Assignment 1

Course: CISC 870 (Cryptography)
Term: Fall 2022

Submitted by:

Kazi Amit Hasan
ID: 20341105
MSc (research-based) Student, School of Computing
Queen's University

Lab Setup: For this assignment, I have installed and used SEED Ubuntu 20.4 on VirtualBox on MacBook Pro (2017 Mid). I have followed the instructions given on the website. [1] [2].

I followed the first approach of using a pre-built SEED VM. Additionally, I have allocated the memory size of 2GB instead of the usual recommendation for better performance of the Virtual machine. Also, we set the video memory to 64MB and increased the number of CPUs to 2 from 1 to increase performance.

Task 1: Generate Encryption Key in a Wrong Way

Approach: To understand the observations in a better way, each code was run multiple times. This approach helped me to understand both of the codes.

Observations before commenting out Line 1: Time and the random number change every time when I run the ./task1 command. The output is shown in Figure 1.1.
Observations after commenting out Line 1: Time changes on each execution but the random number is not changing. The output is shown in Figure 1.2

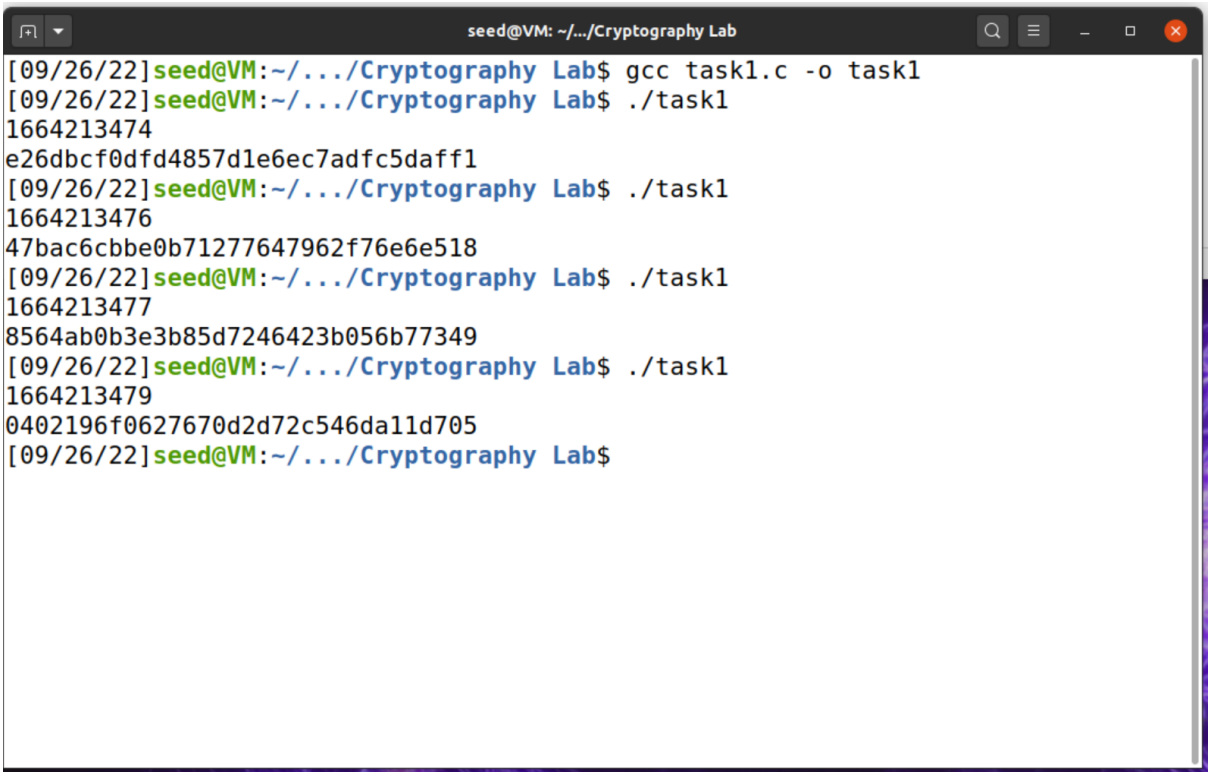

Figure 1.1: Visual representation of terminal before commenting out

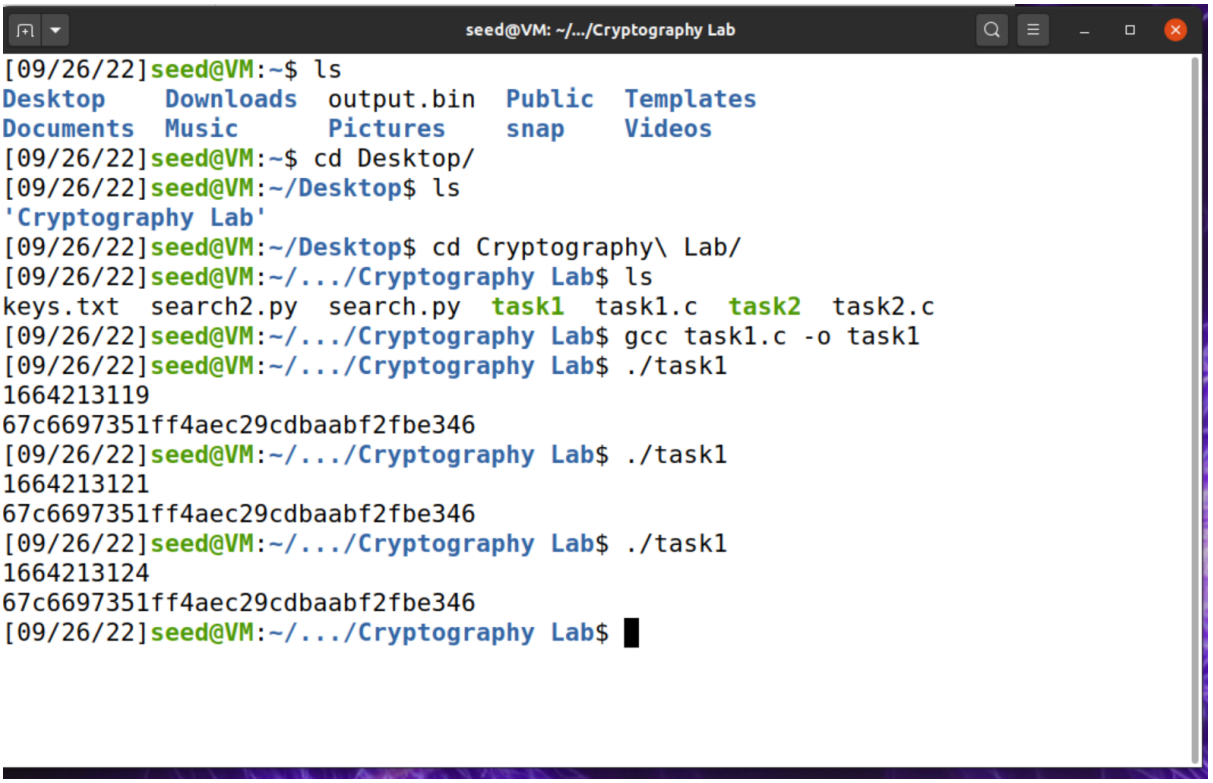

Figure 1.2: Visual representation of terminal after commenting out

Purpose of srand() & time() functions in both observations:

Like the function name, the rand() function is responsible for generating random numbers. [5] The extra 's' with rand() function added new functionality with it. The main purpose of srand() function is to use the computer's current time as seed in order to generate random numbers. And the time() represents the time (in seconds) since 00:00:00 on 1/1/1970. [3]

Here in the code, 'srand(time(NULL));'. It outputs unique random numbers in each execution as it uses the computer's current time as seed, which is unique in every execution.

For example, The '1664213479' represents the time in seconds. This shows the time since 00:00:00 on 1/1/1970. According to the figure, it is clear that the generated numbers are random and unique.

If I comment out the code, the generated random number is the same each time. I tried this with multiple executions, but the result is always the same. The reason behind this 'same' random number is that there is no seed set in the program.

Task 2: Guessing the key

Approach & Observations: I have divided the task into several steps. The steps are described below:

1. Firstly, I generated the seed range. As per the question, the timestamp of the encrypted file was "2018-04-17 23:08:49"' and Bob guessed that it might be generated within a 2 hours time frame. So, using "date -d "2018-04-17 23:08:49" +%s" and "date -d "2018-04-17 21:08:49" +%s" commands, I generated the range. The figure is shown in Fig 2.1
2. Secondly, I used those seeds and generated keys.txt file with the help of the program of task 1. Figures 2.2 and 2.3 are representing programs and keys respectively.
3. For this part, I went with the modularized approach. In the utils folder, I kept the ciphertext, plaintext, and IV in a credential file. Also, I kept the read_keys function aside. Figure 2.4 is representing the program.
4. The output of 'search.py' is shown in Figure 2.5.
5. The structure of the utils folder is shown in Figure 2.6. Also, Figure 2.7, figure 2.8 and figure 2.9 are representing the functions inside the utils folder.
6. In this assignment, the __init__.py function is called as an empty constructor. Screenshot addded in figure 2.9.

