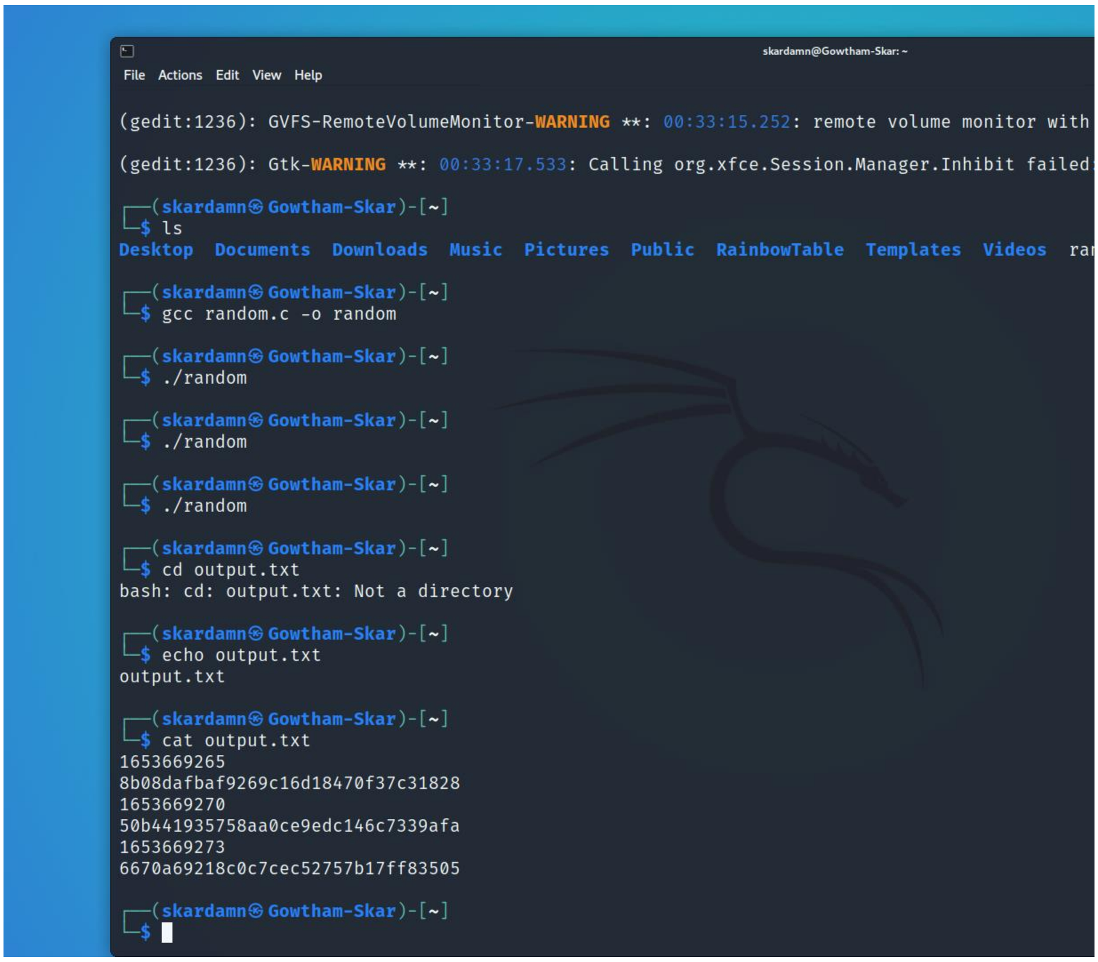
Assignment 1
Pseudo Random Number Generation (PRNG)
Security Tools Lab 2

