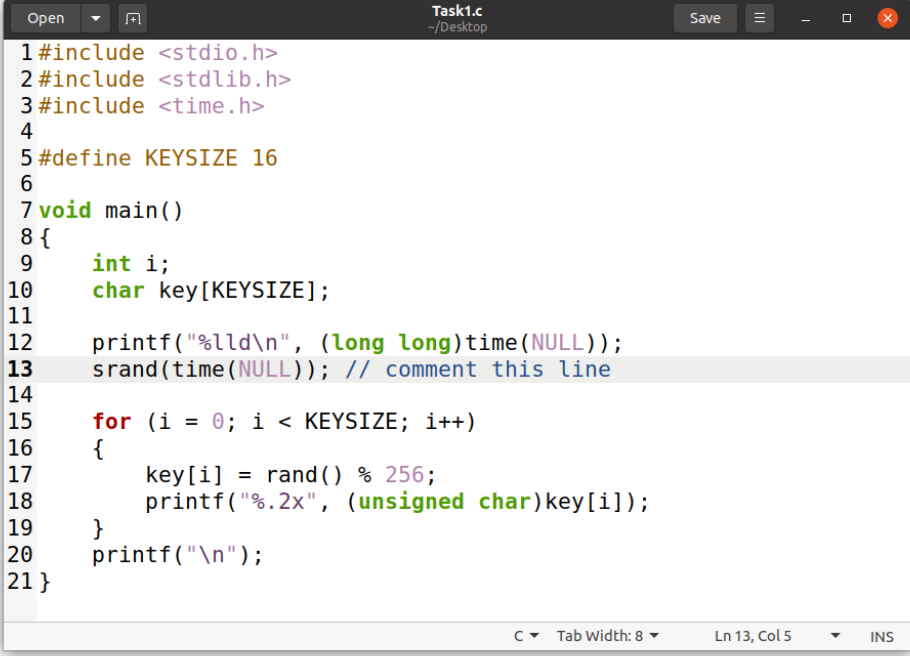


姓名：蔡佩蓉

學號：109511286

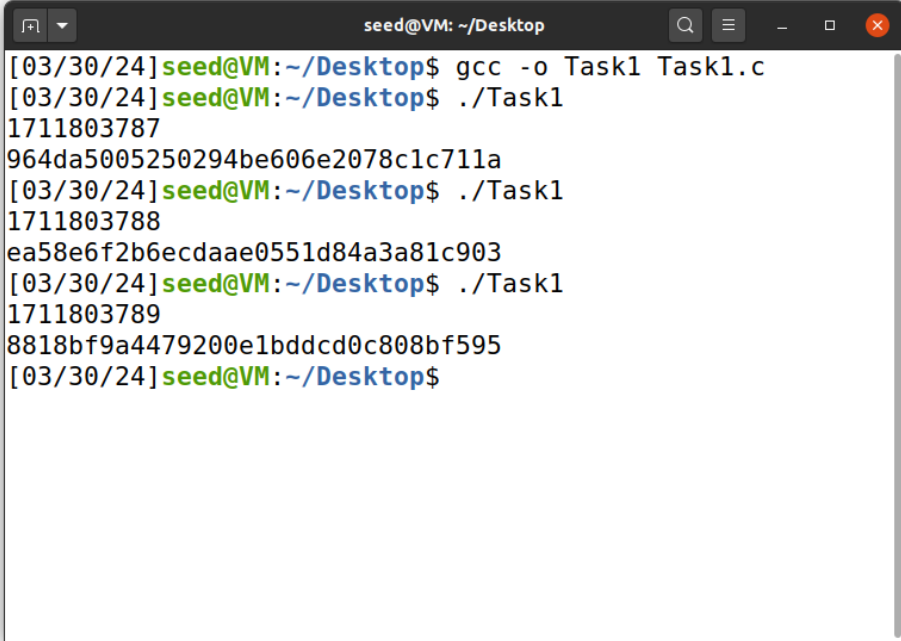
Pseudo Random Number Generation Lab (Lab1)

Task 1: Generate Encryption Key in a Wrong Way



```
1#include <stdio.h>
2#include <stdlib.h>
3#include <time.h>
4
5#define KEYSIZE 16
6
7void main()
8{
9    int i;
10   char key[KEYSIZE];
11
12   printf("%lld\n", (long long)time(NULL));
13   srand(time(NULL)); // comment this line
14
15   for (i = 0; i < KEYSIZE; i++)
16   {
17       key[i] = rand() % 256;
18       printf("%.2X", (unsigned char)key[i]);
19   }
20   printf("\n");
21}
```

