

## **PRN\_LabReport**

### **Task1: Generate Encryption Key in Wrong Way**

- How can we make an outcome unpredictable?

In comparison with the pseudo-random number generator (PRNG), the true random number generator (TRNG) relies on gadgets that promise to create really random numbers.

The gadgets are unpredictable rather than human-defined patterns like an algorithm. For really random numbers, the computer must employ an unpredictable external physical variable, such as radioactive decay of isotopes or airwave static, rather than.

- What is the purpose of srand() and time()?

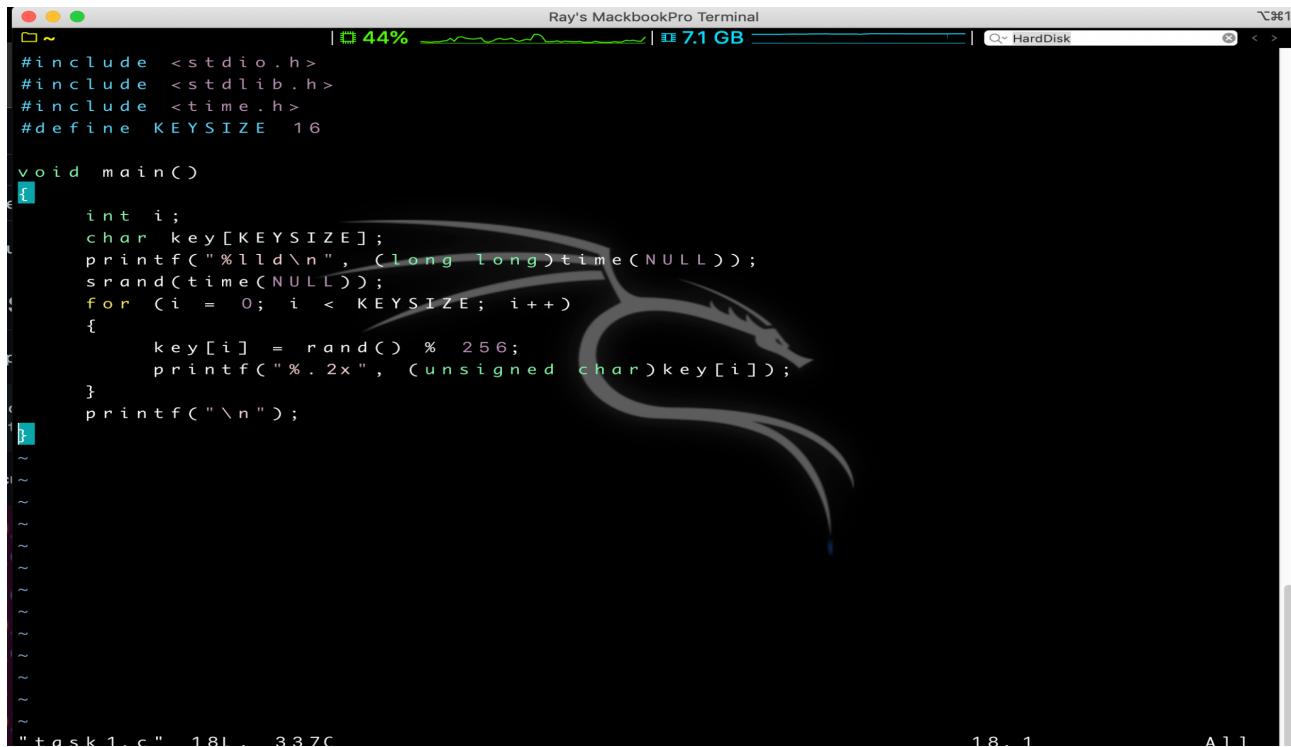
The purpose of srand() is that pass the argument seed to start the pseudo-random number generator. And the time() function means generating a time-based input as an argument for the seed.

- What is the result of commenting out the srand()?

The procedure will not be involved with creating an initial input for generating pseudo-random numbers if srand() is commented out. So, every time the program runs, it will generate an identical series of integers.

### Task 1 Screenshots and Observations

#### Generate random numbers with **rand()** function



A screenshot of a Mac OS X terminal window titled "Ray's MacBookPro Terminal". The window shows a C program named "task\_1.c" with the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
    int i;
    char key[KEYSIZE];
    printf("%lld\n", (long long)time(NULL));
    srand(time(NULL));
    for (i = 0; i < KEYSIZE; i++)
    {
        key[i] = rand() % 256;
        printf("%.2x", (unsigned char)key[i]);
    }
    printf("\n");
}
```

The terminal output shows the current time followed by a series of random bytes represented as hex values:

```
181 . 3.37C
```

Observation:

Because it generates random numbers using the current time as a random seed, the seed is always different in each run.

```

Ray's MackbookPro Terminal
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ sudo vim task1.c
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ gcc task1.c -o task1
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ ./task1
1648832448
02b4748f517bfbf0ca508973ff981f09
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ ./task1
1648832449
04d9471e95363225bc87a8f34e9fac1f
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ ./task1
1648832449
04d9471e95363225bc87a8f34e9fac1f
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ ./task1
1648832464
0de49beabdfcecc33c1df50a253511d9
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ ./task1
1648832466
c94a7889d73a76c02cd81fe627e1739a
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ ./task1
1648832470
fcbb86661df86c7adfd5ed89dbf2c8cf4
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $ |

```

Generate random numbers **without** `rand()` function