Gowtham Baskar
1006523

Observation for Task 1

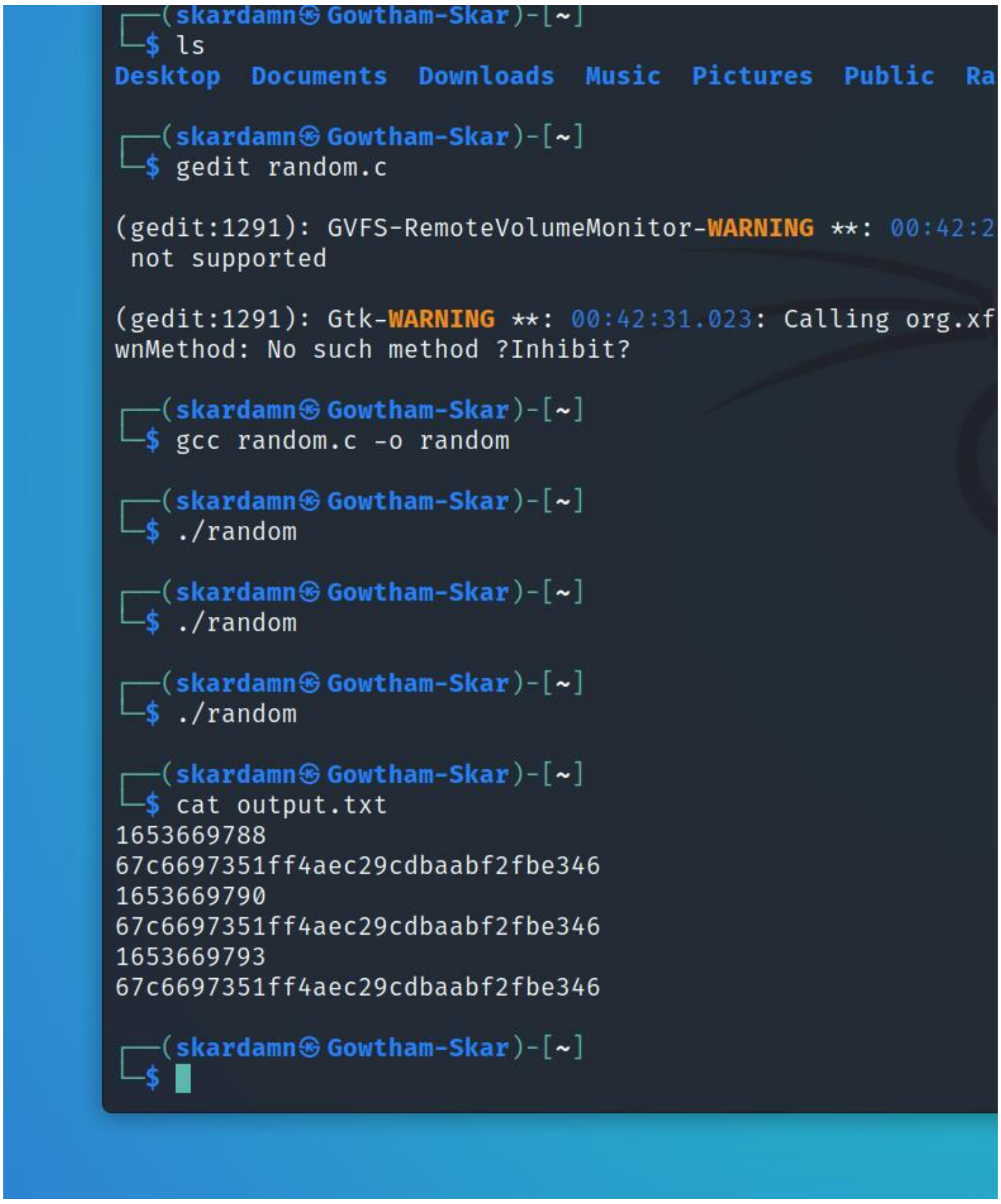
From Image 1, Random number generated each time is to be different based on the number of seconds. This is because the srand (time (NULL)) function is used to seed the random number generation function rand (). Time is a function of C language to obtain the current system time, in seconds, representing how many seconds the current time has passed since the Unix standard timestamp. The time of each run is different, so the random number obtained is also different.



```
(gedit:1236): GVFS-RemoteVolumeMonitor-WARNING **: 00:33:15.252: remote volume monitor with

(gedit:1236): Gtk-WARNING **: 00:33:17.533: Calling org.xfce.Session.Manager.Inhibit failed

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  RainbowTable  Templates  Videos  ra

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ gcc random.c -o random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ./random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ./random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ./random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ cd output.txt
bash: cd: output.txt: Not a directory

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ echo output.txt
output.txt

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ cat output.txt
1653669265
8b08dafbaf9269c16d18470f37c31828
1653669270
50b441935758aa0ce9edc146c7339afa
1653669273
6670a69218c0c7cec52757b17ff83505

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ▮
```

Image 1

From Image 2, we commented the srand (time (NULL)) function. Since no seed is set, the default random number for the seed will be set to 0. Therefore, the random number generated each time the program is executed will give the same value.

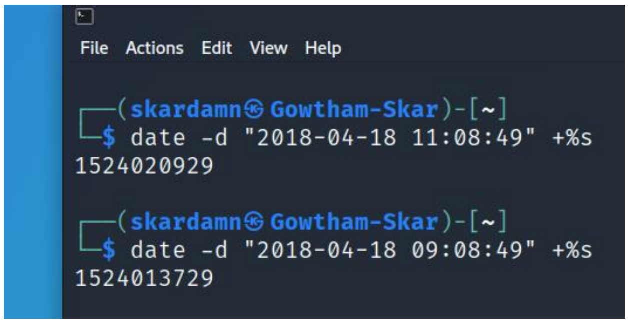


```
┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Ra

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ gedit random.c

(gedit:1291): GVFS-RemoteVolumeMonitor-WARNING **: 00:42:2
 not supported

(gedit:1291): Gtk-WARNING **: 00:42:31.023: Calling org.xf
wnMethod: No such method ?Inhibit?

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ gcc random.c -o random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ./random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ./random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ ./random

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ cat output.txt
1653669788
67c6697351ff4aec29cdbaabf2fbe346
1653669790
67c6697351ff4aec29cdbaabf2fbe346
1653669793
67c6697351ff4aec29cdbaabf2fbe346

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ 
```

