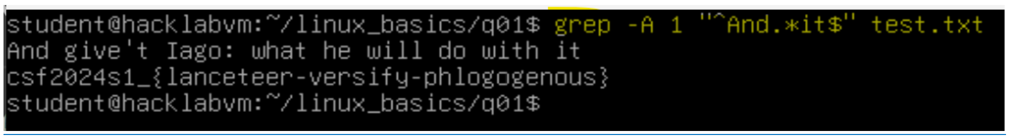
Assignment 0x01

Menno Brandt, a1849852

Part I – Intro

Question 1:

Screenshots:



```
student@hacklabvm:~/linux_basics/q01$ grep -A 1 "^And.*it$" test.txt
And give't Iago: what he will do with it
csf2024s1_{lanceteer-versify-phlogogenous}
student@hacklabvm:~/linux_basics/q01$
```

Explanation: I used grep, with the regular expression, "^And.*it$"

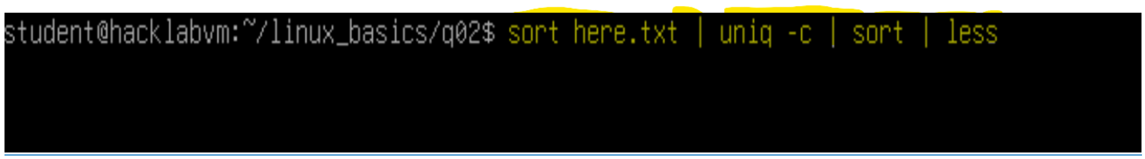
This looks for "And" at the start, "it" at the end, and the .* characters mean that anything (e.g. words) can be in between.

The "-A 1" option tells grep to print the proceeding line as well. In this case, the next line is of course the flag.

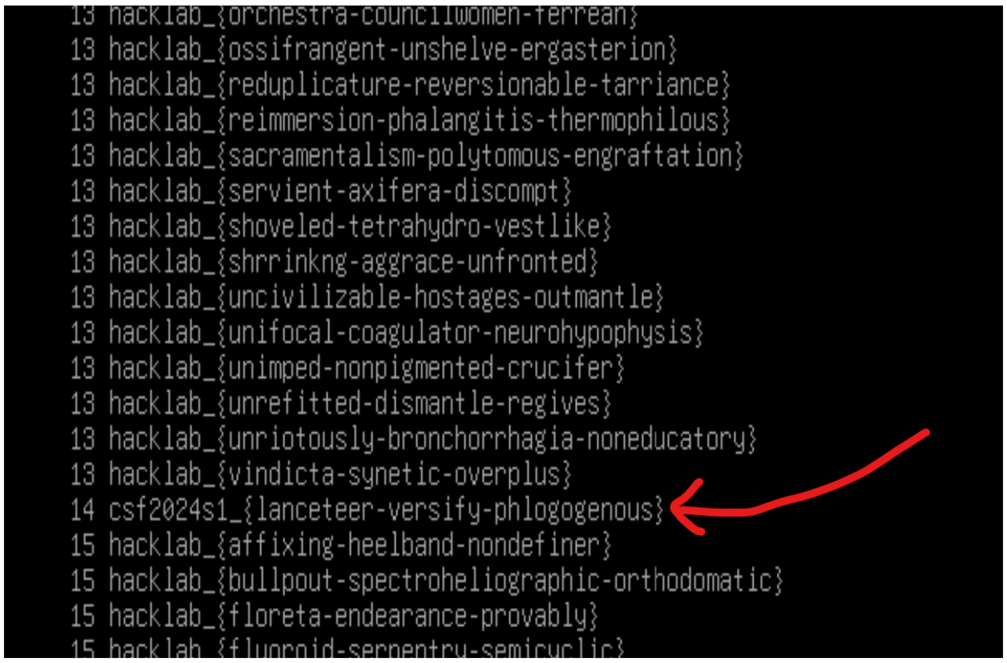
Answer: csf2024s1_{lanceteer-versify-phlogogenous}.

Question 2:

Screenshots:



```
student@hacklabvm:~/linux_basics/q02$ sort here.txt | uniq -c | sort | less
```



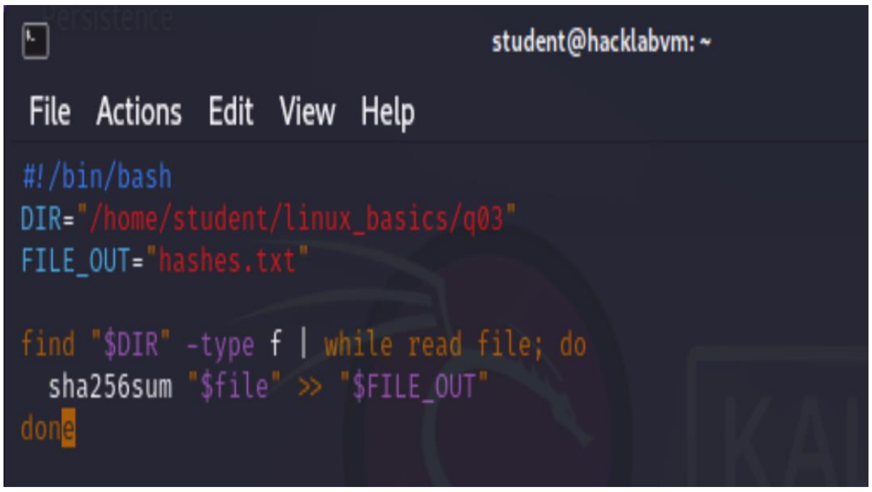
```
     13 hacklab_{orchestra-councilwomen-terrean}
     13 hacklab_{ossifrangent-unshelve-ergasterion}
     13 hacklab_{reduplicature-reversionable-tarriance}
     13 hacklab_{reimmersion-phalangitis-thermophilous}
     13 hacklab_{sacramentalism-polytomous-engraftation}
     13 hacklab_{servient-axifera-discompt}
     13 hacklab_{shoveled-tetrahydro-vestlike}
     13 hacklab_{shrrinkng-aggrace-unfronted}
     13 hacklab_{uncivilizable-hostages-outmantle}
     13 hacklab_{unifocal-coagulator-neurohypophysis}
     13 hacklab_{unimped-nonpigmented-crucifer}
     13 hacklab_{unrefitted-dismantle-regives}
     13 hacklab_{unriotously-bronchorrhagia-noneducatory}
     13 hacklab_{vindicta-synetic-overplus}
     14 csf2024s1_{lanceteer-versify-phlogogenous}
     15 hacklab_{affixing-heelband-nondefiner}
     15 hacklab_{bullpout-spectroheliographic-orthodomatic}
     15 hacklab_{floreta-endearance-provably}
     15 hacklab_{fluoroid-serpentry-semicyclic}
```

Explanation: I'm using piping. The first command, "sort here.txt", sorts all the words alphabetically. Then "uniq -c" counts how many times a consecutive password is repeated in this alphabetical list. From that output, I then use "sort" again. This sorts it by numerical order. The purpose of the "less" command at the end, was just to enable scrolling in the terminal.

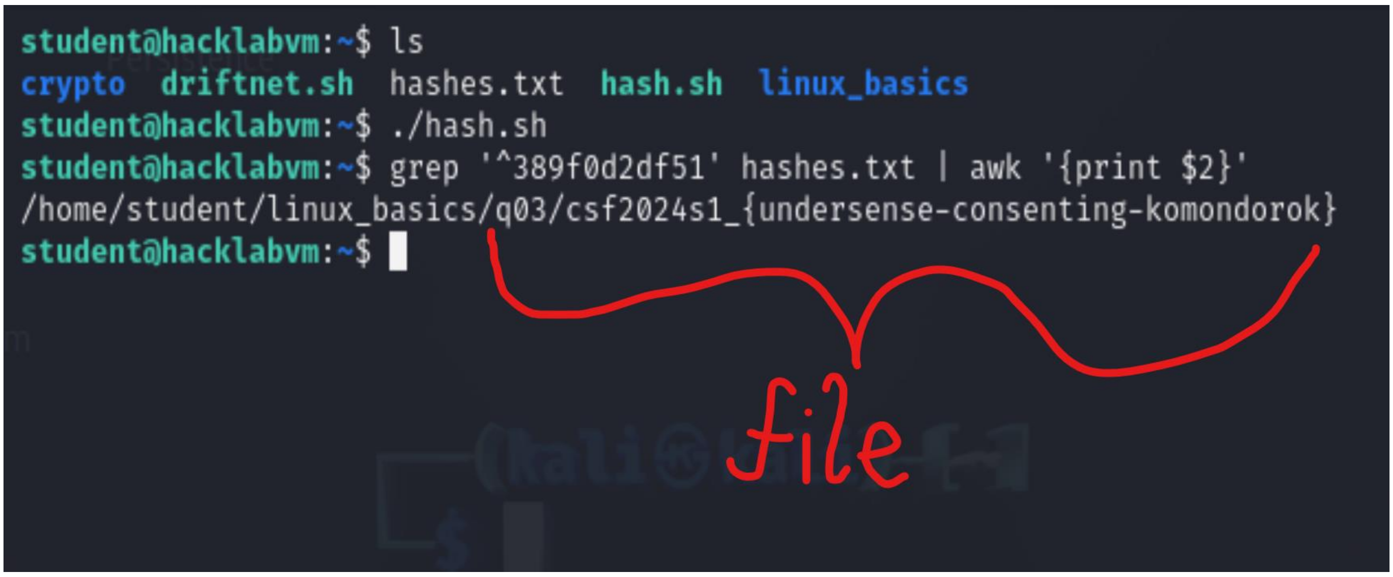
Answer: csf2024s1_{lanceteer-versify-phlogogenous}

Question 3:

Screenshots:



```
#!/bin/bash
DIR="/home/student/linux_basics/q03"
FILE_OUT="hashes.txt"

find "$DIR" -type f | while read file; do
  sha256sum "$file" >> "$FILE_OUT"
done
```



```
student@hacklabvm:~$ ls
crypto  driftnet.sh  hashes.txt  hash.sh  linux_basics
student@hacklabvm:~$ ./hash.sh
student@hacklabvm:~$ grep '^389f0d2df51' hashes.txt | awk '{print $2}'
/home/student/linux_basics/q03/csf2024s1_{undersense-consenting-komondorok}
student@hacklabvm:~$ 
```

Explanation: I SSH'd into Hacklab, with Kali. I created two files: hash.sh, and hashes.txt.