With Line ① `srand(time(NULL))` included:



```
seed@VM: ~/Desktop
[03/30/24] seed@VM:~/Desktop$ gcc -o Task1 Task1.c
[03/30/24] seed@VM:~/Desktop$ ./Task1
1711803787
964da5005250294be606e2078c1c711a
[03/30/24] seed@VM:~/Desktop$ ./Task1
1711803788
ea58e6f2b6ecdaae0551d84a3a81c903
[03/30/24] seed@VM:~/Desktop$ ./Task1
1711803789
8818bf9a4479200e1bddcd0c808bf595
[03/30/24] seed@VM:~/Desktop$
```

The program prints the current time in seconds since the Epoch and generates a 128-bit encryption key by using the current time as the seed for the pseudo-random number generator.

Explanation:

`srand()` initializes the pseudo-random number generator (PRNG) with a seed value.

`time(NULL)` returns the current time in seconds since the Epoch.

By using the current time as the seed, `srand()` ensures that each time the program is executed, the seed value is different, leading to a different sequence of pseudo-random numbers. Thus, the generated encryption key varies with each execution, making it less predictable.

Without Line ① `srand(time(NULL))`:

A terminal window titled 'seed@VM: ~/Desktop' showing the execution of a program. The user runs 'gcc -o Task1 Task1.c' and then './Task1' three times. Each time, the program outputs the same two lines: '1711803840' and '67c6697351ff4aec29cdbaabf2fbe346'. This demonstrates that without seeding, the pseudo-random number generator produces the same sequence of numbers every time the program is run.

```
seed@VM: ~/Desktop
[03/30/24] seed@VM:~/Desktop$ gcc -o Task1 Task1.c
[03/30/24] seed@VM:~/Desktop$ ./Task1
1711803840
67c6697351ff4aec29cdbaabf2fbe346
[03/30/24] seed@VM:~/Desktop$ ./Task1
1711803841
67c6697351ff4aec29cdbaabf2fbe346
[03/30/24] seed@VM:~/Desktop$ ./Task1
1711803842
67c6697351ff4aec29cdbaabf2fbe346
[03/30/24] seed@VM:~/Desktop$
```

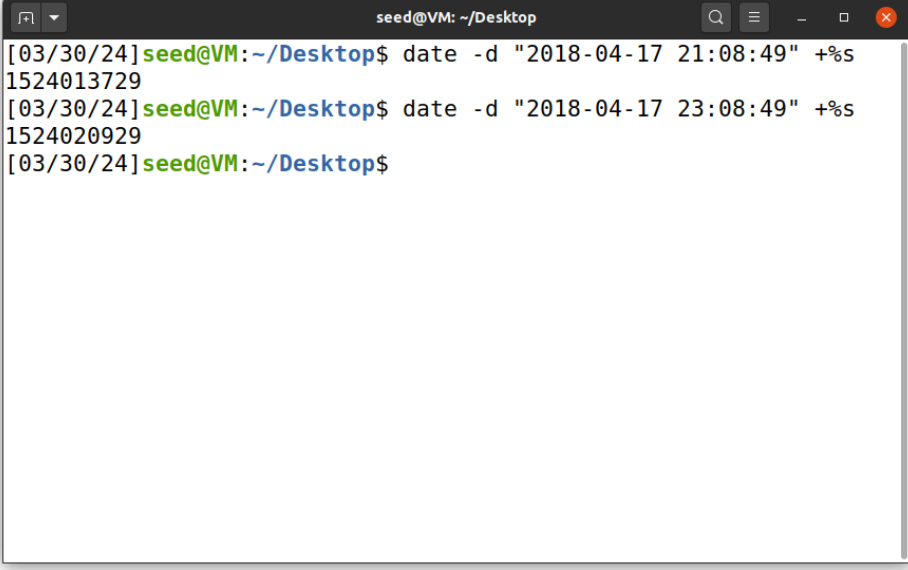
The program still prints the current time. However, the generated encryption key remains the same every time the program is executed without any noticeable change.

Without seeding the PRNG, the same seed value (likely 0) is used every time the program runs. Consequently, the sequence of pseudo-random numbers generated by `rand()` remains the same in each execution. Therefore, the encryption key generated will be the same every time, making it highly predictable and insecure.

In summary, `srand()` with `time(NULL)` as the seed ensures that the pseudo-random number generator starts with a different state each time the program runs, enhancing randomness and security in generating encryption keys.

