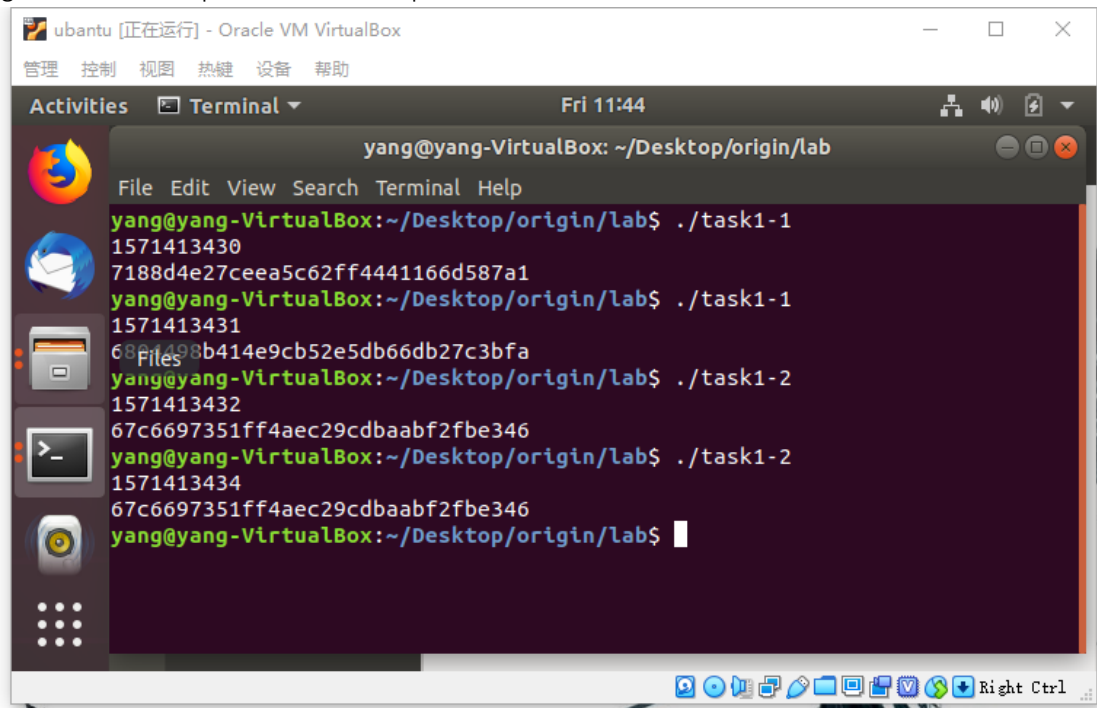


Lab1

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.



```
yang@yang-VirtualBox: ~/Desktop/origin/lab
File Edit View Search Terminal Help
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-1
1571413430
7188d4e27ceea5c62ff4441166d587a1
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-1
1571413431
6887798b414e9cb52e5db66db27c3bfa
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-2
1571413432
67c6697351ff4aec29cdbaabf2fbe346
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-2
1571413434
67c6697351ff4aec29cdbaabf2fbe346
yang@yang-VirtualBox:~/Desktop/origin/lab$
```

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.

```

1 import sys
2 import time
3 import random
4 from Crypto.Cipher import AES
5 from binascii import b2a_hex, a2b_hex
6 plaintext="255044462d312e350a25d0d4c5d80a34"
7 ciphertext="d06bf9d0dab8e8ef880660d2af65aa82"
8 iv = "09080706050403020100A2B2C2D2E2F2"
9
10 def getST(string):
11     timeArray = time.strptime(string, "%Y-%m-%d %H:%M:%S")
12     timestamp = time.mktime(timeArray)
13     return timestamp
14
15 dt1 = getST("2018-04-17 21:08:49")#get timestamp
16 dt2 = getST("2018-04-17 23:08:49")
17
18
19 class prpcrypt():
20     def __init__(self, key,iv):
21         self.key = key
22         self.iv =iv
23         self.mode = AES.MODE_CBC
24
25     def encrypt(self, text):
26         cryptor = AES.new(self.key, self.mode, self.iv)
27         length = 16
28         count = len(text)
29         if(count % length != 0) :
30             add = length - (count % length)
31         else:
32             self.ciphertext = cryptor.encrypt(text)#get plaintext
33             return b2a_hex(self.ciphertext)#output as hex
34
35 if __name__ == '__main__':
36     for i in range(int(dt1),int(dt2)):
37         try:
38             #print(i)
39             random.seed(int(i))#set seed
40             key = int(random.random()*10000000000000000)#generate key
41             pc = prpcrypt(str.encode(str(key)),str.encode(iv[:16])) #initial algorithm
42             e = pc.encrypt(str.encode(plaintext))# encrypt
43             if e == ciphertext:
44                 break
45         except Exception as e:
46             print(e)
47     print(key)

```

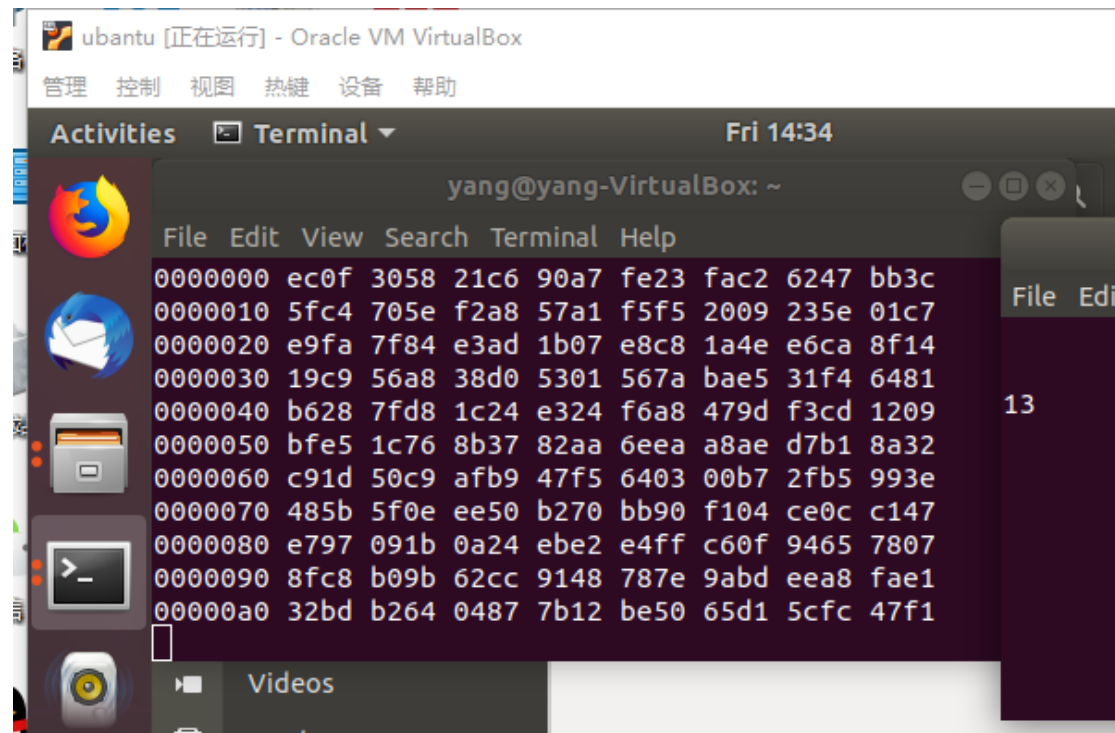
Incorrect AES key length (14 bytes)
2129322710651590

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.

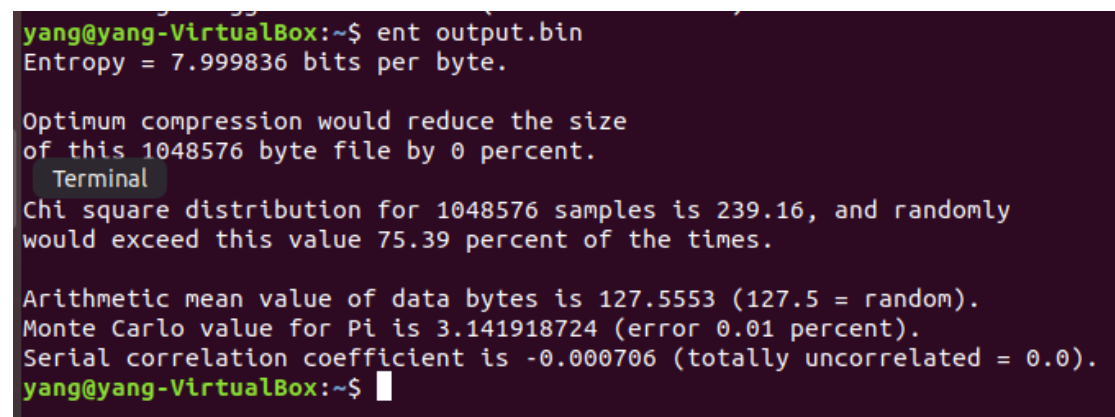


```
yang@yang-VirtualBox: ~  
File Edit View Search Terminal Help  
00000000 ec0f 3058 21c6 90a7 fe23 fac2 6247 bb3c  
00000100 5fc4 705e f2a8 57a1 f5f5 2009 235e 01c7  
00000200 e9fa 7f84 e3ad 1b07 e8c8 1a4e e6ca 8f14  
00000300 19c9 56a8 38d0 5301 567a bae5 31f4 6481  
00000400 b628 7fd8 1c24 e324 f6a8 479d f3cd 1209  
00000500 bfe5 1c76 8b37 82aa 6eea a8ae d7b1 8a32  
00000600 c91d 50c9 afb9 47f5 6403 00b7 2fb5 993e  
00000700 485b 5f0e ee50 b270 bb90 f104 ce0c c147  
00000800 e797 091b 0a24 ebe2 e4ff c60f 9465 7807  
00000900 8fc8 b09b 62cc 9148 787e 9abd eea8 fae1  
00000a00 32bd b264 0487 7b12 be50 65d1 5cfc 47f1
```

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.



```
yang@yang-VirtualBox:~$ ent output.bin  
Entropy = 7.999836 bits per byte.  
  
Optimum compression would reduce the size  
of this 1048576 byte file by 0 percent.  
Terminal  
Chi square distribution for 1048576 samples is 239.16, and randomly  
would exceed this value 75.39 percent of the times.  
  
Arithmetic mean value of data bytes is 127.5553 (127.5 = random).  
Monte Carlo value for Pi is 3.141918724 (error 0.01 percent).  
Serial correlation coefficient is -0.000706 (totally uncorrelated = 0.0).  
yang@yang-VirtualBox:~$
```

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.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define LEN 32 // 256 bits
4
5  void bi(int k){
6      int l = sizeof(k)*8;
7      int i;
8      if(i == 0)
9      {
10         printf("0");
11         return;
12     }
13
14     for(i = l-1; i >= 0; i --)
15     {
16         if(k&(1<<i)) break;
17     }
18     for(;i>=0; i --){
19         printf("%d", (k&(1<<i)) != 0);
20     }
21 }
22
23
24 void main(){
25     unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char)*LEN);
26     FILE* random = fopen("/dev/urandom", "r");
27     fread(key, sizeof(unsigned char)*LEN, 1, random);
28     fclose(random);
29     for( int i = 0; i < LEN; i++ )
30     {
31         int k = key[i];
32         bi(k);
33     }
34 }
```

This screenshot shows the key that generate by the code.

```
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
1000111110001111010011011111101010011010011110111101111100
1101100000101111100011110101100110011100001011101101111001
110111000100101111100101111111011010101001101110111110001
11101110111111000010010110110011101101110011110yang@yang-
VirtualBox:~/Desktop/origin/lab$ ./task5
1000011110001011111001111100101011101001101111101110011111
0011100100101000111001011010111111111011011010110101111000
1101001001000000111001011111101010100000111110111101011111
10111010011111000110011011101100001010001100011yang@yang-V
irtualBox:~/Desktop/origin/lab$ ./task5
10111000101111000100101010101111111011001101111111110011
010100110101110110110010000101000101010001110011110111000
yang@yang-VirtualBox:~/Desktop/origin/lab$
```