The Bash script, hash.sh, loops through every file in q03's directory, and outputs the SHA-256 hash to hashes.txt.

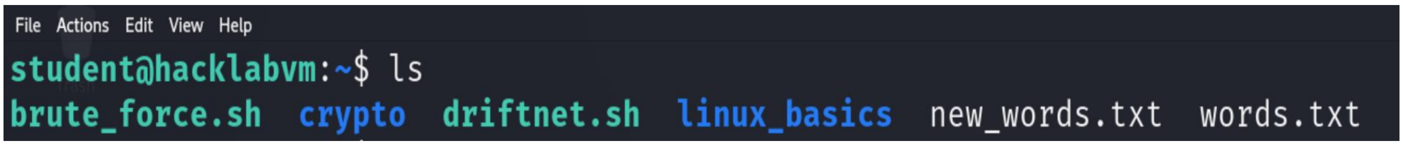
Then, I use grep to search hashes.txt, for the file with the matching hash. The purpose of the "awk" command (that grep pipes to), was to print only the file name (instead of the hash + filename). In my grep expression, I used a shortened version of the hash.

This was because it's very unlikely that 2 hashes would've been similar enough for it to output more than one result.

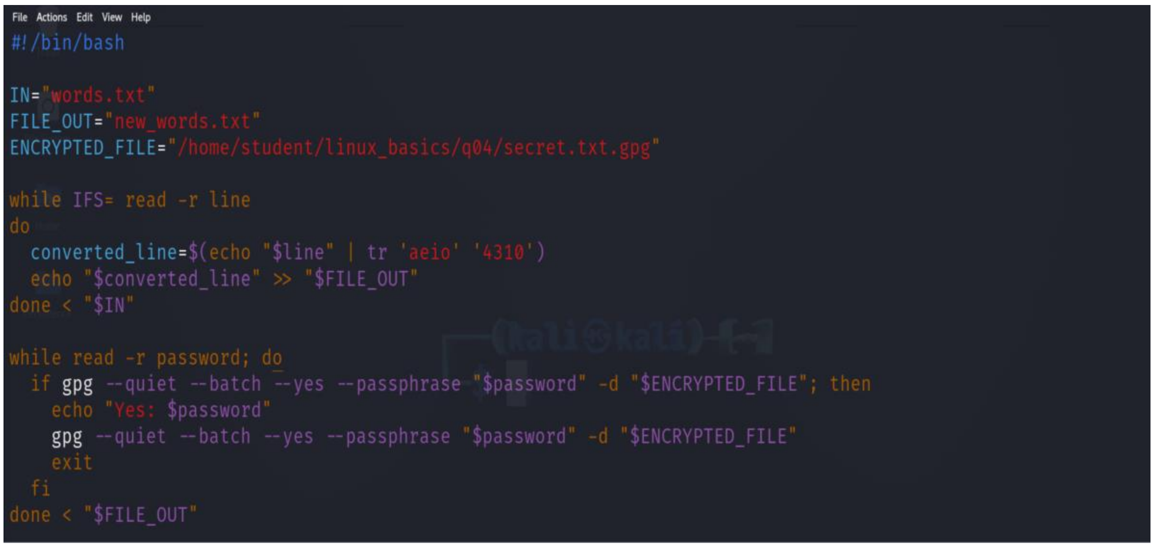
Answer: csf2024s1_{undersense-consenting-komondorok}

Question 4:

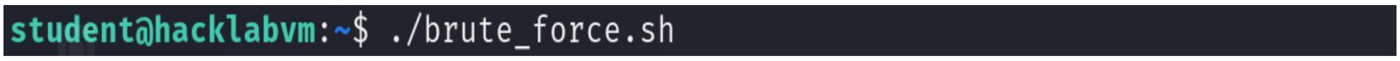
Screenshots:



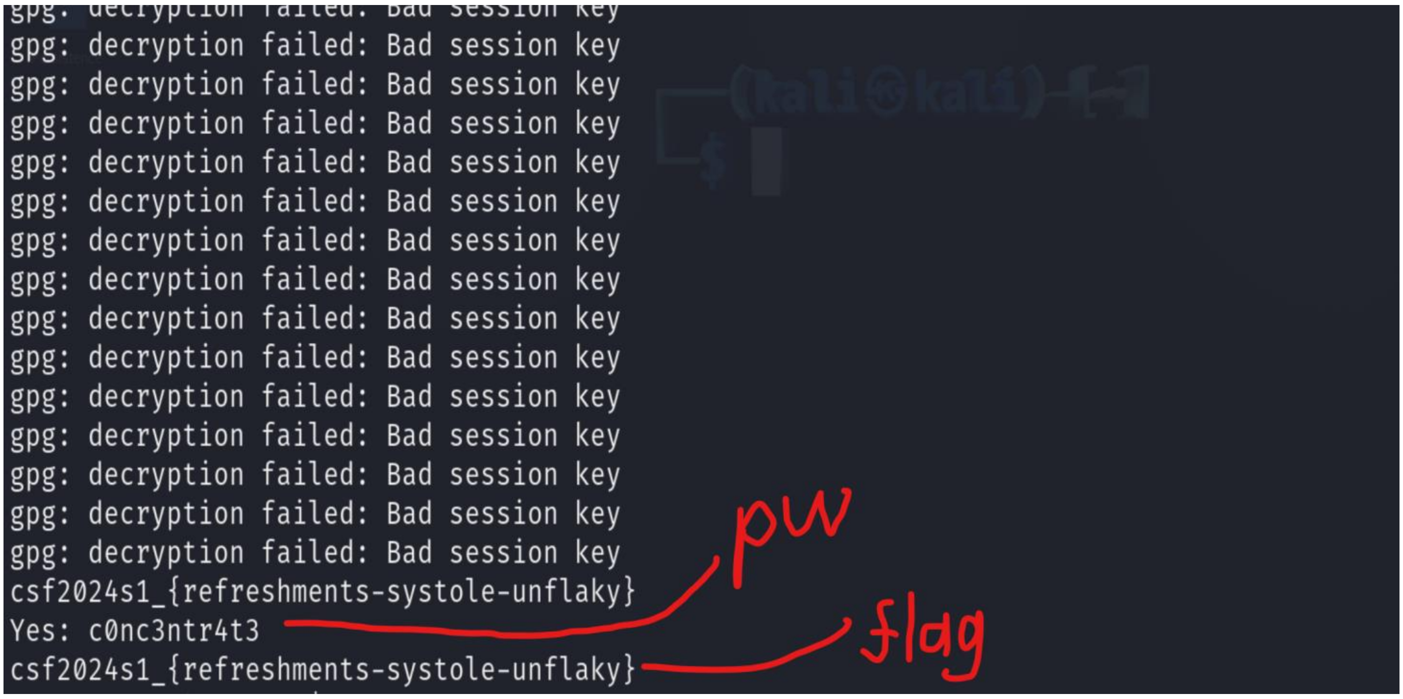
```
student@hacklabvm:~$ ls
brute_force.sh  crypto  driftnet.sh  linux_basics  new_words.txt  words.txt
```



```
#!/bin/bash

IN="words.txt"
FILE_OUT="new_words.txt"
ENCRYPTED_FILE="/home/student/linux_basics/q04/secret.txt.gpg"

while IFS= read -r line
do
  converted_line=$(echo "$line" | tr 'aeio' '4310')
  echo "$converted_line" >> "$FILE_OUT"
done < "$IN"

while read -r password; do
  if gpg --quiet --batch --yes --passphrase "$password" -d "$ENCRYPTED_FILE"; then
    echo "Yes: $password"
    gpg --quiet --batch --yes --passphrase "$password" -d "$ENCRYPTED_FILE"
    exit
  fi
done < "$FILE_OUT"
```



```
student@hacklabvm:~$ ./brute_force.sh
```



```
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
gpg: decryption failed: Bad session key
csf2024s1_{refreshments-systole-unflaky}
Yes: c0nc3ntr4t3
csf2024s1_{refreshments-systole-unflaky}
```

Explanation: The first thing I did was to copy the words.txt file, from q04's directory over to the /home/students/ directory. I then created a second text file, new_words.txt.

The idea behind this, was that instead of doing the l33t conversion in place (e.g., keeping it all in words.txt), it would be simpler to just have two files.

I then wrote my brute_force.sh Bash script. This has two goals: to perform l33t conversion, and to use the l33t converted passwords to brute force the secret.txt.gpg file.

