PSEUDO RANDOM NUMBER GENERATION LAB

Jasmine Joy (500924677)

Task 1: Generate Encryption Key in a Wrong Way

```
[02/07/20]seed@VM:~/.../Lab2$ cat homerandom.c
#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");

}

[02/07/20]seed@VM:~/.../Lab2$ gcc homerandom.c -o random
[02/07/20]seed@VM:~/.../Lab2$ ls
homerandom.c  random
[02/07/20]seed@VM:~/.../Lab2$ ./random
1581089821
4916b092c077acde2a2c7a8a5fb000bd
[02/07/20]seed@VM:~/.../Lab2$
```

```
[02/07/20]seed@VM:~/.../Lab2$ cat homerandom.c
#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");

}

[02/07/20]seed@VM:~/.../Lab2$ gcc homerandom.c -o random
[02/07/20]seed@VM:~/.../Lab2$ ls
homerandom.c  random
[02/07/20]seed@VM:~/.../Lab2$ ./random
1581089973
67c6697351ff4aec29cdbaabf2fbe346
[02/07/20]seed@VM:~/.../Lab2$
```

```
[02/22/20]seed@VM:~/Desktop$ ./random
1582386709
e1617a4158a47d33ec1dc79af0c90751
[02/22/20]seed@VM:~/Desktop$ ./random
1582386756
2bc063b1df5ce8a283b4c0e55dcd8548
[02/22/20]seed@VM:~/Desktop$ ./random
1582386757
e5fcf9960c98de97dfdcccc3b158ebee
[02/22/20]seed@VM:~/Desktop$
```

```
[02/22/20]seed@VM:~/Desktop$ ./random
1582386803
67c6697351ff4aec29cdbaabf2fbe346
[02/22/20]seed@VM:~/Desktop$ ./random
1582386806
67c6697351ff4aec29cdbaabf2fbe346
[02/22/20]seed@VM:~/Desktop$ ./random
1582386808
67c6697351ff4aec29cdbaabf2fbe346
[02/22/20]seed@VM:~/Desktop$
```

`srand()` is a pseudo-random generator that is used to randomize the seed. `srand(time(NULL))` uses the computer's internal clock to control the choice of the seed. Because time constantly changes, the seed also does.

So when the `srand(time(NULL))` was commented out, the seed number remained the same, and the sequence of numbers was repeated for each run.

Task 2: Guessing the Key

Script to generate and stores all the values of the dates to test for a likely key:

```
#!/bin/bash

start=$(date -d "2018-04-17 20:00:00" +%s)
end=$(date -d "2018-04-18 02:00:00" +%s)

while [[ $start -lt $end ]];
do
start=$[$start+1]
echo $start >> datesFile
done
```

```
[02/24/20]seed@VM:~/.../Lab2$ head datesFile
1524013730
1524013731
1524013732
1524013733
1524013734
1524013735
1524013736
1524013737
1524013738
1524013739
[02/24/20]seed@VM:~/.../Lab2$
```

```
% openssl enc ciphertype -e  -in plain.txt -out cipher.bin \
            -K  00112233445566778889aabbccddeeff \
            -iv 0102030405060708
```

Script developed to passing the dates through the openssl command:

```
#!/bin/bash

crosscheck="255044462d312e350a25d0d4c5d80a34"

for line in $(cat datesFile);do

openssl enc -aes-128-cbc -d -in ciphertext -out temp -K $line -iv "0908070605040302010 0A2B2$
openssl base64 -in temp -out temp001

hexvar=$(cat temp001);
plaintxt=$(cat plaintext);
#echo $hexvar

if [[ *$hexvar* == $plaintxt ]]; then
echo $line
break
fi

done
```

```
???[??????C??}c?(?]??)???#?.0F?????????-?????9??[02
 hexdump
0000000 0acc fc9f 5bdd f603 8c93 43f6 e182 637d
0000010 96ca f828 d35d 29dc e886 2390 2eba 4630
0000020 f1c0 e1bf 018c 1c12 e73e ef93 39e5 3ff9
0000030
```

I got the following error on runtime that I was unable to debug. Hence, I was unable to complete the task.

```
[02/24/20]seed@VM:~/.../Lab2$ ./myscript3.save
bad decrypt
3070891712:error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:wrong final block le
ngth:evp_enc.c:518:
bad decrypt
3070731968:error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:wrong final block le
ngth:evp_enc.c:518:
bad decrypt
```

Expected outcome was that the plaintext would be a string contained in decrypted hexdump generated.

