Final Assignment

by

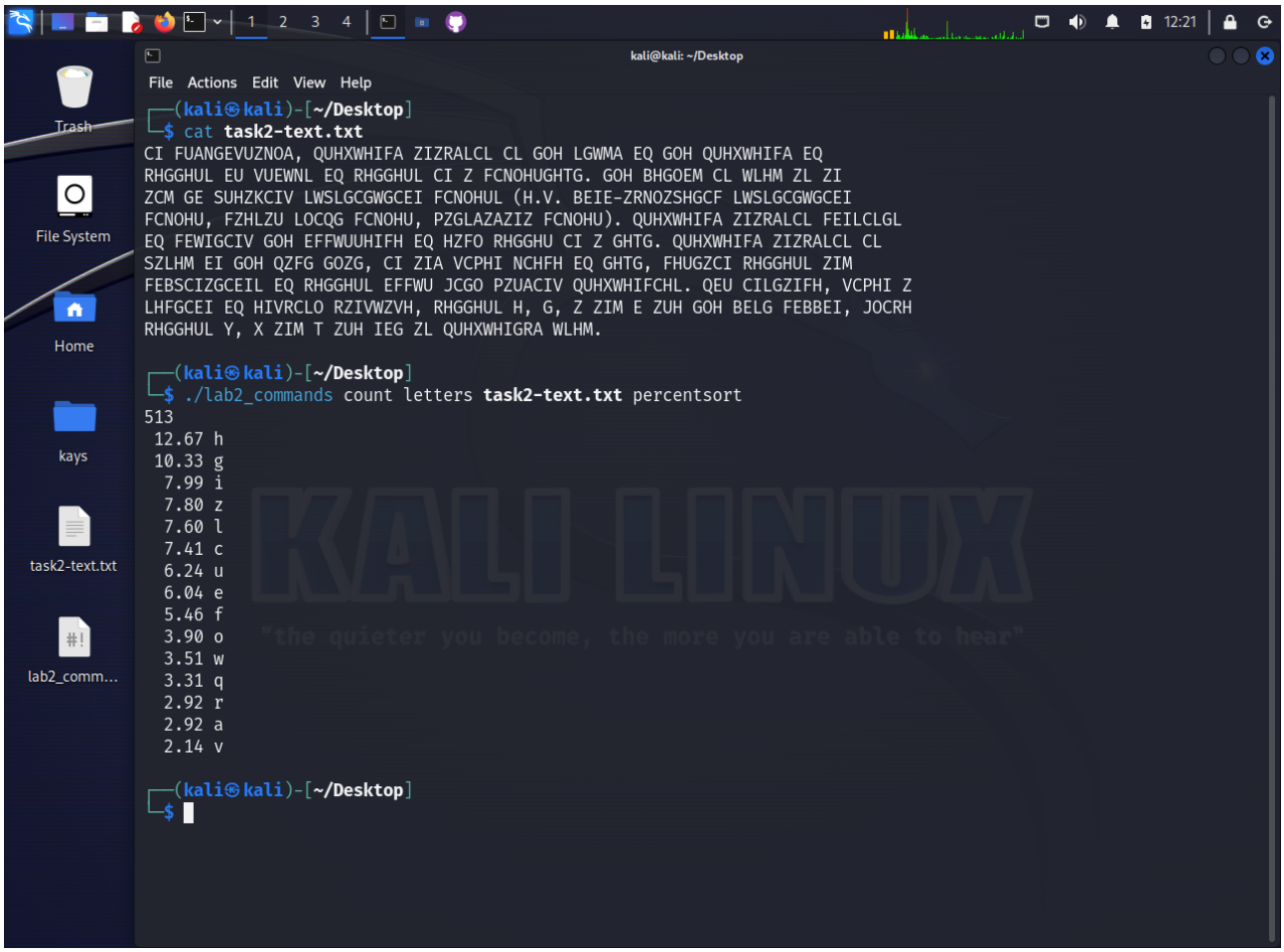
Enrique Juan Gamboa

D23125488

INDEX:

Task II

To accomplish the decryption of the text, we first must perform a frequency analysis on it to determine the most common letters in it. For this, I will be using the code that was given to us in the second lab.



Here, I have created 2 files: one with the text called “task2-text.txt”, which we will be using throughout the task, and “lab2_commands”, the commands given to us for the second lab which I have given executable permissions. This way I can run the command shown, and determine the most common letters.

With these letters, we can start by creating a couple of tables which will switch the common letters in the text with the most common letters in the English language.

```
#Character test lists

list1 = {
    "h" : "e",
    "g" : "t",
    "i" : "a",
    "z" : "o",
    "l" : "n",
    "c" : "i",
    "u" : "s",
    "e" : "h",
    "f" : "r",
    "o" : "l",
    "w" : "d",
    "q" : "u",
    "r" : "c",
    "a" : "m",
    "v" : "p",
}
```

```
list2 = {
    "h" : "e",
    "g" : "t",
    "i" : "a",
    "z" : "o",
    "l" : "n",
    "c" : "i",
    "u" : "s",
    "e" : "h",
    "f" : "r",
    "o" : "l",
    "w" : "d",
    "q" : "u",
    "r" : "c",
    "a" : "m",
    "v" : "p",
    "s" : "y",
    "y" : "s",
}
```