```

Ray's MackbookPro Terminal
[ 0 / 01 / 22] s e e d @ V M : ~ / . . . / P R N G L a b $
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

void main()
{
    int i;
    char key[KEYSIZE];
    printf("%lld\n", (long long)time(NULL));
    /*rand(time(NULL));
    for (i = 0; i < KEYSIZE; i++)
    {
        key[i] = rand() % 256;
        printf("%.2x", (unsigned char)key[i]);
    }
    printf("\n");
}
~ ~

```

Observation:

Regarding commenting out strand() function, the program does not use the current time as a seed to generate random numbers, the random numbers are always the same in each run.

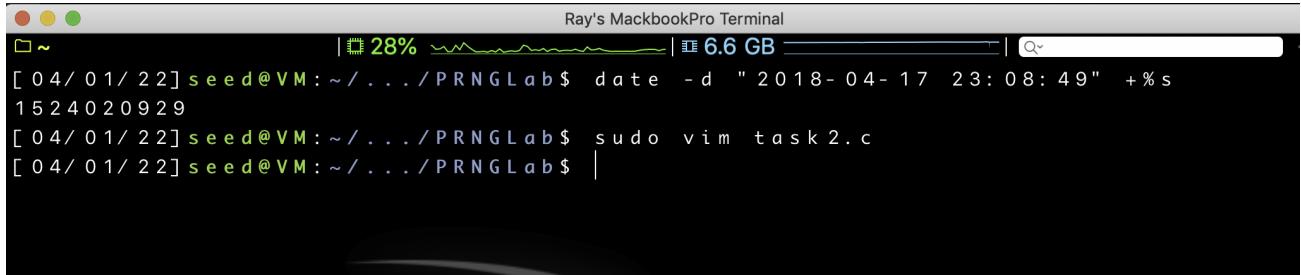
```
[ 04/01/22] se ed@VM:~/.../PRNGLab$ sudo vim task1.c
[ 04/01/22] se ed@VM:~/.../PRNGLab$ gcc task1.c -o task1_a
[ 04/01/22] se ed@VM:~/.../PRNGLab$ ./task1_a
1648832060
67c6697351ff4aec29cdbaabf2fb346
[ 04/01/22] se ed@VM:~/.../PRNGLab$ ./task1_a
1648832063
67c6697351ff4aec29cdbaabf2fb346
[ 04/01/22] se ed@VM:~/.../PRNGLab$ ./task1_a
1648832064
67c6697351ff4aec29cdbaabf2fb346
[ 04/01/22] se ed@VM:~/.../PRNGLab$ ./task1_a
1648832065
67c6697351ff4aec29cdbaabf2fb346
[ 04/01/22] se ed@VM:~/.../PRNGLab$ ./task1_a
1648832066
67c6697351ff4aec29cdbaabf2fb346
[ 04/01/22] se ed@VM:~/.../PRNGLab$ ./task1_a
1648832066
67c6697351ff4aec29cdbaabf2fb346
```

## Task2: Guessing the Key

The purpose of Task 2 is to decrypt the full document using the encryption key. Bob surmised that the key would be generated during a two-hour period before the file was created.

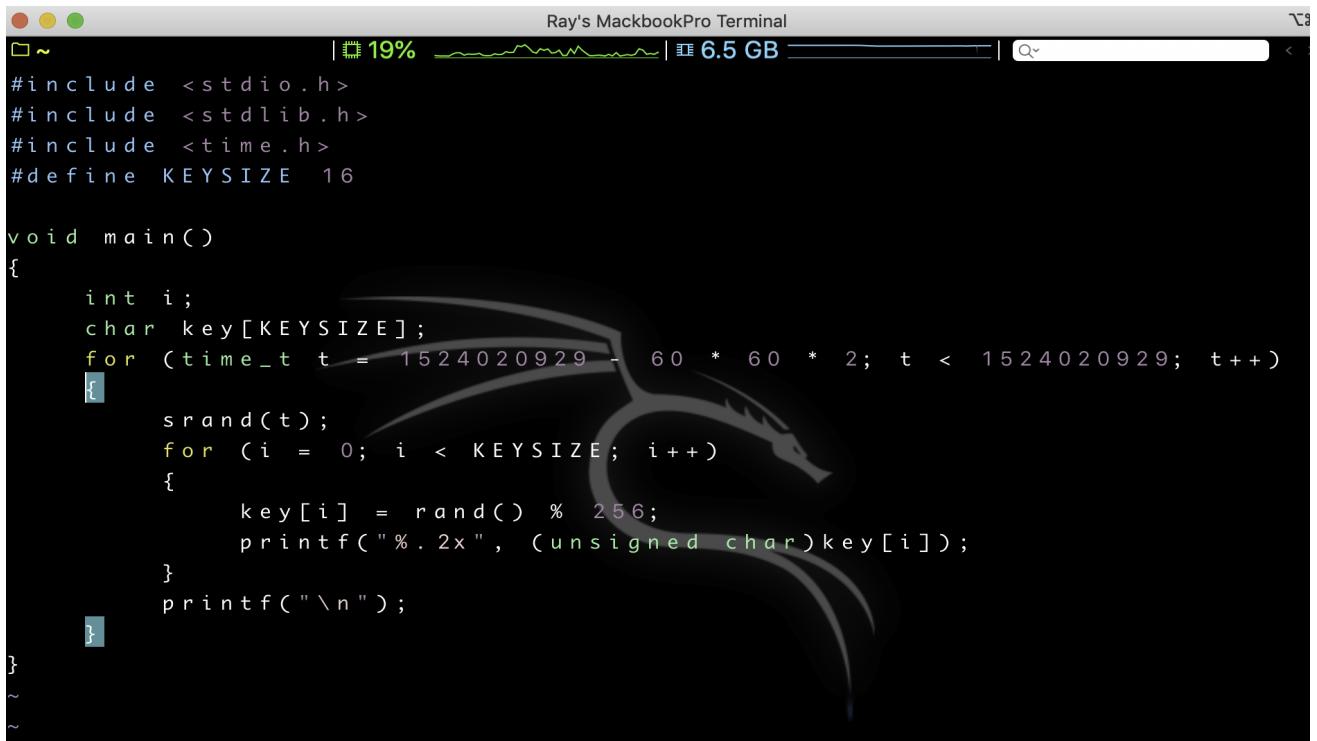
## Task 2 Screenshots and Observations

The timestamp of the encrypted file is "2018-04-17 23:08:49" so calculate its number of seconds.