Image 2

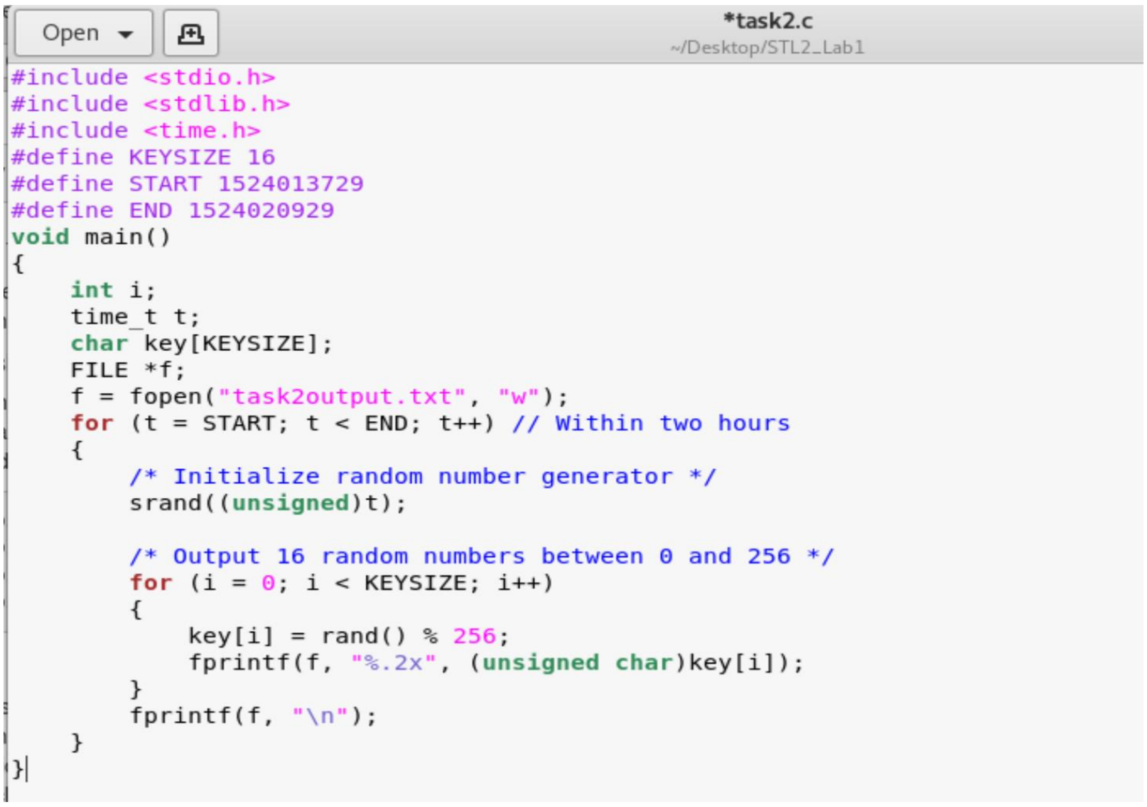
Observation for Task 2

As per the given scenario, I calculated the range since the assumed key generated was within a two-hour window before the encrypted file was created. That is from "2018-04-18 09:08:49" to "2018-04-18 11:08:49" as shown in the below image.



