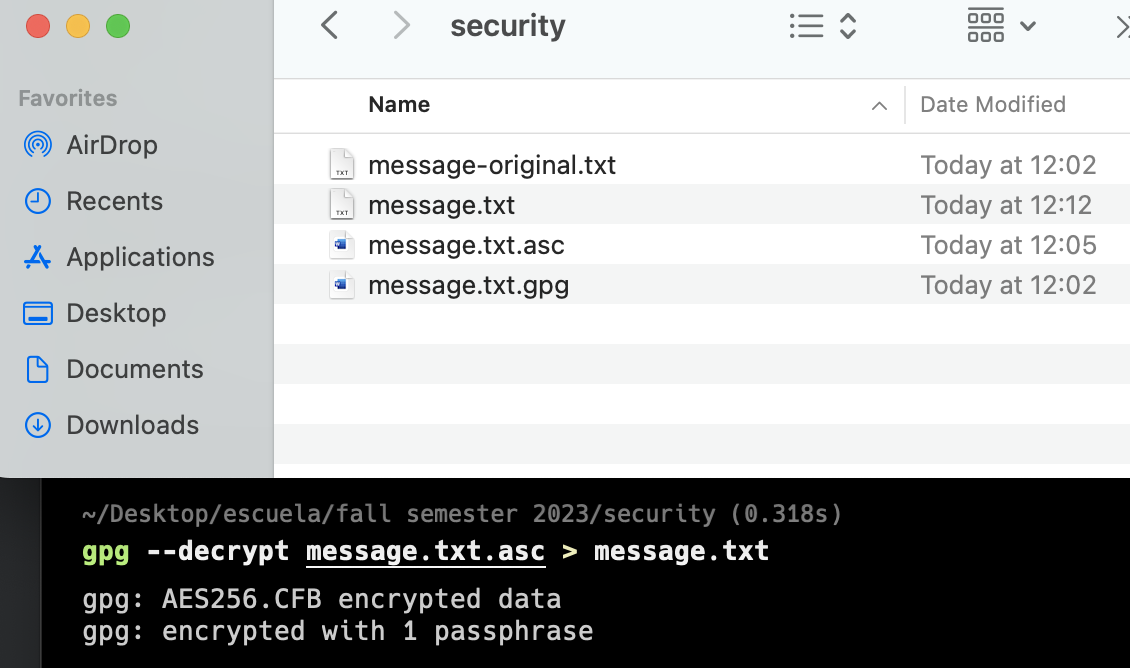
4. Decrypt the encoded text file. It should create message.txt again.