With these lists, we can use the code I have written (“task2-code.py”) to switch the letters:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 task2-code.py task2-text.txt list1
ia rsmNthpsoNlm, useXdearm oaocmnin in tle ntdMm hu tle useXdearm hu
cettesn hs pshdNn hu cettesn ia o riNlesteTt. tle BetlhM in dneM on oa
oiM th SseoKiap ndSntitdtiha riNlesn (e.p. Bhah-ocNloSetir ndSntitdtiha
riNles, roenos nliut riNles, Potnmomoao riNles). useXdearm oaocmnin rhanintn
hu rhdatiap tle hrrdsseare hu eorl cettes ia o teTt. useXdearm oaocmnin in
SoneM ha tle uort tlot, ia oam piPea Niere hu teTt, restoia cettesn oaM
rhBSiaotihan hu cettesn hrrds Jitl Posmiap useXdearien. uhs iantoare, piPea o
nertiha hu eapcinl coapdope, cettesn e, t, o oaM h ose tle Bhnt rhBBha, Jlice
cettesn Y, X oaM T ose aht on useXdeatcm dneM.

┌──(kali㉿kali)-[~/Desktop]
└─$ python3 task2-code.py task2-text.txt list2
ia rsmNthpsoNlm, useXdearm oaocmnin in tle ntdMm hu tle useXdearm hu
cettesn hs pshdNn hu cettesn ia o riNlesteTt. tle BetlhM in dneM on oa
oiM th yseoKiap ndyntitdtiha riNlesn (e.p. Bhah-ocNloyetir ndyntitdtiha
riNles, roenos nliut riNles, Potnmomoao riNles). useXdearm oaocmnin rhanintn
hu rhdatiap tle hrrdsseare hu eorl cettes ia o teTt. useXdearm oaocmnin in
yoneM ha tle uort tlot, ia oam piPea Niere hu teTt, restoia cettesn oaM
rhByiaotihan hu cettesn hrrds Jitl Posmiap useXdearien. uhs iantoare, piPea o
nertiha hu eapcinl coapdope, cettesn e, t, o oaM h ose tle Bhnt rhBBha, Jlice
cettesn s, X oaM T ose aht on useXdeatcm dneM.

┌──(kali㉿kali)-[~/Desktop]
└─$
```

With these results we can determine a couple of repetitions: “tle” will most probably be “the”, and the first word will probably be “in”.

With a couple of trials, errors and more logical substitutions, I reached conclusions such as “oaM” being “and” due to it being a three-letter word at the end of various commas (meaning an enumeration), or “p” being “g” due to the use of parenthesis and two

punctuated letters that usually indicate an example (“e.g.”). This helped to guess a lot of switches further down the road. These were compiled to the list 5 shown below, that gives us the most potential for a solution (shown beside the list):

```
list5 = {
    "h" : "e",
    "g" : "t",
    "i" : "n",
    "z" : "a",
    "c" : "i",
    "u" : "s",
    "e" : "o",
    "f" : "d",
    "o" : "h",
    "w" : "d",
    "q" : "u",
    "r" : "c",
    "a" : "m",
    "v" : "p",
    "s" : "y",
    "y" : "s",
    "m" : "d",
    "n" : "g",
}
```

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 task2-code.py task2-text.txt list5
in cryptography, freXuency anacysis is the study of the freXuency of
cetters or groups of cetters in a cipherteTt. the Bethod is used as an
aid to breaKing substitution ciphers (e.g. Bono-acphabetic substitution
cipher, caesar shift cipher, Patsyayana cipher). freXuency anacysis consists
of counting the occurrence of each cetter in a teTt. freXuency anacysis is
based on the fact that, in any giPen piece of teTt, certain cetters and
coBbinations of cetters occur Jith Parying freXuencies. for instance, giPen a
section of engcish canguage, cetters e, t, a and o are the Bost coBBon, Jhice
cetters z, X and T are not as freXuentcy used.
```

With this, I created a final list with the remaining replacements done. The best example is the word “text”, where I was able to figure “T” was “x” thanks to the program making the switches lowercase. Shown below, the final list and the message with no uppercase letters, meaning a full decryption:

```
solution = {
    "g": "t",
    "o": "h",
    "h": "e",
    "z": "a",
    "i": "n",
    "m": "d",
    "u": "r",
    "b": "m",
    "e": "o",
    "l": "s",
    "f": "c",
    "q": "f",
    "x": "q",
    "w": "u",
    "a": "y",
    "r": "l",
    "c": "i",
    "s": "b",
    "n": "p",
    "v": "g",
    "t": "x",
    "j": "w",
    "p": "v",
    "y": "z",
}
```

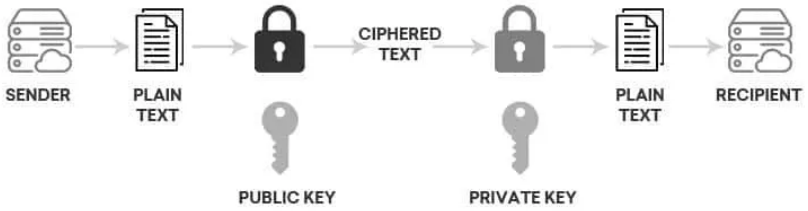
```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 task2-code.py task2-text.txt solution
in cryptography, frequency analysis is the study of the frequency of
letters or groups of letters in a ciphertext. the method is used as an
aid to breaking substitution ciphers (e.g. mono-alphabetic substitution
cipher, caesar shift cipher, vatsyayana cipher). frequency analysis consists
of counting the occurrence of each letter in a text. frequency analysis is
based on the fact that, in any given piece of text, certain letters and
combinations of letters occur with varying frequencies. for instance, given a
section of english language, letters e, t, a and o are the most common, while
letters z, q and x are not as frequently used.
```

The file “task2-code.py” is sent alongside this document. It includes the method used to switch the letters, most of the lists used and the boot code that allows to choose between lists. This code reads a .txt file as its first argument, and the list to be used must be named as the second argument.

