Faculty of Engineering & Technology
Electrical & Computer Engineering Department

APPLIED CRYPTOGRAPHY

ENCS4320

Pseudo Random Number Generation Lab

Prepared by:

ShahEd Jamhour 1180654

Instructor: Dr. Ahmad Alsadeh

Section:1

April 2022

Table of Contents

Abstract:

Our goal in this lab is to learn why random number generation is not appropriate for generating secrets such as encryption keys. In addition, the lab will provide us with a standard way for generating pseudo-random numbers good for security reasons.

Task 1: Generate Encryption Key in a Wrong Way

As a starting point, we must begin with something that is random; otherwise, the outcome would be quite predictable.
With the **current time** seeded into the pseudo random number generator, the following program is run.

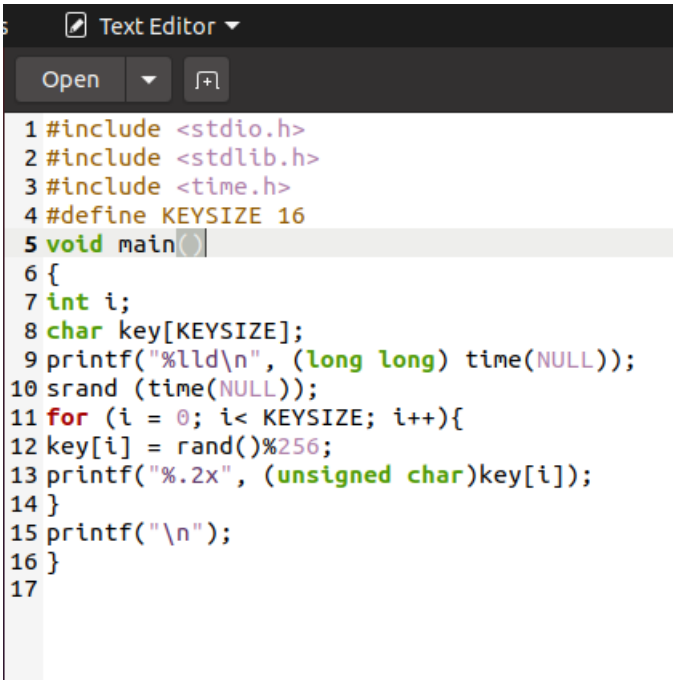


Figure 1: Task 1 code -" Generating a 128-bit encryption key

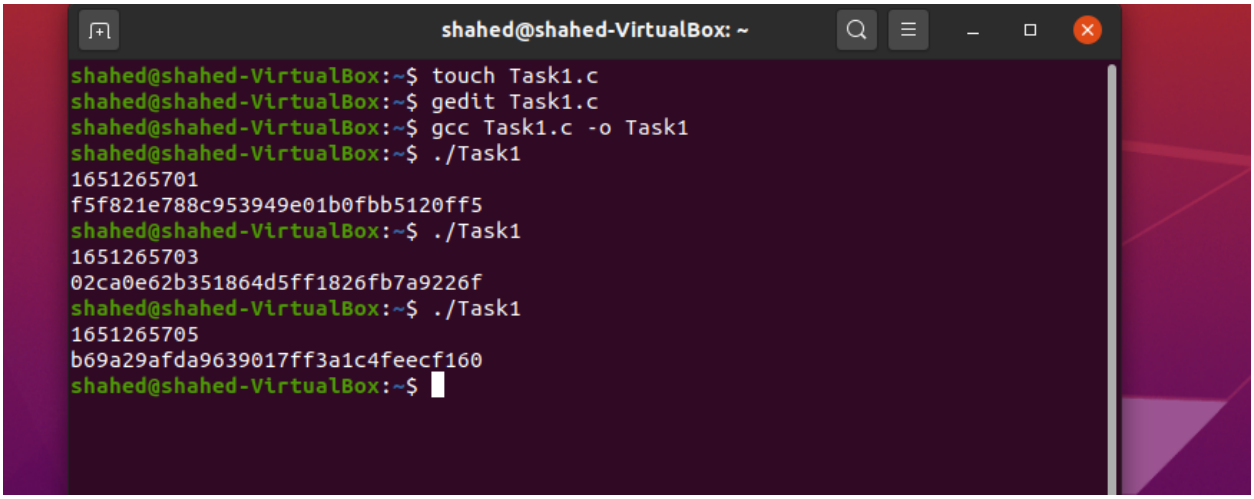


Figure 2:Executing the code

As can be seen, executing "RandomTime" several times always gives a different result

Since the **current time** is used as a random seed, the seed is always different every time the program runs.

