

Kamran Shaikh

1)

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

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

```
[10/14/19]seed@VM:~/Lab1$ ./encryption
1524020929
93c4ad2033792205fb2a29d1dc7c4f50
[10/14/19]seed@VM:~/Lab1$
```

Observation)

This program uses the current time as a seed for the pseudo random number generator. We can see from the screenshot above that it produces the time as the number of seconds since the Epoch. It then produces the random number key.

Explanation)

The srand() function is used to set the starting value for generating a sequence of pseudo-random integer values.

The time(NULL) function makes use of the computer's internal clock to control the choice of the seed. Since the time is always changing, the seed is forever changing. If the seed number remains the same, the sequence of numbers will be repeated for each run of the program.

For this program we get one encryption key.

2)

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

void main()
{
    for(int k = 1524020929; k>=1524013729;k--)
    {
        int i;
        char key[KEYSIZE];
        srand(k);
        for (i = 0; i< KEYSIZE; i++){
            key[i] = rand()%256;
            printf("%.2x", (unsigned char)key[i]);
        }
        printf("\n");
    }
}
```

692d25aa34de29699a79fee6b2888da1
ee55ee9ff3230b3fd20f0a8bb5cb2886
dbdd02aece9df7e4077ce33506740438
c18e4625050de9c78f6602a65d30a133
f26857a3087a798e2cac32f053b1937a
eac7b36829ab99ec734c4adf04485c2b
e7035da6a3d5c06f0943ed1cd3ca03ea
c4d89bd04416f36c3e71fc079c40ee1d
1f55c5ffbd1403ad1aff099109bf29aa
010555a823cd8362db61f9984f8794fc
bc9b9006058e484801775ff6f7b60d9f
b0aed0f2342c2919ac36db972768c12f
9ccb92724e91fc98cd93b2e60d7d1a4a
b0a2eb064c6ccecec03606292f9d5972
c5e5a38575cffffc39e4a1062ac341e6
dce18ac7051b131b38cdf000c7c3a730
01bce0f8850a6b8118594d5a4e6afe81
54bf88cbb83f31cb6a29afe1251e728e
854fd1b49e36e3f816ff4ad67d902b7e
ad13b5c13135b634f2ef7dd19325d13b
d248df44e0cd62d88fd27cdc6b849aba
adaa5f4a65568ddb1209bbbc58f02611
20a0f91bf20d85bd49f6dd413a09d4db
76f7c28d2af7eb0b0c7a58eab2719bfc
999dbdfb5f0346c76a7ca4f8857397be
53b5cc0209b7c0408e6ac0eb593a3a27
51ece46fabcefa221662b5402f936a43
30e00a61a142eef0fd9854532998ff8c
1c89ae5750c46d06a96a14caeea8174c
74bc3aaae878f1bdbd8998ba25db2492
d7062df4f066930951d07d474d05f7d5
0ecbf24d0d8f9c5c9ad4e51edcf79242

Observation)

We first edit the encryption file to gather a key for each time from the start of the encryption to two hours before the encryption. We gather this time from using the date command, starting at 1524020929 and ending at 1524013729. Here we pass in the time to the srand function to get a new random key. I then copied these encryption keys to an excel file so we can decrypt each one using Python.

Observation)

We gather our each encryption code from the csv file which gathered the codes from the c file above. We pass the csv file as a dataframe using pandas. Then we create a decipher algorithm using the encryption key and iv and pass the known ciphertext as a parameter. This will decrypt the plaintext Bob wants. We then compare the known plaintext from the lab to the plaintext for each encryption key and see if they are the same. If they are the same, we found out the correct encryption key.

3)

```
[10/14/19]seed@VM:~/Lab1$ cat /proc/sys/kernel/random/entropy_avail  
956
```

```
Every 0.1s: cat /proc/sys/... Mon Oct 14 13:24:01 2019  
1020
```

#### Observation)

The first screenshot shows the randomness of the key. It's measured using the entropy. This simply tells how many bits of random numbers the system currently has.

The second screenshot monitors the change of the entropy. It runs the program every .1 seconds.

#### Explanation)

When we moved the mouse, typed commands, or visited a website we saw that the entropy increased significantly. It especially increased when we visited a website.

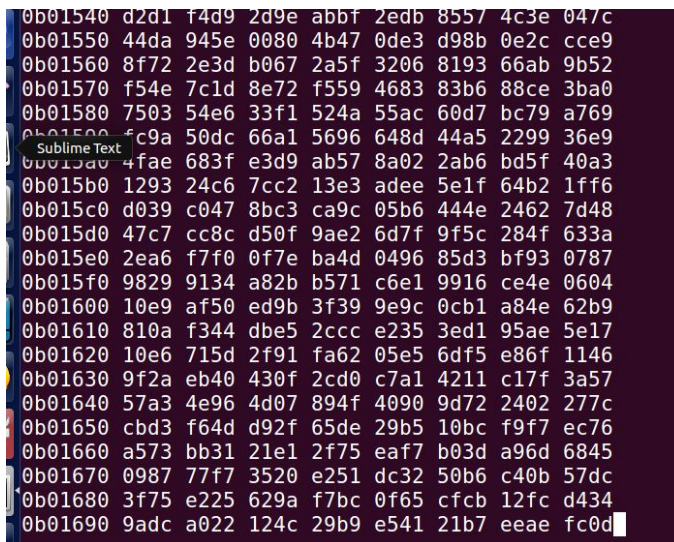
4)