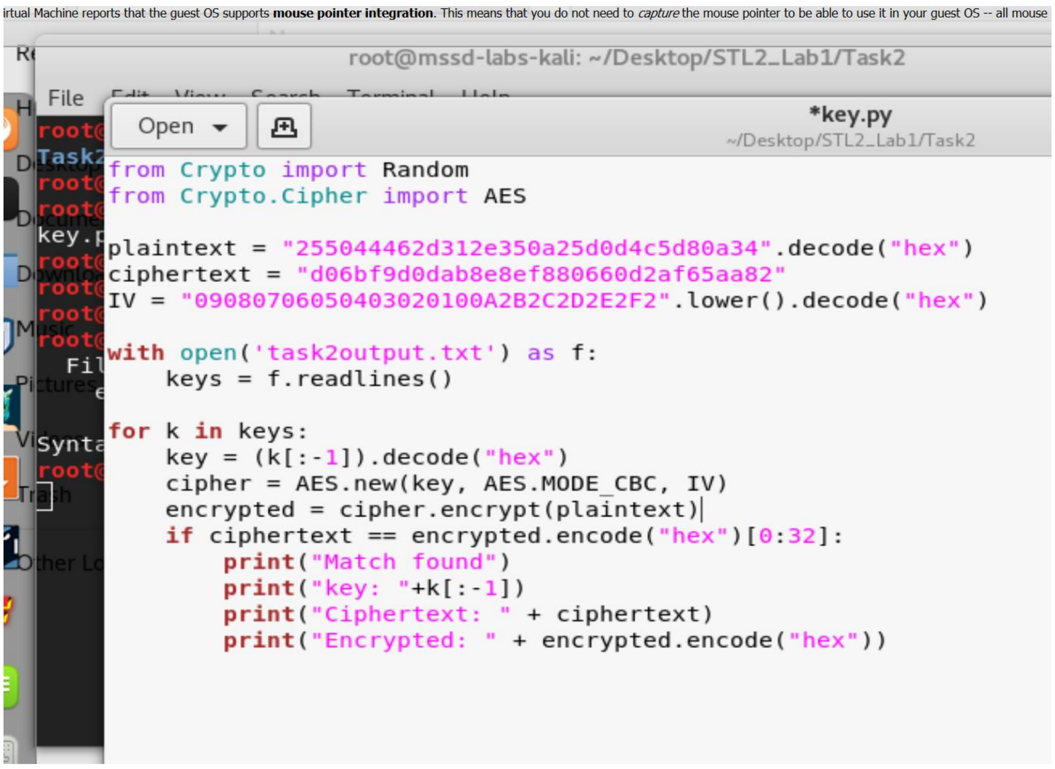
```
┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ date -d "2018-04-18 11:08:49" +%s
1524020929

┌──(skardamn㉿Gowtham-Skar)-[~]
└─$ date -d "2018-04-18 09:08:49" +%s
1524013729
```

After compiling the data within range, **Task2output.txt** file was created similar to Task 1.



```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16
#define START 1524013729
#define END 1524020929
void main()
{
    int i;
    time_t t;
    char key[KEYSIZE];
    FILE *f;
    f = fopen("task2output.txt", "w");
    for (t = START; t < END; t++) // Within two hours
    {
        /* Initialize random number generator */
        srand((unsigned)t);

        /* Output 16 random numbers between 0 and 256 */
        for (i = 0; i < KEYSIZE; i++)
        {
            key[i] = rand() % 256;
            fprintf(f, "%.2x", (unsigned char)key[i]);
        }
        fprintf(f, "\n");
    }
}
```

Based on the meta data of the encrypted file, we know that the file is encrypted using aes-128-cbc. Therefore, a python script was created which tries to match the plaintext, ciphertext and IV for all the keys generated in task2ouput.txt until one of them succeeds in finding the match. Inserted the Script **(key.py)** below for reference. In this script, I have imported the cryptographic module and used the AES encryption to identify encrypted key same as the cipher text.