```
Ray's MacBookPro Terminal
[04/01/22] seed@VM:~/.../PRNGLab$ date -d "2018-04-17 23:08:49" +%s
1524020929
[04/01/22] seed@VM:~/.../PRNGLab$ sudo vim task2.c
[04/01/22] seed@VM:~/.../PRNGLab$
```

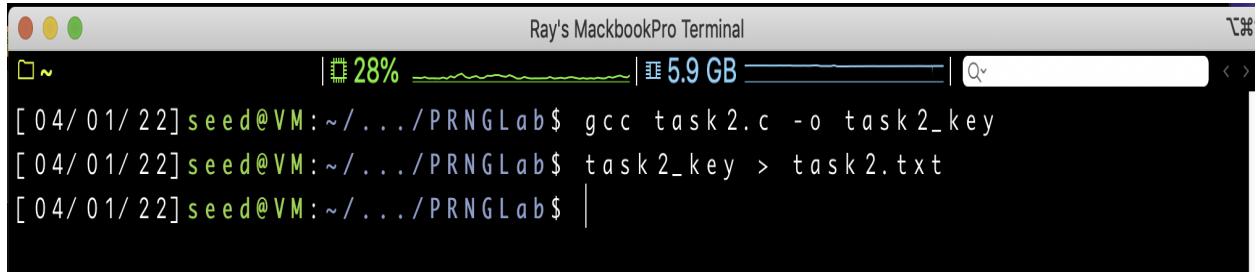
The below program can list the possible pseudo-random generator numbers. And the file was created during a two-hour window so the program needs to minus the number of seconds in 2 hours total.



```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

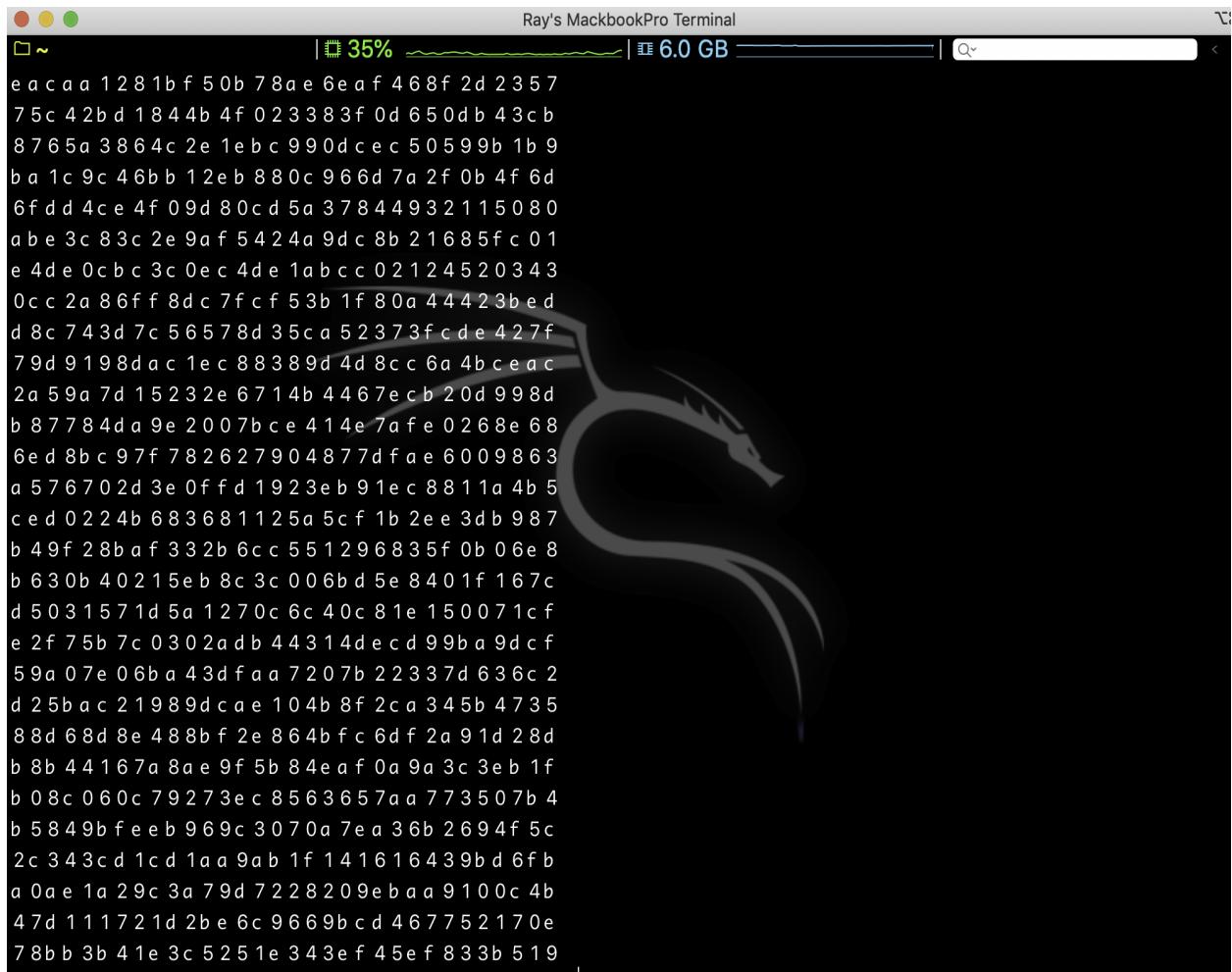
void main()
{
    int i;
    char key[KEYSIZE];
    for (time_t t = 1524020929 - 60 * 60 * 2; t < 1524020929; t++)
    {
        srand(t);
        for (i = 0; i < KEYSIZE; i++)
        {
            key[i] = rand() % 256;
            printf("% .2x", (unsigned char)key[i]);
        }
        printf("\n");
    }
}
```

GCC compiles C program and redirects stdout to in a .txt file as a dictionary