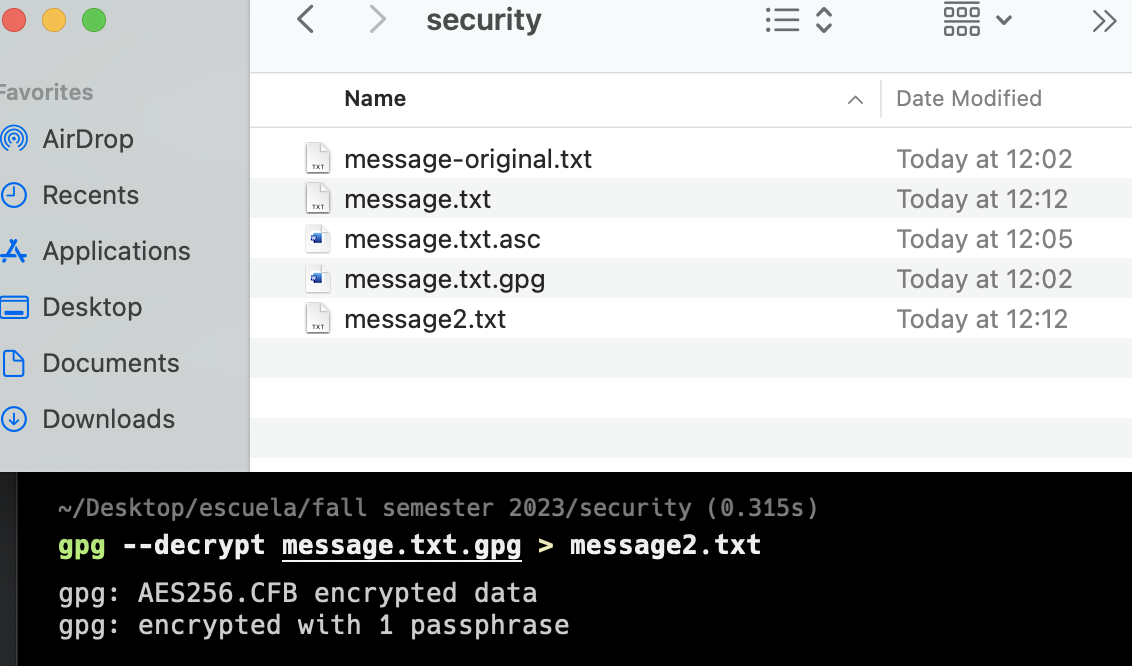
gpg --decrypt message.txt.asc 

gpg --decrypt message.txt.gpg



Each of these commands will only display the message on the screen. To redirect the output back to a file, type in the commands as follows:

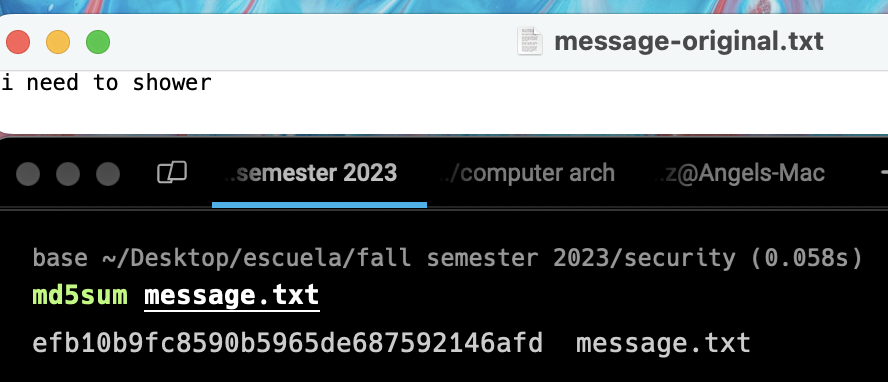
gpg --decrypt message.txt.asc > message.txt

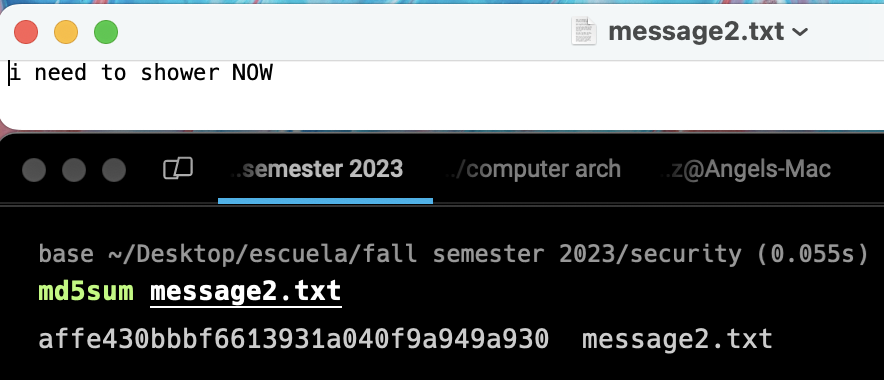
gpg --decrypt message.txt.gpg > message2.txt

You should be able to open the decrypted message. Hopefully it has not changed, but can we prove that it has not changed?