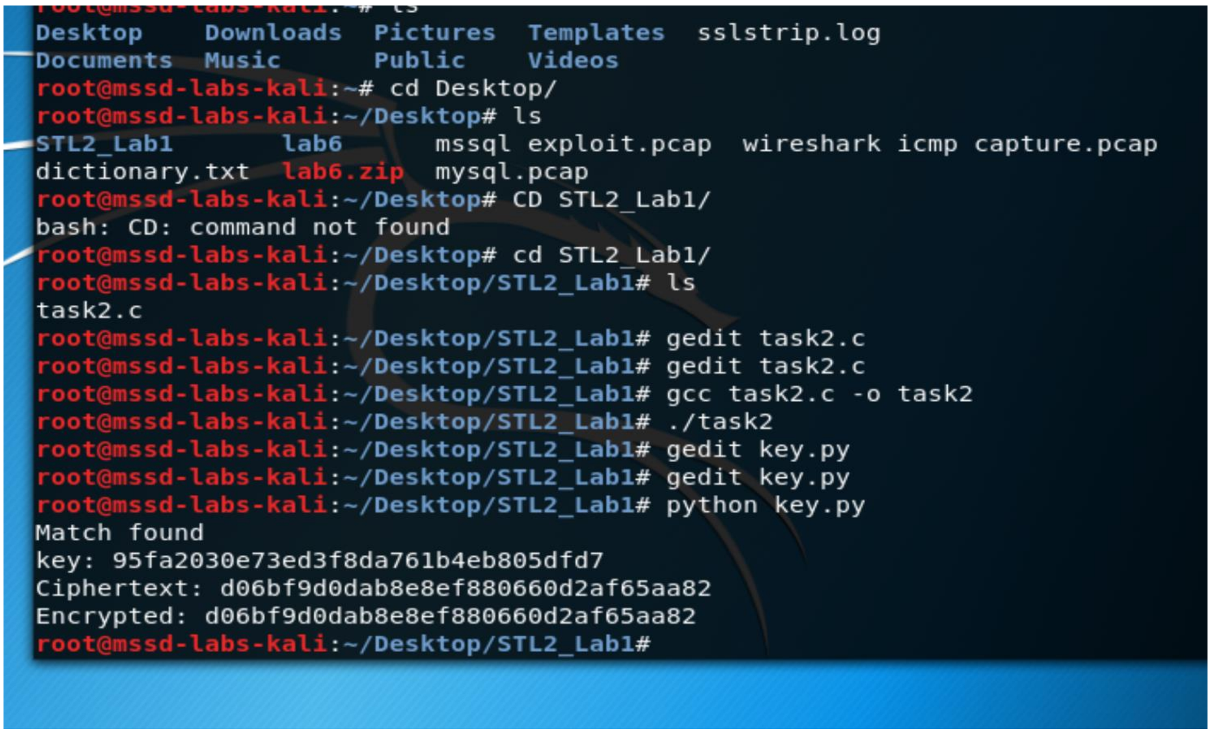


irtual Machine reports that the guest OS supports **mouse pointer integration**. This means that you do not need to *capture* the mouse pointer to be able to use it in your guest OS -- all mouse

root@mssd-labs-kali: ~/Desktop/STL2_Lab1/Task2
File
Open
*key.py
~/Desktop/STL2_Lab1/Task2

```python
from Crypto import Random
from Crypto.Cipher import AES

plaintext = "255044462d312e350a25d0d4c5d80a34".decode("hex")
ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
IV = "09080706050403020100A2B2C2D2E2F2".lower().decode("hex")

with open('task2output.txt') as f:
    keys = f.readlines()

for k in keys:
    key = (k[:-1]).decode("hex")
    cipher = AES.new(key, AES.MODE_CBC, IV)
    encrypted = cipher.encrypt(plaintext)
    if ciphertext == encrypted.encode("hex")[0:32]:
        print("Match found")
        print("key: "+k[:-1])
        print("Ciphertext: " + ciphertext)
        print("Encrypted: " + encrypted.encode("hex"))
```

After Running the script, we found the key as **“95fa2030e73ed3f8da761b4eb805dfd7”** as shown below.

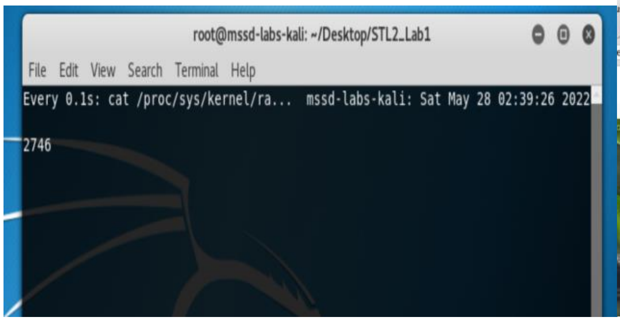


```
root@mssd-labs-kali:~# ls
Desktop    Downloads  Pictures  Templates  sslstrip.log
Documents  Music      Public    Videos
root@mssd-labs-kali:~# cd Desktop/
root@mssd-labs-kali:~/Desktop# ls
STL2_Lab1        lab6      mssql exploit.pcap  wireshark icmp capture.pcap
dictionary.txt  lab6.zip  mysql.pcap
root@mssd-labs-kali:~/Desktop# CD STL2_Lab1/
bash: CD: command not found
root@mssd-labs-kali:~/Desktop# cd STL2_Lab1/
root@mssd-labs-kali:~/Desktop/STL2_Lab1# ls
task2.c
root@mssd-labs-kali:~/Desktop/STL2_Lab1# gedit task2.c
root@mssd-labs-kali:~/Desktop/STL2_Lab1# gedit task2.c
root@mssd-labs-kali:~/Desktop/STL2_Lab1# gcc task2.c -o task2
root@mssd-labs-kali:~/Desktop/STL2_Lab1# ./task2
root@mssd-labs-kali:~/Desktop/STL2_Lab1# gedit key.py
root@mssd-labs-kali:~/Desktop/STL2_Lab1# gedit key.py
root@mssd-labs-kali:~/Desktop/STL2_Lab1# python key.py
Match found
key: 95fa2030e73ed3f8da761b4eb805dfd7
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
Encrypted: d06bf9d0dab8e8ef880660d2af65aa82
root@mssd-labs-kali:~/Desktop/STL2_Lab1#
```

It is not recommended to keep Time as the seed value as it is not a real random number and can easily be predicted.