```
[10/14/19]seed@VM:~$ cat /dev/random | hexdump
00000000 2fd0 57af b95c 8d9b 4120 2c8d a159 3a4c
00000010 6003 6c71 0e95 fde5 da53 bfd8 3deb f849
00000020 f30b f545 6d2f c887 18a1 ef55 8946 7f53
00000030 3de5 60c0 ab08 8681 3517 e84d 5f19 90e9
00000040 9dfe 92f1 e709 1d5a 9946 ebd2 d3de b26a
00000050 d8f9 bd29 568e f8a2 b0aa e621 786f 3b94
00000060 9d19 fb09 04f2 abfd 1e74 02db a10f 5e97
00000070 f6c7 3a39 99fb 5d55 a7b6 b7a2 ecab c8be
00000080 3a1c 71f5 083b 9d3c 27cf bfea eda5 4e01
00000090 507d 3d34 6e2a c719 37e4 2c6d 9945 53a7
000000a0 67b2 6a3d f8e1 ced5 d986 1b9b 6b3f 8ad4
000000b0 3908 fbda a56d 673e 666a 94cd 6bca 1e19
000000c0 ddda 5f1d ca2c c001 d5ec c4e6 7bfc 4e60
000000d0 0c6f 0f2f 87b8 564d 705f a5cb 6517 b165
000000e0 3221 70d5 0d4c 4c98 191e 4d61 6028 58f2
000000f0 07b5 0278 c1ff a2e9 2a46 5add f621 fd99
```

Observation)

This function observes the behavior of the `/dev/random` device. It continuously reads the pseudo random numbers from `/dev/random`. The output it then sent out to `hexdump` to make it easier to read.

5)

A screenshot of a hex dump showing random data from /dev/urandom. The data is displayed in a grid of hexadecimal values, with a 'Sublime Text' window title visible on the left. The values range from 0b01540 to 0b01690, showing a sequence of random bytes.

```
0b01540 d2d1 f4d9 2d9e abbf 2edb 8557 4c3e 047c
0b01550 44da 945e 0080 4b47 0de3 d98b 0e2c cce9
0b01560 8f72 2e3d b067 2a5f 3206 8193 66ab 9b52
0b01570 f54e 7c1d 8e72 f559 4683 83b6 88ce 3ba0
0b01580 7503 54e6 33f1 524a 55ac 60d7 bc79 a769
0b01590 fc9a 50dc 66a1 5696 648d 44a5 2299 36e9
0b015a0 4fae 683f e3d9 ab57 8a02 2ab6 bd5f 40a3
0b015b0 1293 24c6 7cc2 13e3 adee 5e1f 64b2 1ff6
0b015c0 d039 c047 8bc3 ca9c 05b6 444e 2462 7d48
0b015d0 47c7 cc8c d50f 9ae2 6d7f 9f5c 284f 633a
0b015e0 2ea6 f7f0 0f7e ba4d 0496 85d3 bf93 0787
0b015f0 9829 9134 a82b b571 c6e1 9916 ce4e 0604
0b01600 10e9 af50 ed9b 3f39 9e9c 0cb1 a84e 62b9
0b01610 810a f344 db5e 2ccc e235 3ed1 95ae 5e17
0b01620 10e6 715d 2f91 fa62 05e5 6df5 e86f 1146
0b01630 9f2a eb40 430f 2cd0 c7a1 4211 c17f 3a57
0b01640 57a3 4e96 4d07 894f 4090 9d72 2402 277c
0b01650 cbd3 f64d d92f 65de 29b5 10bc f9f7 ec76
0b01660 a573 bb31 21e1 2f75 eaf7 b03d a96d 6845
0b01670 0987 77f7 3520 e251 dc32 50b6 c40b 57dc
0b01680 3f75 e225 629a f7bc 0f65 cfc3 12fc d434
0b01690 9adc a022 124c 29b9 e541 21b7 eeae fc0d
```

Observation:

The screenshot shows the output of running `cat /dev/urandom | hexdump` command.

A screenshot of a C program source code. The code includes `<stdio.h>` and `<stdlib.h>`, defines a constant `LEN` as 32, and implements a `main` function that generates a random key using `malloc` and `fread` from `/dev/urandom`.

```
#include <stdio.h>
#include <stdlib.h>

#define LEN 32 // 128
int main(void)
{
    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);
    return 0;
}
```

Observation)

We change the len to 32 so that we can generate a 256 bit encryption key instead of a 128 encryption key.



```
10/14/19]seed@VM:~$ ent output.bin
Entropy = 7.999817 bits per byte.

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

Chi square distribution for 1048576 samples is 266.24,
and randomly
would exceed this value 30.15 percent of the times.

Arithmetic mean value of data bytes is 127.5205 (127.5
= random).
Monte Carlo value for Pi is 3.141781394 (error 0.01 per
cent).
Serial correlation coefficient is -0.001161 (totally un
correlated = 0.0).
10/14/19]seed@VM:~$
```