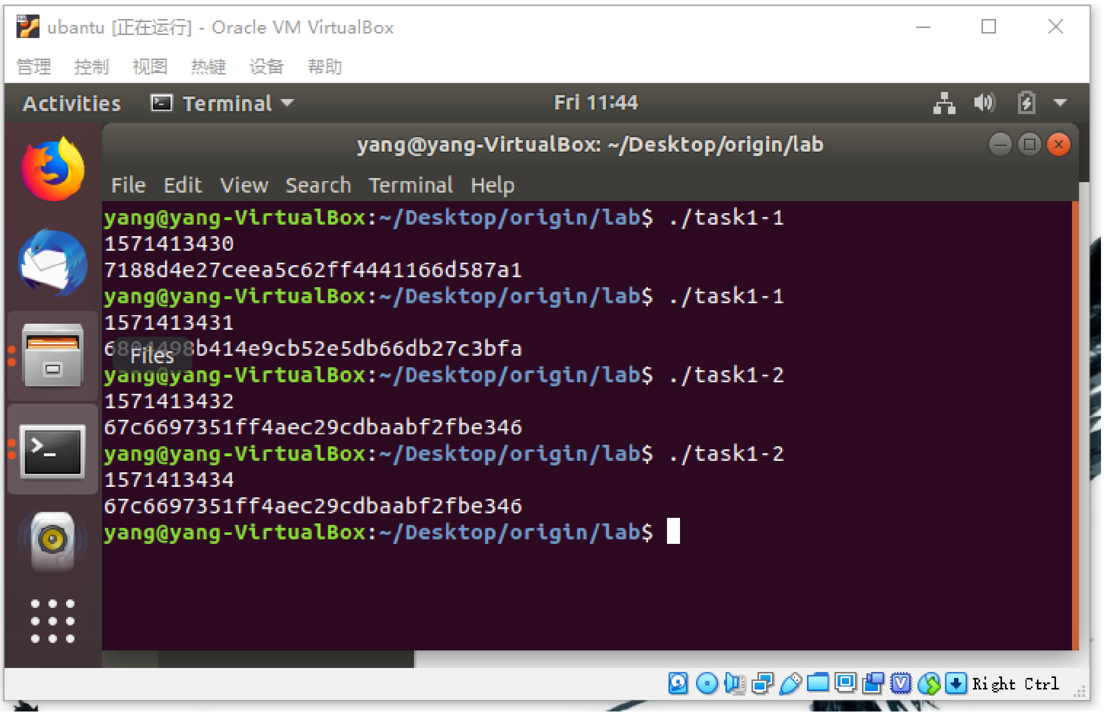
Lab1

Ziyang lin (zlin32)

1 Task1: Generate Encryption Key in a Wrong Way

The task1-1 is the original code's output, and the task1-2 is the code that command out the "srand()" line. I found that in task1-1 the key will change with the timestamp but in the task1-2 the key will be the same no matter whether the time change. "time(NULL)" is the function to give a timestamp, and the "srand()" use the current timestamp as seed to generate random number. If command out the line, the seed will be the same, which means the random number generate will output that same sequence of number.