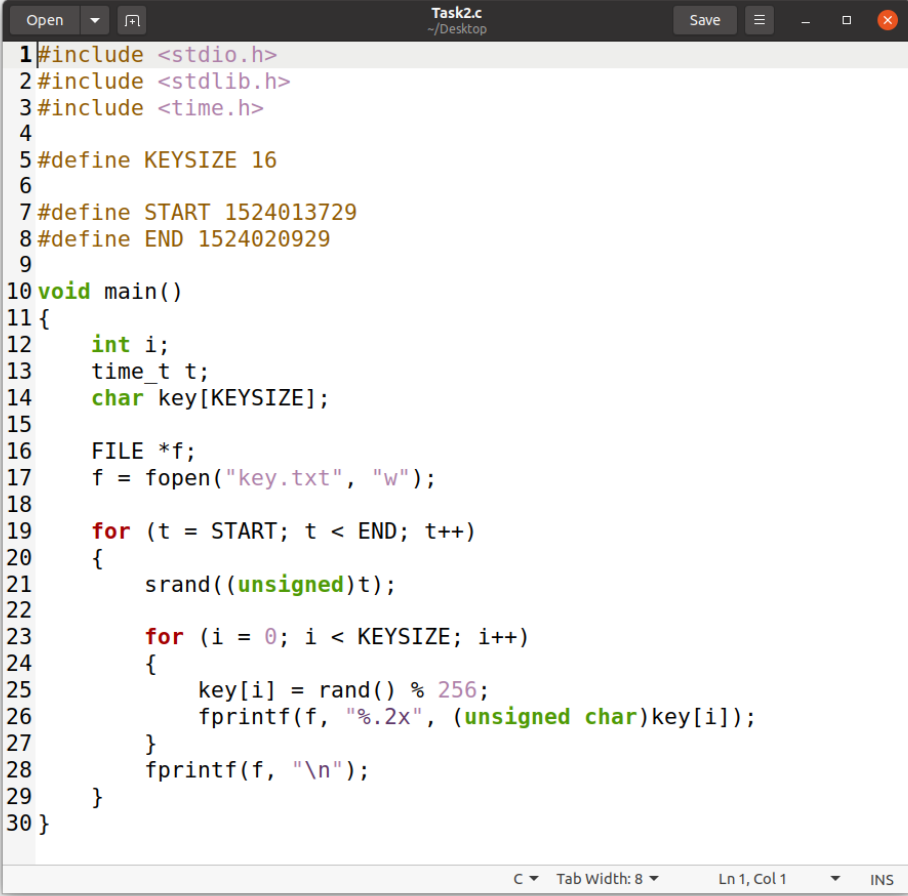
Task 2: Guessing the Key

Get the epoch of 2018-04-17 21:08:49 and 2018-04-17 23:08:49 (2 hours) by:



```
seed@VM: ~/Desktop
[03/30/24] seed@VM:~/Desktop$ date -d "2018-04-17 21:08:49" +%s
1524013729
[03/30/24] seed@VM:~/Desktop$ date -d "2018-04-17 23:08:49" +%s
1524020929
[03/30/24] seed@VM:~/Desktop$
```

Then we list all possible random numbers generated by Tack2.c (output: key.txt) and use a brute-force method to crack the key from key.txt



```
Task2.c
~/Desktop

1#include <stdio.h>
2#include <stdlib.h>
3#include <time.h>
4
5#define KEYSIZE 16
6
7#define START 1524013729
8#define END 1524020929
9
10void main()
11{
12    int i;
13    time_t t;
14    char key[KEYSIZE];
15
16    FILE *f;
17    f = fopen("key.txt", "w");
18
19    for (t = START; t < END; t++)
20    {
21        srand((unsigned)t);
22
23        for (i = 0; i < KEYSIZE; i++)
24        {
25            key[i] = rand() % 256;
26            fprintf(f, "%.2x", (unsigned char)key[i]);
27        }
28        fprintf(f, "\n");
29    }
30}
```

```
Open Task2.py ~/Desktop Save
1 from Crypto.Cipher import AES
2
3 plaintext = bytes.fromhex("255044462d312e350a25d0d4c5d80a34")
4 ciphertext = bytes.fromhex("d06bf9d0dab8e8ef880660d2af65aa82")
5 IV = bytes.fromhex("09080706050403020100A2B2C2D2E2F2")
6
7 with open('key.txt', 'r') as f:
8     keys = f.readlines()
9
10 for k in keys:
11     key = bytes.fromhex(k.strip())
12     cipher = AES.new(key, AES.MODE_CBC, IV)
13     encrypted = cipher.encrypt(plaintext)
14     if ciphertext == encrypted[:len(ciphertext)]:
15         print("Match found")
16         print("key:", k.strip())
17         print("Ciphertext:", ciphertext.hex())
18         print("Encrypted:", encrypted.hex())
```

```
seed@VM: ~/Desktop
[03/30/24] seed@VM:~/Desktop$ date -d "2018-04-17 21:08:49" +%s
1524013729
[03/30/24] seed@VM:~/Desktop$ date -d "2018-04-17 23:08:49" +%s
1524020929
[03/30/24] seed@VM:~/Desktop$ gcc -o Task2 Task2.c
[03/30/24] seed@VM:~/Desktop$ ./Task2
[03/30/24] seed@VM:~/Desktop$ python3 Task2.py
Match found
key: 95fa2030e73ed3f8da761b4eb805dfd7
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
Encrypted: d06bf9d0dab8e8ef880660d2af65aa82
[03/30/24] seed@VM:~/Desktop$
```

