CIS 628 SEED Crypto_Random_Number Lab Report

Introduction to Cryptography

Syracuse University

Fall 2022

Homework 2

Part 2, Question (b)

Saket Kiran Thombre

SU ID: 899913802

NetID: sthombre

Email: sthombre@syr.edu

Task 1: Generate Encryption Key in a Wrong Way:

Solution:

We are using a program that uses the current time as a seed for pseudo-random number generator:

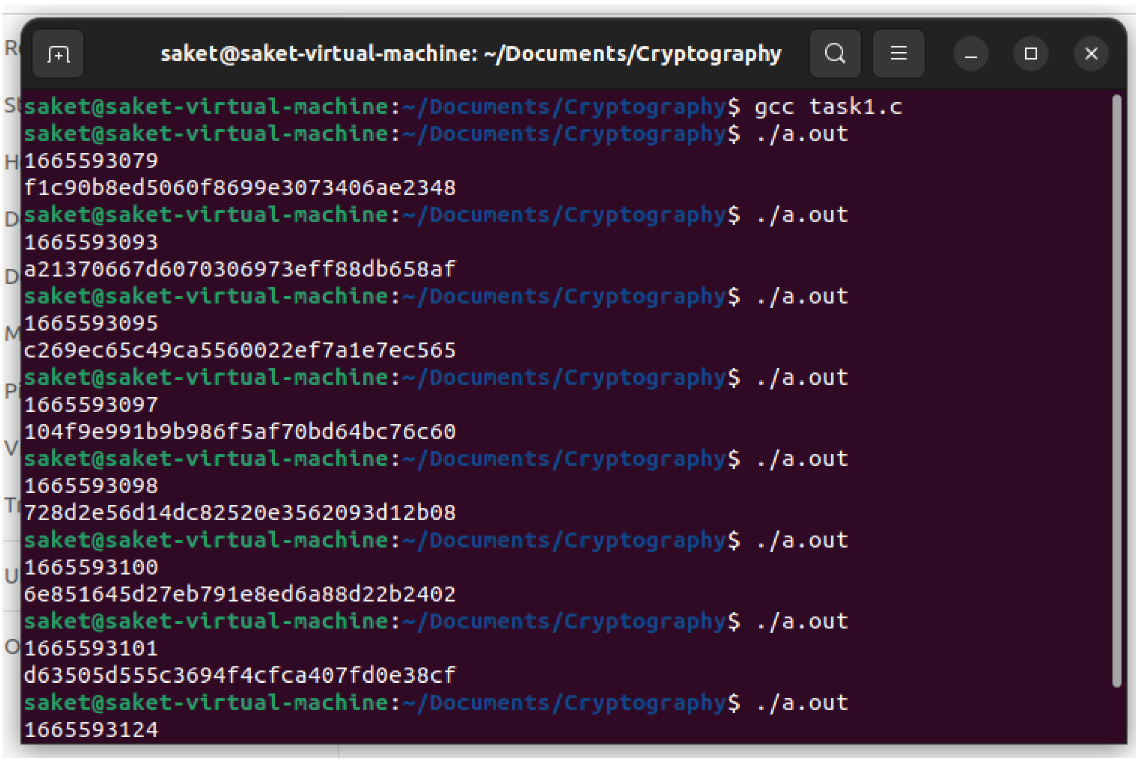


Figure 01: Show the output of program after running it multiple times.

In this output screenshot, we can see that the program task1.c is giving different sets of outputs after running multiple times. This occurs because we are considering the current time every time, we run the program. This will in turn generate a different seed in every iteration.