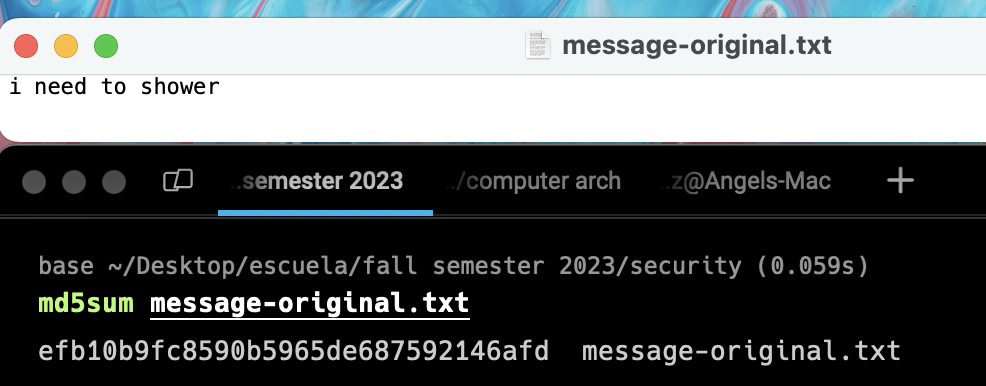
**Verify that the file has not changed with a Hash.**

We will run a hashing function to get a digital signature on the file before and after the encryption process.

md5sum message.txt

md5sum message2.txt

This should generate a hash value unique to this text file.

md5sum message-original.txt

The hashes should match.

1. Check the original file with a hash function.

2. Check the file that was decrypted with the same hash function. The two hashes should match to prove that the contents are the same.

You can use the “>” operator to dump console messages to a text file.