Task 3: Measure the Entropy of Kernel

Running the command:

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

allows us to monitor the entropy available in the kernel in real-time.

Moving the mouse cursor, clicking the mouse and typing on the keyboard increases the entropy significantly (can be observed clearly). On the other hand, read a large file and visit a website didn't show obvious increase of the entropy (as considered the increase entropy may be the effect of keyboard typing and mouse movement).

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

Running the command `cat /dev/random | hexdump` continuously reads pseudo-random numbers from `/dev/random` and displays them using `hexdump`.

Initially, the entropy level in the kernel may be high, allowing `/dev/random` to generate pseudo-random numbers without blocking. However, as the pseudo-random numbers are continuously read from `/dev/random`, the entropy level gradually decreases. Eventually, if there is no additional source of entropy (such as mouse movement or keyboard typing), the entropy level continues to decrease until certain values (may reach zero), causing `/dev/random` to block. This means that the `cat` command will pause and wait until more entropy is collected before continuing to output random numbers.

When you randomly move your mouse, the entropy level in the kernel increases due to the additional source of randomness. As a result, `/dev/random` is able to replenish its entropy pool, allowing the `cat` command to continue generating pseudo-random numbers without blocking.

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.

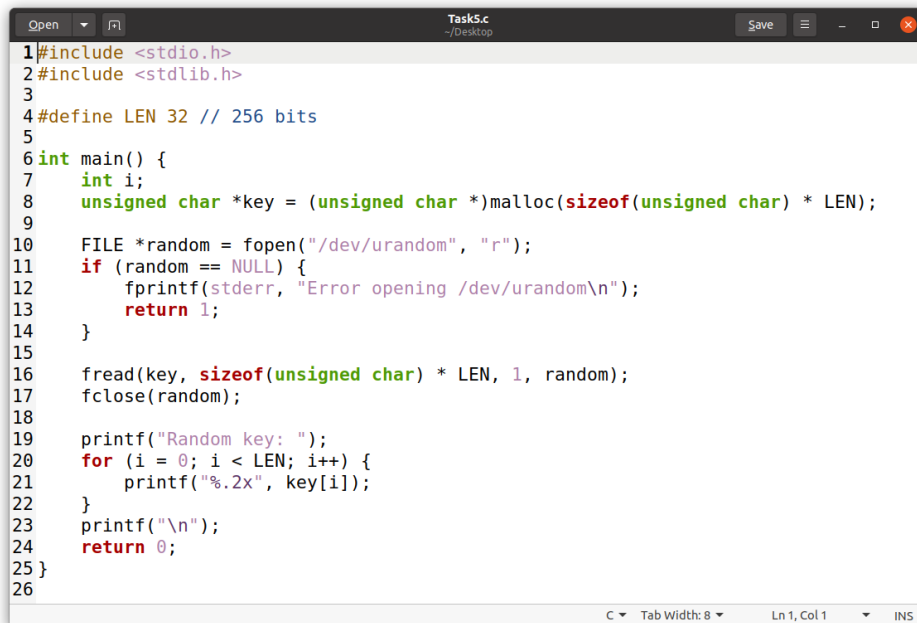
Using `/dev/random` to generate random session keys opens the server to a potential Denial-of-Service (DoS) attack. An attacker could flood the server with connection requests that require the generation of session keys. As `/dev/random` relies on the entropy pool, which may deplete over time, the server could become unable to generate new session keys when the entropy runs out. Consequently, legitimate clients attempting to establish connections with the server would be unable to do so, leading to a denial of service. By repeatedly initiating connection requests and exhausting the server's entropy pool, the attacker could sustain the DoS attack, disrupting the server's normal operation and preventing legitimate clients from accessing its services.