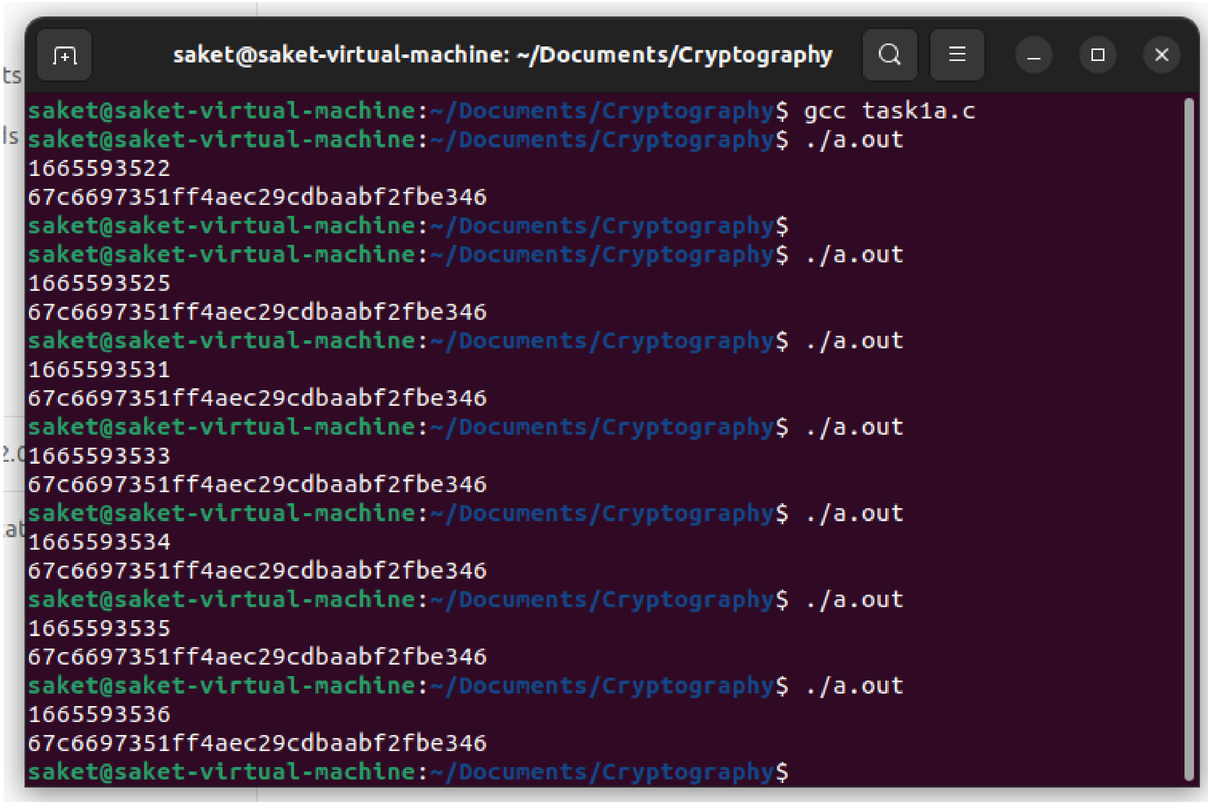


Figure 02: After commenting the line "***srand(time(NULL));***" We again run the program multiple times.

Now we can see that the numbers generated are the same in multiple iterations.

Role of *srand()* function and *time()*:

This function srand() is used to initialize the pseudo-random number generator by passing the argument seed to all subsequent rand() calls.

Syntax of the srand() function in C language is:

void srand(unsigned int seed);

Where the seed value determines a particular sequence of random numbers when calling the rand() function. If the seed value remains the same as in this case, then the rand() function will not generate new sequence numbers.

In our example, we remove/comment the srand() function, and hence all the calls to rand() function generates the same sequence.

The time() function returns the number of seconds passed since Jan 01 1971. It is usually a very long integer whose value changes every second.

This function pair of srand() and rand() is usually paired with time function() and use its return value.

Task 2: Guessing the Key

Solution:

Finding the time stamp of “2018-04-17 23:08:49” we get

```
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ date -d "2018-04-17 23:08:49" +%s
1524020929
saket@saket-virtual-machine:~/Documents/Cryptography/task2$
```

We get the value “1524020929”

Now we find the value of all possible numbers generated within 2 hours of timestamp generation.

We use task2a.c to generate these values.

Excerpts from the c program to generate all values.

```c
    //Now we create a for loop to generate and traverse through all the timestamps within 2 hour window
    for (time_t t = 1524020929 - 60 * 60 * 2; t < 1524020929; t++)
    {
        //initiating the srand() function
        srand(t);
        for (i = 0; i < KEYSIZE; i++)
        {
            seed[i] = rand() % 256;

            //Printing all the seed values
            printf("%.2x", (unsigned char)seed[i]);
        }
        printf("\n");
    }
```

We now store the output in a txt file called listTask2a.txt

```
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ gcc task2a.c
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ ./a.out > listTask2a.txt
saket@saket-virtual-machine:~/Documents/Cryptography/task2$
```

Further on, we now use brute force to crack the key from the generated txt file.

For this, we write a python script since we can directly use inbuilt “pycryptodome” library in the code “task2b.py”.

Excerpts from the python code:

```python
with open('listTask2a.txt') as f:
    possibleKeys = f.readlines()

for seed in keys:
#removing all trailing characters at the end of the srring
    seed = seed.rstrip('\n')

#Now we return a new bytearray object initialized from a seed string
    possibleKeys = bytearray.fromhex(seed)

#using AES cipher library to encrypt
    cipher = AES.new(possibleKeys=possibleKeys, mode=AES.MODE_CBC, iv=iv)

    guess = cipher.encrypt(data)

    if guess == ciphertext:
        print("The Key is:", seed)
        exit(0)

```

The python code traverses through all the values present in the txt file and encrypts them one by one. Subsequently it compares the values with the given key and provides the matching output.

```
saket@saket-virtual-machine: ~/Documents/Cryptography/t...
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ python3 task2b.py
The Key is: 95fa2030e73ed3f8da761b4eb805dfd7
saket@saket-virtual-machine:~/Documents/Cryptography/task2$
```

Thus, we find the key which is “95fa2030e73ed3f8da761b4eb805dfd7”

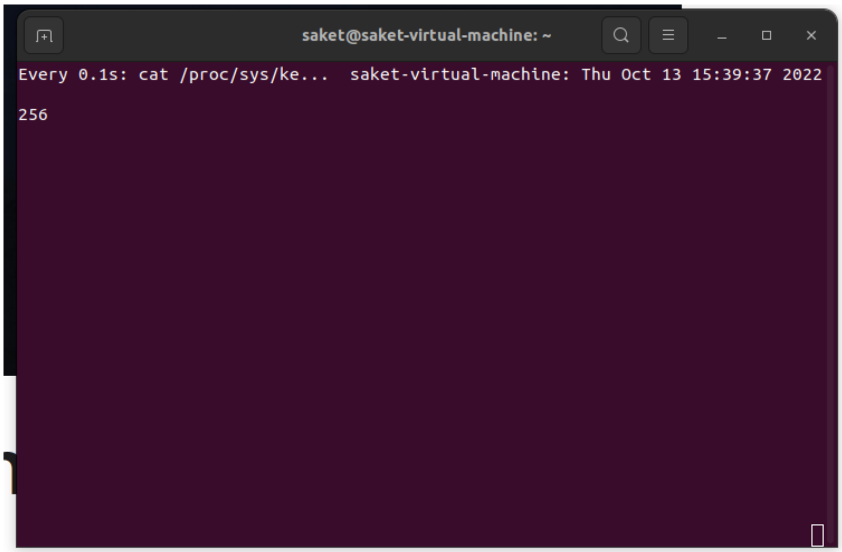
Hence, we can say that Bob has successfully generated the correct key using Alice’s file timestamp for her tax return PDF file.

Task 3: Measure the Entropy of Kernel

Solution:

We ran the following command Ubuntu, and we get the following output

"$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail"



Entropy is a way to measure the difficulty it needs to guess the strength of passwords using a secret value drawn from a pool.

A Linux machine pulls entropy from the last few timestamp digits fired by mouse, keyboards, disk drive or other hardware events. These new data points are then mixed up with the old points in the entropy pool resulting in even higher entropy. A keystroke or a mouse movement eventually adds up a couple of bits to the entropy in the pool, however it also tracks the number of bytes you pull from the function. In addition, as the network traffic rises, many network drivers may turn ON or OFF which in turn reduces the entropy of the pool.

There are many untapped entropy sources like temperature sensors, fan speed, audio inputs from surrounding etc. that can increase the entropy of the pool significantly. However, this can be only implemented in normal machines but not in virtual machines. For virtual machines, we need to use a pseudo-random number generator which will have a lesser entropy.

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

Solution:

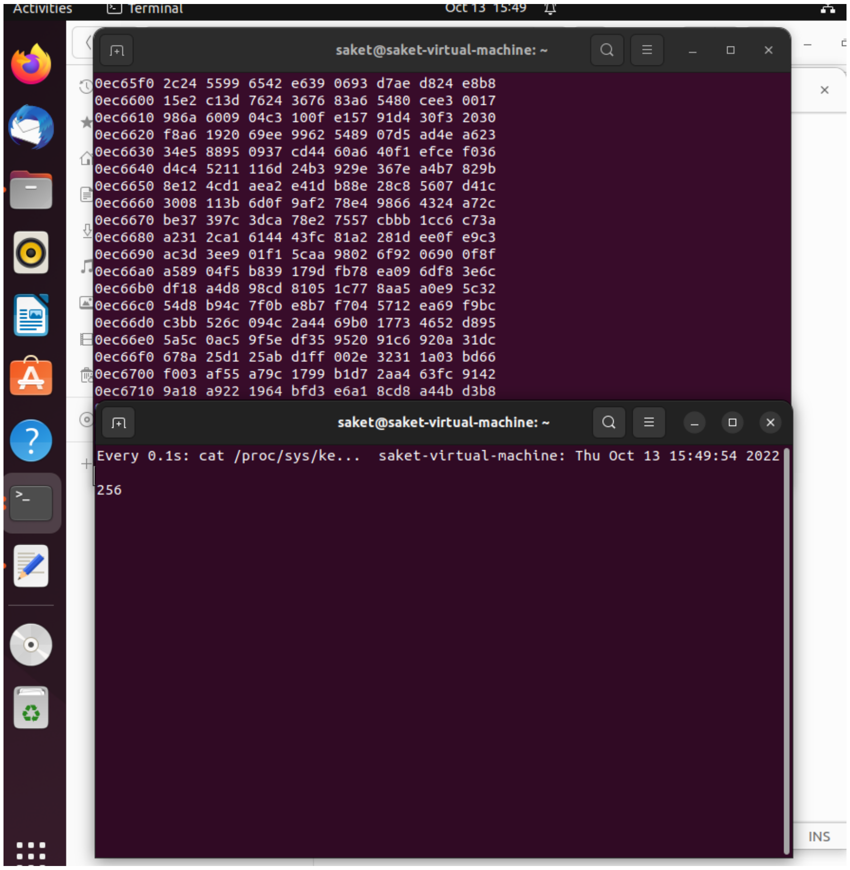
First, we run the command

"$ cat /dev/random | hexdump"

And then we run

“$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail”

We get the following output



When we run the hexdump command, we can see the entropy slowly decrease to zero. When we do a mouse movement, we can see temporarily that the entropy pool stops decreasing and start again.

When we stop the hexdump command, we can see that the entropy pool starts building up again and slowly increases as you do mouse movement and keystrokes.

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.

Solution:

As we know that /dev/random is used to generate random numbers with highest possible entropy, it also gets blocked when the pool gets exhausted.

The attack relies on this property of /dev/random. The attacker will request a huge amount of session IDs which will require private keys to be generated. These private keys are generated using the random numbers. This by effect reduces the pool entropy and may also stop the service.

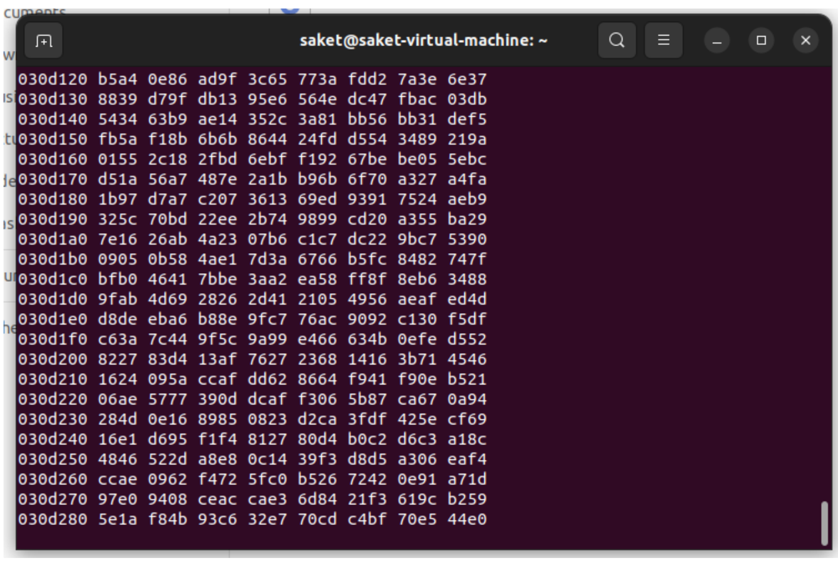
/dev/random will block all request till the entropy pool is filled again with the intention of providing random numbers with the highest possible entropy.

Task 5: Get Random Numbers from /dev/urandom

Solution:

First, we run the command for /dev/urandom

We get the following output



LINUX/UNIX systems have 2 main types of random number generators namely, /dev/random and /dev/urandom.

These generators gather external noise from hardware drivers and collect it in a entropy pool. This entropy pool is further used to create random numbers.

However, when there is a need to generate high number of random numbers, the pool might get exhausted and, in that case, /dev/random function will stop but /dev/urandom will reuse the pool and keep generating random numbers. This in turn reduces the entropy for /dev/urandom.

When we run a hexdump for /dev/urandom, the entropy starts reducing slowly, when we generate some mouse movements, the entropy seems to increase and become stable momentarily and once the mouse movements stop, its starts decreasing again.

In the second part, we generate a 1MB file consisting of pseudo-random numbers from /dev/urandom and we save it in output.bin file.

Then we run "ent" on this file to check its entropy.

We get the following output

```
saket@saket-virtual-machine: ~
saket@saket-virtual-machine:~$ head -c 1M /dev/urandom > output.bin
saket@saket-virtual-machine:~$ ent output.bin
Entropy = 7.999794 bits per byte.

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

Chi square distribution for 1048576 samples is 299.46, and randomly
would exceed this value 2.91 percent of the times.

Arithmetic mean value of data bytes is 127.5447 (127.5 = random).
Monte Carlo value for Pi is 3.141644064 (error 0.00 percent).
Serial correlation coefficient is -0.000962 (totally uncorrelated = 0.0).
saket@saket-virtual-machine:~$
```

"ent" performs produces standard output by performing various tests on a stream of bytes.

In our case, we look for the entropy, it is the information density of the contents of a file which are expressed as a number of bits per character.

In our case, we get the Entropy = 7.999794. this value of entropy is supposed to be very high. This means that the random numbers to be generated are going to be of a good value.

Along with entropy, "ent" function also provides optimum compression, chi square, arithmetic mean, Monte Carlo value and serial correlation coefficient.

In part 3

We are given a code snippet for 128 bits, and we need to convert it to 256 bits and generate random number.

So, we modify the code to get 256-bit number

```
task5c.c
~/Documents/Cryptography/task5
#include <stdio.h>
#include <stdlib.h>
#define LEN 32 // 256 bits

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);
```

After running the updated code, we get the following output