Now after commenting line 10, it can be noticed that the generated random number remains unchanged in every run.

Figure 3:Executing the code after commenting SRAND

At first, we noticed that the random number generated and the number of seconds were different each time the program executed. This is because the function SRAND uses time (NULL) to set a different seed.
"Time (NULL)" returns the number (after conversion) of seconds since about midnight 1970-01-01, so that number changes every second.

But then, when srand (time (NULL)) is commented out, the default is a random number seed 0 because time is not seeded, so every time you run the program, the resulting random number is the same.

Task 2: Guessing the Key

First, we need to get the epoch of 2018-04-17 23:08:49 by:

```
date -d "2018-04-15 23:08:49" +%s
```

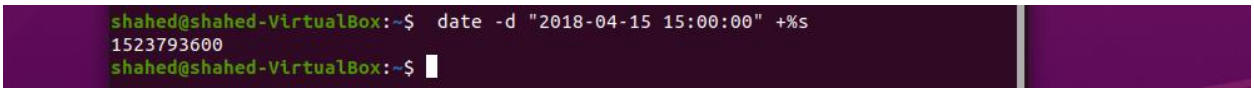


Figure 4:get the epoch of 2018-04-15

it returns 1523793600.

Then we list all possible random numbers generated by Task1.c within the two hours, This was don't by adding a loop before line 12 .

The file was named GuessingTime.c:

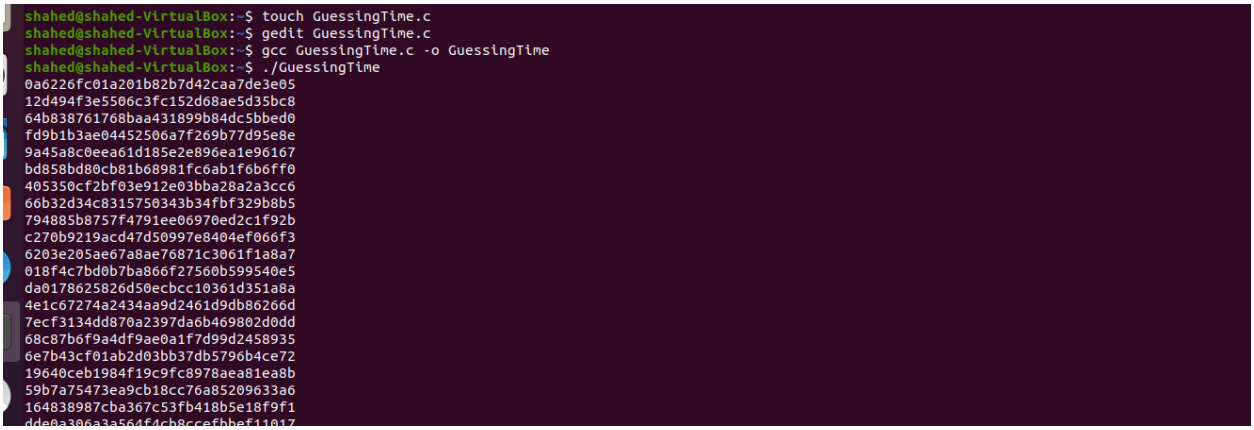


Figure 5:Generate all possible keys

The list of keys was obtained then redirect it to a txt file:

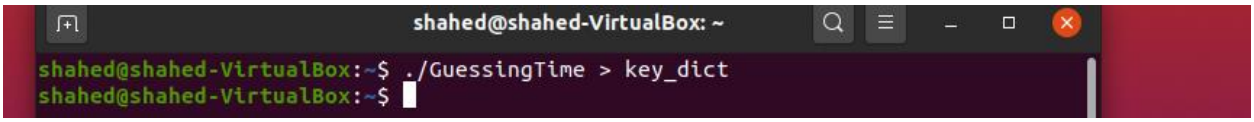


Figure 6:Redirect list of possible keys to txt file

Then brute-force method was used to crack the key from key_dict.txt:

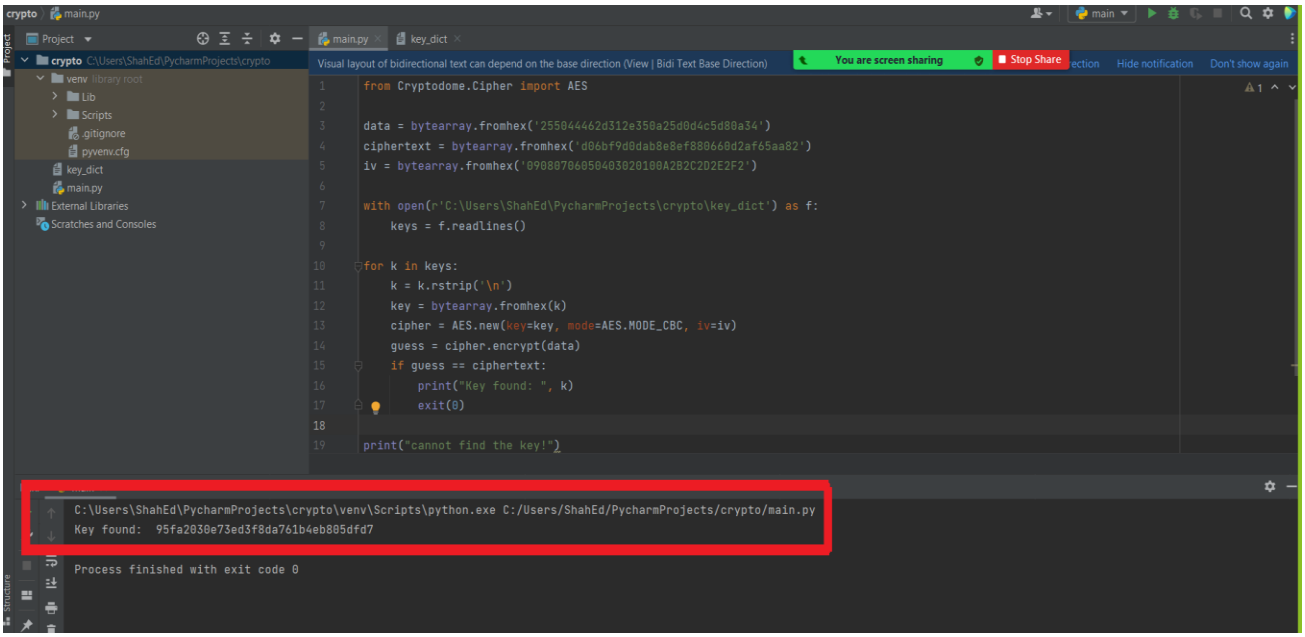


Figure 7:Guessing Key

Key was found = 95fa2030e73ed3f8da761bb4eb805dfd7

- Time as a seed value is not a true random number, and it is not advisable to generate random numbers with time.

Task 3: Measure the Entropy of Kernel

An entropy measure is used to determine randomness. It indicates how many random bits the system currently possesses. The following command tells us how much entropy the kernel currently possesses:

cat /proc/sys/kernel/random/entropy_avail

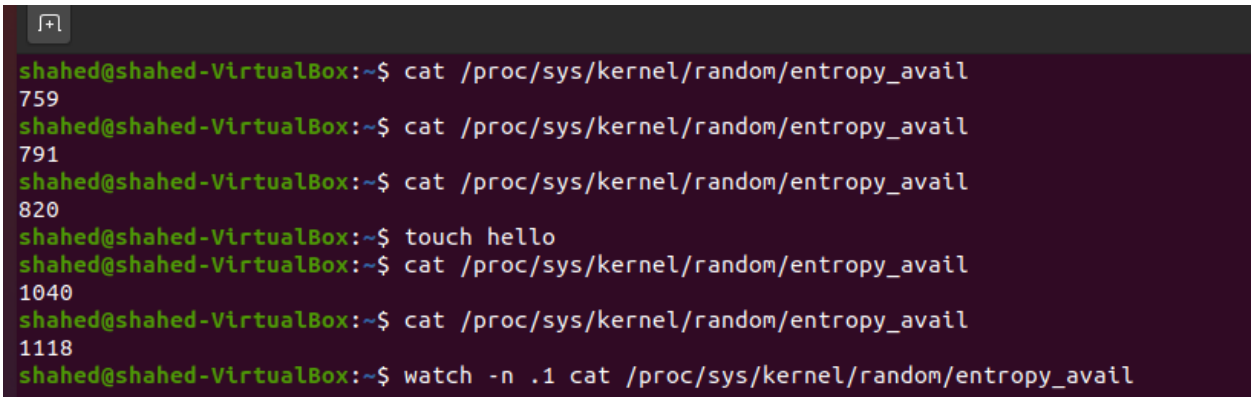


Figure 8:Measuring the Entropy of Kernel

It was found that every time you move the mouse, tap the keyboard, etc., it will cause a change in entropy.

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

Random data collected from the physical resources are stored in Linux's random pool and turned into pseudo-random numbers by using two devices. These two devices are /dev/random and /dev/urandom.

The main difference between /dev/random, /dev/urandom is that /dev/random blocks if the entropy is not indicating sufficient randomness, /dev/urandom does not block ever, even when the pseudo-random number generator is not fully seeded.

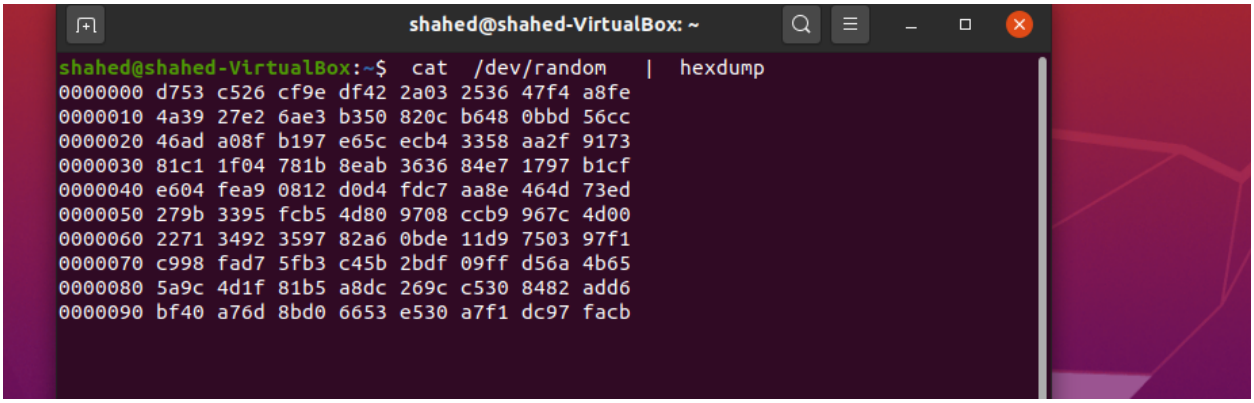


Figure 9:Get Pseudo Random Numbers from /dev/random

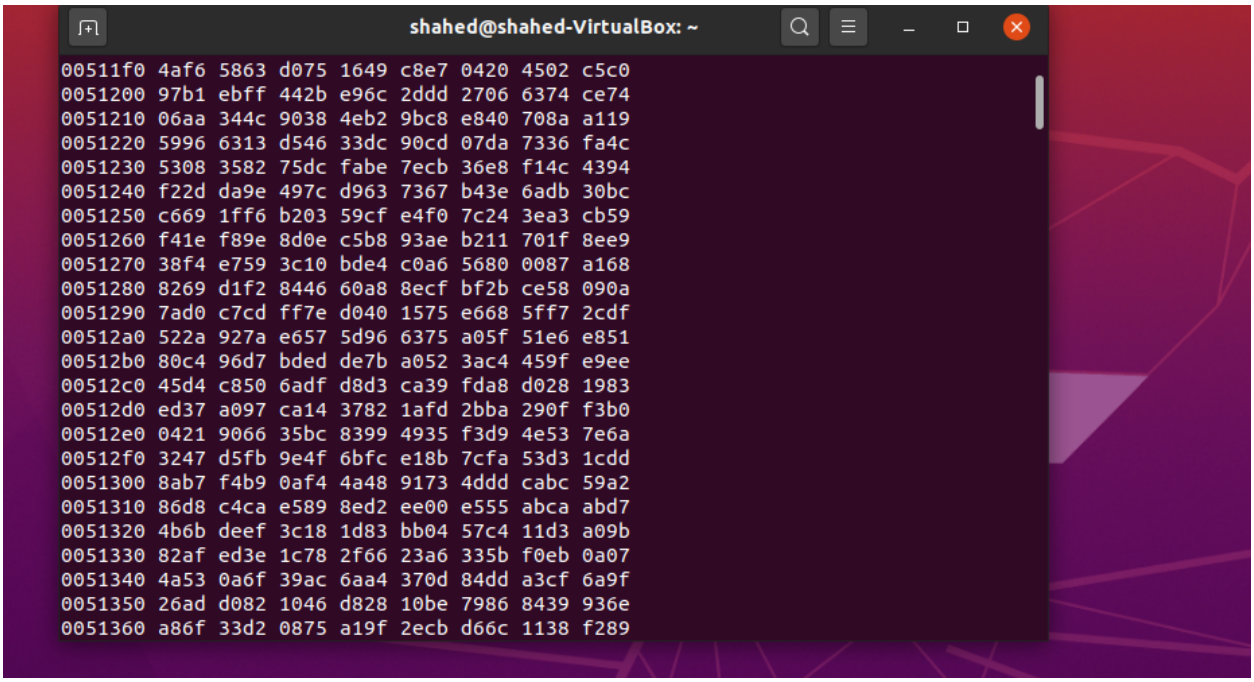
- Question: If a server uses /dev/random to generate the random session key with a client. Please describe how you can launch a Denial-Of-Service (DOS) attack on such a server?