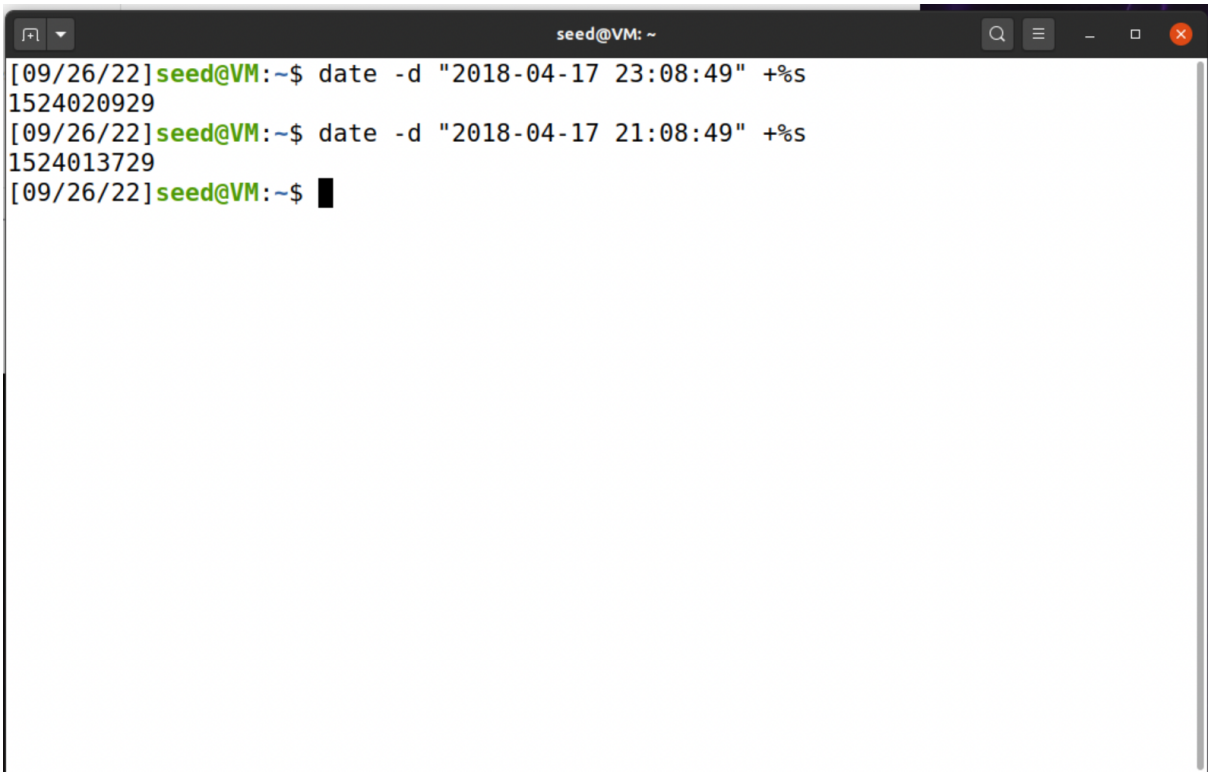

Figure 2.1: Range of seeds.

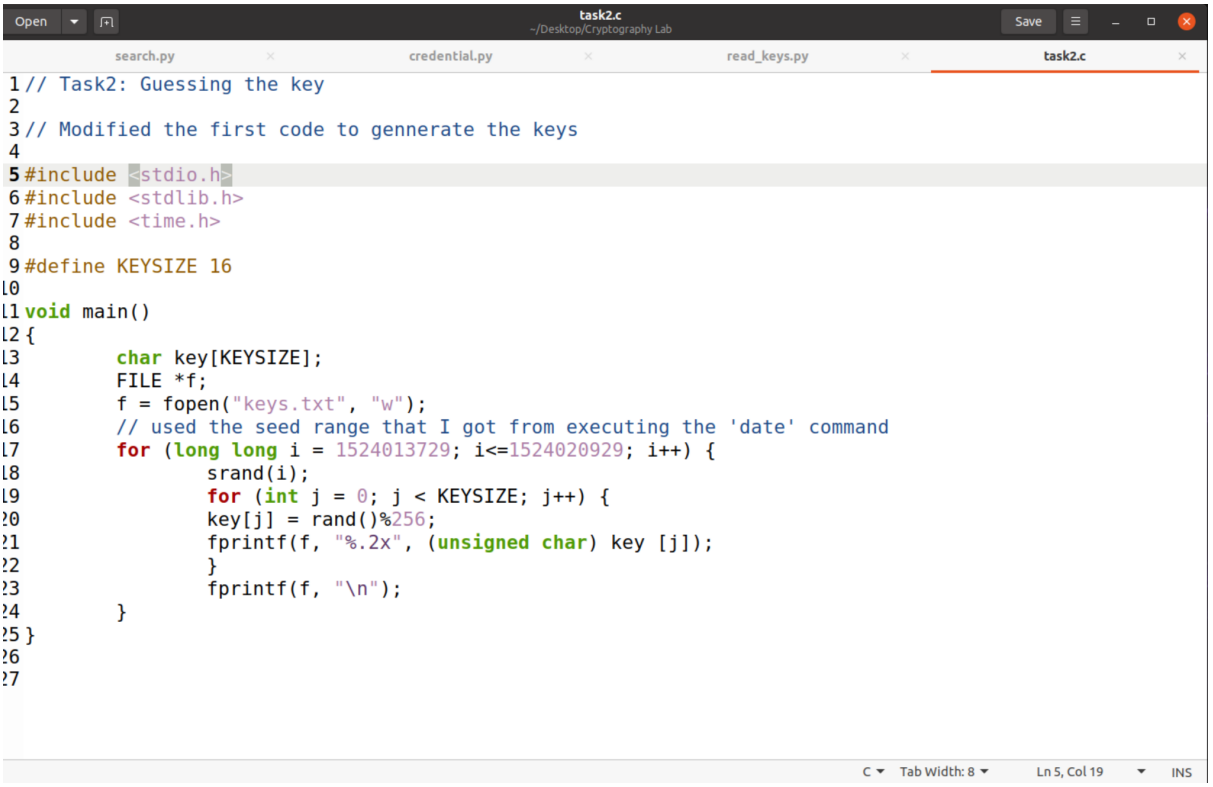

Figure 2.2: Program to generate the keys.

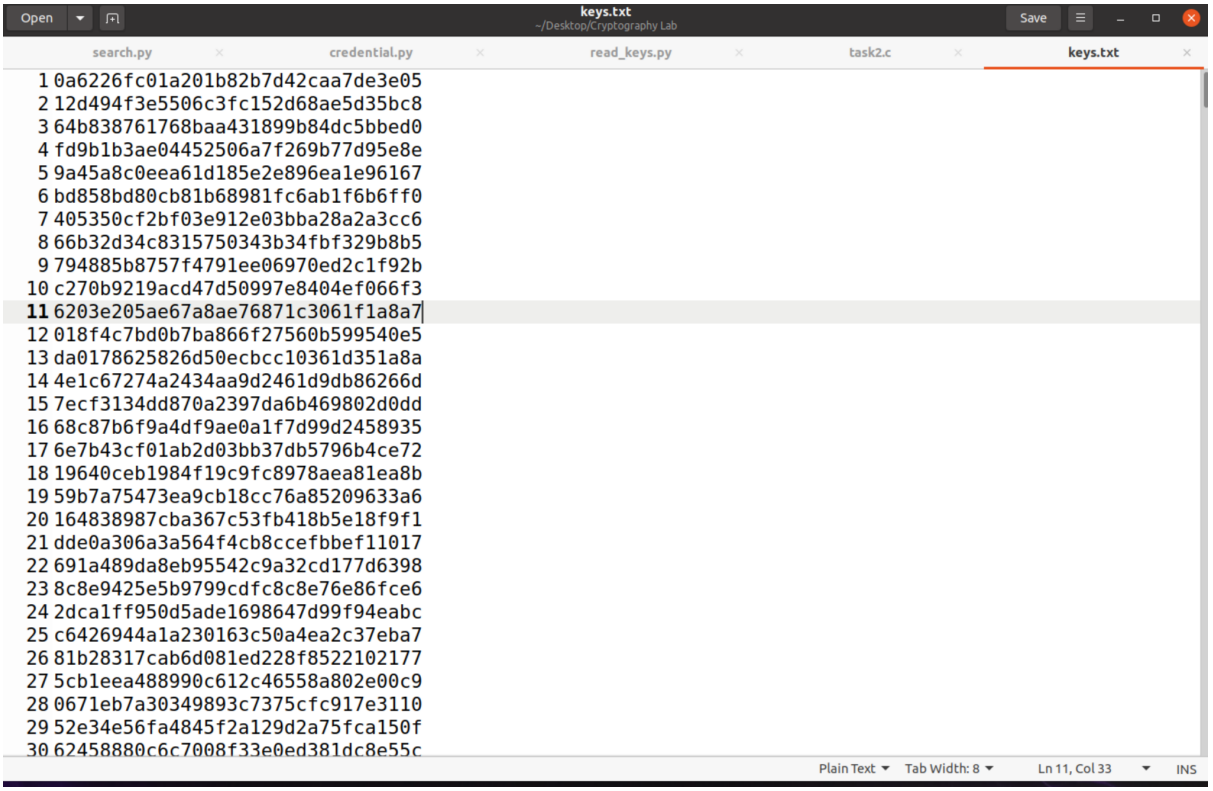

Figure 2.3: Visual representation of keys.txt.