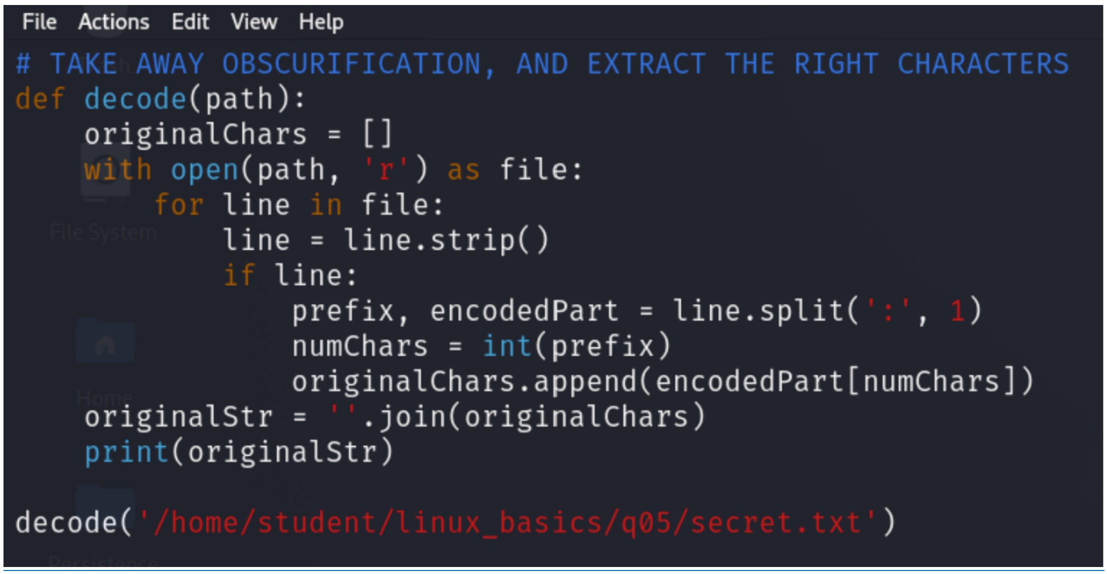
These two goals can be seen in the Bash file's while loops. The first one very simply replaces letters a->4, e->3, i -> 1, o -> 0, and adds it to new_words.txt. The second while loop, does a gpg decryption with new_words.txt's lines as passphrases. If it's successful, it breaks the while loop, prints the password that worked, and also the decrypted output.

Although I could have made the script more efficient, by combining the while loops and immediately trying every l33t converted password, it works fine. As shown by the screenshot, the flag and password were still found.

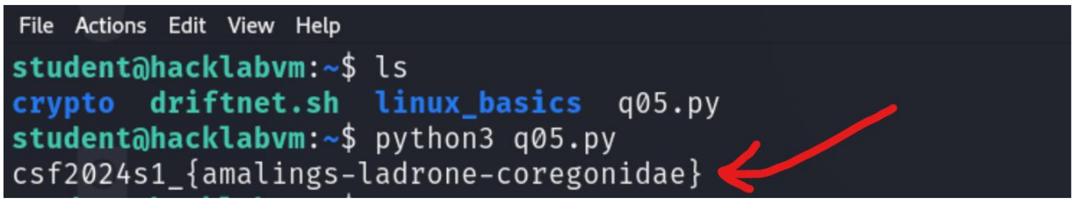
Answer: The correct password is c0nc3ntr4t3. The content of the decrypted file, is cs2024s1_{refreshments-systole-unflaky}

Question 5:

Screenshots:



```
# TAKE AWAY OBSCURIFICATION, AND EXTRACT THE RIGHT CHARACTERS
def decode(path):
    originalChars = []
    with open(path, 'r') as file:
        for line in file:
            line = line.strip()
            if line:
                prefix, encodedPart = line.split(':', 1)
                numChars = int(prefix)
                originalChars.append(encodedPart[numChars])
    originalStr = ''.join(originalChars)
    print(originalStr)

decode('/home/student/linux_basics/q05/secret.txt')
```



```
student@hacklabvm:~$ ls
crypto   driftnet.sh   linux_basics   q05.py
student@hacklabvm:~$ python3 q05.py
csf2024s1_{amalings-ladrone-coregonidae}
```

Explanation: The first thing I did, was open cyber.py and try to understand some of the code. By looking at it, I could see that its general purpose was to take in a string as its input and encode its characters. It does so by generating a 100-character long line, that has a random number of characters in front of and behind of the specific character.

Before the 100-character long line, it would also have 4 digits that specify the position of the obscured character. For example, 0042 would mean that the real character is in the 42nd spot. This meant that it was not really encrypted, but instead padded a random number of times by randomly generated characters.

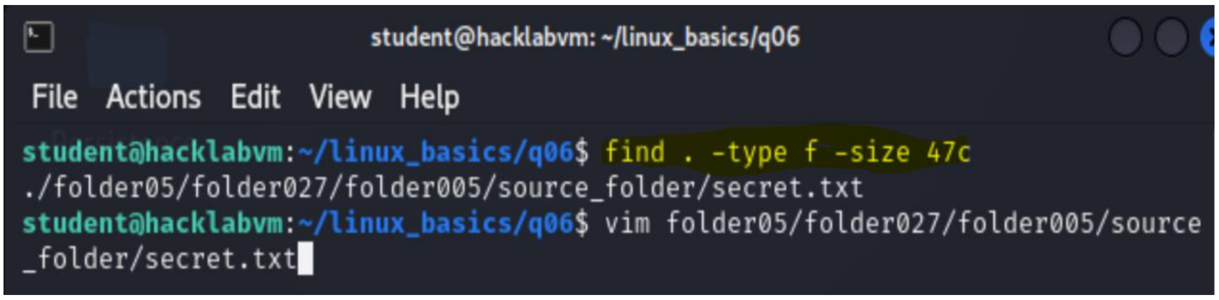
To solve this problem, I realised I had to extract the obscured characters from the right position and then combine them into a single string. I did this with the python script, q05.py, that's pictured above. This script takes each line in the file, and splits it into two parts: prefix (the 4 digit number at the start), and encodedPart (the 100-character long). 'Prefix' is then converted to an int, that's used to access the target character in encodedPart. This character is of course appended to a string, and after going through the entire file, it prints out the answer.

The process of running the script, and the answer it gave is shown in the 2nd image.

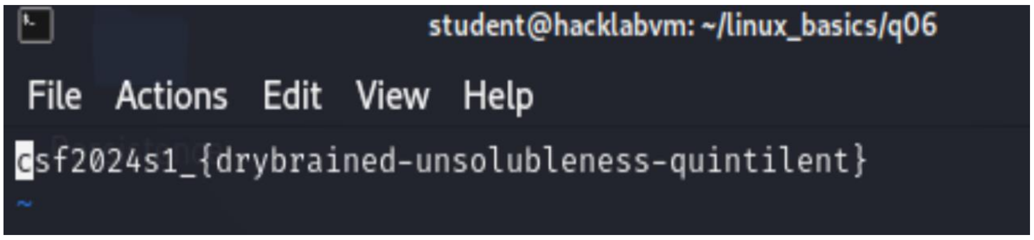
Answer: The answer is csf2024s1_{amalings-ladrone-coregonidae}

Question 6:

Screenshots:


student@hacklabvm: ~/linux_basics/q06
File Actions Edit View Help

```
student@hacklabvm:~/linux_basics/q06$ find . -type f -size 47c
./folder05/folder027/folder005/source_folder/secret.txt
student@hacklabvm:~/linux_basics/q06$ vim folder05/folder027/folder005/source_folder/secret.txt
```


student@hacklabvm: ~/linux_basics/q06
File Actions Edit View Help

```
csf2024s1_{drybrained-unsolubleness-quintilent}
~
```

Explanation: In order to locate this file, I used the command: $ find . -type f -size 47c

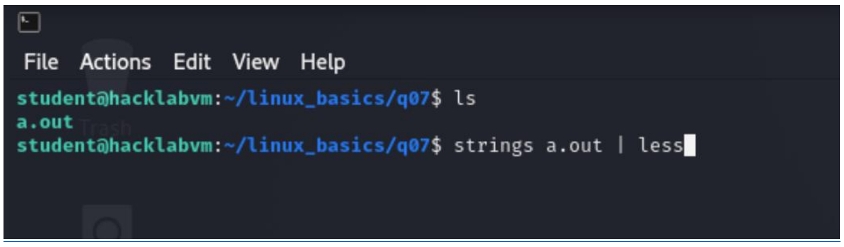
This "find" command does a recursive search through everything in q06. I used the option "-type f" to get a file (instead of a directory/folder), along with the option "-size 47c" to get the one that is exactly 47 bytes long. The command's output is the path to the file that matches these two criteria.

I then went into the folder containing secret.txt, opened it in vim, and saw the flag.

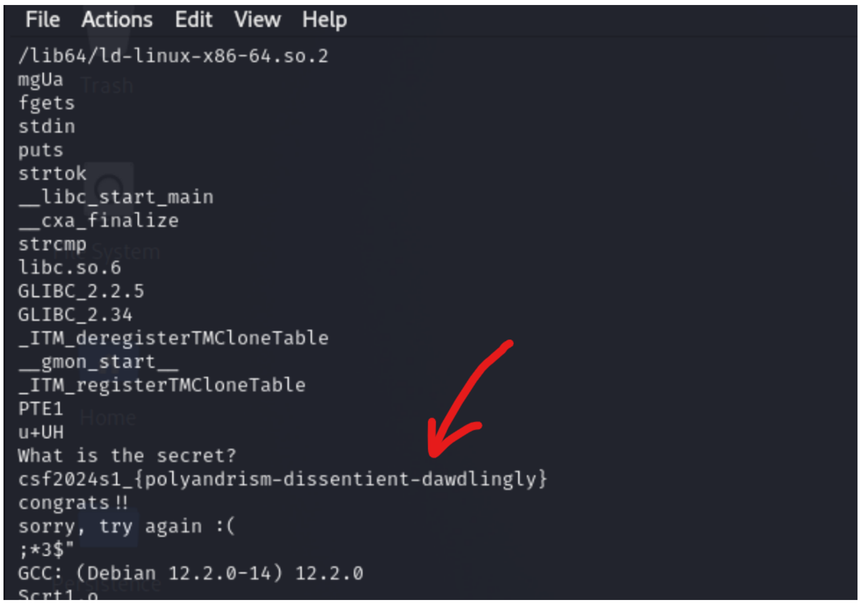
Answer: csf2024s1_{drybrained-unsolubleness-quintilent} - (in secret.txt)

Question 7:

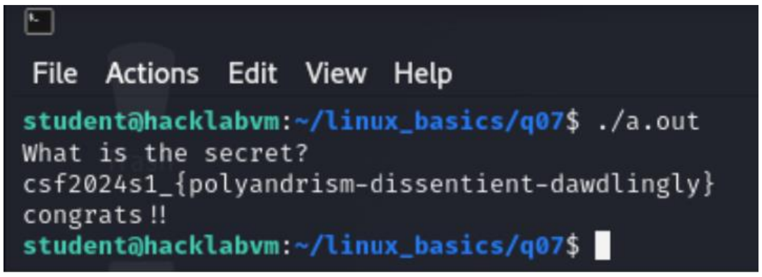
Screenshots:



```
student@hacklabvm:~/linux_basics/q07$ ls
a.out
student@hacklabvm:~/linux_basics/q07$ strings a.out | less
```



```
/lib64/ld-linux-x86-64.so.2
mgUa
fgets
stdin
puts
strtok
__libc_start_main
__cxa_finalize
strcmp
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
What is the secret?
csf2024s1_{polyandrism-dissentient-dawdlingly}
congrats !!
sorry, try again :(
;*3$"
GCC: (Debian 12.2.0-14) 12.2.0
```



```
student@hacklabvm:~/linux_basics/q07$ ./a.out
What is the secret?
csf2024s1_{polyandrism-dissentient-dawdlingly}
congrats !!
student@hacklabvm:~/linux_basics/q07$
```

Explanation: My approach to this question, involved multiple attempts. At first, I ran the program and randomly entered a word. Naturally, my answer failed. Following this, I used a few commands (e.g., $ file a.out, and also the command $objdump -D a.out) to try and find out more about the file and try and decompile it.

This was not fruitful and did not get me closer to an answer. I considered the installation of some external decomplication tools on Kali, such as Ghidra, and trying to look at the assembly. This seemed too complicated, so I opted to keep searching for built-in terminal commands that could help me.

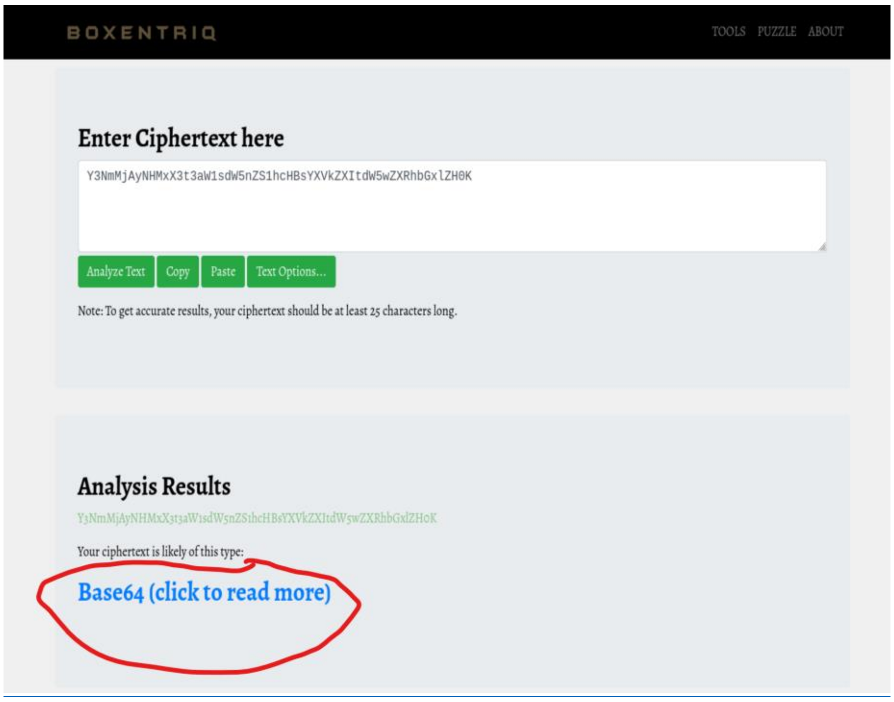
I came across the "strings" command, and read some of its documentation. It extracts readable strings from executables. The programs asks for a specific word, and that word or if-condition could perhaps be extracted by the "strings" command.

As shown by the photo, I used: $ string a.out | less, and spotted the flag it was asking for. verify that this was indeed the right flag, I ran ./a.out again and typed the flag in. This time, I was met with the message "congrats!!".

Answer: csf2024s1_{polyandrism-dissentient-dawdlingly}

Question 8:

Screenshots:



Enter Ciphertext here

```
Y3NmMjAyNHMxX3t3aW1sdW5nZS1hcHBsYXVkZXItdW5wZXRhbGxlZH0K
```

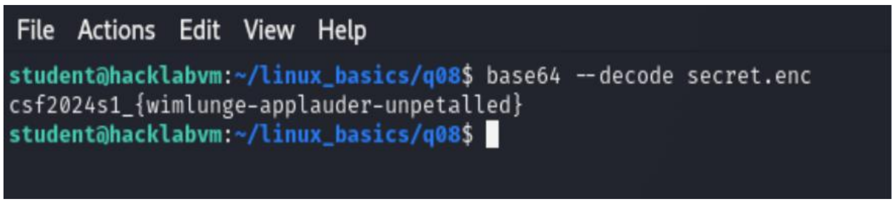
Note: To get accurate results, your ciphertext should be at least 25 characters long.

Analysis Results

Y3NmMjAyNHMxX3t3aW1sdW5nZS1hcHBsYXVkZXItdW5wZXRhbGxlZH0K

Your ciphertext is likely of this type:

Base64 (click to read more)



```
student@hacklabvm:~/linux_basics/q08$ base64 --decode secret.enc
csf2024s1_{wimlunge-applauder-unpetalled}
student@hacklabvm:~/linux_basics/q08$
```

Explanation:

My approach to this problem, was to first look up ".enc" files, and encoding formats. I learned about some different types, but ultimately made the decision to use an external tool to detect the type. This simplified the process and meant that I did not have to ascertain the type of encryption, from the .enc file's structure alone.

By copying the .enc file's contents into the online "Cypher Identifier and Analyzer" tool by Boxentriq (https://www.boxentriq.com/code-breaking/cipher-identifier), I was able to see that the file (likely) used Base64 encoding. To verify that this was indeed the encoding type and hence decode the file, I used the following terminal command:

```
$ base64  - -decode secret.enc
```

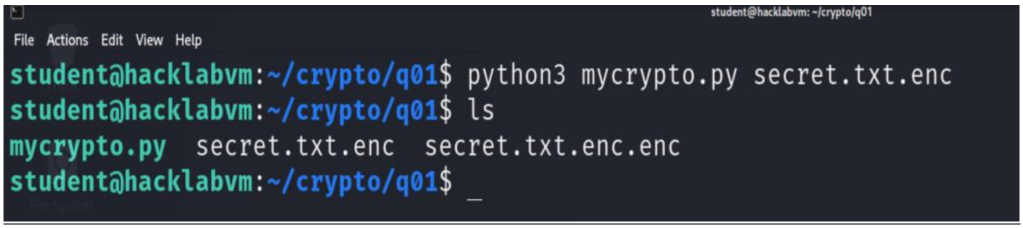
This uses the built-in base64 tool to decode the file and outputted the answer.

Answer: csf2024s1_{wimlunge-applauder-upetalled}

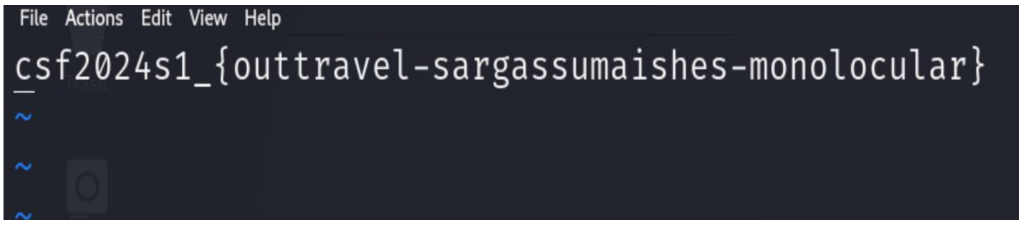
Part II – Cryptography:

Question 1:

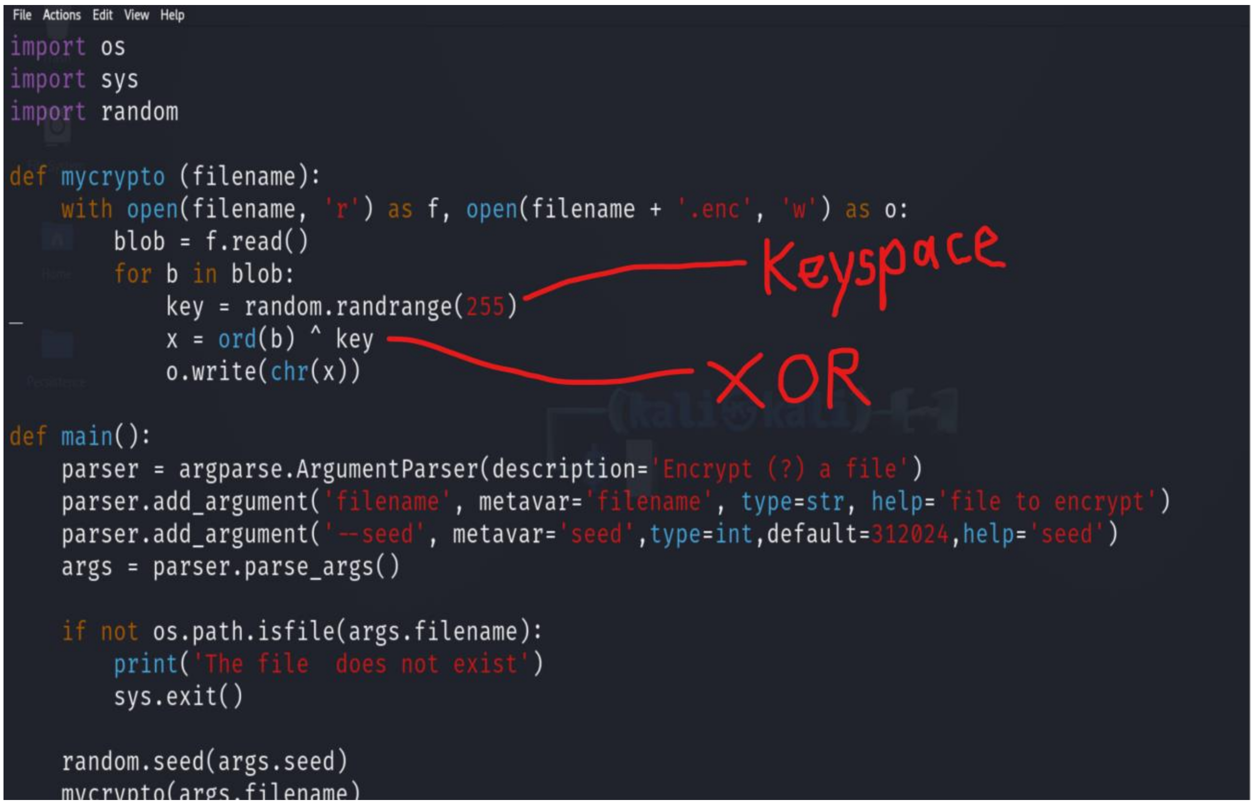
Screenshots:



```
student@hacklabvm:~/crypto/q01$ python3 mycrypto.py secret.txt.enc
student@hacklabvm:~/crypto/q01$ ls
mycrypto.py  secret.txt.enc  secret.txt.enc.enc
student@hacklabvm:~/crypto/q01$ _
```



```
csf2024s1_{outtravel-sargassumaishes-monolocular}
~
~
~
```



```
import os
import sys
import random

def mycrypto (filename):
    with open(filename, 'r') as f, open(filename + '.enc', 'w') as o:
        blob = f.read()
        for b in blob:
            key = random.randrange(255)
            x = ord(b) ^ key
            o.write(chr(x))

def main():
    parser = argparse.ArgumentParser(description='Encrypt (?) a file')
    parser.add_argument('filename', metavar='filename', type=str, help='file to encrypt')
    parser.add_argument('--seed', metavar='seed',type=int,default=312024,help='seed')
    args = parser.parse_args()

    if not os.path.isfile(args.filename):
        print('The file  does not exist')
        sys.exit()

    random.seed(args.seed)
    mycrypto(args.filename)
```

Explanation: To decrypt secret.txt.enc, I first opened mycrypto.py and had a look at the code. Its general structure was clear. It takes a file as a parameter, it passes the file into the mycrypto() function and encrypts it.

By having a look at the line, “x = order(b) ^ key”, I realised from the “^” operator that this was actually XOR encryption. It takes the file, XOR’s each byte with a key generated from a seed, and adds the .enc file extension to the final encrypted file.

Using the cryptographic property (that I had learned in Week 2’s content and Cryptography III), that:

M XOR K = C, and C XOR K = M,

I figured that I could perhaps decrypt it by just passing the same file into the program. I assumed, that the file given to us was generated by the default seed, 312024, and that none was specified. As shown by the first 2 images, I simply used the algorithm on the encrypted file. By not specifying the seed, I let it use the default one, and it ended up working.

If this didn't work, I probably would have needed to brute force the seed or use another approach.

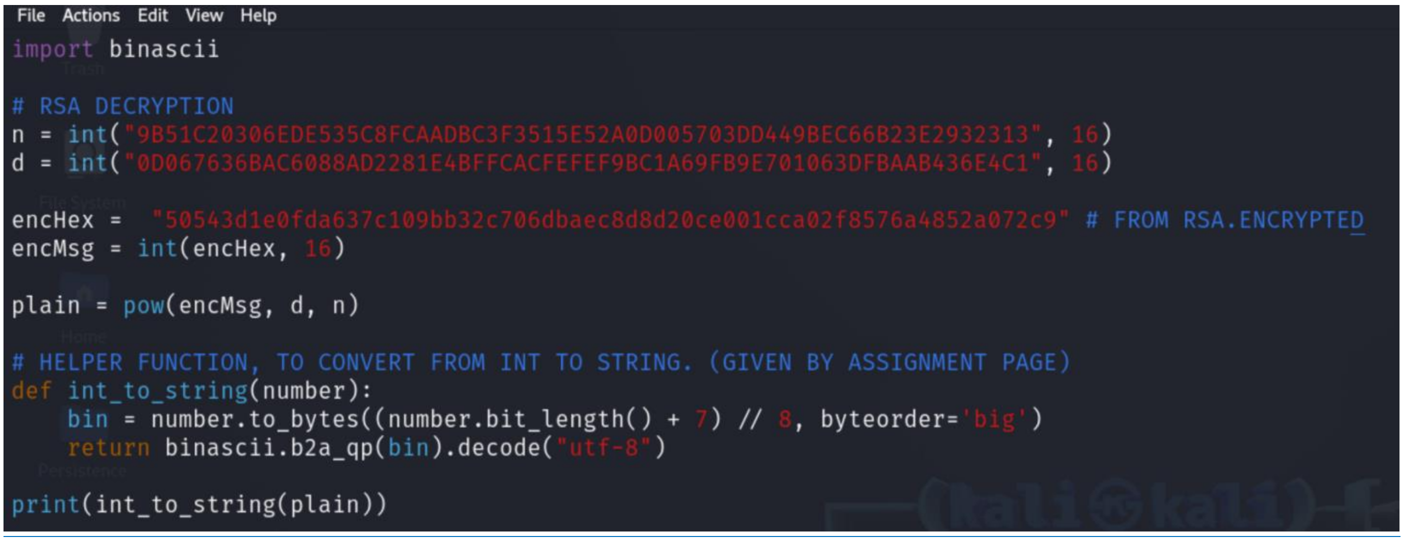
Answer: The secret is csf2024s1_{outtravel-sargassumaishes-monolocular}.

This is not good encryption, because the seed is guessable or can be brute forced. This is worsened, by the fact that reusing a seed will result in the generation of an identical sequence of keys. Hence meaning they can be reused. There is nothing in the algorithm, that guarantees their uniqueness, and this means that keys are insufficiently random. This lack of diversity in the keys, means that it's very vulnerable to attacks and key generation is predictable.

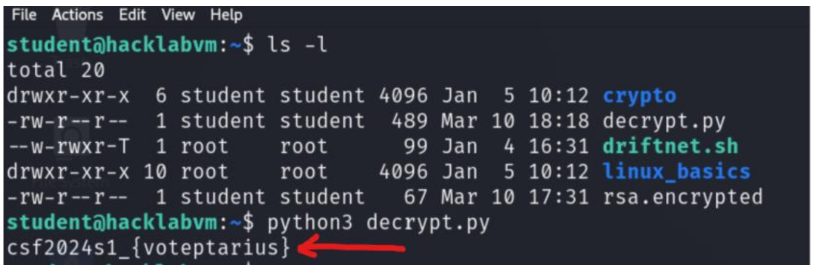
The key space of this algorithm, is 255^n. This is because, the key generated for each byte is in the range [0, 254]. This is shown in the line key = random.randrange(255). This is repeated, for an input that is n-bytes long. Therefore, the key space is 255^n. For example, in n = 3, the key space would be 255 x 255 x 255 = 255^3.

Question 2:

Screenshots:



```
import binascii

# RSA DECRYPTION
n = int("9B51C20306EDE535C8FCAADBC3F3515E52A0D005703DD449BEC66B23E2932313", 16)
d = int("0D067636BAC6088AD2281E4BFFCACFEFEF9BC1A69FB9E701063DFBAAB436E4C1", 16)

encHex =  "50543d1e0fda637c109bb32c706dbaec8d8d20ce001cca02f8576a4852a072c9" # FROM RSA.ENCRYPTED
encMsg = int(encHex, 16)

plain = pow(encMsg, d, n)

# HELPER FUNCTION, TO CONVERT FROM INT TO STRING. (GIVEN BY ASSIGNMENT PAGE)
def int_to_string(number):
    bin = number.to_bytes((number.bit_length() + 7) // 8, byteorder='big')
    return binascii.b2a_qp(bin).decode("utf-8")

print(int_to_string(plain))
```



```
student@hacklabvm:~$ ls -l
total 20
drwxr-xr-x  6 student student 4096 Jan  5 10:12 crypto
-rw-r--r--  1 student student  489 Mar 10 18:18 decrypt.py
--w-rwxr-T  1 root    root      99 Jan  4 16:31 driftnet.sh
drwxr-xr-x 10 root    root    4096 Jan  5 10:12 linux_basics
-rw-r--r--  1 student student   67 Mar 10 17:31 rsa.encrypted
student@hacklabvm:~$ python3 decrypt.py
csf2024s1_{voteptarius}
```

Explanation: In order to decrypt the RSA message, I wrote a python script. This python script was a combination of the workshop code, and the helper function that was supplied on the assignment page.