  An attacker keeps requesting connection establishments, making /dev/random run out of entropy. At that point, random number generation stops working.

Task5: Get Random Numbers from /dev/urandom

urandom is a PRNG that's periodically re-seeded from the system's entropy pools when they contain enough estimated entropy.

Let's look at the behavior of /dev/urandom. again, we use cat to get pseudo-random numbers from this device.



The console will frantically print data., so we truncate the first 1 MB outputs into a file named output.bin.

Then we used **ent** to evaluate its information density:

- ent output.bin

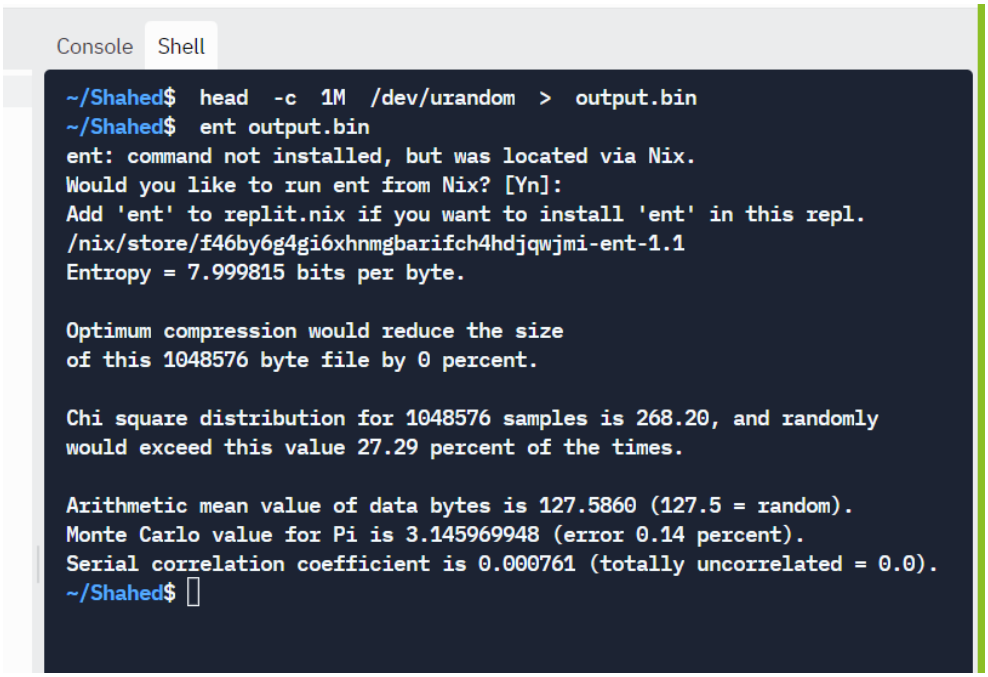


Figure 10:evaluate information density

The given code was modified to generate a 256-bit encryption key.

```c
/*   task5.c   */
#include <stdio.h>
#include <stdlib.h>
#define LEN 32 //  256  bits

void main()
{

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

```
shahed@shahed-VirtualBox:~$ touch task5.c
shahed@shahed-VirtualBox:~$ gedit task5.c
shahed@shahed-VirtualBox:~$ gcc task5.c -o task5
shahed@shahed-VirtualBox:~$ ./task5
d17ceff3430c3be494fc600185e75eaefb084a828cec58ab5c539bc545766569
shahed@shahed-VirtualBox:~$
```

Figure 11:Task5 execution

This is a true random number because it is read from/dev/urandom