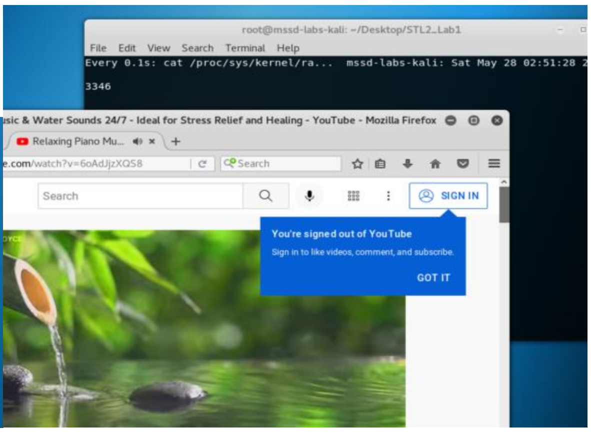
Observation for Task 3

According to the task given, every time the mouse is moved or clicked, every keystroke and every action performed in the system like opening a browser will change the entropy. I have recorded the entropy on the start and end in the below image. During this time, actions were performed as mentioned in the list below.



root@mssd-labs-kali: ~/Desktop/STL2_Lab1
File Edit View Search Terminal Help

```
Every 0.1s: cat /proc/sys/kernel/ra...  mssd-labs-kali: Sat May 28 02:39:26 2022

2746
```



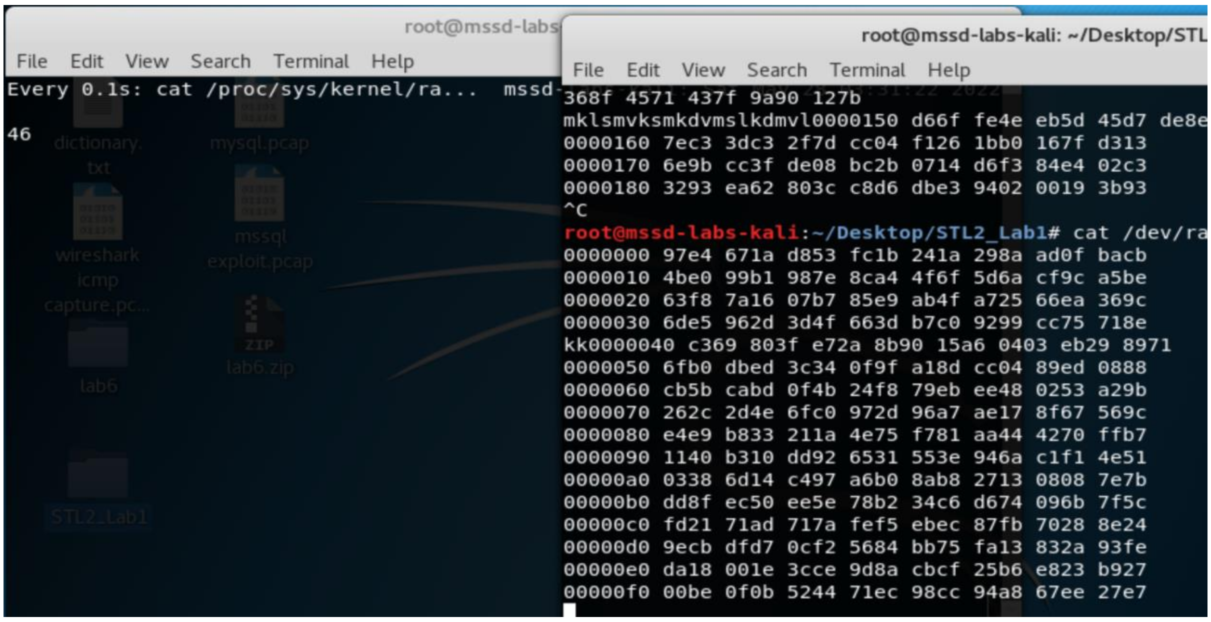
root@mssd-labs-kali: ~/Desktop/STL2_Lab1
File Edit View Search Terminal Help
sic & Water Sounds 24/7 - Ideal for Stress Relief and Healing - YouTube - Mozilla Firefox
Relaxing Piano Mu...
Search
SIGN IN
You're signed out of YouTube
Sign in to like videos, comment, and subscribe.
GOT IT

```
Every 0.1s: cat /proc/sys/kernel/ra...  mssd-labs-kali: Sat May 28 02:51:28 2

3346
```

List of Activities performed.

• Move Cursor/Mouse	• Opened Website
• Clicked Mouse (Left and Right)	• Ping 8.8.8.8
• Random Keyboard clicks	• YouTube Video (Auto Entropy Change)
• Opened Application	