Task 5: Get Random Numbers from /dev/urandom

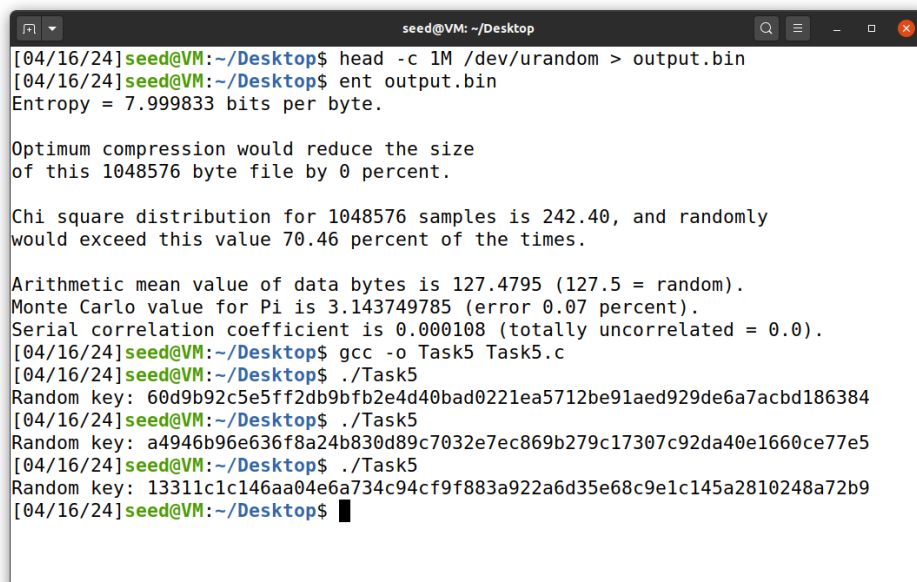
Run the command:

```
cat /dev/urandom | hexdump
```

It keeps printing out random numbers (even if the mouse stops or moves). We truncate the first 1 MB outputs into a file named `output.bin`, then use `ent` to evaluate its information density.



```
1#include <stdio.h>
2#include <stdlib.h>
3
4#define LEN 32 // 256 bits
5
6int main() {
7    int i;
8    unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
9
10    FILE *random = fopen("/dev/urandom", "r");
11    if (random == NULL) {
12        fprintf(stderr, "Error opening /dev/urandom\n");
13        return 1;
14    }
15
16    fread(key, sizeof(unsigned char) * LEN, 1, random);
17    fclose(random);
18
19    printf("Random key: ");
20    for (i = 0; i < LEN; i++) {
21        printf("%.2x", key[i]);
22    }
23    printf("\n");
24    return 0;
25 }
26
```



```
seed@VM: ~/Desktop
[04/16/24]seed@VM:~/Desktop$ head -c 1M /dev/urandom > output.bin
[04/16/24]seed@VM:~/Desktop$ ent output.bin
Entropy = 7.999833 bits per byte.

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

Chi square distribution for 1048576 samples is 242.40, and randomly
would exceed this value 70.46 percent of the times.

Arithmetic mean value of data bytes is 127.4795 (127.5 = random).
Monte Carlo value for Pi is 3.143749785 (error 0.07 percent).
Serial correlation coefficient is 0.000108 (totally uncorrelated = 0.0).
[04/16/24]seed@VM:~/Desktop$ gcc -o Task5 Task5.c
[04/16/24]seed@VM:~/Desktop$ ./Task5
Random key: 60d9b92c5e5ff2db9bfb2e4d40bad0221ea5712be91aed929de6a7acbd186384
[04/16/24]seed@VM:~/Desktop$ ./Task5
Random key: a4946b96e636f8a24b830d89c7032e7ec869b279c17307c92da40e1660ce77e5
[04/16/24]seed@VM:~/Desktop$ ./Task5
Random key: 13311c1c146aa04e6a734c94cf9f883a922a6d35e68c9e1c145a2810248a72b9
[04/16/24]seed@VM:~/Desktop$
```

It looks random in most measures. Use `/dev/urandom` to generate a 256-bit random number as a session key. No denial of service occurred during this process. The difference between the random numbers generated each time was also very large, and the quality of the random numbers was also very high.