**Part 2 - Asynchronous Encryption - Public Key**

1. Create your own private and public key files

gpg --gen-key

Follow the steps through the process

1 - RSA

2048 - key length

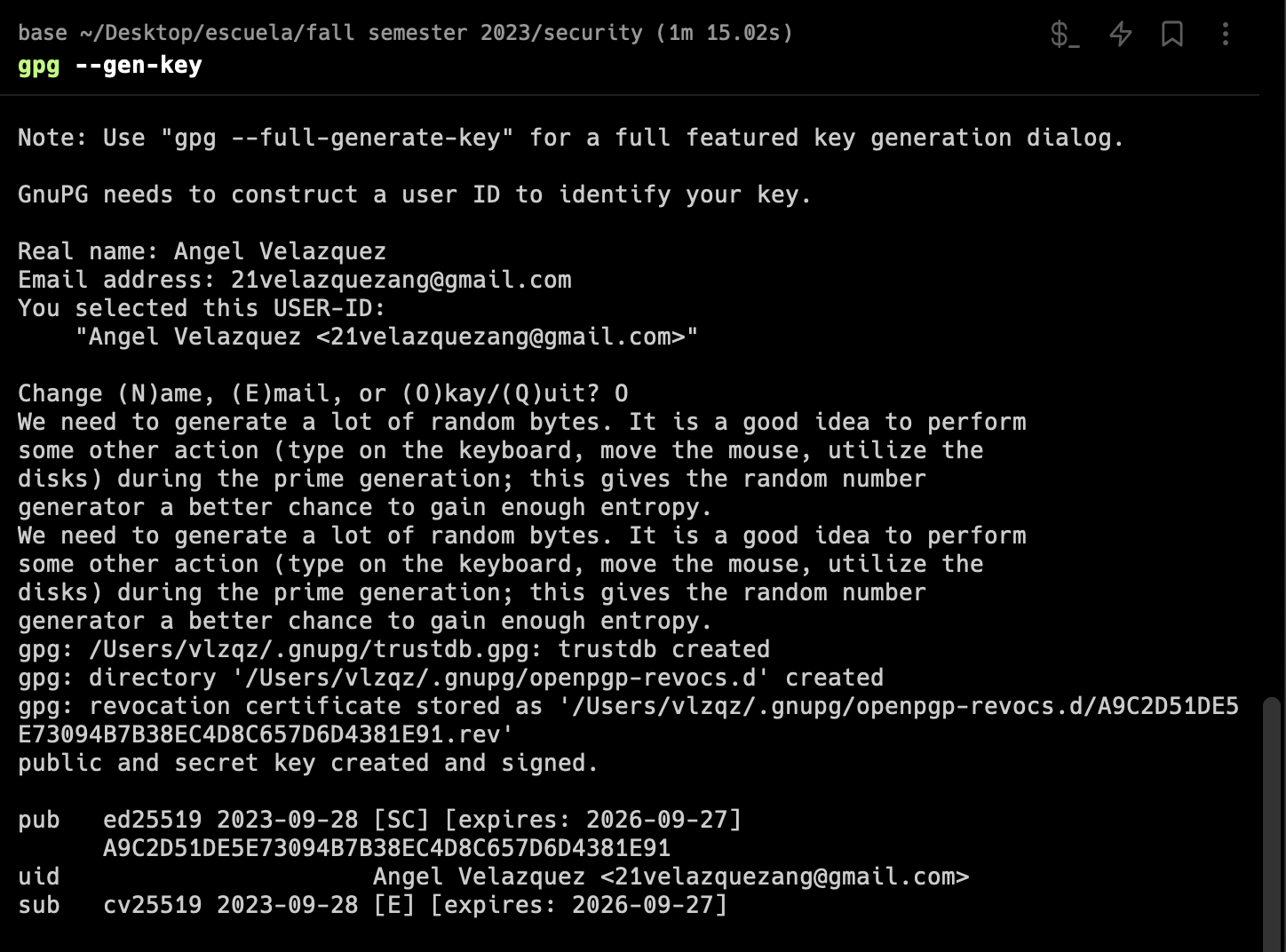
5y - expire date

shad - real name

shad.sluiter@gcu.edu - email

Always use encryption if you are paranoid - comment