```
[04/01/22] seed@VM:~/.../PRNGLab$ gcc task2.c -o task2_key
[04/01/22] seed@VM:~/.../PRNGLab$ task2_key > task2.txt
[04/01/22] seed@VM:~/.../PRNGLab$
```

The below screenshot shows the possible pseudo-random generator numbers which are used to encrypt the file.



```
e a a a 1281bf50b78ae6ea f468f2d2357
75c42bd1844bf4f023383f0d650db43cb
8765a3864c2e1ebc990dccec50599b1b9
ba1c9c46bb12eb880c966d7a2f0b4f6d
6fdd4ce4f09d80cd5a37844932115080
abe3c83c2e9af5424a9dc8b21685fc01
e4de0cbc3c0ec4de1abcc02124520343
0cc2a86ff8dc7fcf53b1f80a44423bed
d8c743d7c56578d35ca52373fcde427f
79d9198dac1ec88389d4d8cc6a4bc eac
2a59a7d15232e6714b4467ecb20d998d
b87784da9e2007bce414e7afe0268e68
6ed8bc97f782627904877dfae6009863
a576702d3e0ffd1923eb91ec8811a4b5
ced0224b683681125a5cf1b2ee3db987
b49f28ba f332b6cc551296835f0b06e8
b630b40215eb8c3c006bd5e8401f167c
d5031571d5a1270c6c40c81e150071cf
e2f75b7c0302adb44314dec99ba9dcf
59a07e06ba43dfaa7207b22337d636c2
d25bac21989dc a e104b8f2ca345b4735
88d68d8e488bf2e864bfcc6df2a91d28d
b8b44167a8ae9f5b84ea f0a9a3c3eb1f
b08c060c79273ec8563657aa773507b4
b5849bf eeb969c3070a7ea36b2694f5c
2c343cd1cd1aa9ab1f141616439bd6fb
a0ae1a29c3a79d7228209eba a9100c4b
47d111721d2be6c9669bcd467752170e
78bb3b41e3c5251e343ef45ef833b519
```

As given the information as belows,

Plaintext: 255044462d312e350a25d0d4c5d80a34

Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82

IV: 09080706050403020100A2B2C2D2E2F2

To crack the key from task2.txt, utilize a brute-force approach (dictionary approach).



```
Ray's MacBookPro Terminal
22% 6.4 GB
~/
! /usr/bin/python
from Crypto.Cipher import AES

Plaintext = bytearray.fromhex('255044462d312e350a25d0d4c5d80a34')
Ciphertext = bytearray.fromhex('d06bf9d0dab8e8ef880660d2af65aa82')
Iv = bytearray.fromhex('09080706050403020100A2B2C2D2E2F2')

with open('task2.txt') as f:
    keys = f.readlines()

for k in keys:
    k = k.rstrip('\n')
    key = bytearray.fromhex(k)
    cipher = AES.new(key=key, mode=AES.MODE_CBC, iv=Iv)
    Guess_Ciphertext = cipher.encrypt(Plaintext)
    if Guess_Ciphertext == Ciphertext:
        print("find the encryption key:", k)
        exit(0)

print("cannot find the encryption key!")
~
```

Find the encryption key: **95fa2030e73ed3f8da761b4eb805dfd7**