Task 3: Measuring the Entropy of the Kernel

```
[02/23/20]seed@VM:~/Desktop$ cat /proc/sys/kernel/random/entropy_avail
1682
[02/23/20]seed@VM:~/Desktop$ cat /proc/sys/kernel/random/entropy_avail
1698
```

The `entropy_avail` field reads the input_pool.entropy_count.
On running `$watch -n .1 cat /proc/sys/kernel/random/entropy_avail` to observe the behaviour, we see the following:

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail          Sun Feb 23 18:23:55 2020

1775
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail          Sun Feb 23 18:24:13 2020

1822
```

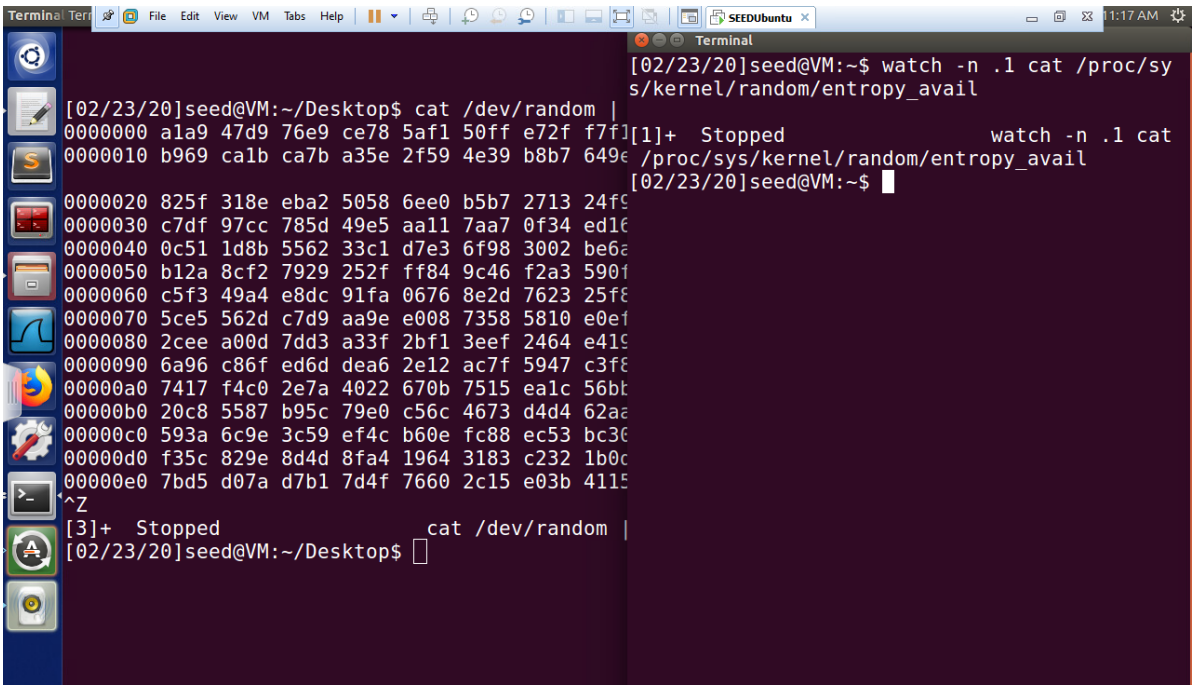
```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail          Sun Feb 23 18:24:31 2020

1871
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail          Sun Feb 23 18:24:48 2020

1918
```

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



When I do not move the mouse or type anything, the output is delayed. Then, when I randomly move the mouse, the output is generated more frequently. This is, it works on mouse movements and hard disk velocity.

`/dev/random` require waiting for the result as it uses an *entropy pool*, and random data may not be available at a given moment. Generating a DoS attack would exhaust `/dev/random`'s entropy. Under certain conditions we could implement a DoS attack by requesting a large number of session ID's.

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

Behaviour of **`/dev/urandom`**