After running “# cat /dev/random | hexdump” command along with the watch command, the entropy is changed on every 64th bit as shown below. From this observation, we see that the kernel generates random data and stops when the pool is exhausted until it gets (slowly) refilled. According to observation, the value ranges from 0 -63.



```
Every 0.1s: cat /proc/sys/kernel/ra...   mssd-

46
```

```
368f 4571 437f 9a90 127b
mklsmvksmkdvmslkdmvl0000150 d66f fe4e eb5d 45d7 de8e
0000160 7ec3 3dc3 2f7d cc04 f126 1bb0 167f d313
0000170 6e9b cc3f de08 bc2b 0714 d6f3 84e4 02c3
0000180 3293 ea62 803c c8d6 dbe3 9402 0019 3b93
^C
root@mssd-labs-kali:~/Desktop/STL2_Lab1# cat /dev/ra
0000000 97e4 671a d853 fc1b 241a 298a ad0f bacb
0000010 4be0 99b1 987e 8ca4 4f6f 5d6a cf9c a5be
0000020 63f8 7a16 07b7 85e9 ab4f a725 66ea 369c
0000030 6de5 962d 3d4f 663d b7c0 9299 cc75 718e
kk0000040 c369 803f e72a 8b90 15a6 0403 eb29 8971
0000050 6fb0 dbed 3c34 0f9f a18d cc04 89ed 0888
0000060 cb5b cabd 0f4b 24f8 79eb ee48 0253 a29b
0000070 262c 2d4e 6fc0 972d 96a7 ae17 8f67 569c
0000080 e4e9 b833 211a 4e75 f781 aa44 4270 ffb7
0000090 1140 b310 dd92 6531 553e 946a c1f1 4e51
00000a0 0338 6d14 c497 a6b0 8ab8 2713 0808 7e7b
00000b0 dd8f ec50 ee5e 78b2 34c6 d674 096b 7f5c
00000c0 fd21 71ad 717a fef5 ebec 87fb 7028 8e24
00000d0 9ecb dfd7 0cf2 5684 bb75 fa13 832a 93fe
00000e0 da18 001e 3cce 9d8a cbcf 25b6 e823 b927
00000f0 00be 0f0b 5244 71ec 98cc 94a8 67ee 27e7
```

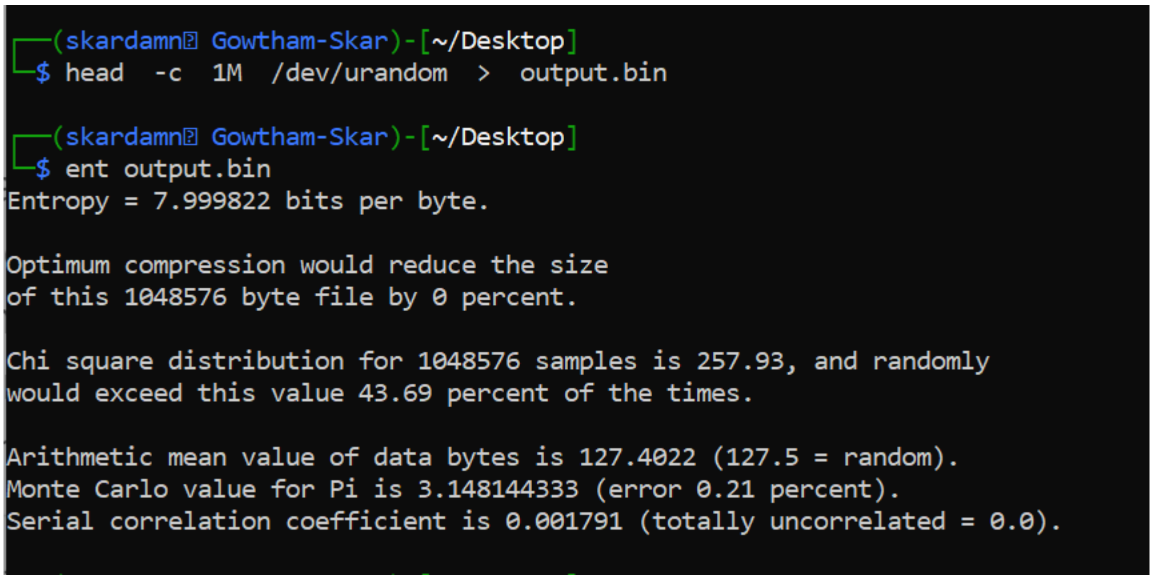
If a server uses /dev/random to generate the random session key with a client, there will not be any reaction since the mouse is not being moved. A new row of data will only appear when the mouse is moved to a certain extent. DOS attack can occur by exhausting /dev/random's entropy. Reading from /dev/random will block on some systems if there is not enough entropy available. Quickly drawing entropy from the system by creating lots of session ids might thus lead to the application blocking when creating another session id, thus making at least this specific part of the application would not respond for some time. This leads to Denial of Service.

Observation for Task 4

After running “cat /dev/urandom | hexdump” command, Entropy keeps changing and it does not depend on the mouse or keyboard action.

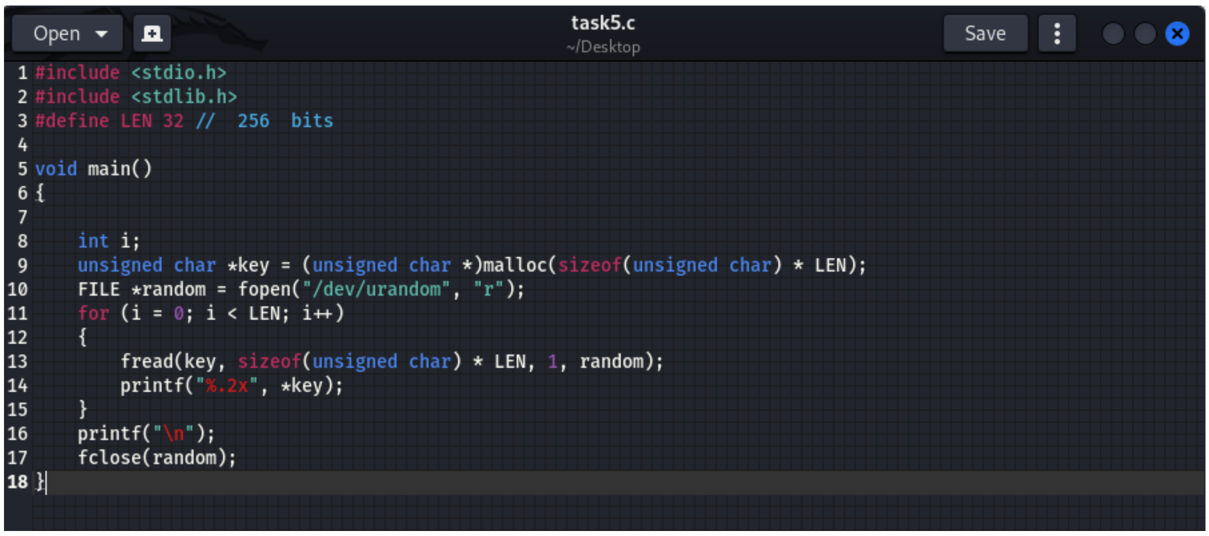
Read on Window’s 10 PRNG in this article https://aka.ms/win10rng, Windows are using cryptogen randomization and several other entropies to increase the source of the seed. It is not completely different as they stated that “In general, low sources are assumed to provide unconditioned entropy events, such as mouse movements. High sources are assumed to provide high-quality random bytes. All sources can provide entropy data at any time, but a pull source is additionally notified whenever the system reseeds the root PRNG from the entropy pools. This allows an on-demand source to provide entropy for every reseed without having its own timer logic”. The also have primary APIs for random number generation in windows 10 such as SystemPrng, ProcessPrng, BCryptGenRandom, CryptGenRandom and RtlGenRandom.

Observation for Task 5



```
┌──(skardamn㉿ Gowtham-Skar)-[~/Desktop]
└─$ head  -c  1M  /dev/urandom  >  output.bin

┌──(skardamn㉿ Gowtham-Skar)-[~/Desktop]
└─$ ent output.bin
Entropy = 7.999822 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.

Chi square distribution for 1048576 samples is 257.93, and randomly
would exceed this value 43.69 percent of the times.

Arithmetic mean value of data bytes is 127.4022 (127.5 = random).
Monte Carlo value for Pi is 3.148144333 (error 0.21 percent).
Serial correlation coefficient is 0.001791 (totally uncorrelated = 0.0).
```

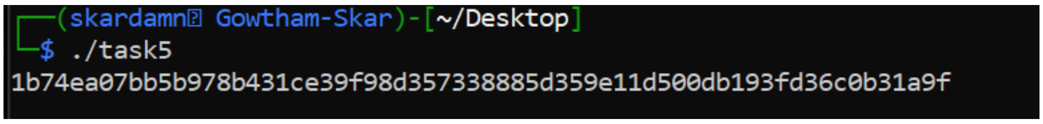
In principle, / dev/random devices are safer, but in fact, the "seed" utilized by / dev/random is arbitrary and unpredictable (every time new random data becomes available, / dev/random will reseed)/ Blocking can lead to denial-of-service attacks, which is a major issue with dev/blocking random's behaviour. As a result, using / dev/urandom to generate random numbers is suggested.



task5.c
~/Desktop

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define LEN 32 //  256  bits
4  
5  void main()
6  {
7  
8      int i;
9      unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
10     FILE *random = fopen("/dev/urandom", "r");
11     for (i = 0; i < LEN; i++)
12     {
13         fread(key, sizeof(unsigned char) * LEN, 1, random);
14         printf("%.2x", *key);
15     }
16     printf("\n");
17     fclose(random);
18 }
```

From Line 10, I have also increased the length by modifying it from 16 to 32 (256 bits), thus making it more secure with a 256-bit encryption key. We can see that the file is opening and reading from the dev/urandom. From this we can confirm that this is a real random number as shown below.



```
┌──(skardamn㉿ Gowtham-Skar)-[~/Desktop]
└─$ ./task5
1b74ea07bb5b978b431ce39f98d357338885d359e11d500db193fd36c0b31a9f
```