```
Ray's MacBookPro Terminal
19% 6.4 GB
[04/01/22] seed@VM:~/.../PRNGLab$ sudo chmod 777 task2.py
[04/01/22] seed@VM:~/.../PRNGLab$ task2.py
find the encryption key: 95fa2030e73ed3f8da761b4eb805dfd7
[04/01/22] seed@VM:~/.../PRNGLab$ |
```

Observation:

Utilize the given date converting to the number of seconds to generate the possible pseudo-random numbers, and store these numbers in a dictionary file. Then, guess the encryption key by brute force way with the given plaintext, ciphertext, IV and encryption algorithm.

### **Task 3: Measure the entropy of the Kernel**

- From what resources does Linux get randomness? (name 3 and explain each one).
  1. input randomness: This is usually involved in games, it is when something random happens before the player has any input into the game.
  2. device randomness: an evenly-distributed integer random number generator that generates non-deterministic random numbers
  3. environmental noise randomness: This refers to an external source of randomness, such as the user's interactions with the keyboard and mouse, as well as the signal on an audio input line.
- What does entropy mean?

The meaning of entropy is the randomness gathered by a system for use in algorithms that require random data, which simply refers to how many random number bits the system currently possesses. In other words, entropy is a measure of the unpredictability or variety of a data-generating function in cyber security. A lack of good entropy might leave a cryptosystem insecure and unable to encrypt data securely.

### *Task 3 Screenshots and Observations*

When I input short words, it appears that the entropy figures climb slowly.

Ray's MackbookPro Terminal

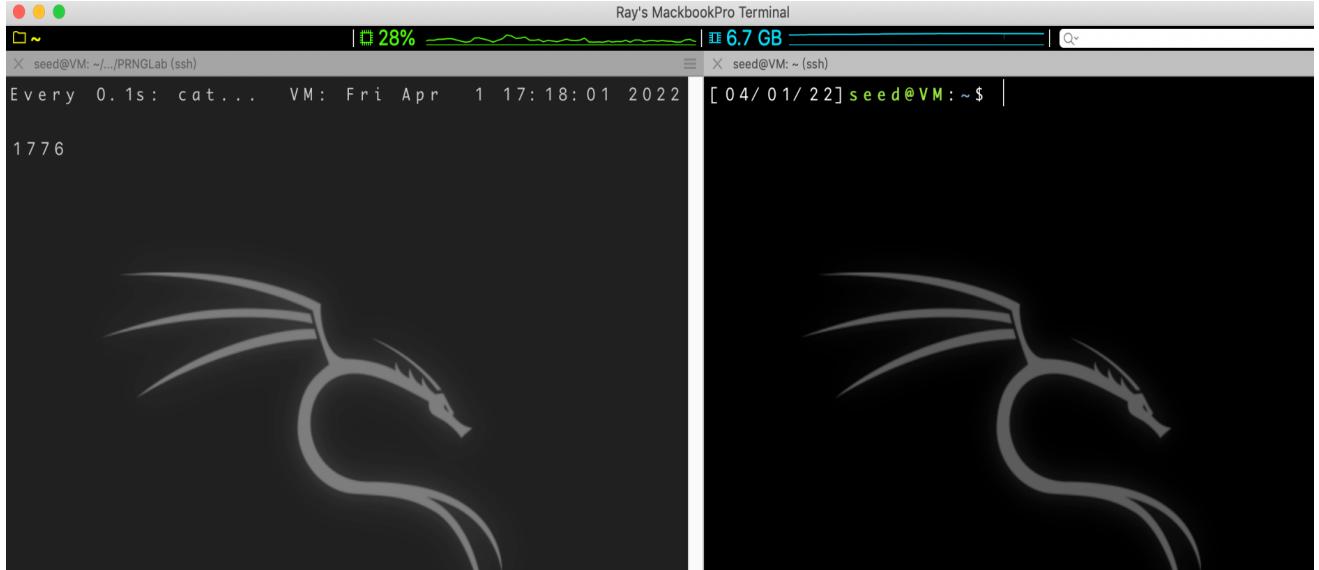
41% 6.5 GB

seed@VM: ~|/PRNGLab (ssh)