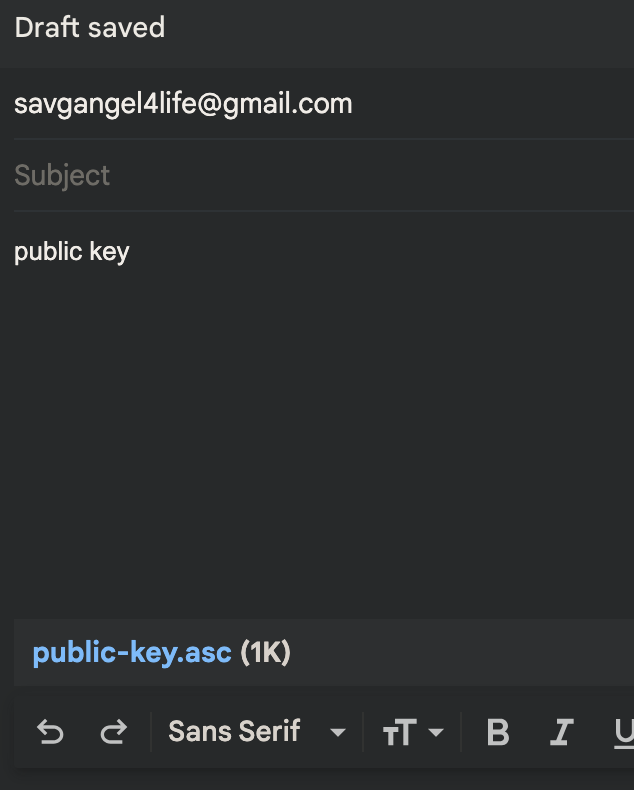
o - OK

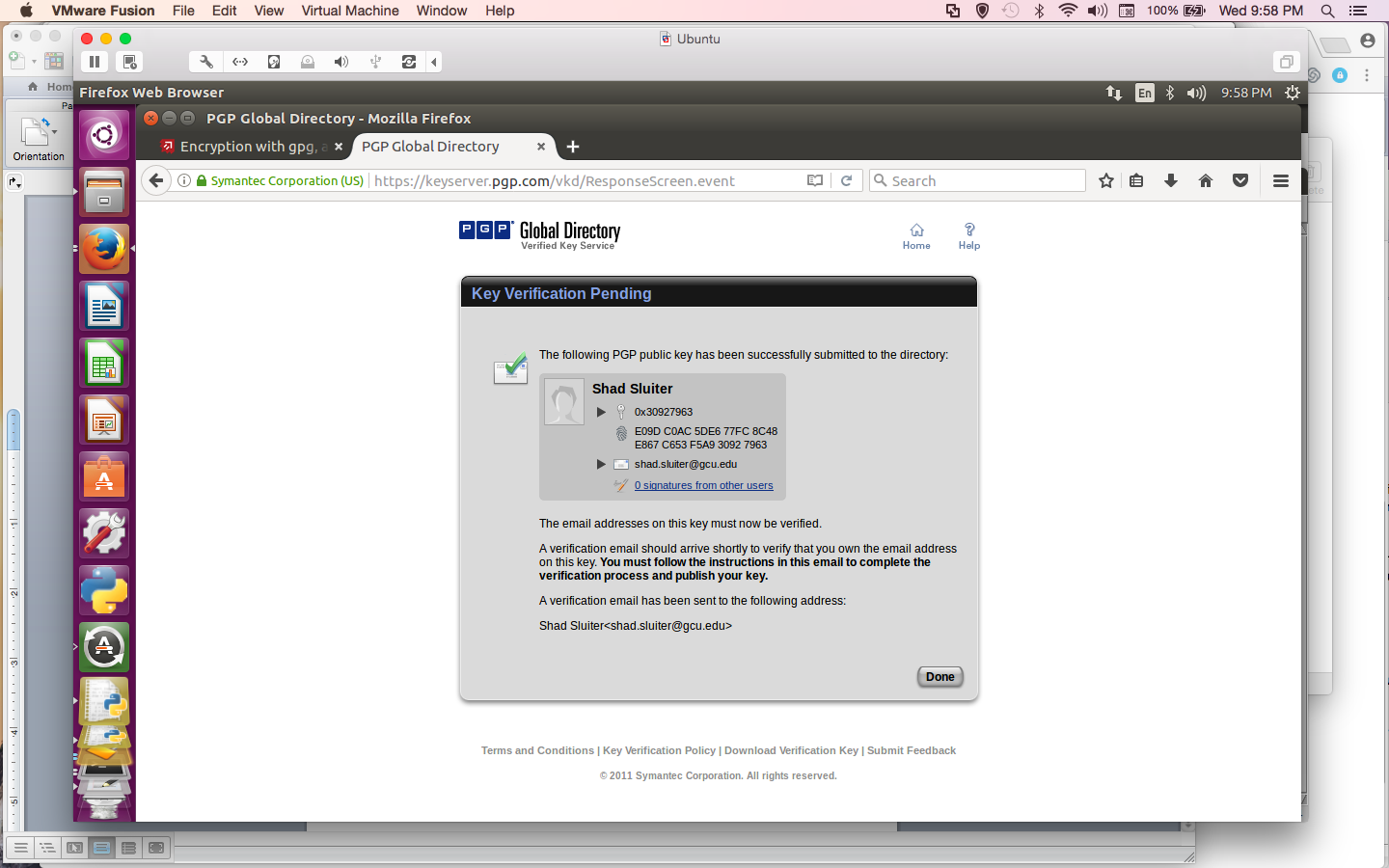
password - passphrase for your private key

1. Share your key with another user. Use the --armor option to make it an ascii file. Dump it to a file.

**gpg --export --armor youremail@email.com > shadsluiter-public-key.asc**

Share your public key in one or more of these ways.