```
01da000 c960 0ae2 6715 cb76 0aee e0c6 0dfe 5098
01da010 57be 3ddd 02fe a85a ade1 6b19 3a3b 5ac2
01da020 cd4c f8cf 37b5 1663 b31a 8cd5 2a42 044e
01da030 f5bb 2768 fbe7 1ea2 475e 4827 88f6 4464
01da040 85bd 8753 0a4e 68a8 4a78 7d32 f309 68b3
01da050 3a0d 9e71 cc66 d6ca 54ac 8af3 90d0 6a09
01da060 ca12 32c2 8655 5597 9b0d f19b 8c25 62ff
01da070 07ab ec45 d04a 77d8 10a3 455d 9182 831b
01da080 54b8 cdf5 e0d4 0986 cd7d 93b3 4503 d7b5
01da090 13d8 ec83 0ae1 9804 2f7d 2b54 3aae d97d
01da0a0 76c7 3727 2046 8818 38be 3049 2d8e 515b
01da0b0 4099 53bb b571 e5e4 d4b6 4da7 5ba1 2ce8
01da0c0 2e33 19af 7002 437c 9cc8 b197 e975 5a06
01da0d0 4e83 e53e b2ea c103 45e9 e15b faf6 d5a3
01da0e0 3688 9e16 1c53 e0d4 ace0 c864 fb0b 178b
01da0f0 37e6 5405 6e98 6357 347f e25f 247c de29
01da100 e02d 4be9 f943 fb4e 6184 f295 a947 dcd4
01da110 7543 f438 18bf fee9 164b 3c11 cb5d 986e
01da120 d17c 9ff0 88ef 2f3d a043 b8f0 3204 9b47
01da130 5ba7 e497 fbe6 61eb c4ed d959 c3e0 ee60
01da140 bc8d af15 5ee3 9570 eac5 9ca6 17ab dfd5
01da150 494c 4a4d b2c8 7f5c 30fe b3b1 a6bc b014
01da160 2872 55a4 7128 9e0c e114 2260 86da f8f6
01da170 336d 1b27 10c8 d22b 4514 8b3a 1ede 5fee
01da180 b3b4 5f64 4e2a 39c6 84a6 929c 23c4 7637
01da190 e3d6 7bad b1be dca2 7894 40c7 b8ae 2315
01da1a0 b361 5590 06ff 29fc 681e 944e 4574 ee37
^Z
[4]+  Stopped                 cat /dev/urandom | hexdump
[02/23/20]seed@VM:~/Desktop$
```

```
[02/24/20]seed@VM:~/.../Lab2$ head -c 1M /dev/urandom > output.bin
[02/24/20]seed@VM:~/.../Lab2$ ent output.bin
Entropy = 7.999821 bits per byte.

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

Chi square distribution for 1048576 samples is 259.56, and randomly
would exceed this value 40.90 percent of the times.

Arithmetic mean value of data bytes is 127.5940 (127.5 = random).
Monte Carlo value for Pi is 3.138210824 (error 0.11 percent).
Serial correlation coefficient is 0.001818 (totally uncorrelated = 0.0).
[02/24/20]seed@VM:~/.../Lab2$
```

Since the file compression is at the minimum (0%), and random data cannot be compressed.

Chi-squared value is between 10% and 90%.

Arithmetic mean value is greater than the threshold for random.

And the Monte Carlo value for Pi is only 11% off from the theoretical pi value. And would probably converge when the number of samples increases.

Correlation value is also approximately 0.0. Therefore, the generated is a good set of pseudo-random data behave like a set of random numbers.

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define LEN 64

char hexe(int n){
char array[16] = {'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'};
return array[n];
}

int main(){
unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
int a[LEN], i;
char en[LEN]="";

FILE* random = fopen("/dev/urandom", "r");
fread(key , sizeof(unsigned char)*LEN, 1, random);

fclose(random);

for(int i=0; i<LEN; i++){
a[i]=((int)key[i])%16;
}

for(int i=0;i<LEN;i++){
en[i]=hexe(a[i]);
}
printf("Encryption key:\n");

for(int i=0;i<LEN;i++){
printf("%c",en[i]);
}
printf("\n");
return 0;
}
```

The above code did not compile properly in Ubuntu. I got the following error:

```
[02/24/20]seed@VM:~/.../Lab2$ ./task5.c
./task5.c: line 6: syntax error near unexpected token `('
./task5.c: line 6: `char hexe(int n)'
[02/24/20]seed@VM:~/.../Lab2$
```

So, I ran it in an online C compiler, and below are the results of 3 runs:

```
Encryption key:
2262f3d235c3829402c159f28cc8f33ec0c2704abda65527256e8729e660a583

Encryption key:
118492df6b83b6ba1da5a88f7713cda691ce6eba6646b848209905d3cc4f3a61

Encryption key:
aae78e70d8556423fd553067c229a2995c8b018447017774c974e75ae7012172
```