```
Every 0.1s: cat...  VM: Fri Apr  1 17:18:12 2022 [ 04/01/22] seed@VM:~$ nyitv  
nyitv: command not found  
[ 04/01/22] seed@VM:~$ |
```

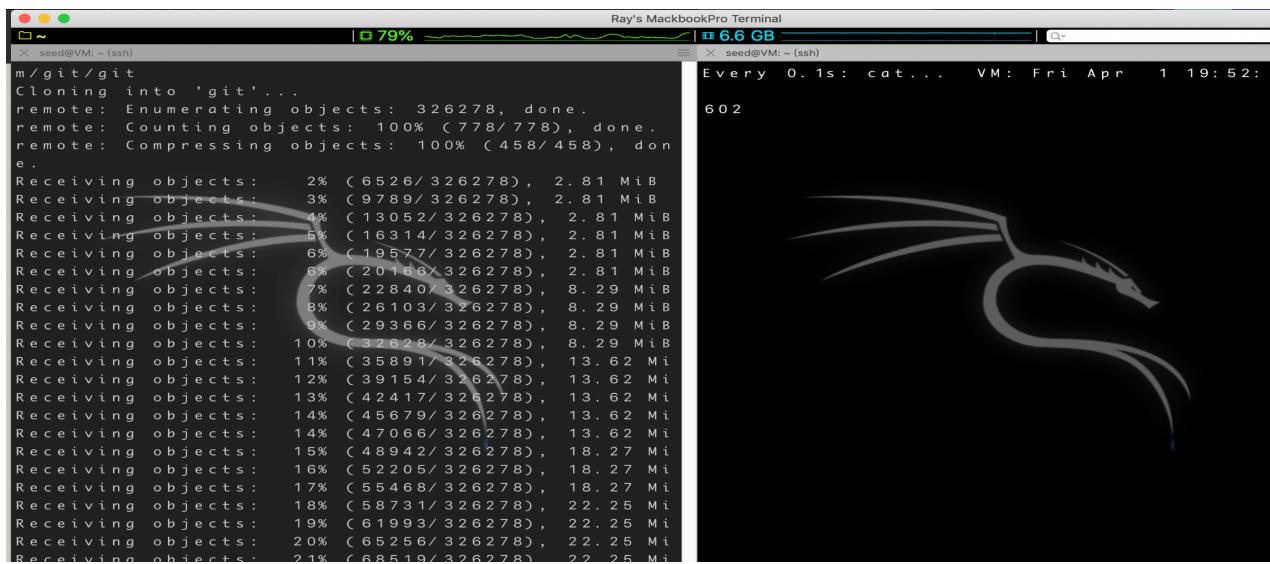
The below screenshots show that printing the content of a 76 bytes file via the cat command, increases the entropy number quickly compared to typing some words and moving the mouse.

Furthermore, if I move the mouse quickly, the entropy number also climbs tediously.



The below screenshot show when I open a new monitor window and clone a file from Github,

The number of entropy increases rapidly which is the fastest in this experiment such as moving the mouse, typing some words and printing the content of the file.



Overall, according to the observations, it might say that **/dev/random** utilizes the available entropy generated by user interactions to generate new random numbers.

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

- Where does Linux store the collected random data?

The Linux kernel gathers noise from device drivers and other sources and stores them in its entropy pool. The generator also maintains track of the number of noise bits in the entropy pool. The random numbers are generated from this entropy pool.

- What are “/dev/random” and “/dev/urandom”?

The meaning of **/dev/random** provides a maximum of the number of bits of randomness stored in the entropy pool. Also, **/dev/random** should be suited for applications that require very high-quality randomization. When the entropy pool is reduced, reads from **/dev/random** will be blocked until more environmental noise is obtained.

And, the meaning of **/dev/urandom** is reading from **/dev/urandom** will not be interrupted while waiting for more entropy. As a result, if the entropy pool is reduced, the returned values are theoretically open to a cryptographic attack on the driver's algorithms.

- What is the difference between “/dev/random” and “/dev/urandom”?

The difference between “/dev/random” and “/dev/urandom” is that “/dev/random” may necessitate some waiting for an entropy pool, where random data may not be accessible at the time while “/dev/urandom” only responds the number of bytes requested by the user, making it less unpredictable than /dev/random.

- What is your observation for this task if you do not move your mouse or type anything?

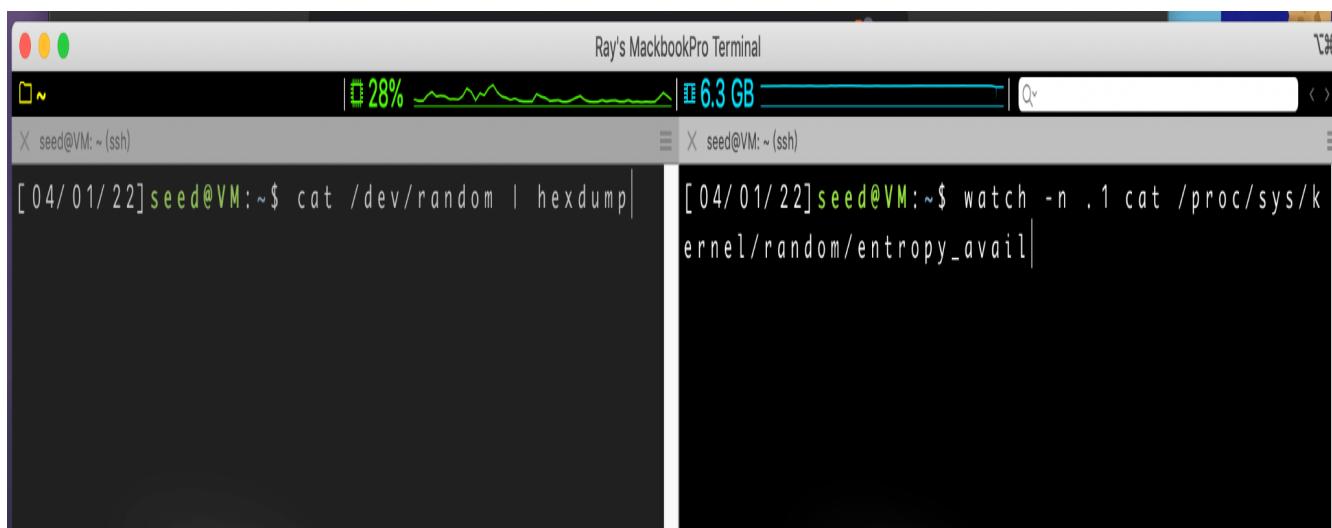
When I remain motionless, the number of entropy generates slowly and almost becomes stuck.

- What is your observation for this task if you move your mouse or type anything?

When typing some words on the left window and moving the mouse, it is indicated that the number of entropy increases on the right monitor window.

#### *Task 4 Screenshots and Observations*

The below screenshots show the /dev/random entropy usage and monitor simultaneously.



When typing some words on the left window and moving the mouse, the number of entropy increases.



When I stop typing some words on the left window and stop moving the mouse, the number of entropy seems stuck and if the number of entropy decreases, some new line of random numbers appears.



## Task 5: Get Random Numbers from /dev/urandom

- What is ent tool?

Ent tool is a good command-line utility for calculating entropy in a simple and quick manner and does a number of tests on the file's bytes stream. It can be used to evaluate pseudo-random number generators for encryption and statistical sampling applications, compression methods, and other applications where the information density of a file is important.

The below screenshot shows manual ent via the “man ent” command.

```

Ray's MackbookPro Terminal
| 32% | 7.1 GB | < > | ~ | man(1) | q | ↵

ent(1)                                ent(1)

NAME
    ent - pseudorandom number sequence test

SYNOPSIS
    ent [options] [file]

DESCRIPTION
    ENT Logo

    ent performs a variety of tests on the
    stream of bytes in file (or standard
    input if no file is specified) and produces
    output on standard output; for example:

    Entropy = 7.980627 bits per character.

    Optimum compression would reduce the size
    of this 51768 character file by 0 percent.

    Chi square distribution for 51768 samples is 1542.26, and randomly
    would exceed this value 0.01 percent of the times.

    Arithmetic mean value of data bytes is 125.93 (127.5 = random).
    Monte Carlo value for Pi is 3.169834647 (error 0.90 percent).
    Serial correlation coefficient is 0.004249 (totally uncorrelated =
    0.0).