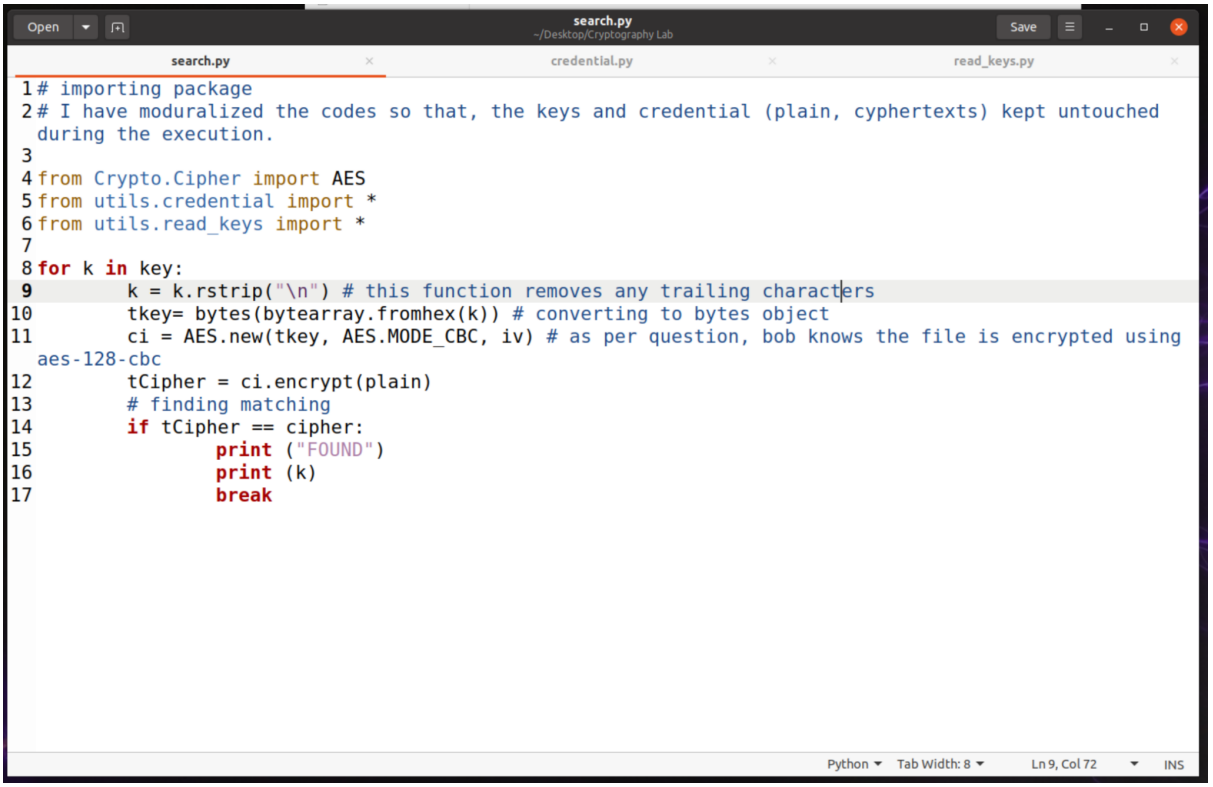

Figure 2.4: Program to find the correct key.

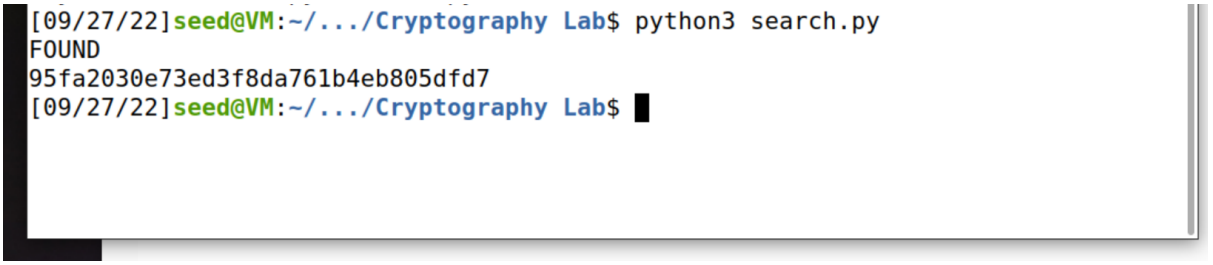

Figure 2.5: Output of search.py

Utils function:

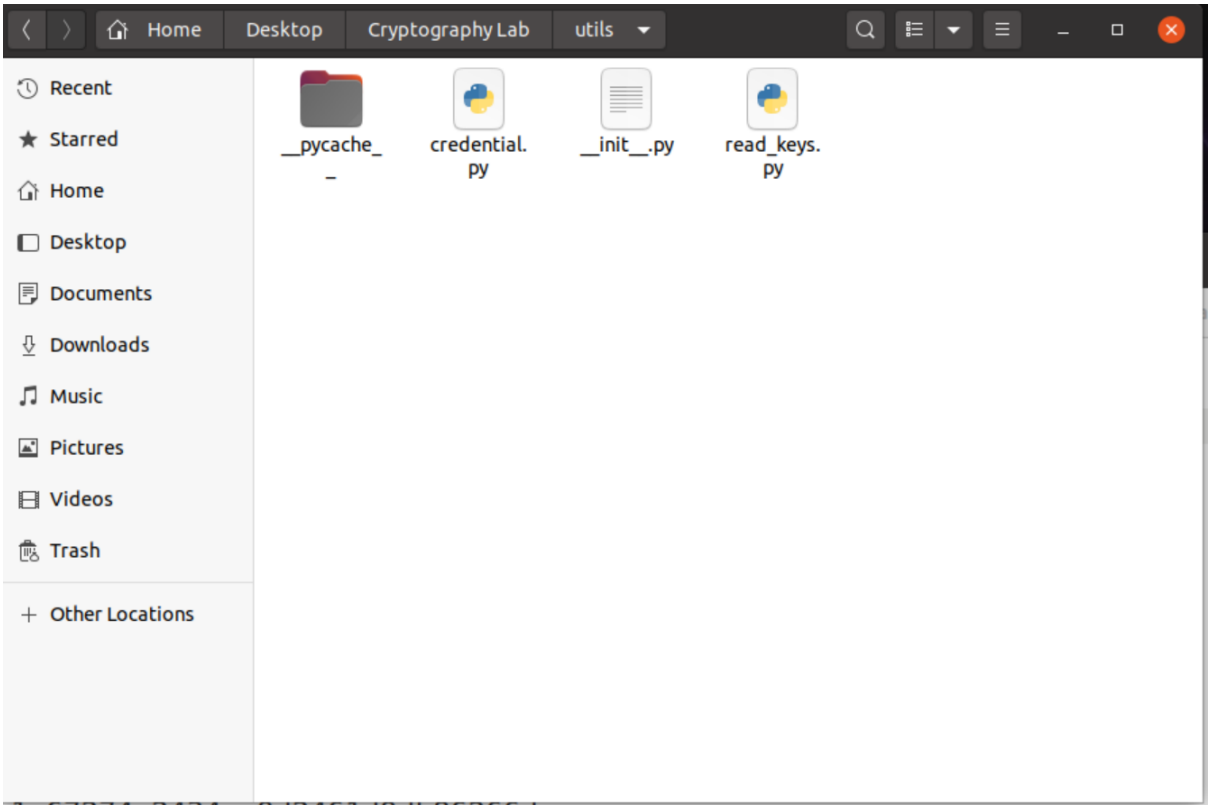

Figure 2.6: Structure of utils folder

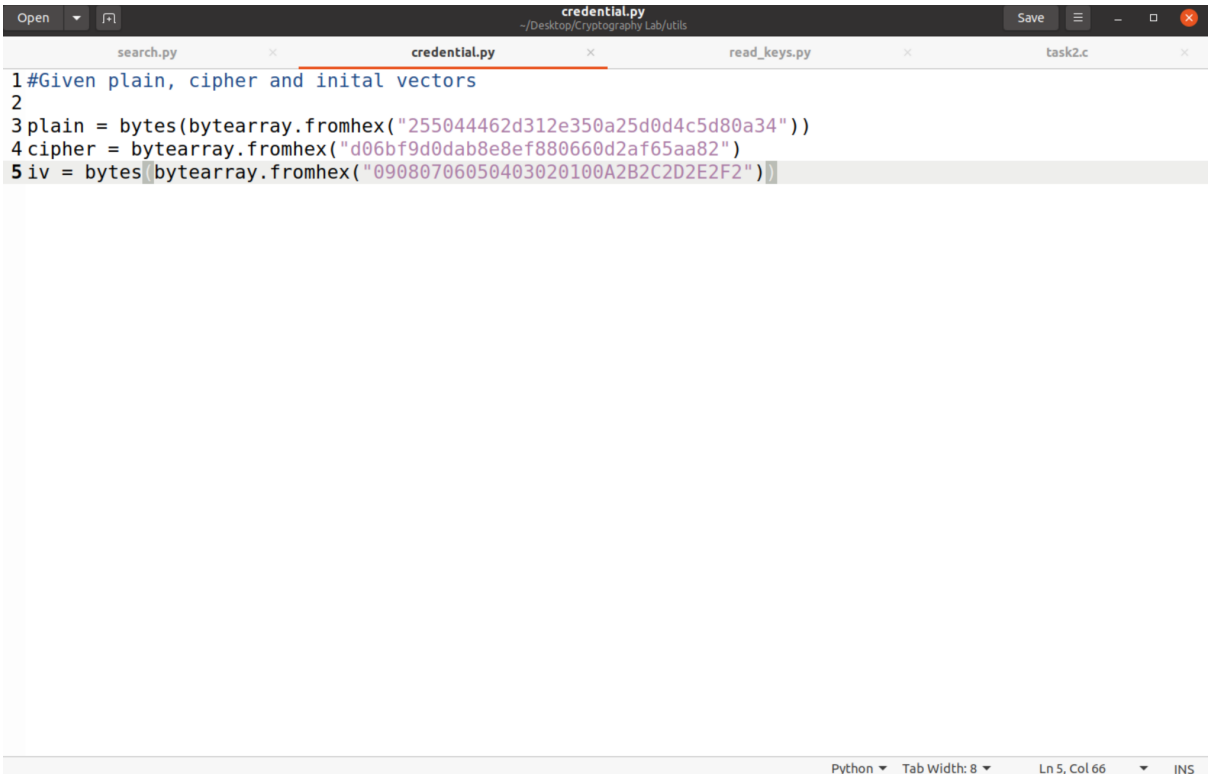

Figure 2.7: Given credentials in the utils folder

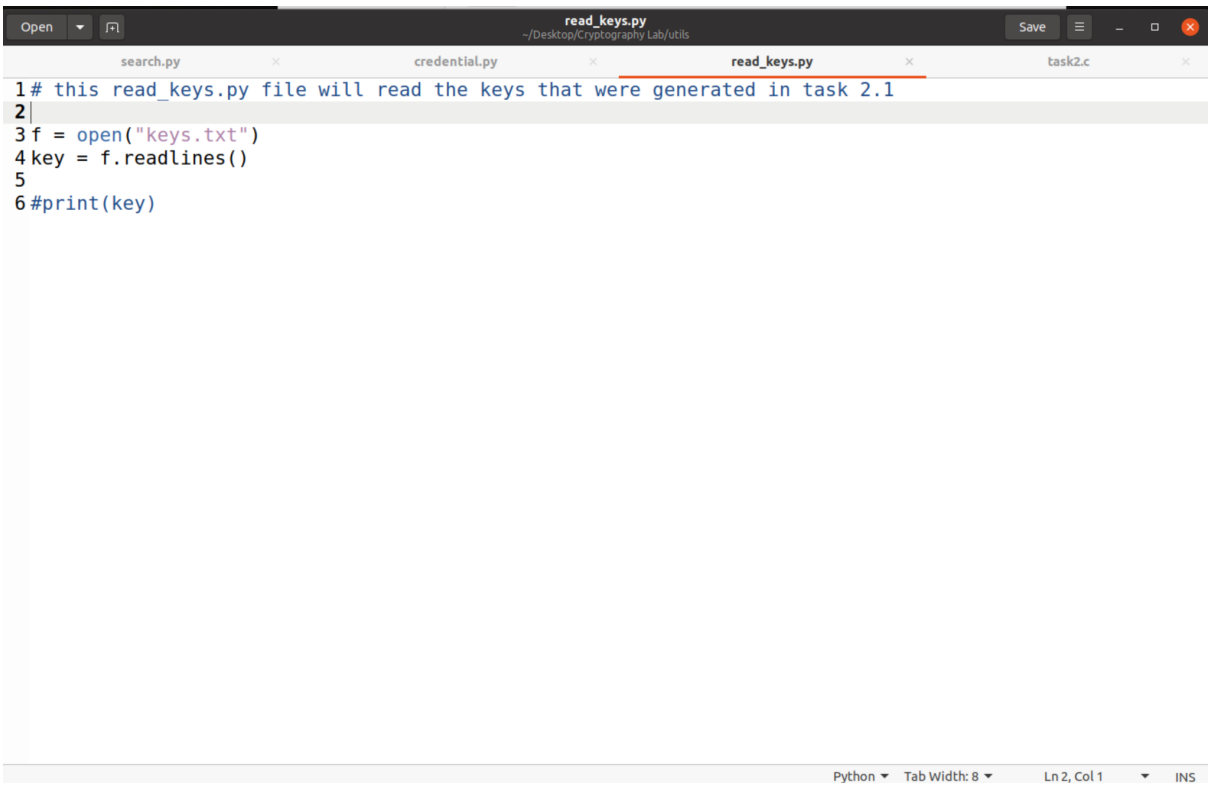

Figure 2.8: read keys in the utils folder

Figure 2.9: Representation of __init__.py file

Task 3: Measure the Entropy of Kernel

Approach & Observations: For this task, firstly, I run the '$ cat /proc/sys/kernel/random/entropy_avail' command to check the current entropy. Then, I added the 'watch' command to it and observed the changes in entropy. The final command was 'watch -n .1 cat /proc/sys/kernel/random/entropy_avail'.
Figure 3.1 shows the current entropy of the system which is 2605. Then I executed the command with 'watch' and observed the changes. The figure 3.2 represents the entropy while running the 'watch -n .1 cat /proc/sys/kernel/random/entropy_avail' command only. Figure 3.3 shows the output of entropy while moving the mouse, or typing something. It was observed that the entropy increased while typing something really fast. Figure 3.4 shows that the entropy also increased while reading a file. Also, I saw a sudden increase in entropy while visiting a website, figure 3.5 represents the increase. After executing all these tasks, I encountered that reading large files and visiting websites increase the entropy of the system significantly.