Task III

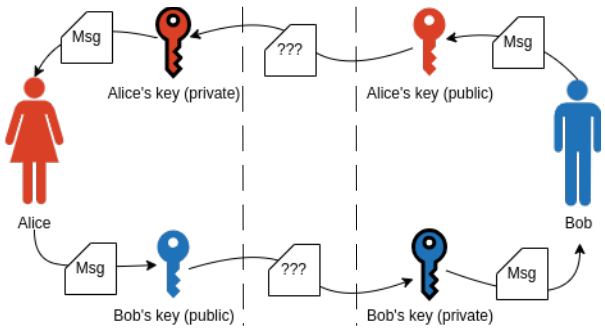
If the purpose is to securely send and receive messages using the Internet, we will have to start with the use the RSA (Rivest-Shamir-Adleman) encryption algorithm. This algorithm is based upon the use of two different keys: the receiver's public key, used by the sender, to encrypt the message; and the receiver's private key, which is used by the receiver to decrypt it.

Here's a simplified schematic of how it works:



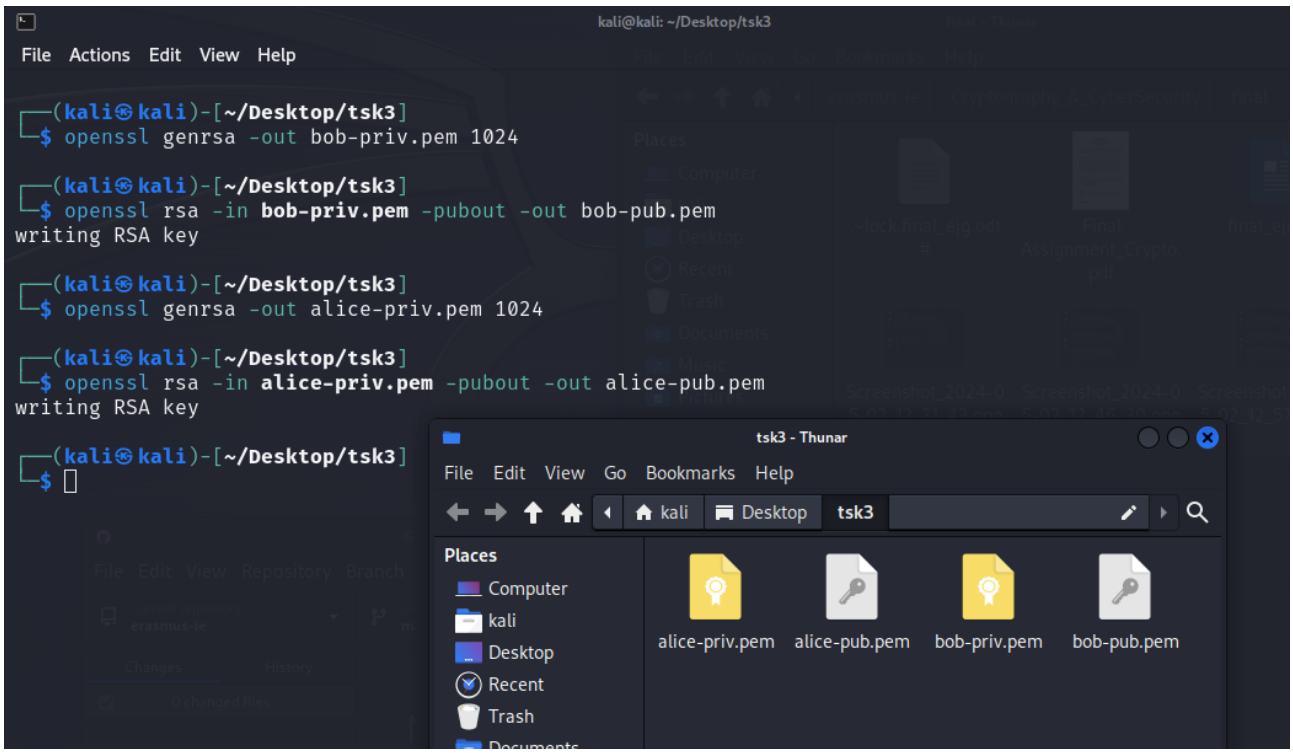
This algorithm covers the basic functionality that we want: to send and receive text messages securely. If we wanted to send files or larger texts, having multiple keys would be very impractical. That's why this algorithm is usually used to establish a connection and share the keys for other types of algorithms that allow a larger file/text size but aren't as secure to initiate a connection with.