Manual page ent( ) line   (press h for help or q to quit)

```

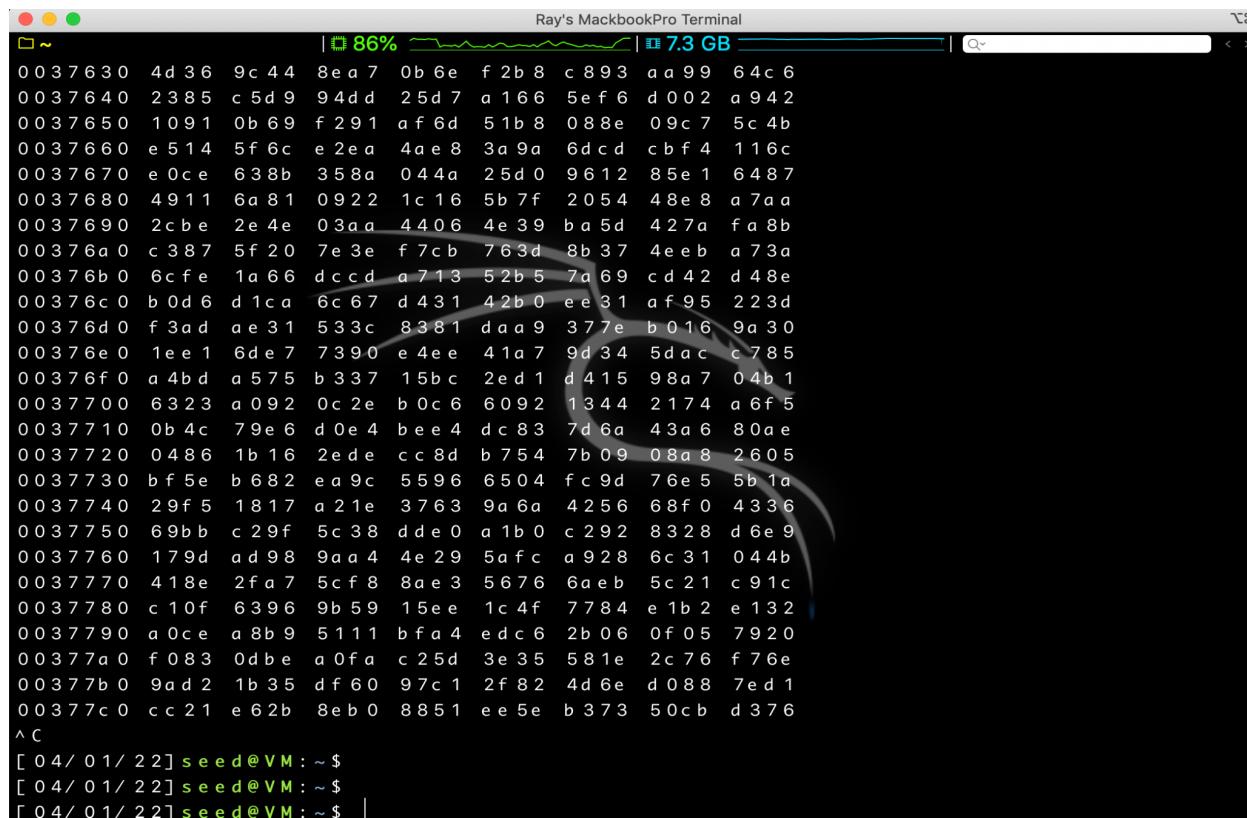
- Which one is more secure “/dev/urandom” or “/dev/random”?

The /dev/random is more secure than the dev/urandom.

The fundamental distinction between /dev/random and /dev/urandom is that /dev/random blocks if the entropy does not indicate adequate randomness, whereas /dev/urandom never blocks, even if the pseudo-random number generator is not fully seeded when booted. In short, /dev/random is the safest, followed by /dev/urandom, which is the least safe.

### *Task 5 Screenshots and Observations*

The below screenshots shows the result of “cat /dev/urandom | hexdump”, which indicates that it continues to generate random integers.



The screenshot shows a terminal window titled "Ray's MackbookPro Terminal". The window has a dark background with light-colored text. At the top, there is a status bar showing battery level (86%), signal strength, and disk usage (7.3 GB). The main area of the terminal displays the output of the command "cat /dev/urandom | hexdump". The output consists of a long sequence of random hexadecimal digits. The terminal window has a standard OS X look with red, yellow, and green window control buttons at the top left. The cursor is visible at the bottom of the terminal window.

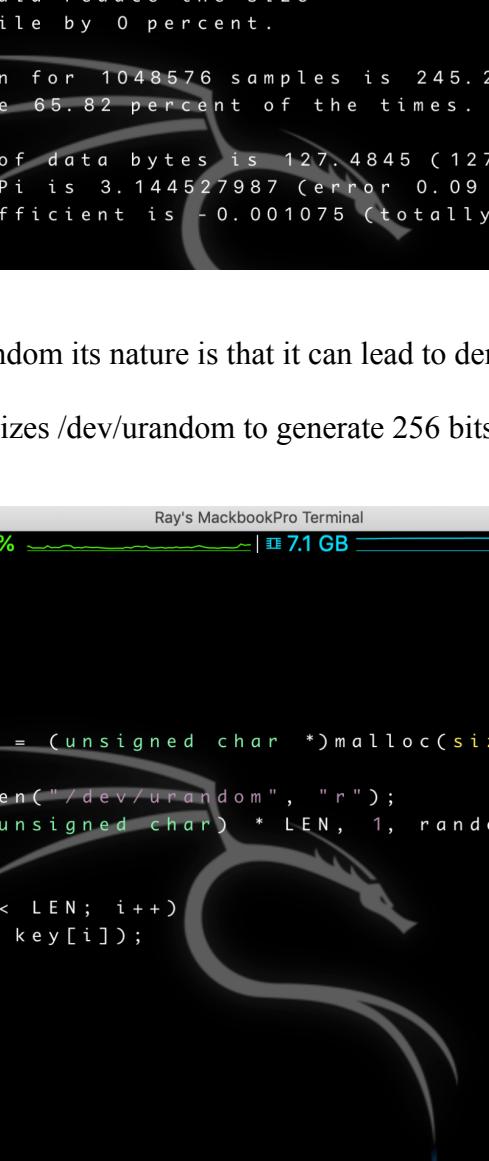
```