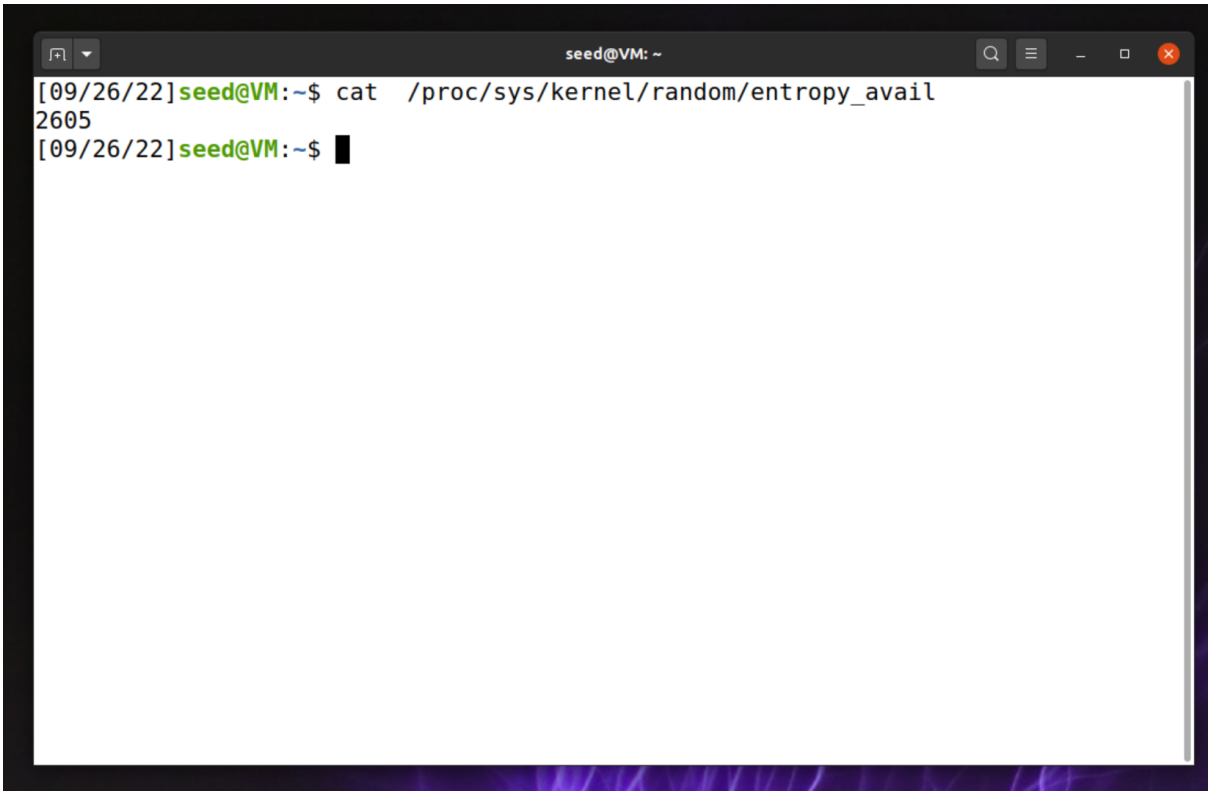

Figure 3.1: Current entropy

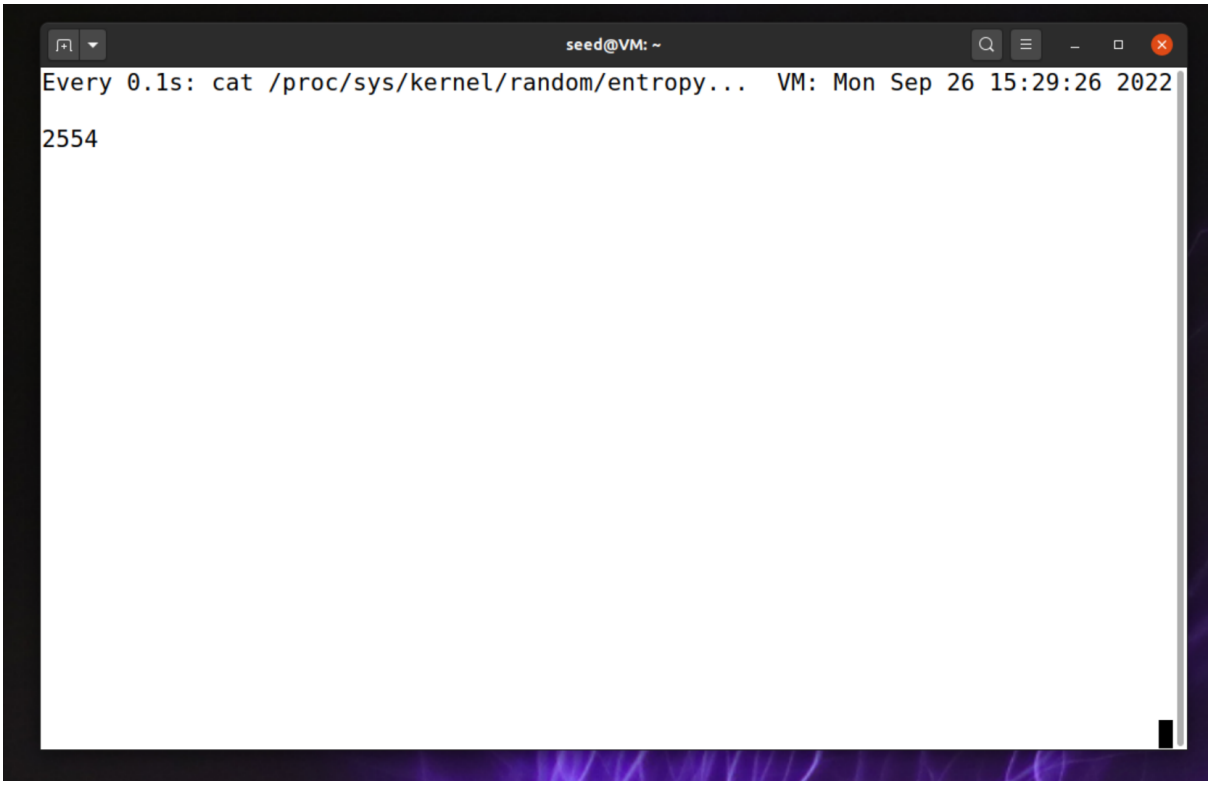

Figure 3.2: Change in entropy while keeping everything as it is.