1. eMail. Copy the contents of the public-key.asc file into an email. Send it to your friend.
2. Publish it to your personal web page or social media app.
3. Publish it on a key server where other people can copy it. <http://keyserver.ubuntu.com> <https://pgp.mit.edu> and <https://keyserver.pgp.com>

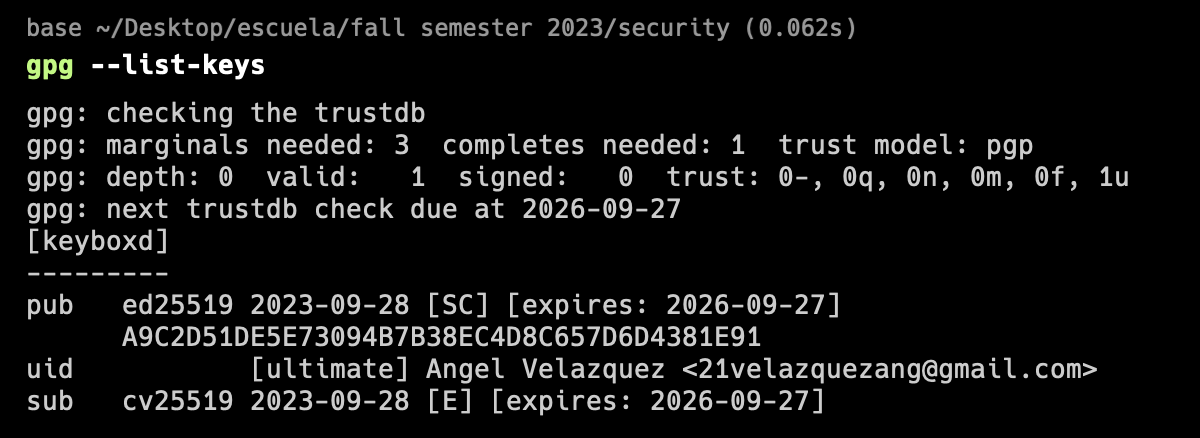


The fingerprint is a hash of your public key. You can use it to verify that a public key has not been tampered with.

1. Import a public key made by another person. This example shows how to import the file called shad-gcu-pub-key.asc

**gpg -- import shad-gcu-pub-key.asc**

1. How to view the public keys you have imported into your computer.

**gpg -- list-keys **

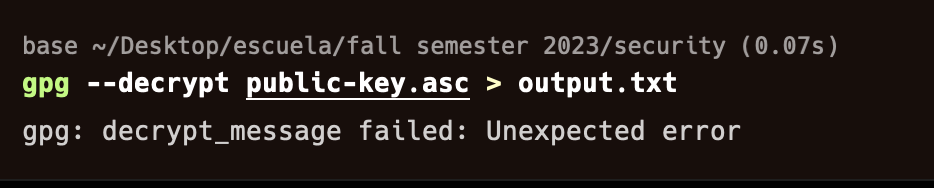
Your list of public keys will grow as you work with more individuals and share public keys.

1. Encrypt a message in preparation for sharing it with another user. In this example, I am going to share a file (filename.txt) with my friend Bill Gates.

**gpg –-encrypt –-recipient --armor** [**bill.gates@microsoft.com**](mailto:bill.gates@microsoft.com) **filename.txt**

This will use the public key to encrypt the text file. There should be a new, encrypted file in the same directory as filename.txt. Type ls to confirm it is there.

1. Send the encoded message to your friend who will then decrypt it. You can copy the encrypted text into the body of an email message and safely send it via email, one of the most non-private forms of Internet communication.
2. Decrypt a message sent to you.
3. You must first share your public key with a friend.
4. He/she uses your public key to encrypt a message.
5. He/she sends you an email.
6. Save the encrypted message to a file.
7. Run the decrypt command to create an unencrypted text file.

**gpg –-decrypt message-from-friend.txt.asc > output.txt**

1. You must provide the password associated with your private key.