Regardless, because we only want to send 3 short text messages, we can directly use RSA to do it. Because the communication is between 2 individuals, we will need 4 keys: one public and one private for each.

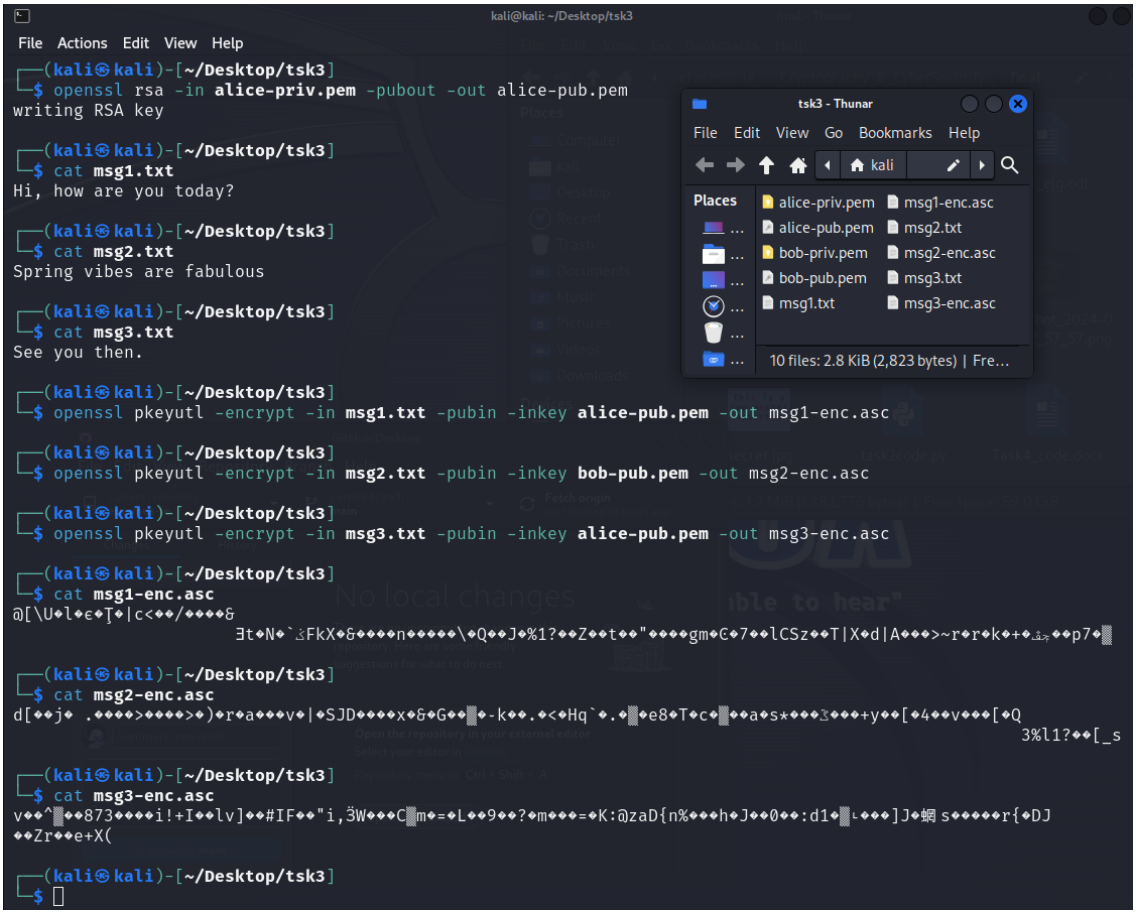


I will be mocking the exchange between Bob and Alice in my Kali VM. To do this, I will use the OpenSSL library which is pre-installed in Kali. This library gives us the basic functionality to generate keys both public and private, encrypt messages and decrypt them.

