ECE 571: Lab 2

> *Date: 2022-02-28*
>
> *Author: Zhaohui Yang*

Task 1: Generate Encryption Key in a Wrong Way

By compiling the sample source file and run the executable twice subsequently as follows,

```
gcc -o random random.c
# the first running
./random

# the second running
./random
```

different key values are obtained:

1. time: 1645518067; key: 524f397697036dadad0c6c45879fc96f
2. time: 1645518091; key: 8b69bb11eb5776070c301acc6da2a30b

Because function `time()` is used to get current time from 1970-01-01 00:00:00 +0000 (UTC), whose type is integer. In different running epochs, different current times (in second) are used as random seeds of the PC random generator, through the `srand()` function. Different time values naturally lead to different random key values.

With the ① line commented, results will be the same in different running epochs. In my PC, the result is 67c6697351ff4aec29cdbaabf2fbe346, because the same "default" random seed is designed.

Task 2: Guessing the Key

Strategy

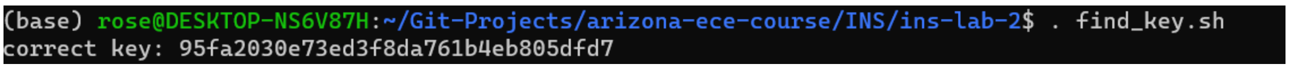
With the time "`2018-04-15 15:00:00`" (1523818800 in integer value) as benchmark, we can calculate the integer representation of the encryption time window [`2018-04-17 21:08:49`, `2018-04-17 23:08:49`] is [`1524013729`, `1524013729`]. Then our decryption strategy is as follows.

1. Use each time value $t \in [1524013729, 1524013729]$ as random seed, generate a 128-bit key k with the above program
2. Use the key k to encrypt the known plain, with corresponding cipher c generated
3. Compare the generated c with the known cipher, if they are the same, then the key is found

The attached source files `find_keys.sh`, `gene_keys.cpp` and `find_keys.py` are the core implementation of this strategy. See `find_key.sh` for details.

Result

7201 keys are generated and stored into `keys.txt`. Corresponding ciphers are also generated using AES-128-CBC algorithm, stored into `cipher.bin` and `cipher.txt`. The searched correct key is `95fa2030e73ed3f8da761b4eb805dfd7` as the figure below.



```
(base) rose@DESKTOP-NS6V87H:~/Git-Projects/arizona-ece-course/INS/ins-lab-2$ . find_key.sh
correct key: 95fa2030e73ed3f8da761b4eb805dfd7
```

Task 3: Measure the Entropy of Kernel

When running the below command to view how many "entropy" in the system initially,

```
cat /proc/sys/kernel/random/entropy_avail
```

the output is about "3566". Then after running command

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

and start other activities (mouse moving, keyboard beating, opening other programs, etc.) at the same time, the entropy got fluctuatingly increasing, e.g., 3644, 3648, ..., etc.

That phenomenon shows that varieties of events the OS accepted will affect the systematic noise, leading to the entropy available increasing.

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

Operation:

- First, run `watch` command to view available entropy intermediately.
- Second, run `cat /dev/random | hexdump` command to view generated random number in hexagon.

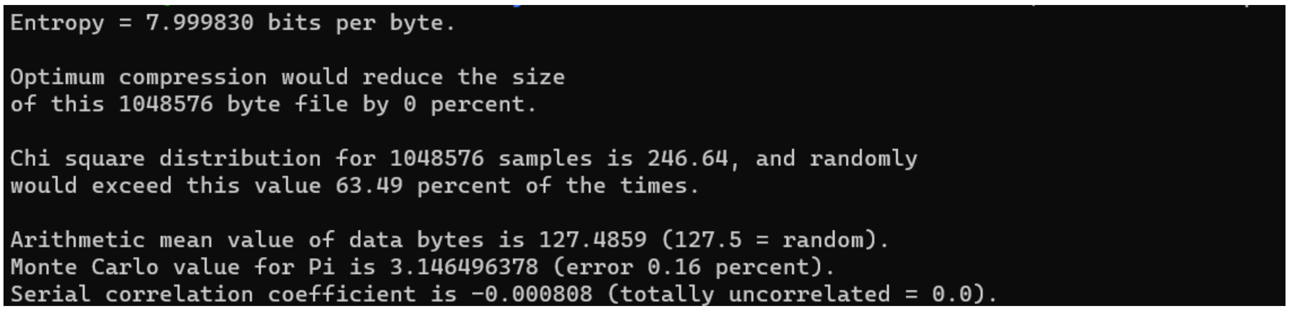
Result:

- A series of random values were rapidly generated and printed in the terminal, while the available entropy value rapidly decreased to near zero.
- Then without any human intervention, the entropy value stayed in fluctuation, while no any other random number was generated.
- Gain by moving my mouse and making keyboard typing, the available entropy rapidly increased, some new random numbers were generated and printed and the available entropy got rapidly consumed again, i.e., the random generating process got blocked gain eventually.
- The above phenomenon corresponds to our expectation. That means `/dev/random` generate random numbers via consuming available entropy, when the "available entropy" is consumed to near zero (or below some threshold), the random number generating process gets blocked until there is enough randomness in the system. Human intervention rapidly increases the randomness in system to increase available entropy, which helps activated the blocked generating process.

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

Randomness of `dev/urandom`

By running `head -c 1M /dev/urandom > output.bin && ent output.bin` command, the following output was obtained.



```
Entropy = 7.999830 bits per byte.

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

Chi square distribution for 1048576 samples is 246.64, and randomly
would exceed this value 63.49 percent of the times.

Arithmetic mean value of data bytes is 127.4859 (127.5 = random).
Monte Carlo value for Pi is 3.146496378 (error 0.16 percent).
Serial correlation coefficient is -0.000808 (totally uncorrelated = 0.0).
```

From the output it can be concluded that the randomness of these generated numbers is pretty good. For instance, the "entropy" is around 8 bits per byte, the optimum compression will almost not reduce the file's size, the "arithmetic mean value" is near to the random benchmark 127.5 and the "serial correlation coefficient" is small enough.

Generating a 256-bit key using `/dev/urandom`

Core code snippets after modification:

The source file `urandom.c` are attached in the submission folder.

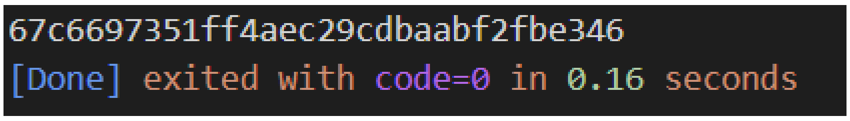
```c
#include <stdlib.h>
#include <stdio.h>
#define LEN 16 // 128 bits

int main(int argc, char const *argv[])
{
    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);

    int i;
    for (i = 0; i < LEN; i++){
        key[i] = rand() % 256;
        printf("%.2x", (unsigned char)key[i]);
    }

    return 0;
}
```

Result:



```
67c6697351ff4aec29cdbaabf2fbe346
[Done] exited with code=0 in 0.16 seconds
```

The result is the same as that in Task-1 in the situation of using default random seed. That is corresponding to the very principle of `/dev/urandom` and C random number library.