It works, by first converting the hexadecimal values of n, d, and rsa.encrypted's value to integers. Then, it applies the RSA decryption formula, m = c^d mod n, with the line: plain = pow(encMsg, d, n)

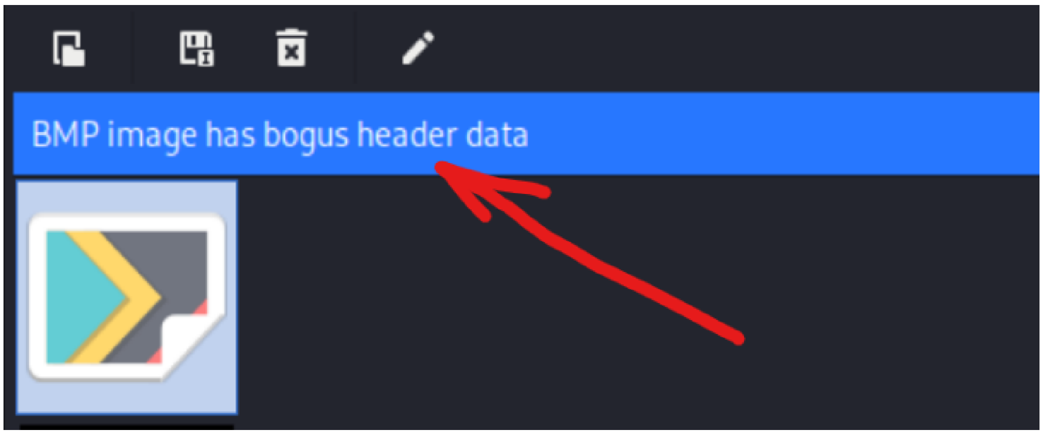
Now that it's obtained the decrypted message in its integer form, it has to be converted into a string. This is done with the helper function, int_to_string(number), and finally it is printed.

The 1st image shows the python script, and the 2nd image shows the process of running it, along with the generated decrypted output.

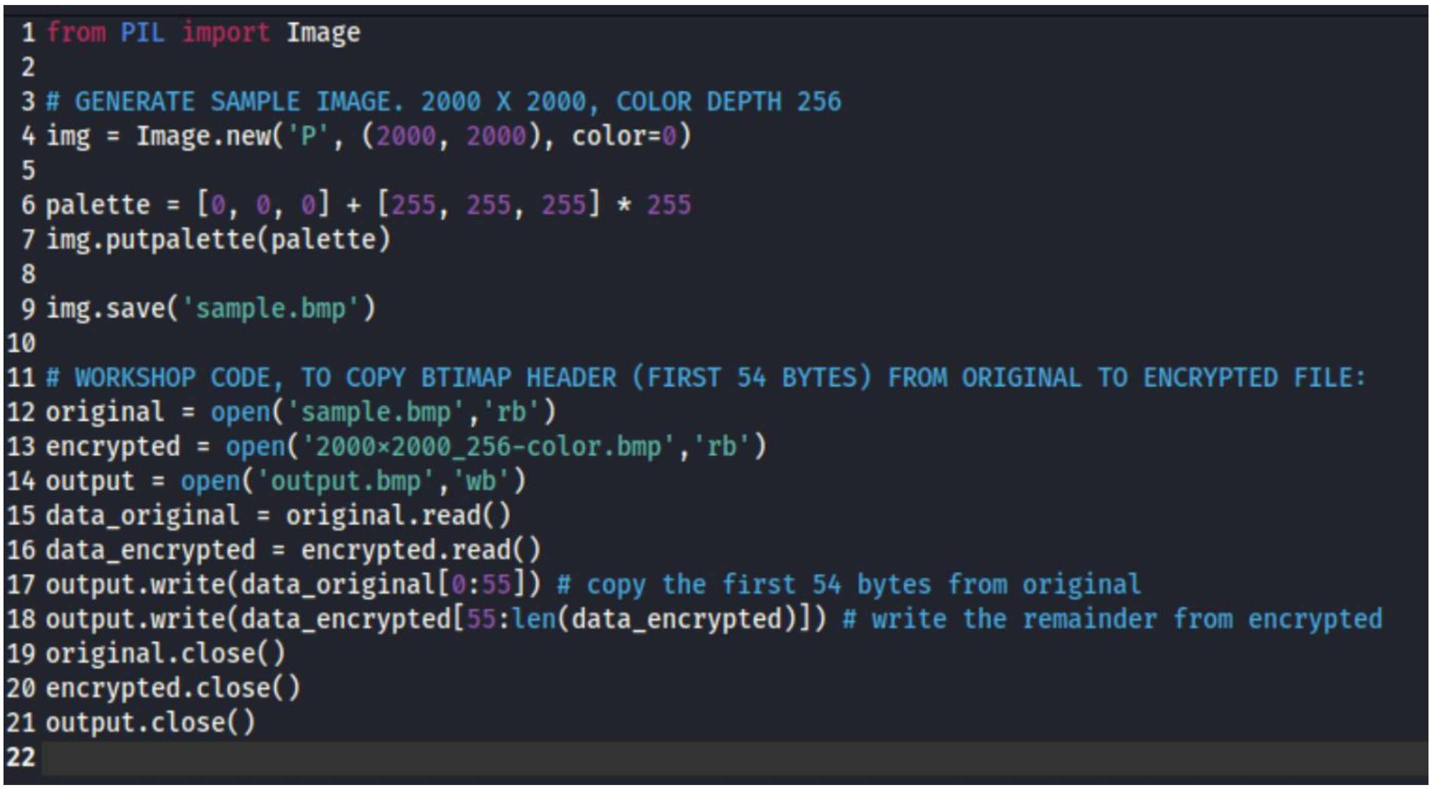
Answer: The answer/decrypted message is csf2024s1_{voteptarius}

Question 3:

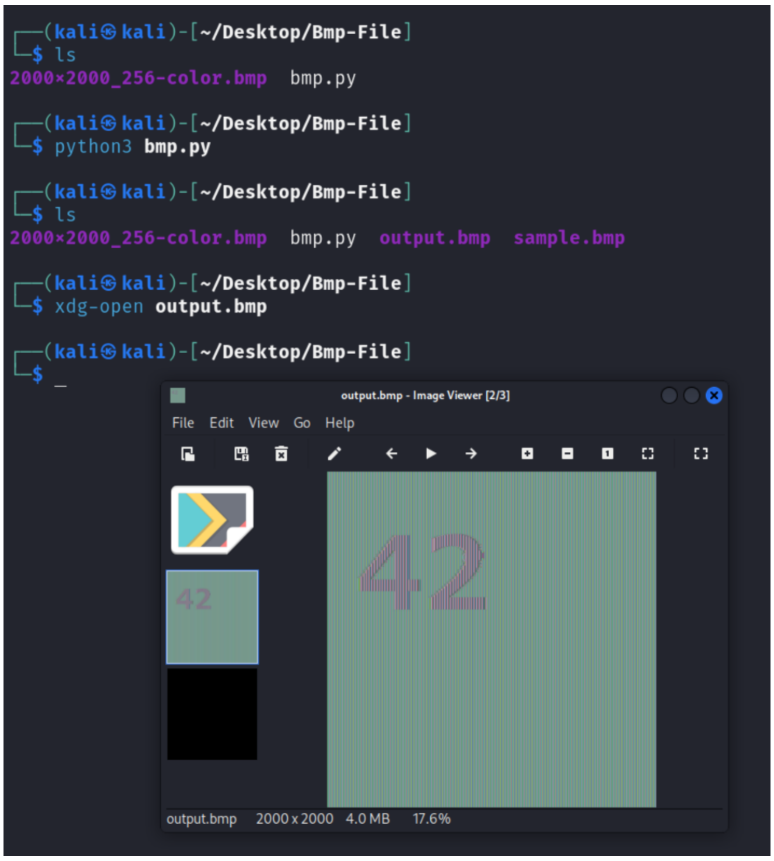
Screenshots:



BMP image has bogus header data



```python
from PIL import Image

# GENERATE SAMPLE IMAGE. 2000 X 2000, COLOR DEPTH 256
img = Image.new('P', (2000, 2000), color=0)

palette = [0, 0, 0] + [255, 255, 255] * 255
img.putpalette(palette)

img.save('sample.bmp')

# WORKSHOP CODE, TO COPY BTIMAP HEADER (FIRST 54 BYTES) FROM ORIGINAL TO ENCRYPTED FILE:
original = open('sample.bmp','rb')
encrypted = open('2000×2000_256-color.bmp','rb')
output = open('output.bmp','wb')
data_original = original.read()
data_encrypted = encrypted.read()
output.write(data_original[0:55]) # copy the first 54 bytes from original
output.write(data_encrypted[55:len(data_encrypted)]) # write the remainder from encrypted
original.close()
encrypted.close()
output.close()

```



```
┌──(kali㉿kali)-[~/Desktop/Bmp-File]
└─$ ls
2000×2000_256-color.bmp  bmp.py

┌──(kali㉿kali)-[~/Desktop/Bmp-File]
└─$ python3 bmp.py

┌──(kali㉿kali)-[~/Desktop/Bmp-File]
└─$ ls
2000×2000_256-color.bmp  bmp.py  output.bmp  sample.bmp

┌──(kali㉿kali)-[~/Desktop/Bmp-File]
└─$ xdg-open output.bmp

┌──(kali㉿kali)-[~/Desktop/Bmp-File]
└─$ _
```

output.bmp - Image Viewer [2/3]
File Edit View Go Help
42
output.bmp 2000 x 2000 4.0 MB 17.6%

Explanation: I thought at first, that the secret was hidden in a steganographic way. Perhaps I could decompile the image, or dump its information, and simply see the secret. Maybe it was hidden in the least significant bits or encoded some way.

I went along with this idea and used “file” to find out about its structure, and “hexdump” to look at the bmp file’s image data. Ultimately, these did not really work. I also uploaded the image to CyberChef, but it only showed incomprehensible symbols. I gave up on the idea that the secret was hidden in the image's literal byte data or structure.

I then removed the .encrypted extension from the file name, and tried opening it. As shown by the screenshot, the image viewer told me that the "BMP Image has bogus header data" (this is shown in the first image). From this, I gathered that it must be like Workshop 2's ECB vs CBC example. I would have to copy the header data from a functional bmp file to the encrypted one, and then I'd be able to (maybe) view it. Although it would be encrypted, the encryption would not really be good enough to completely obscure its contents.

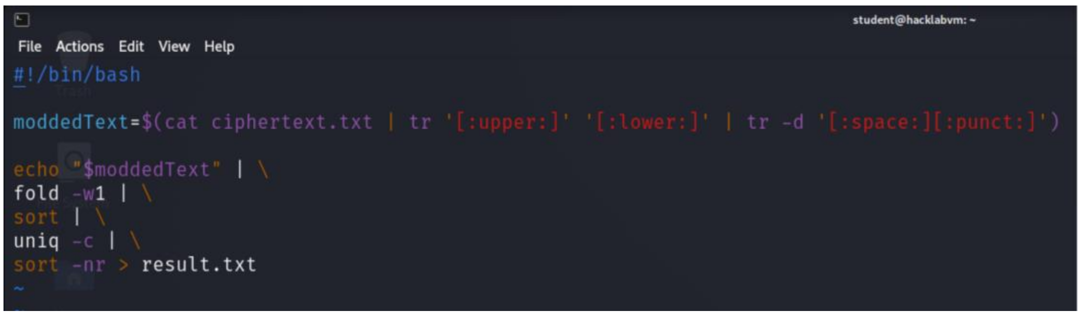
After this, I set about to write a python script. The python script would first and foremost, generate a sample bmp image. It would have the dimensions of 2000 x 2000 pixels, and a matching colour depth of 256. (I gathered these requirements, from the filename "2000x2000_256-color.bmp.encrypted"). Then, I'd use Workshop 2's sample python code to merge the two images by copying the header from the generated image to the encrypted image. This python script is displayed in the 2nd image.

As shown in the 3rd image, I ran my script (bmp.py), and opened the result (output.bmp). It worked and clearly displayed the secret number "42".

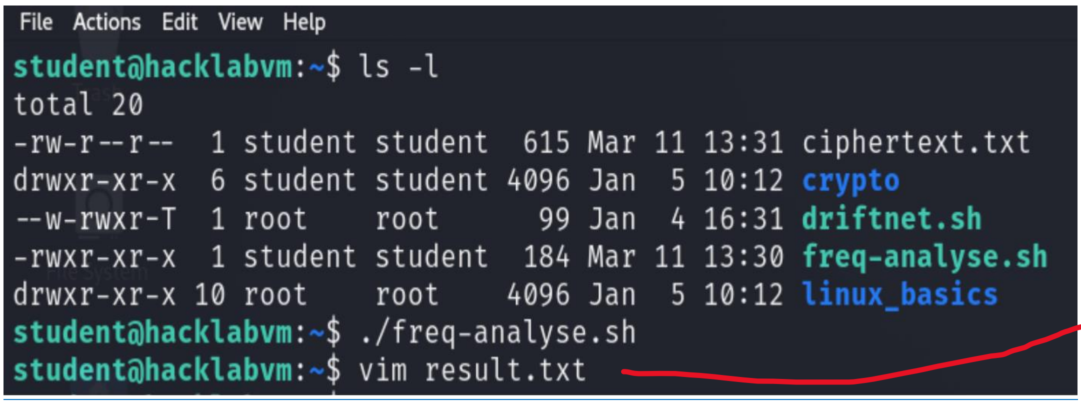
Answer: The message hidden inside of the encrypted bitmap image, was "42".

Question 4:

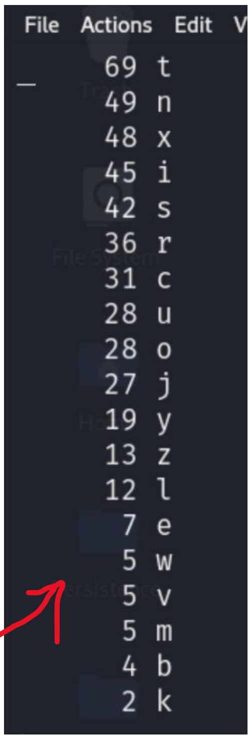
Screenshots:



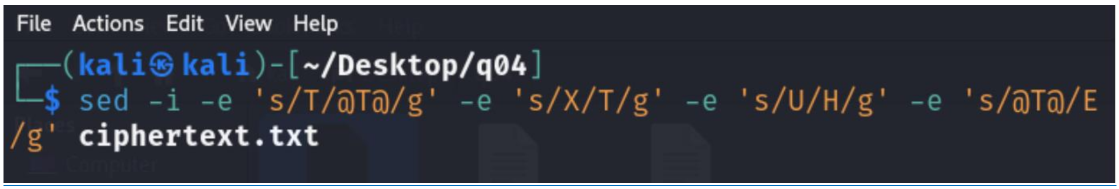
```
#!/bin/bash

moddedText=$(cat ciphertext.txt | tr '[:upper:]' '[:lower:]' | tr -d '[:space:][:punct:]')

echo "$moddedText" | \
fold -w1 | \
sort | \
uniq -c | \
sort -nr > result.txt
~
```



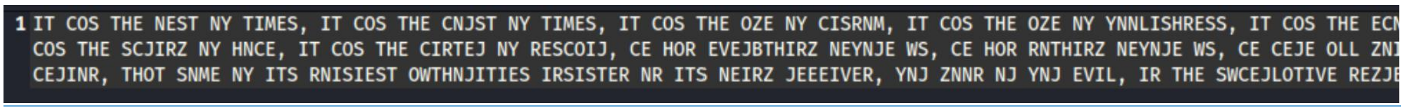
```
student@hacklabvm:~$ ls -l
total 20
-rw-r--r--   1 student student  615 Mar 11 13:31 ciphertext.txt
drwxr-xr-x   6 student student 4096 Jan  5 10:12 crypto
--w-rwxr-T   1 root    root      99 Jan  4 16:31 driftnet.sh
-rwxr-xr-x   1 student student  184 Mar 11 13:30 freq-analyse.sh
drwxr-xr-x  10 root    root    4096 Jan  5 10:12 linux_basics
student@hacklabvm:~$ ./freq-analyse.sh
student@hacklabvm:~$ vim result.txt
```



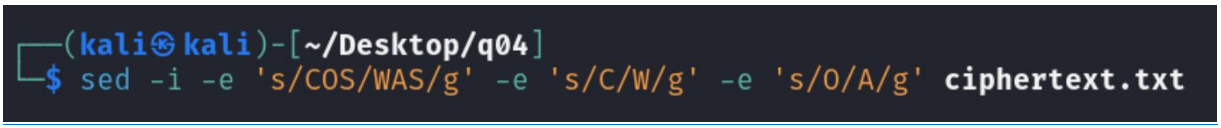
```
     69 t
     49 n
     48 x
     45 i
     42 s
     36 r
     31 c
     28 u
     28 o
     27 j
     19 y
     13 z
     12 l
      7 e
      5 w
      5 v
      5 m
      4 b
      2 k
```



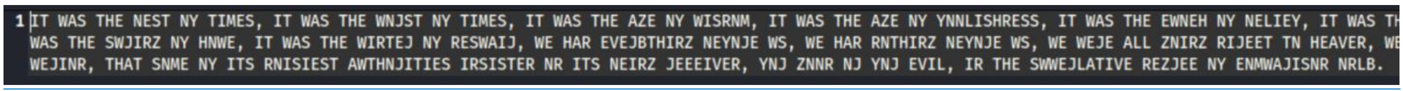
```
┌──(kali㉿kali)-[~/Desktop/q04]
└─$ sed -i -e 's/T/@T@/g' -e 's/X/T/g' -e 's/U/H/g' -e 's/@T@/E
/g' ciphertext.txt
```



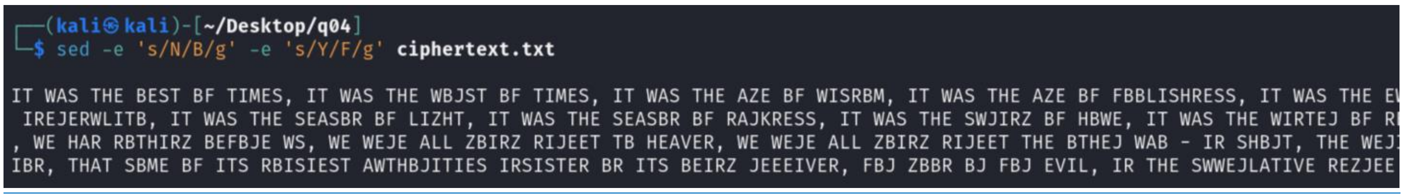
```
IT COS THE NEST NY TIMES, IT COS THE CNJST NY TIMES, IT COS THE OZE NY CISRNM, IT COS THE OZE NY YNNLISHRESS, IT COS THE ECN
COS THE SCJIRZ NY HNCE, IT COS THE CIRTEJ NY RESCOIJ, CE HOR EVEJBTHIRZ NEYNJE WS, CE HOR RNTHIRZ NEYNJE WS, CE CEJE OLL ZNI
CEJINR, THOT SNME NY ITS RNISIEST OWTHNJITIES IRSISTER NR ITS NEIRZ JEEEIVER, YNJ ZNNR NJ YNJ EVIL, IR THE SWCEJLOTIVE REZJE
```



```
┌──(kali㉿kali)-[~/Desktop/q04]
└─$ sed -i -e 's/COS/WAS/g' -e 's/C/W/g' -e 's/O/A/g' ciphertext.txt
```

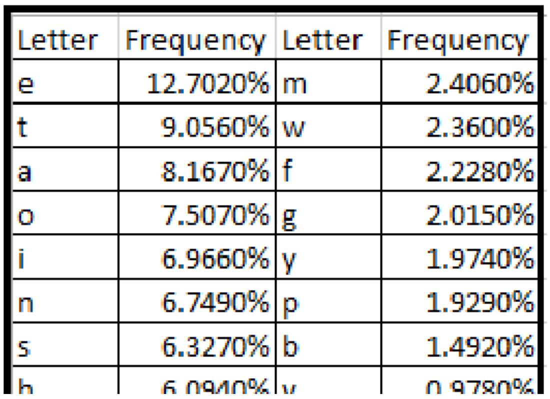


```
IT WAS THE NEST NY TIMES, IT WAS THE WNJST NY TIMES, IT WAS THE AZE NY WISRNM, IT WAS THE AZE NY YNNLISHRESS, IT WAS THE EWNEH NY NELIEY, IT WAS TH
WAS THE SWJIRZ NY HNWE, IT WAS THE WIRTEJ NY RESWAIJ, WE HAR EVEJBTHIRZ NEYNJE WS, WE HAR RNTHIRZ NEYNJE WS, WE WEJE ALL ZNIRZ RIJEET TN HEAVER, WE
WEJINR, THAT SNME NY ITS RNISIEST AWTHNJITIES IRSISTER NR ITS NEIRZ JEEEIVER, YNJ ZNNR NJ YNJ EVIL, IR THE SWWEJLATIVE REZJEE NY ENMWAJISNR NRLB.
```



```
┌──(kali㉿kali)-[~/Desktop/q04]
└─$ sed -e 's/N/B/g' -e 's/Y/F/g' ciphertext.txt

IT WAS THE BEST BF TIMES, IT WAS THE WBJST BF TIMES, IT WAS THE AZE BF WISRBM, IT WAS THE AZE BF FBBLISHRESS, IT WAS THE E
 IREJERWLITB, IT WAS THE SEASBR BF LIZHT, IT WAS THE SEASBR BF RAJKRESS, IT WAS THE SWJIRZ BF HBWE, IT WAS THE WIRTEJ BF R
, WE HAR RBTHIRZ BEFBJE WS, WE WEJE ALL ZBIRZ RIJEET TB HEAVER, WE WEJE ALL ZBIRZ RIJEET THE BTHEJ WAB - IR SHBJT, THE WEJ
IBR, THAT SBME BF ITS RBISIEST AWTHBJITIES IRSISTER BR ITS BEIRZ JEEEIVER, FBJ ZBBR BJ FBJ EVIL, IR THE SWWEJLATIVE REZJEE
```

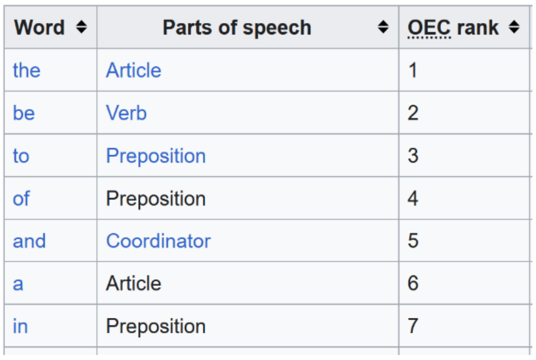
Explanation: I started off with the idea, that this was not just a shift cipher but instead a substitution cipher. One in which every letter, was substituted with another. As shown in the 1st image, I wrote a bash script to do a frequency analysis of the letters. It first converts UPPERCASE to lowercase, and then removes spaces and punctuation. The next commands separate the characters into their own lines, sorts them alphabetically, counts them with "uniq", and then displays them in descending order of appearances. Clearly, the most common letters are t, n, x, i, s, etc.



Letter	Frequency	Letter	Frequency
e	12.7020%	m	2.4060%
t	9.0560%	w	2.3600%
a	8.1670%	f	2.2280%
o	7.5070%	g	2.0150%
i	6.9660%	y	1.9740%
n	6.7490%	p	1.9290%
s	6.3270%	b	1.4920%
h	6.0940%	v	0.9780%

With this information, I looked up the most common letters in English words. My goal was to substitute the ciphertext's most common letters, with the most common letters in English. I tried this a few times, with some different arrangements of the letters shown here on the left. This did not work and produced incomprehensible results. The ciphertext remained unreadable.

After failing to substitute the letters, I decided to look at words instead. Perhaps by making educated guesses, I could get closer. I saw that the word "the" was the most common 3 letter one in English. Using this information, I substituted XUT with THE, as XUT appeared (from visual inspection) to be the most common

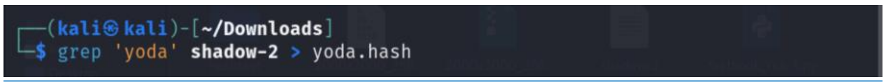


Word	Parts of speech	OEC rank
the	Article	1
be	Verb	2
to	Preposition	3
of	Preposition	4
and	Coordinator	5
a	Article	6
in	Preposition	7

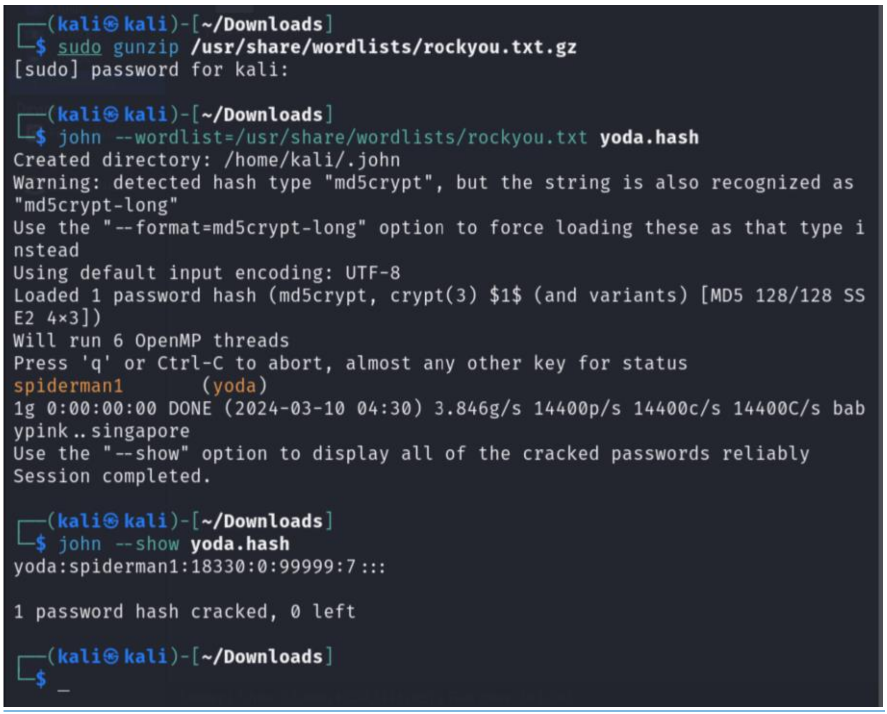
3-letter word. As shown in the screenshots section, I did this replacement with the "sed" command and switched specific letters. X with T, U with H, and T with E. As an educated guess, I also figured that COS, likely translated to WAS. So, I switched C with W, O with A, and looked at the output. By now, it was very clear to me that the decrypted passage, was the quote from A Tale of Two Cities. Although there were some letter replacement conflicts, meaning that I couldn't replace some letters without modifying words that were already done, I assumed I had the answer. The full quote is shown below.

Answer: "IT WAS THE BEST OF TIMES, IT WAS THE WORST OF TIMES, IT WAS THE AGE OF WISDOM, IT WAS THE AGE OF FOOLISHNESS, IT WAS THE EPOCH OF BELIEF, IT WAS THE EPOCH OF INCREDULITY, IT WAS THE SEASON OF LIGHT, IT WAS THE SEASON OF DARKNESS, IT WAS THE SPRING OF HOPE, IT WAS THE WINTER OF DESPAIR, WE HAD EVERYTHING BEFORE US, WE HAD NOTHING BEFORE US, WE WERE ALL GOING DIRECT TO HEAVEN, WE WERE ALL GOING DIRECT THE OTHER WAY – IN SHORT, THE PERIOD WAS SO FAR LIKE THE PRESENT PERIOD, THAT SOME OF ITS NOISIEST AUTHORITIES INSISTED ON ITS BEING RECEIVED, FOR GOOD OR FOR EVIL, IN THE SUPERLATIVE DEGREE OF COMPARISON ONLY."

Question 5:



```
┌──(kali㉿kali)-[~/Downloads]
└─$ grep 'yoda' shadow-2 > yoda.hash
```



```
┌──(kali㉿kali)-[~/Downloads]
└─$ sudo gunzip /usr/share/wordlists/rockyou.txt.gz
[sudo] password for kali:

┌──(kali㉿kali)-[~/Downloads]
└─$ john --wordlist=/usr/share/wordlists/rockyou.txt yoda.hash
Created directory: /home/kali/.john
Warning: detected hash type "md5crypt", but the string is also recognized as
"md5crypt-long"
Use the "--format=md5crypt-long" option to force loading these as that type i
nstead
Using default input encoding: UTF-8
Loaded 1 password hash (md5crypt, crypt(3) $1$ (and variants) [MD5 128/128 SS
E2 4×3])
Will run 6 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
spiderman1       (yoda)
1g 0:00:00:00 DONE (2024-03-10 04:30) 3.846g/s 14400p/s 14400c/s 14400C/s bab
ypink..singapore
Use the "--show" option to display all of the cracked passwords reliably
Session completed.

┌──(kali㉿kali)-[~/Downloads]
└─$ john --show yoda.hash
yoda:spiderman1:18330:0:99999:7:::

1 password hash cracked, 0 left

┌──(kali㉿kali)-[~/Downloads]
└─$ _
```

Explanation: My process to crack the password is shown above. The first thing I did, was to isolate the "yoda" user's hash from the shadow file and put it into its own file. This was done with: grep 'yoda' shadow-2 > yoda.hash

I then noticed that the rockyou wordlist was compressed by default, so I used: sudo gunzip /usr/share/wordlists/rockyou.txt.gz to uncompress it.

Now that I had both the .hash file and the wordlist, I could use John the Ripper (a tool installed on Kali). This was the command I entered:

```
john – -worldlist=/usr/share/wordlists/rockyou.txt yoda.hash
```

Although it produced a warning at first, it seemed that the tool was still able to crack the hash, and find the password. As shown by the orange text (and as shown by the output of the command: john - -show yoda.hash), the password for the ‘yoda’ user was spiderman1.

Answer: The login to the user account yoda, is spiderman1