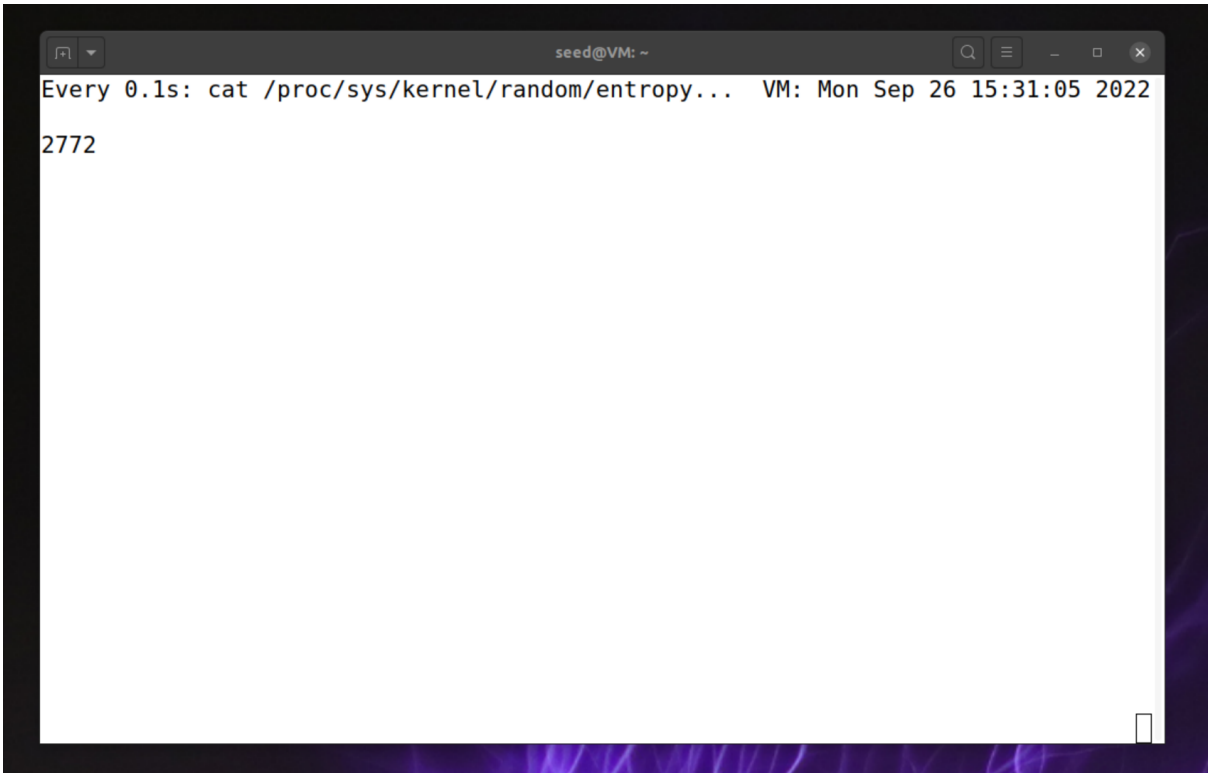

Figure 3.3: Change in entropy while moving mouse, typing something

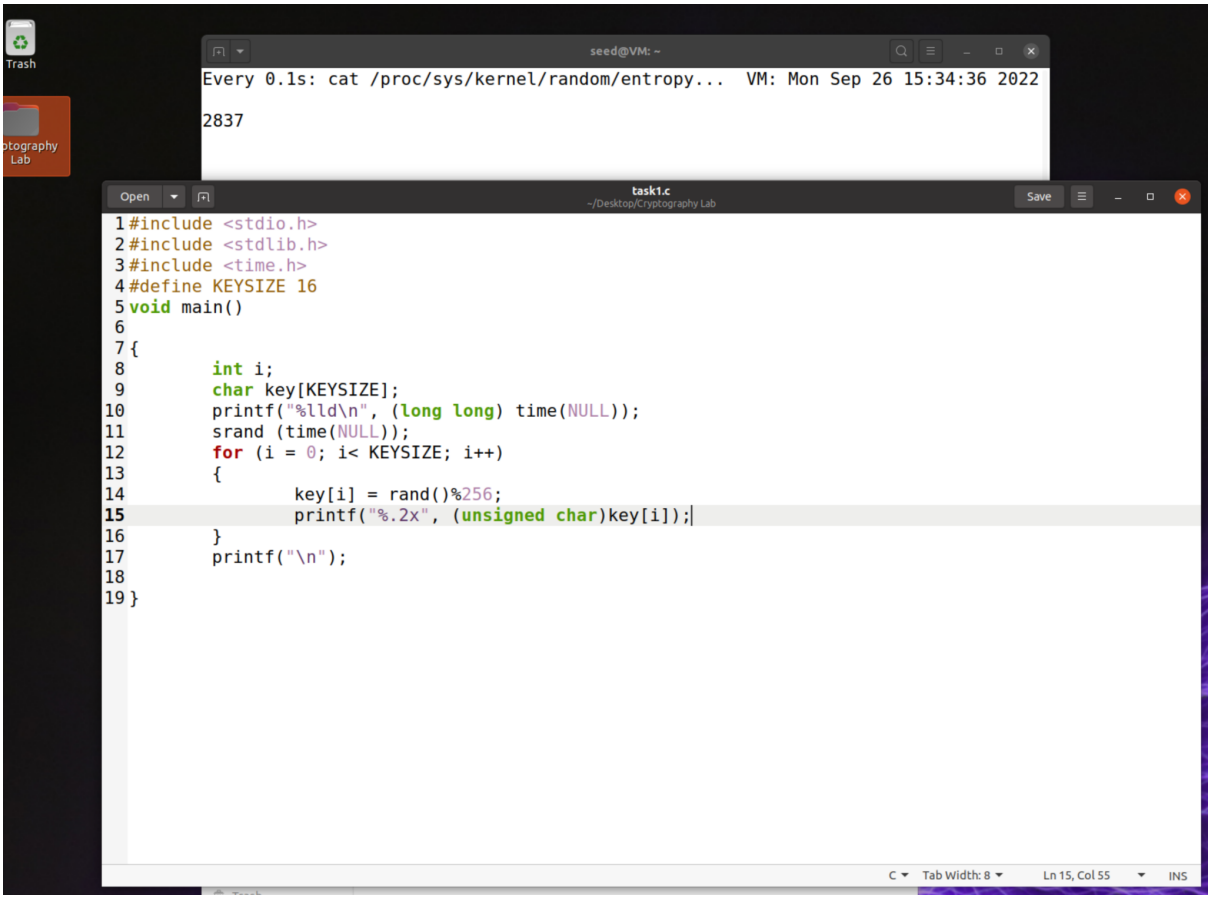

Figure 3.4: Change in entropy while reading a file

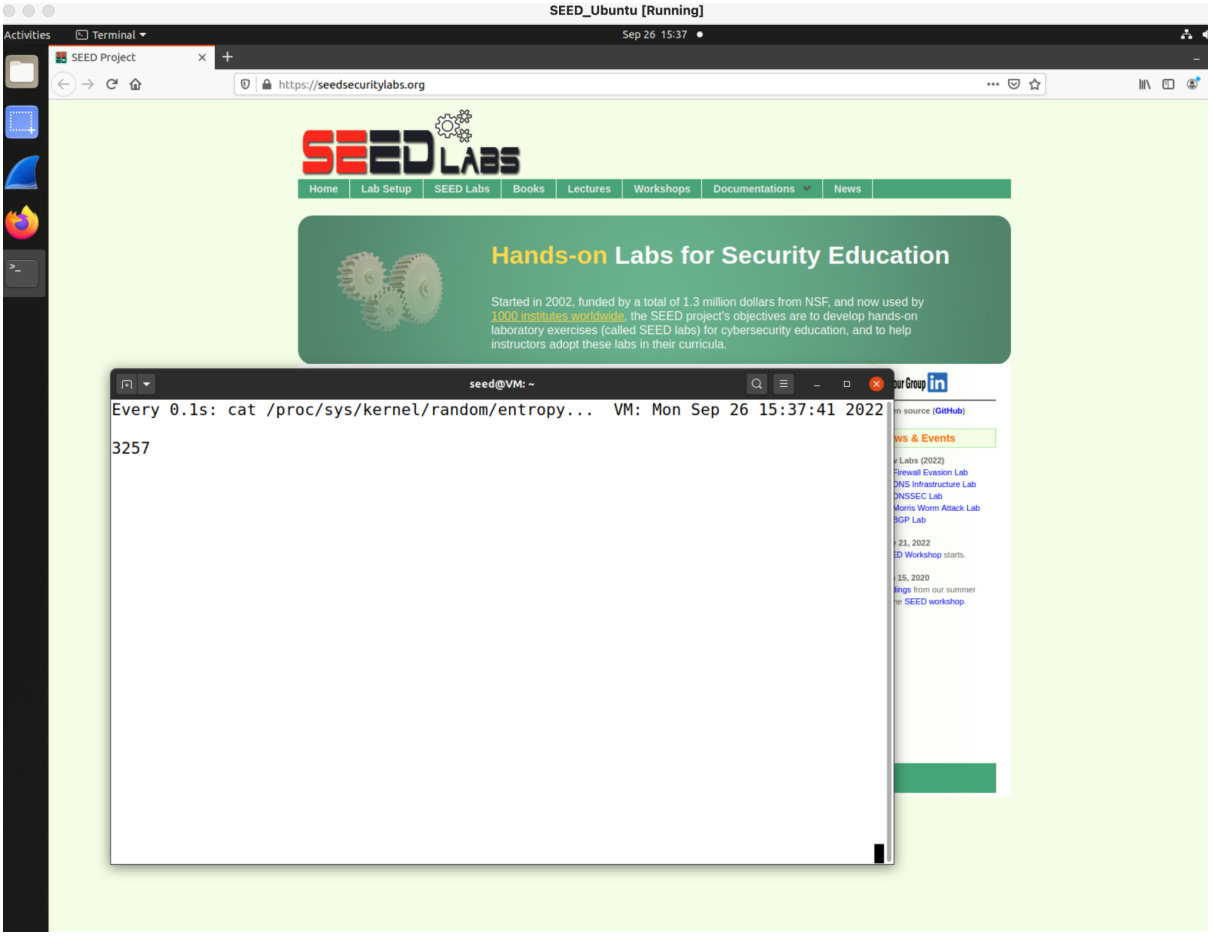

Figure: 3.5: Change in entropy while visiting a website

Task 4: Get Pseudo Random Numbers from /dev/random

Approach: For this task, I executed the command 'cat /dev/random | hexdump' and observed the entropy at the same time by using the "watch -n .1 cat /proc/sys/kernel/random/entropy_avail" command. There are some interesting observations while doing this task.

Observations:

1. **Do not move mouse/type anything:** If I do not move mouse or type anything, the change in entropy is comparatively slower than other tasks. Eventually, it starting coming to lower side gradually. I observed close to '1' a few times. Also, the generation of numbers is really really slow. Even this could be compared to no output. The visuals are added in figure 4.1. The figure might seem confusing. But the screenshot was taken at the initial time, the entropy started dropping after some moment.
2. **Randomly moving mouse:** When I randomly moved the mouse, I observed a good speed in changing the entropy in the terminal. Also, the generation of numbers is

really faster than the previously ‘not moving’ mouse. The visuals are added in figure 4.2

Explanations and learnings:
As it is described in the task, /dev/random device is a blocking device. As the entropy drops, the speed of random number generation also drops. Thus, when the entropy becomes zero, the /dev/random command will stop/pause some functionalities until it achieves some randomness. [6]

