CPS633 Section 07 Fall2021

Lab 05 Report

Pseudo Random Number Generation Lab

Name: Tusaif Azmat (group leader)
Student#: 500660278.
And
Name: Ankit Sodhi
Student#: 500958004

Group 04.

CPS 633 - Lab 5 Report

Pseudo Random Number Generation Lab

2 Lab Tasks:

2.1 Task 1: Generate Encryption Key in a Wrong Way

compile time_random.c and run

```
gcc time_random.c -o time_random
./time_random
```

```
#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));        ①

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

```
$ ./time_random
1590723119
bcd977367a0d4c382412d098af250e97
```

```
$ ./time_random
1590723124
f3f247b0dc8b63d9f759759828a02611
$ ./time_random
1590723128
2b6b53f258245cf6a5ee37c401745f9f
$ ./time_random
1590723130
31446f823d9e255fdbdff9b59912dfce
$ ./time_random
1590723226
5f30338736751df161a0d1be697f2b14
```

As you can see above, it always gives different results: Because it uses the current time as a random seed to generate random numbers, which guarantee the seed is always different in each run.

Note: When comment out the line srand(time(NULL));, recompile it and run, the numbers generated are the same now:

```
$ ./time_random
1590723565
67c6697351ff4aec29cdbaabf2fbe346
$ ./time_random
1590723567
67c6697351ff4aec29cdbaabf2fbe346
```

```
$ ./time_random
1590723568
67c6697351ff4aec29cdbaabf2fbe346
$ ./time_random
1590723569
67c6697351ff4aec29cdbaabf2fbe346
$ ./time_random
1590723571
67c6697351ff4aec29cdbaabf2fbe346
```

2.2 Task 2: Guessing the Key

Get the epoch of `2018-04-17 23:08:49` by:

```
date -d "2018-04-17 23:08:49" +%s
```

It returns **1524020929.**
Then we list all possible random numbers generated by time_random.c **within the two hours** by adding a loop before line 12 in it as time_guess.c:

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

void main()
{
    int i;
    char key[KEYSIZE];
    for (time_t t = 1524020929 - 60 * 60 * 2; t < 1524020929; t++) // within 2h window
    {
        srand(t);
        for (i = 0; i < KEYSIZE; i++)
        {
            key[i] = rand() % 256;
            printf("%.2x", (unsigned char)key[i]);
        }
        printf("\n");
    }
}
```

get the list:

```
gcc time_guess.c -o time_guess
time_guess > key_dict.txt
```

Use a brute-force method to crack the key from `key_dict.txt` as guess_key.py:

```
#!/usr/bin/python3
from Crypto.Cipher import AES

data = bytearray.fromhex('255044462d312e350a25d0d4c5d80a34')
ciphertext = bytearray.fromhex('d06bf9d0dab8e8ef880660d2af65aa82')
iv = bytearray.fromhex('09080706050403020100A2B2C2D2E2F2')

with open('key_dict.txt') as f:
    keys = f.readlines()

for k in keys:
    k = k.rstrip('\n')
    key = bytearray.fromhex(k)
    cipher = AES.new(key=key, mode=AES.MODE_CBC, iv=iv)
    guess = cipher.encrypt(data)
    if guess == ciphertext:
        print("find the key:", k)
        exit(0)

print("cannot find the key!")
```

It finds out the key:

```
$ chmod u+X guess_key.py
$ guess_key.py
find the key: 95fa2030e73ed3f8da761b4eb805dfd7
```

2.3 Task 3: Measure the Entropy of Kernel

Let us monitor the change of the entropy by running the above command via watch, which executes a program periodically, showing the output in full screen. The following command runs the cat program every 0.1 second.

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

Note: When I move the mouse or type something, the value increases fast. Every time it decreases, one line of new random numbers appears. So we can say

that /dev/random consumes the available entropy produced by user's behaviors to generate new random numbers.

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

```
cat /dev/random | hexdump
```

It generates output slowly and almost gets stuck when I keep unmoved. Monitor the entropy in another terminal :

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.

Attackers keep asking for establishing connections, which makes the server run out of the available entropy for /dev/random. Then the random number generator is blocked.

Note: When I move the mouse or type something, the value increases fast. Every time it decreases, one line of new random numbers appears. So we can say
that /dev/random consumes the available entropy produced by user's behaviors to generate new random numbers.

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

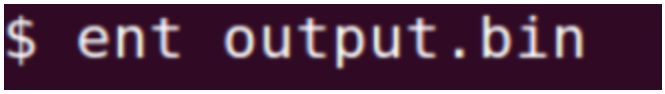
```
cat /dev/urandom | hexdump
```

It keeps printing out random numbers. We truncate the first 1 MB outputs into a file named output.bin:

```
head -c 1M /dev/urandom > output.bin
```

Then use ent to evaluate its information density:

```
ent output.bin
```



```
Entropy = 7.999827 bits per byte.

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

Chi square distribution for 1048576 samples is 251.58,
and randomly
would exceed this value 54.87 percent of the times.

Arithmetic mean value of data bytes is 127.4018 (127.5
= random).
Monte Carlo value for Pi is 3.153065312 (error 0.37 per
cent).
Serial correlation coefficient is 0.000007 (totally unc
orrelated = 0.0).
```

See the documentation. It looks random in most measures.

Use **/dev/urandom** to generate a 256-bit random number as a session key by read_random_key.c:

```c
#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);
    printf("k = ");
    for (int i = 0; i < LEN; i++)
        printf("%.2x", key[i]);
    printf("\n");
    return 0;
}
```

Compile:

```
gcc read_random_key.c -o read_random_key
```

Run:

```
$ gcc read_random_
key.c -o read_random_key
$ ./read_random_ke
y
k = dc8629b5005e34e22a17e678475fa548267eadee87fea9a0ff0
f2e655e740d59
```

Key= dc8629b5005e34e34e22a17e678475fa548267eadee87fea9a0ff0fe655e740d59