2 Task2: Guessing the Key

The following is my Python code to break the key. The code use the timestamp as seed to generate key, and then traverse all key to encrypt the plaintext message. Comparing the output and the ciphertext, if they are the same, it means the key generator get the correct key that Alice used. As shown below, the key is 2129322710651590.

```python
import sys
import time
import random
from Crypto.Cipher import AES
from binascii import b2a_hex, a2b_hex
plaintext="255044462d312e350a25d0d4c5d80a34"
ciphertext="d06bf9d0dab8e8ef880660d2af65aa82"
iv = "0908070605040302010OA2B2C2D2E2F2"

def getST(string):
    timeArray = time.strptime(string, "%Y-%m-%d %H:%M:%S")
    timestamp = time.mktime(timeArray)
    return timestamp

dt1 = getST("2018-04-17 21:08:49")#get timestamp
dt2 = getST("2018-04-17 23:08:49")


class prpcrypt():
    def __init__(self, key,iv):
        self.key = key
        self.iv =iv
        self.mode = AES.MODE_CBC

    def encrypt(self, text):
        cryptor = AES.new(self.key, self.mode, self.iv)
        length = 16
        count = len(text)
        if(count % length != 0) :
            add = length - (count % length)
        else:
            self.ciphertext = cryptor.encrypt(text)#get plaintext
        return b2a_hex(self.ciphertext)#output as hex

if __name__ == '__main__':
    for i in range(int(dt1),int(dt2)):
        try:
            #print(i)
            random.seed(int(i))#set seed
            key = int(random.random()*10000000000000000)#generate key
            pc = prpcrypt(str.encode(str(key)),str.encode(iv[:16])) #initial algorithm
            e = pc.encrypt(str.encode(plaintext))# encrypt
            if e == ciphertext:
                break
        except Exception as e:
            print(e)
    print(key)
```

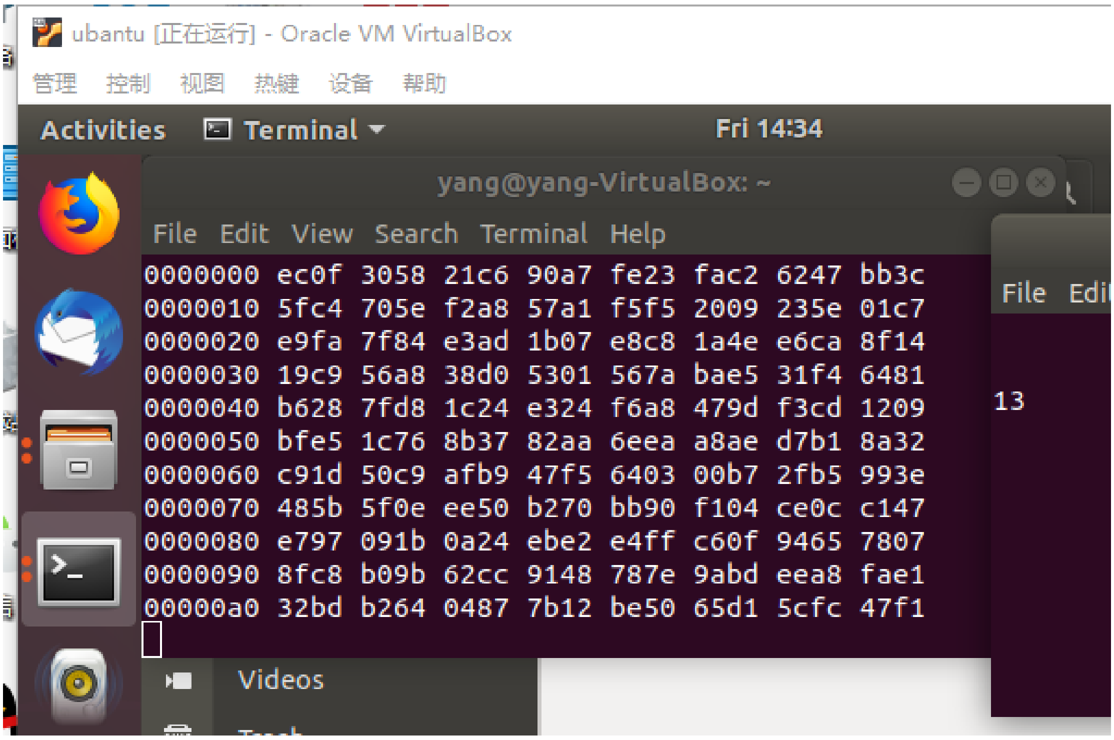
```
Incorrect AES key length (14 bytes)
2129322710651590
```

2.3 Task3: Measure the Entropy of Kernel

When moving the mouse, the number increased by 2 or 3 each time. The click action will make number increase by 3 or 4. The typing will make number increase 5. The reading would not make entropy increase unless using mouse or keyboard to turn page. And visiting a website will make number keep going by 1 without other actions. If we only consider the step of the number, typing increased the entropy significantly, but if we consider the all action we need to do, visiting a website will be the champion.

2.4 Task4: Get Pseudo Random Numbers from /dev/random

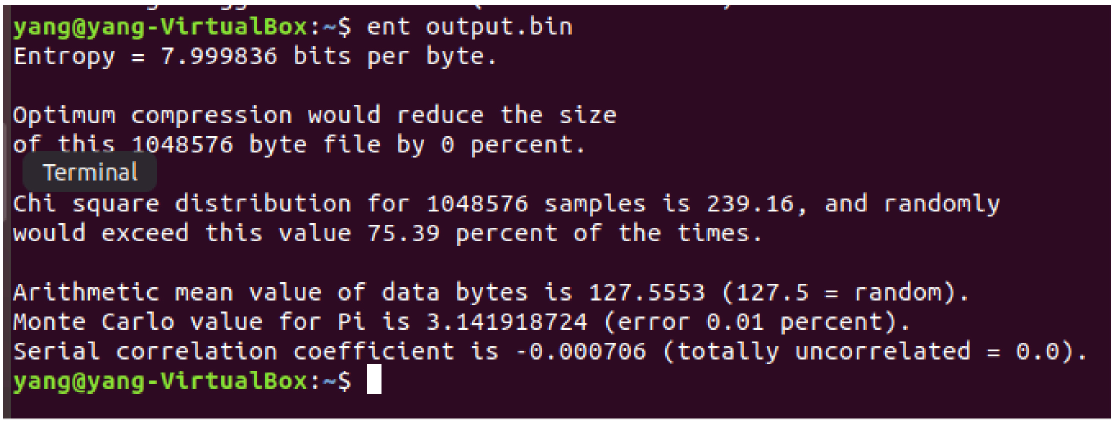
If we do not do any action, the number will only increase 1 each time, and if we randomly move mouse the number will increase much faster. No matter how faster the number increases, it will go back to 1 once it reach 64, with the increase in first 6 bit of output of hexdump. The output of hexdump is hex. The reason that number increase faster is that the increase is the sum of default number and the number triggered by actions.



Question: If a server uses this to generate session key, DoS will generate a large number of session key, which may run out of all available session id and cannot provide enough entropy for normal service request.

2.5 Task5: Get Random Numbers from /dev/urandom

In this case, the number generate very fast, and it seems that there is no affect by the moving the mouse on the outcome. The following is the outcome of random number analysis.



It shows that the quality of random number is good, because the number it generate is close to the truly random number.

The following is the code to generate a 256-bit encryption key.

```c
#include<stdio.h>
#include<stdlib.h>
#define LEN 32 // 256 bits


void bi(int k){
    int x=0,a[10];
    for(int i = 0; i < 8; i++)
    {
    a[x++]=k%2;
    k/=2;
    }
    --x;
    while(x>=0)
    printf("%d",a[x--]);


}


void 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);
    for( int i = 0; i < LEN; i++ )
    {   int k = key[i];
        bi(k);
    }
}
```

This screenshot shows the key that generate by the code.

```
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
00001101110110011010001000110001101100101000100010111110001101010010001001111101
11010101010100011010100011101111101010100110111110010001001100110000110000100111
10011100100000001000111100110110010101000100110101101111011110011010101010011110
0111101010010011yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
01100111101011001001000010010000001101101101011111100011100100010000111100110011
1001100010111110011100010110101001111010101111100111101010010111000011010000110
10101011001011001100110100011110001000010001000000011100111011000000111000111111
0011101111011010yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
1110000001011110010111100100110000000101011011001001110010010000111011001001001001
00001010001111101100111101110111100001111001000011000011001100101000001000110110
01111010111011011111110101000100110010110011010100001110111011010011010000010000
1110100010100111yang@yang-VirtualBox:~/Desktop/origin/lab$
```