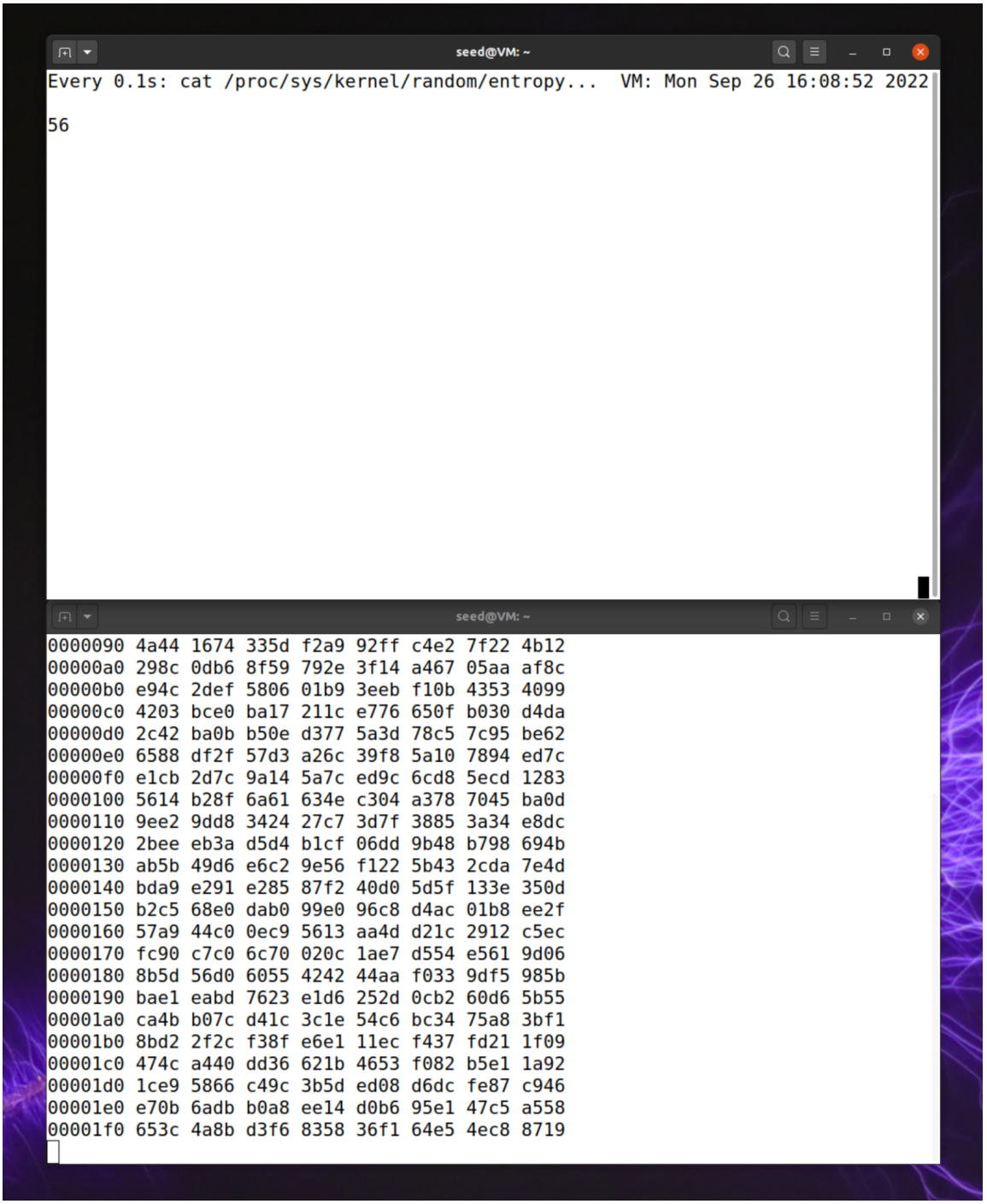

Figure 4.1: Do not moving mouse/type anything

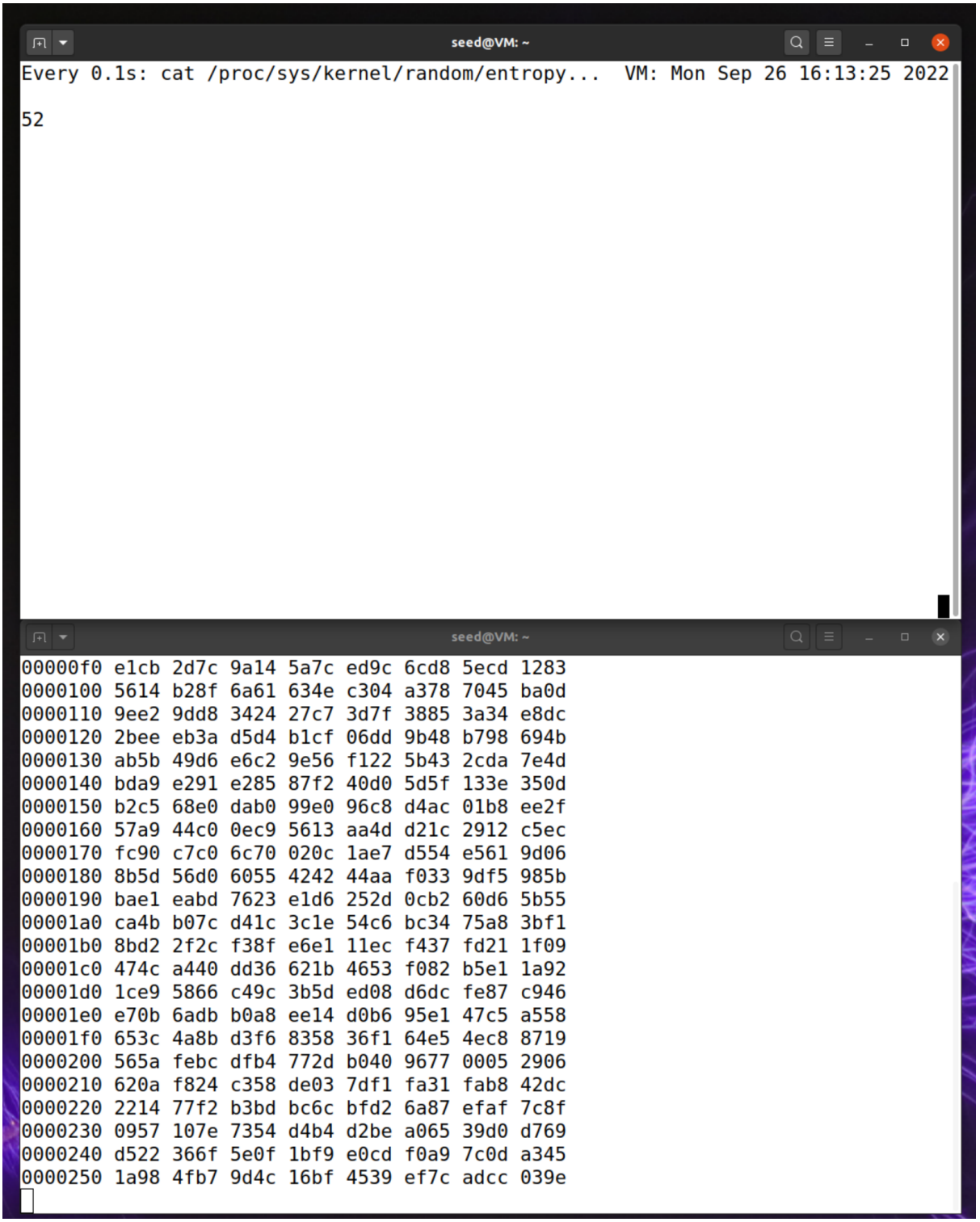

Figure 4.2: Randomly moving mouse:

Question: If a server uses /dev/random to generate the random session key with a client. Please describe how you can launch a Denial-Of-Service (DOS) attack on such a server.
Answer: As we know, /dev/random device is a blocking device, the entropy will drop gradually and will pause functionalities if there's not enough entropy found. At the time of such situations, a Denial of Service (DOS) attack could launch by sending request. [8]

Task 5: Get Random Numbers from /dev/urandom

Approach:

1. The first task was to observe the behavior of "cat /dev/urandom | hexdump". For this, I executed this command on the terminal. The observation is described in a later section.
2. The next task was to evaluate the quality of random numbers. For this, I executed two commands "head -c 1M /dev/urandom > output.bin" and "ent output.bin" respectively. After running these, I got some interesting parameters to explore. The observations are described in a later section.
3. The last task was to generate a 256-bit encryption key. For this, I added and modified the code of task 2 and added it in figure 5.3 and the output is also shown in figure 5.4. The code snipped was already provided in the instructions.

Observations:

1. **After ruining cat/dev/urandom**, it is shown that the terminal was generating random numbers constantly at an excellent increased speed. I moved the mouse and typed, but it has no impact on number generation. The numbers were genenrating rapidly.
2. **Measuring quality of random number**: Figure 5.2 represents the outcome of 1 MB of pseudo-random number from /dev/urandom. Here, each outcome has its own significant meaning. [7]
    a. **Entropy**: 7.999842 bit per byte. This represents the density of information. The achieved score of 7.999842 is considered a very better score. [7]
    b. **Optimum Compression**: As per Figure 5.2, the optimum compression is 0%. This reveals that the generated data is random. [7]
    c. **Chi-square distribution:** According to Figure 5.2, 87.59% was achieved. [7]
    d. **Arithmetic means:** I got the arithmetic mean of 127.65 which is close to 127.5. [7]
    e. **Monte Carlo value of pi:** I got the value of 3.122848422 with a 0.1% error. This refers to a good outcome. [7]
    f. **Serial correlation coefficient:** 0.000318 and it's close to 0.

        After observing these parameters, we can say that the quality of random number is good.
3. **Generating 256-bit encryption key:** As the code snipped was already given, it was given 128 bits, so I had to generate 256 bits. So I changed the length to 32 from 16.

Figure 5.1: After running cat/dev/urandom.

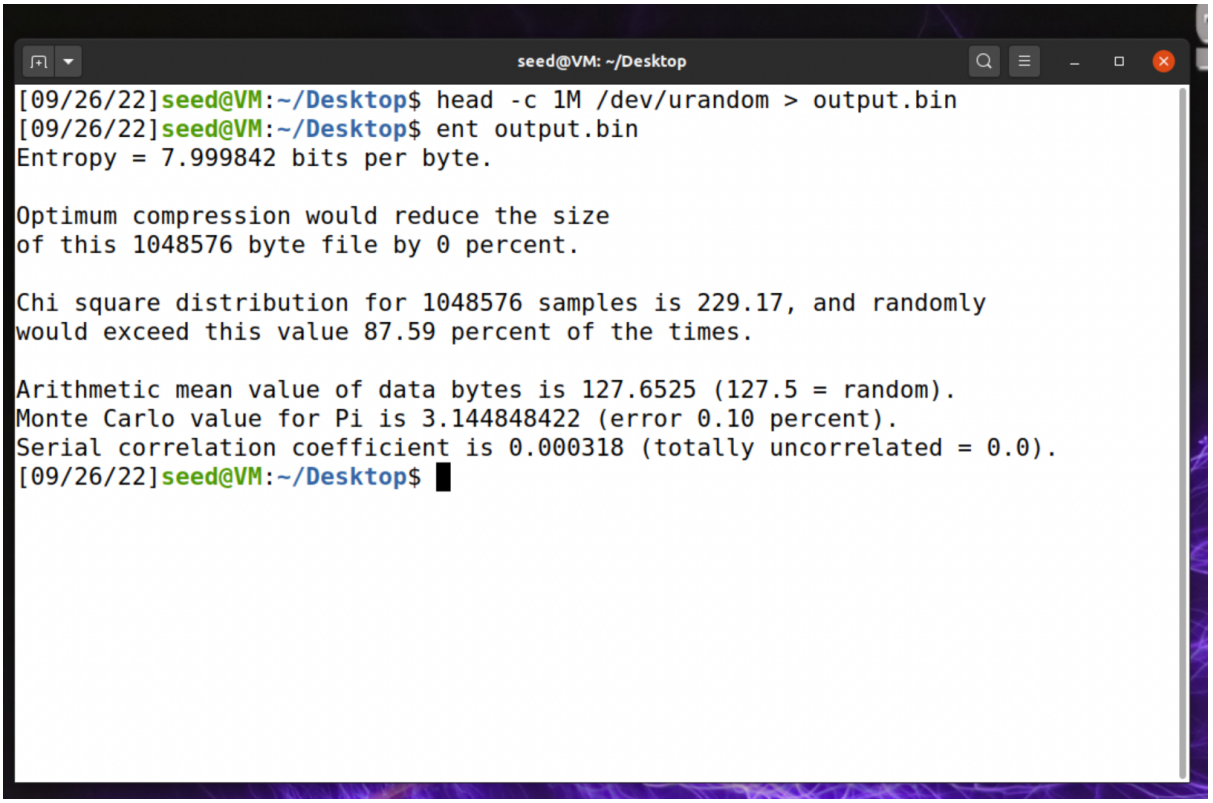

Figure 5.2: Measuring entropy of 1MB of pseudo-random number

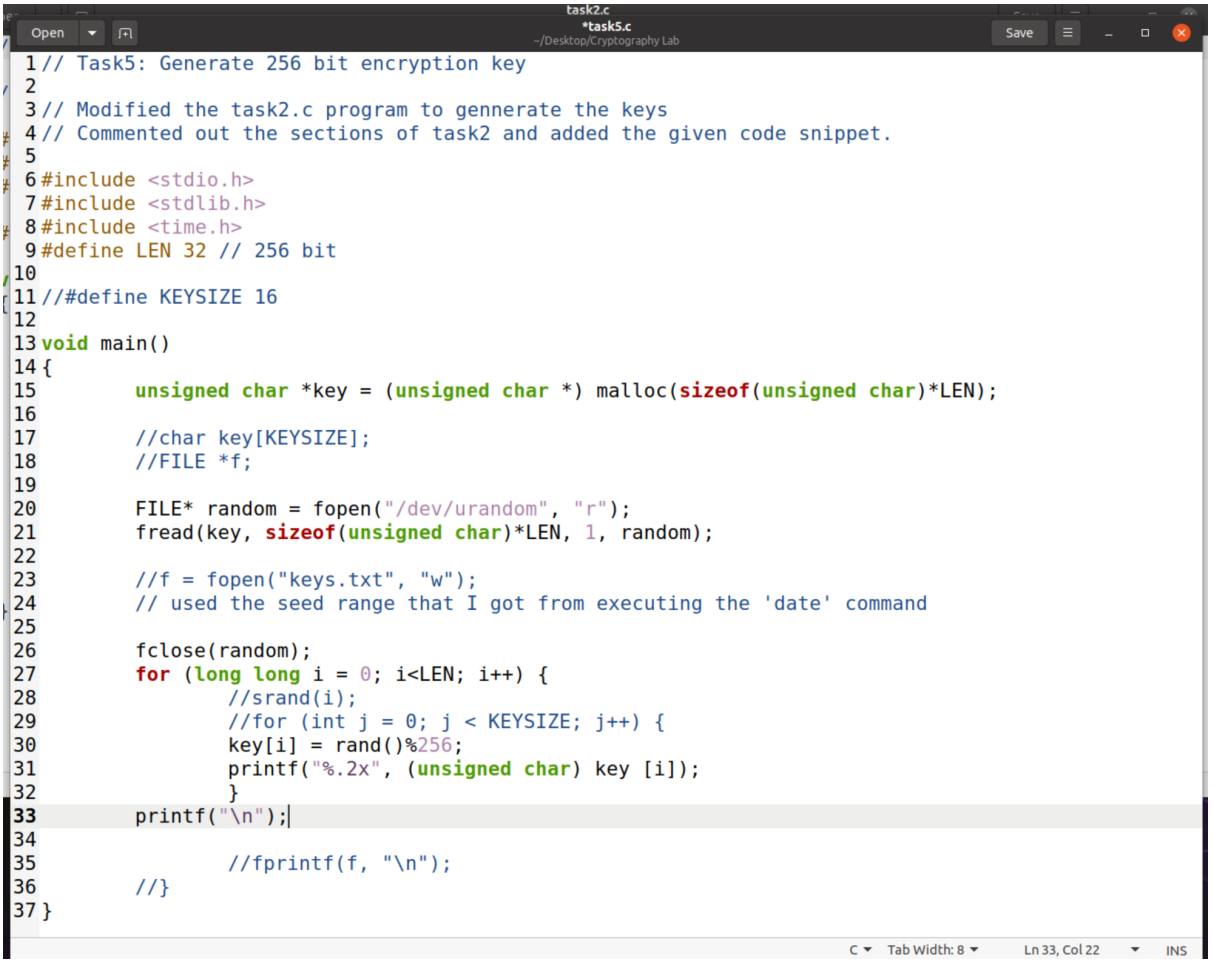

Figure 5.3: Generating 256 bit encryption key

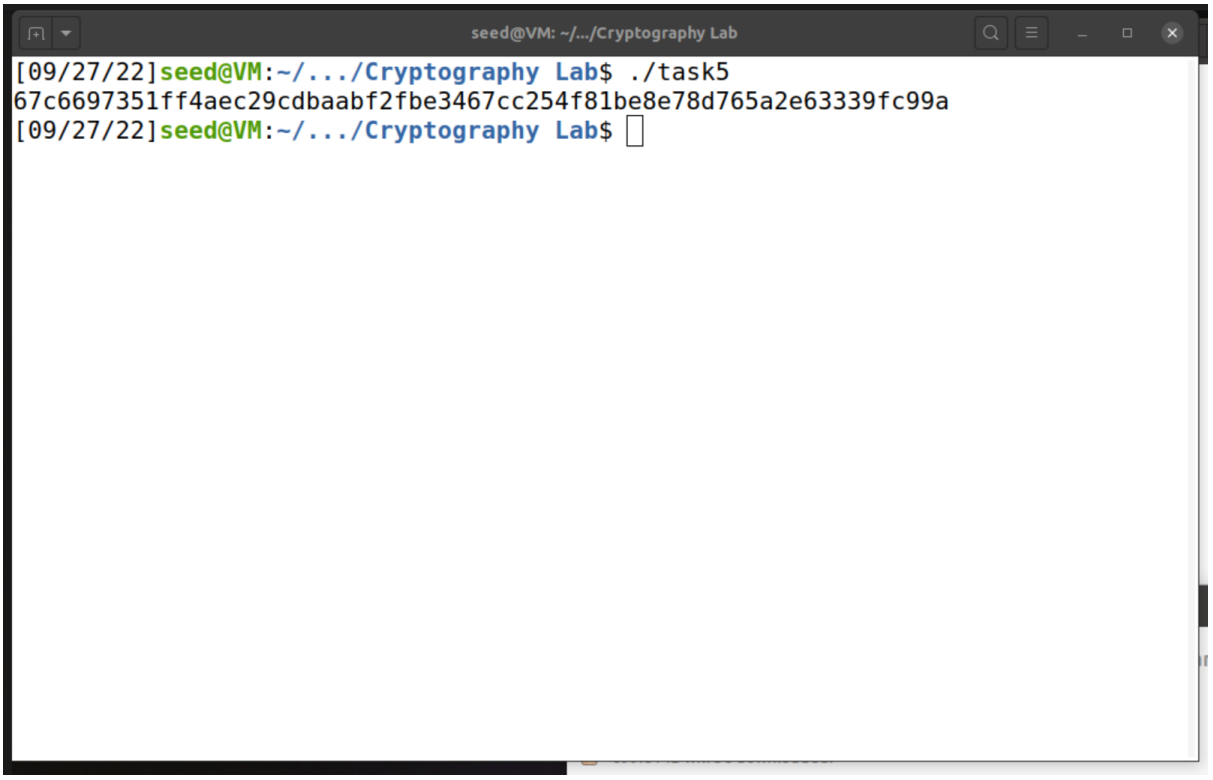

Figure 5.4: The output of 256-bit generated key.

Conclusion:

In this assignment, I learned about pseudo-random number generation. Generating encryption keys random/urandom devices. Some outputs were really interesting and were fun to explore and observe. I had spent a good amount of time installing VM and Ubuntu 20.4 on my personal machine. In the first 3 attempts, I got errors and faced a lot of problems. Then for the 4th time I reinstalled Ubuntu 20.04 on VM with some extra cores and allocated some more space. Still, the machine felt a bit slow while doing the assignments. However, this assignment was really informative and interesting.

References:

1. (Available online) https://seedsecuritylabs.org/labsetup.html
2. (Available online) https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md
3. (Available online) https://www.geeksforgeeks.org/time-function-in-c/#:~:text=The%20time()%20function%20is,object%20pointed%20to%20by%20second.
4. (Available online) https://cplusplus.com/reference/cstdlib/srand/
5. (Available online) https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/?ref=gcse

6. (Available online) https://seedsecuritylabs.org/Labs_20.04/Files/Crypto_Random_Number/Crypto_Random_Number.pdf
7. (Available online) https://programmersought.com/article/21636379515/#22_Task_2_Guessing_the_Key_57
8. ((Available online)) https://security.stackexchange.com/questions/194958/dev-random-generate-session-key