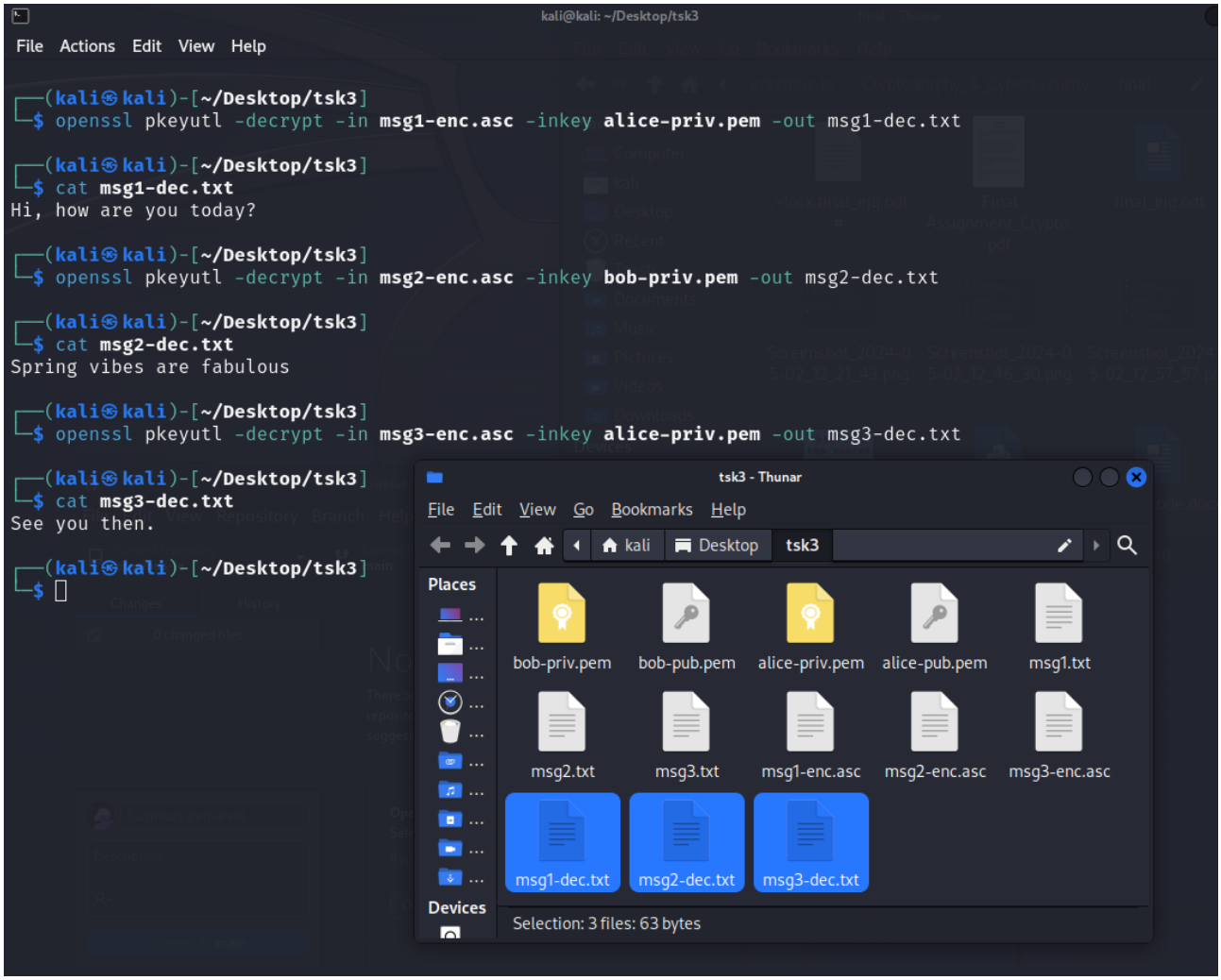
Firstly, I must generate the RSA keys: one public and one private for both Bob and Alice. This is done using the "genrsa" command given with OpenSSL for the private keys, and the "rsa" command with the public ones. The private key must be passed as an argument for the public keys as to make them associated with their private counterparts.



Once this is done, I will create a text file for each message of the conversation and encrypt them accordingly: the first one is sent by Bob so he will encrypt the message with Alice's public key, the second will be vice-versa and the third, the same again.



Finally we must decrypt each message with the corresponding private key: the first and last one with Alice’s, and the second one with Bob’s.



The encryption/decryption would happen in real time, one message after the other, but to demonstrate functionality I encrypted and decrypted all messages at the same time. We will also suppose that both Bob and Alice have already received each other’s public key to communicate successfully.