Ray's MackbookPro Terminal
| 86% | 7.3 GB |
0037630 4d 36 9c 44 8e a7 0b 6e f 2b 8 c 893 aa 99 64c 6
0037640 2385 c 5d 9 94dd 25d 7 a 166 5e f 6 d 002 a 942
0037650 1091 0b 69 f 291 a f 6d 51b 8 088e 09c 7 5c 4b
0037660 e 514 5f 6c e 2e a 4a e 8 3a 9a 6d c d c b f 4 116c
0037670 e 0ce 638b 358a 044a 25d 0 9612 85e 1 6487
0037680 4911 6a 81 0922 1c 16 5b 7f 2054 48e 8 a 7aa
0037690 2cbe 2e 4e 03aa 4406 4e 39 b a 5d 427a f a 8b
00376a0 c 387 5f 20 7e 3e f 7cb 763d 8b 37 4e e b a 73a
00376b0 6c fe 1a 66 d c c d a 713 52b 5 7a 69 c d 42 d 48e
00376c0 b 0d 6 d 1c a 6c 67 d 431 42b 0 e e 31 a f 95 223d
00376d0 f 3ad a e 31 533c 8381 d a a 9 377e b 016 9a 30
00376e0 1ee 1 6d e 7 7390 e 4e e 41a 7 9d 34 5d a c c 785
00376f0 a 4bd a 575 b 337 15b c 2e d 1 d 415 98a 7 04b 1
0037700 6323 a 092 0c 2e b 0c 6 6092 1344 2174 a 6f 5
0037710 0b 4c 79e 6 d 0e 4 b e e 4 d c 83 7d 6a 43a 6 80a e
0037720 0486 1b 16 2e d e c c 8d b 754 7b 09 08a 8 2605
0037730 b f 5e b 682 e a 9c 5596 6504 f c 9d 76e 5 5b 1a
0037740 29f 5 1817 a 21e 3763 9a 6a 4256 68f 0 4336
0037750 69b b c 29f 5c 38 d d e 0 a 1b 0 c 292 8328 d 6e 9
0037760 179d a d 98 9a a 4 4e 29 5a f c a 928 6c 31 044b
0037770 418e 2f a 7 5c f 8 8a e 3 5676 6a e b 5c 21 c 91c
0037780 c 10f 6396 9b 59 15e e 1c 4f 7784 e 1b 2 e 132
0037790 a 0c e a 8b 9 5111 b f a 4 e d c 6 2b 06 0f 05 7920
00377a0 f 083 0d b e a 0f a c 25d 3e 35 581e 2c 76 f 76e
00377b0 9a d 2 1b 35 d f 60 97c 1 2f 82 4d 6e d 088 7e d 1
00377c0 c c 21 e 62b 8e b 0 8851 e e 5e b 373 50c b d 376
^C
[ 04/ 01/ 22] seed@VM: ~ $
[ 04/ 01/ 22] seed@VM: ~ $
[ 04/ 01/ 22] seed@VM: ~ $ |

```

Because of continually generating random integers, use head to generate 1 MB of pseudo-random numbers from /dev/urandom and save them to output.bin.

And analyzing the result via “ent” tool.



```
Ray's MackbookPro Terminal
[ 04/01/22] seed@VM:~$ head -c 1M /dev/urandom > output.bin
[ 04/01/22] seed@VM:~$ ent output.bin
Entropy = 7.999831 bits per byte.

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

Chi square distribution for 1048576 samples is 245.25, and randomly
would exceed this value 65.82 percent of the times.

Arithmetic mean value of data bytes is 127.4845 (127.5 = random).
Monte Carlo value for Pi is 3.144527987 (error 0.09 percent).
Serial correlation coefficient is -0.001075 (totally uncorrelated = 0.0).
[ 04/01/22] seed@VM:~$ |
```

One major issue with /dev/random its nature is that it can lead to denial of service attacks. As a result, the below program utilizes /dev/urandom to generate 256 bits of random numbers.



```
Ray's MackbookPro Terminal
[ 04/01/22] seed@VM:~$ 39% | 7.1 GB | Q~|
```

```
#include <stdio.h>
#include <stdlib.h>
#define LEN 32 //bytes

int main()
{
    unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
    FILE *random = fopen("/dev/urandom", "r");
    fread(key, sizeof(unsigned char) * LEN, 1, random);
    fclose(random);
    printf("k = ");
    for (int i = 0; i < LEN; i++)
        printf("% .2x", key[i]);
    printf("\n");
    return 0;
}
```

After the running program, the result is

fe21f8e1f9e216c4e78b1906af5eb570c5732566c4d57cdcc996952bb146fd6f

```
[ 04/ 01/ 22] s e e d @ V M : ~ $ sudo vim task5.c
[ 04/ 01/ 22] s e e d @ V M : ~ $ gcc task5.c -o task5
[ 04/ 01/ 22] s e e d @ V M : ~ $ ./task5
k = fe21f8e1f9e216c4e78b1906af5eb570c5732566c4d57cdcc996952bb146fd6f
[ 04/ 01/ 22] s e e d @ V M : ~ $ |
```

## References

Understanding and Managing Entropy (n.d.). Whitewood Encryption Systems.

<https://www.blackhat.com/docs/us-15/materials/us-15-Potter-Understanding-And-Managing-Entropy-Usage-wp.pdf>

Stack overflow. What keeps draining entropy?.

<https://unix.stackexchange.com/questions/96847/what-keeps-draining-entropy>

random add\_input\_randomness.

[https://docs.huihoo.com/doxygen/linux/kernel/3.7/drivers\\_2char\\_2random\\_8c.html](https://docs.huihoo.com/doxygen/linux/kernel/3.7/drivers_2char_2random_8c.html)

/dev/random vs /dev/urandom and are they secure?.

[https://linuxhint.com/dev\\_random\\_vs\\_dev\\_urandom/](https://linuxhint.com/dev_random_vs_dev_urandom/)