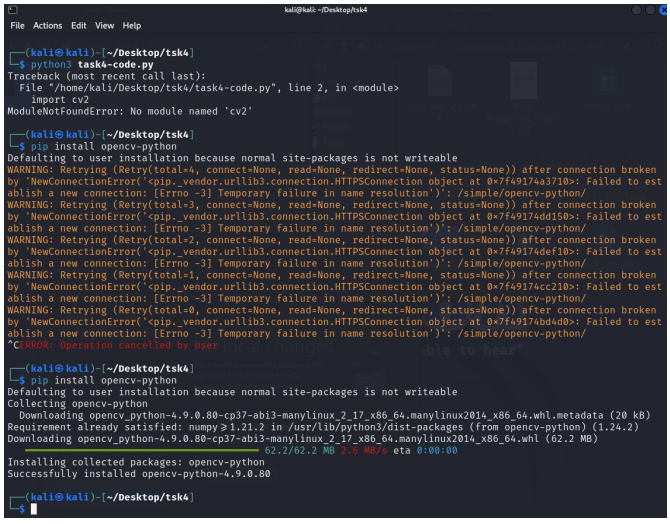
Task IV

The document attached is a python program that uses the AES (Advanced Encryption Standard) algorithm in ECB (Electronic CodeBook) mode to firstly encrypt an image and then decrypt it back. We can determine where the image is encrypted and decrypted in the first and second images below, respectively.

```
17 keyPress = cv2.waitKey(20)
18 imageOrigBytes = imageOrig.tobytes()
19 key = get_random_bytes(keySize)
20 iv = get_random_bytes(ivSize)
21 cipher = AES.new(key, AES.MODE_CBC, iv) if mode == AES.MODE_CBC else AES.new(key, AES.MODE_ECB)
22 imageOrigBytesPadded = pad(imageOrigBytes, AES.block_size)
23 ciphertext = cipher.encrypt(imageOrigBytesPadded)
24 paddedSize = len(imageOrigBytesPadded) - len(imageOrigBytes)
25 void = columnOrig * depthOrig - ivSize - paddedSize
26 ivCiphertextVoid = iv + ciphertext + bytes(void)
27 imageEncrypted = np.frombuffer(ivCiphertextVoid, dtype = imageOrig.dtype).reshape(rowOrig + 1, columnOrig, depthOrig)
28 cv2.imshow("Encrypted image", imageEncrypted)
```

```
29 keyPress = cv2.waitKey(20)
30 imageOrigBytes = imageOrig.tobytes()
31 rowEncrypted, columnOrig, depthOrig = imageEncrypted.shape
32 rowOrig = rowEncrypted - 1
33 encryptedBytes = imageEncrypted.tobytes()
34 iv = encryptedBytes[:ivSize]
35 imageOrigBytesSize = rowOrig * columnOrig * depthOrig
36 paddedSize = (imageOrigBytesSize // AES.block_size + 1) * AES.block_size - imageOrigBytesSize
37 encrypted = encryptedBytes[ivSize : ivSize + imageOrigBytesSize + paddedSize]
38 cipher = AES.new(key, AES.MODE_CBC, iv) if mode == AES.MODE_CBC else AES.new(key, AES.MODE_ECB)
39 decryptedImageBytesPadded = cipher.decrypt(encrypted)
40 decryptedImageBytes = unpad(decryptedImageBytesPadded, AES.block_size)
41 decryptedImage = np.frombuffer(decryptedImageBytes, imageEncrypted.dtype).reshape(rowOrig, columnOrig, depthOrig)
42 cv2.imshow("Decrypted Image", decryptedImage)
```

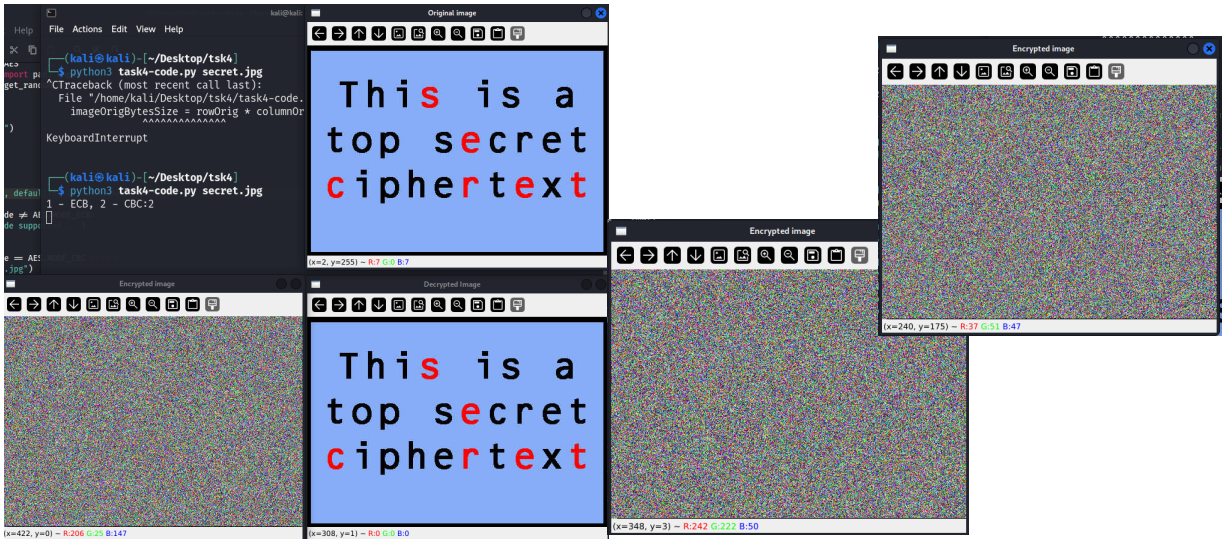
To run this code, we need to install the opencv library. This library allows the python code to create windows to display the images. Also, for the code to show all images at the same time, some changes must be made in lines 17, 29 and 43-47.




```
8 mode = AES.MODE_ECB
9 if mode ≠ AES.MODE_CBC and mode ≠ AES.MODE_ECB:
10     print('Only CBC and ECB mode supported...')
11     sys.exit()
12 keySize = 32
13 ivSize = AES.block_size if mode == AES.MODE_CBC else 0
14 imageOrig = cv2.imread("secret.jpg")
15 rowOrig, columnOrig, depthOrig = imageOrig.shape
16 cv2.imshow("Original image", imageOrig)
17 #keyPress = cv2.waitKey(20)
18 imageOrigBytes = imageOrig.tobytes()
19 key = get_random_bytes(keySize)
20 iv = get_random_bytes(ivSize)
21 cipher = AES.new(key, AES.MODE_CBC, iv) if mode == AES.M
22 imageOrigBytesPadded = pad(imageOrigBytes, AES.block_siz
23 ciphertext = cipher.encrypt(imageOrigBytesPadded)
24 paddedSize = len(imageOrigBytesPadded) - len(imageOrigBy
25 void = columnOrig * depthOrig - ivSize - paddedSize
26 ivCiphertextVoid = iv + ciphertext + bytes(void)
27 imageEncrypted = np.frombuffer(ivCiphertextVoid, dtype =
28 cv2.imshow("Encrypted image", imageEncrypted)
29 #keyPress = cv2.waitKey(20)
30 imageOrigBytes = imageOrig.tobytes()
31 rowEncrypted, columnOrig, depthOrig = imageEncrypted.sha
32 rowOrig = rowEncrypted - 1
33 encryptedBytes = imageEncrypted.tobytes()
34 iv = encryptedBytes[:ivSize]
35 imageOrigBytesSize = rowOrig * columnOrig * depthOrig
36 paddedSize = (imageOrigBytesSize // AES.block_size + 1)
37 encrypted = encryptedBytes[ivSize : ivSize + imageOrigBy
38 cipher = AES.new(key, AES.MODE_CBC, iv) if mode == AES.M
39 decryptedImageBytesPadded = cipher.decrypt(encrypted)
40 decryptedImageBytes = unpad(decryptedImageBytesPadded, A
41 decryptedImage = np.frombuffer(decryptedImageBytes, imag
42 cv2.imshow("Decrypted Image", decryptedImage)
43 while True:
44         keyPress = cv2.waitKey(1) & 0xFF
45         if key == ord('q'):
46             break
47 cv2.destroyAllWindows()
```

By doing this, we can finally see all windows at the same time:

As we can see, the encrypted image isn't that… encrypted. It is relatively easy to read the text, and running the code various times will render the same encrypted image. This is due to the use of the ECB mode. By adding a console input to decide between ECB and CBC (Cipher Block Chaining) mode at the start of the code, we can choose the latter and see that, thanks to its use of diffusion and other parameters, it is much more secure and changes each execution. The modified code will be sent alongside this document as "task4-code.py".



Task V

1. What is Kali? How is it different from other Linux distributions? How does it support cyber security?

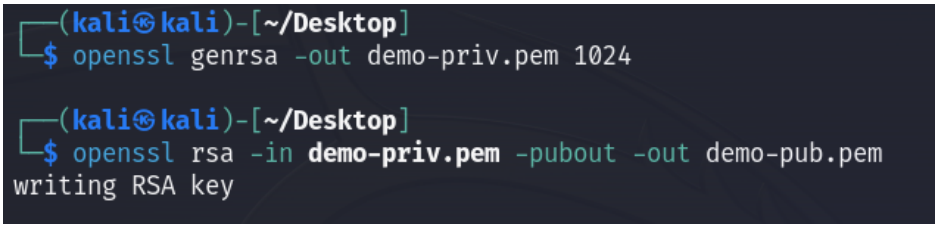
Kali is a privacy and cyber-security centred, Debian based Linux distribution. It differs from other distros due to its commitment to offer easy access to cyber security tools with its plethora of libraries, open kernel and regular updates.

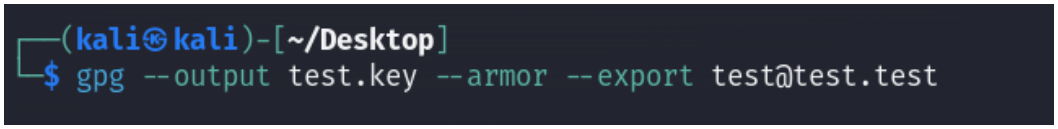
Kali prides itself on its Open Source community, its privacy tools and the easy access to training material and documentation. This makes it an essential operating system for those interested in learning cyber security, those who want to protect their privacy and even professionals who need it for their work.

2. What command line is used for sharing your public key in a file?

Both GnuPG and OpenSSL can be used for this purpose. The choice between either of them depends on what encryption is going to be used: OpenSSL offers various types of encryption, while GnuPG is held to the OpenPGP standard.

The images below show how to generate public keys using OpenSSL and GnuPG, respectively:





After this, the only thing left to do is to share the file generated with whoever we want to give access to.

3. How can you run a brute force attack?

A brute force attack is trying out any combination imaginable for a given security feature until going through. This solely relies on performance and speed rather than algorithms, which means that the greater the power of the device you are running the brute force attack from the less time it will take to “get in”.

It is assumed that, for example, to force entry into an account we won’t use the same combinations as to force break the decryption of a message. Not changing approaches would mean that entry would be impossible regardless of performance.

To run this type of attack in Kali, we can use the Hydra library. This library gives us a plethora of commands to use, so in the image below I will be showing the command to run a brute force attack upon the login page of an example website:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ hydra -l admin -P passwords.txt example.com http-post-form "/login.php:user=^USER^&passw
ord=^PASS^:Invalid password!" -V
```

“-l admin” is the username we are trying to brute force into, and “-P passwords.txt” is the file with the list of passwords we are going to attempt. “example.com” is the target website, and what follows is the post form that we will be checking to see if we failed the entry and should continue attacking.

4. How can you create targeted wordlist and run hashcat?

To create a targeted wordlist we need to gather information upon the targets we want to break the hash from and brainstorm what their passwords could be, compiling them to a list.

Alternatively, we can use the CeWL (Custom Word List generator) library to scan websites for words and text. This is a more general approach and not as precise, but it is less time consuming and doesn’t involve invading people’s privacy. To use CeWL we only need to give it the target website and the output file like so:

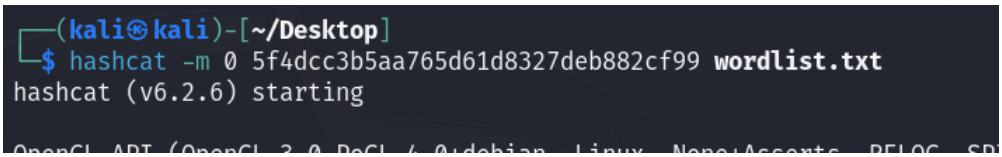
```
File  Actions  Edit  View  Help

┌──(kali㉿kali)-[~/Desktop]
└─$ cewl http://example.com -w wordlist.txt
CeWL 6.1 (Max Length) Robin Wood (robin@digi.ninja) (https://digi.ninja/)

┌──(kali㉿kali)-[~/Desktop]
└─$ cat wordlist.txt
Example
Domain
```

After this, we just need to run hashcat with the word list. This gives us the potential of decrypting private passwords to gain access to various systems. The hashcat

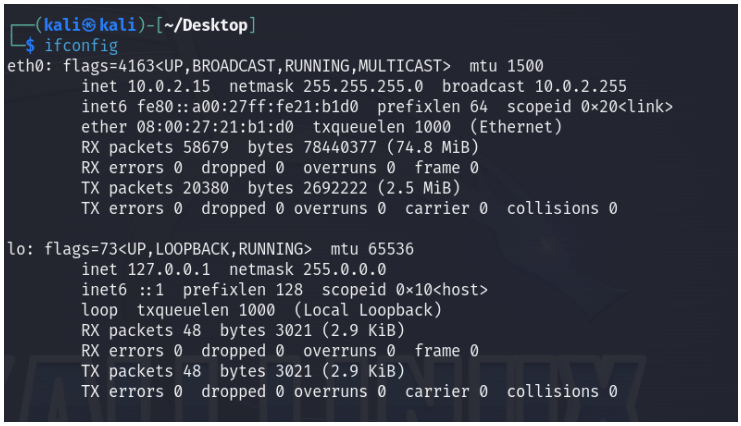
command takes as parameters a file where hashes are stored (alternatively, a hash can be pasted in this field) and another file where the wordlist is stored. Hashcat will display its progress, and there we must see if any passwords were cracked or not.



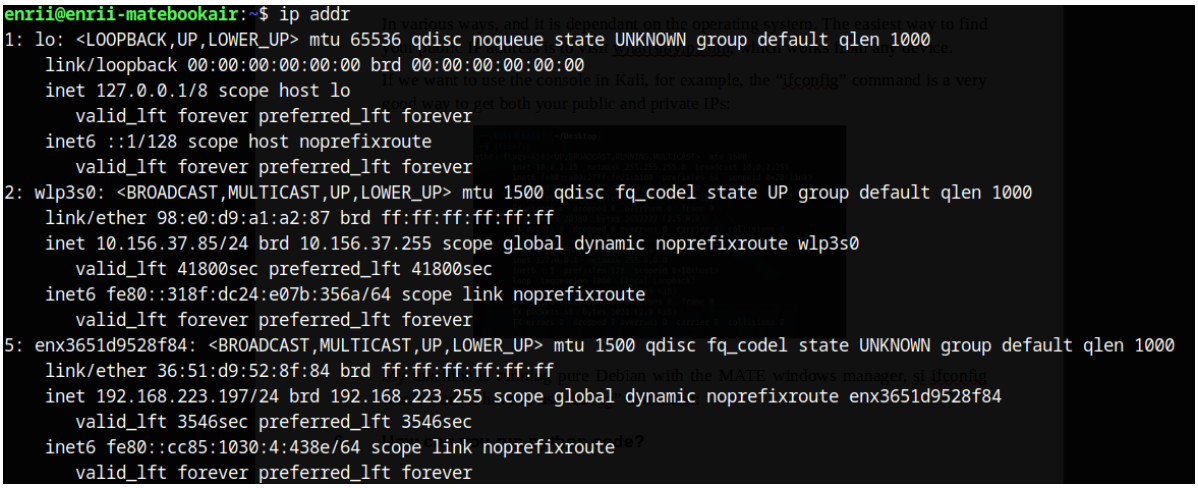
5. How can you get your IP computer address?

In various ways, and it is dependant on the operating system. The easiest way to find your public IP address is to visit whatismyip.com, which works from any device.

If we want to use the console in Kali, for example, the “ifconfig” command is a very good way to get your public and private IPs:



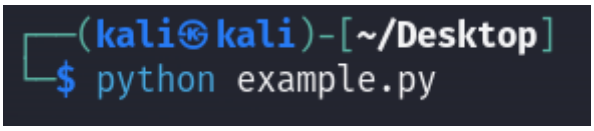
Another example: my machine is running pure Debian with the MATE windows manager, so “ifconfig” won’t work. I have to use the “ip addr” command:



6. How can you run python code?

To run Python code a Python installation is needed in the machine of your choice, if its not pre-installed. You can check this if any of the following commands exist in your machine's terminal: "py3", "python", "pip" or "python3". If not, you must follow the installation instructions for your operating system in the python.org website.

Once done, you can simply open a console window in the directory where the python file you want to run is located and execute the working command for your machine (refer to the list I made earlier) with the filename as the argument. If the code asks for more arguments, you can append them after the filename.



Alternatively, you can directly write code in the command line by just writing the command, pressing enter, writing the code and pressing enter again: