Task 1)

```
→  Lab1 ./test
1632697774
5e9dbcdffdc77887c0ab1a11e9ae5126
→  Lab1 ./test
1632697775
c69b0d77b7a2941456c9cda73067cbb7
→  Lab1 ./test
1632697776
491fa61fe930708084962fef90e61eb0
→  Lab1 ./test
1632697778
2037df7e602e3c4a2b419b63267a68c1
```

```
→  Lab1 gcc -o test test.c
→  Lab1 ./test
1632696402
07ca5102380a67912a7e3f76e29cc0e1
→  Lab1
```

```
test.c (~/Crypto/Lab1) - gedit
Open

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

After creating, compiling, and running the given program, I observe that it prints the time and "random" key generated. I comment out line 1, the srand() function and re-compile. After running this version of the code multiple times, I see that the time changes, but the random key does not. Clearly, the srand() function sets the seed for the random function used later in the code. When they seed is not set, the default seed runs, and the rand() function becomes deterministic, returning the same value every time. Notice below, the first line (time) changes but the second line (key) does not.

```
→  Lab1 ./test
1632696559
67c6697351ff4aec29cdbaabf2fbe346
→  Lab1 ./test
1632696563
67c6697351ff4aec29cdbaabf2fbe346
→  Lab1 ./test
1632696565
67c6697351ff4aec29cdbaabf2fbe346
→  Lab1 ./test
1632696566
67c6697351ff4aec29cdbaabf2fbe346
→  Lab1
```

Task 2) First, we need to generate all of the possible keys that could happen 2 hours before Alice's timestamp all the way up to the time stamp. To do this, we must find the epoch of Alice's timestamp. Below is a screenshot.

```
seed@VM:~ 80x24
→ ~ date -d "2018-04-17 23:08:49" +%s
1524020929
→ ~
```

Then, we use the provided code to list out the possible random numbers generated within the two hour window. I add a loop to print out all generated keys. After compiling, I put the output to a text file called possible_keys.txt. This text file will be run through my python program that will test every single key in the txt file to see if the key produces the ciphertext from the plaintext given.

```
seed@VM:~/Crypto/Lab1
seed@VM:~/Crypto/Lab1 80x7
→ Lab1 gcc generate_keylist.c -o generate_keylist
→ Lab1 ./generate_keylist > possible_keys.txt
→ Lab1
```

```
generate_keylist.c (~/Crypto/Lab1) - gedit
Open
```

```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");
    }
}
```

Below is the program I wrote to test all the keys from the text file generated above. First, it makes a byte array from the known hex values of the plaintext block, ciphertext, and initial vector. Then, it opens the list of keys that were generated above using the loop of times as seeds. Then, it encrypts the plaintext and tests the ciphertext. When the generated ciphertext and the given ciphertext are a match, we know that we have found the key.

```python
from Crypto.Cipher import AES

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

with open('possible_keys.txt') as possible:
    keys = possible.readlines()

for k in keys:
    k = k.rstrip('\n')
    key = bytearray.fromhex(k)
    cipher = AES.new(key=key, mode=AES.MODE_CBC, iv=iv)
    generated_ciphertext = cipher.encrypt(plaintext)
    if generated_ciphertext == ciphertext:
        print("correct key:", k)
        exit(0)
```

I run my python script and Bob has correctly guessed the key!

```
→  Lab1 python3 test_all_keys.py
correct key: 95fa2030e73ed3f8da761b4eb805dfd7
→  Lab1
```

Task 3) I ran the entropy command for a solid 30+ seconds. It started off with a relative increase just by watching the terminal, however, as I started to act on the machine, the number increased significantly.

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_...  Sun Sep 26 18:58:58 2021

1700
```

Just from opening the browser, the entropy went from 1700 to 1936. Again, I sat with the browser open for a few seconds and just watched the number slowly rise. Then, I closed the browser, and the entropy dropped to 1918.

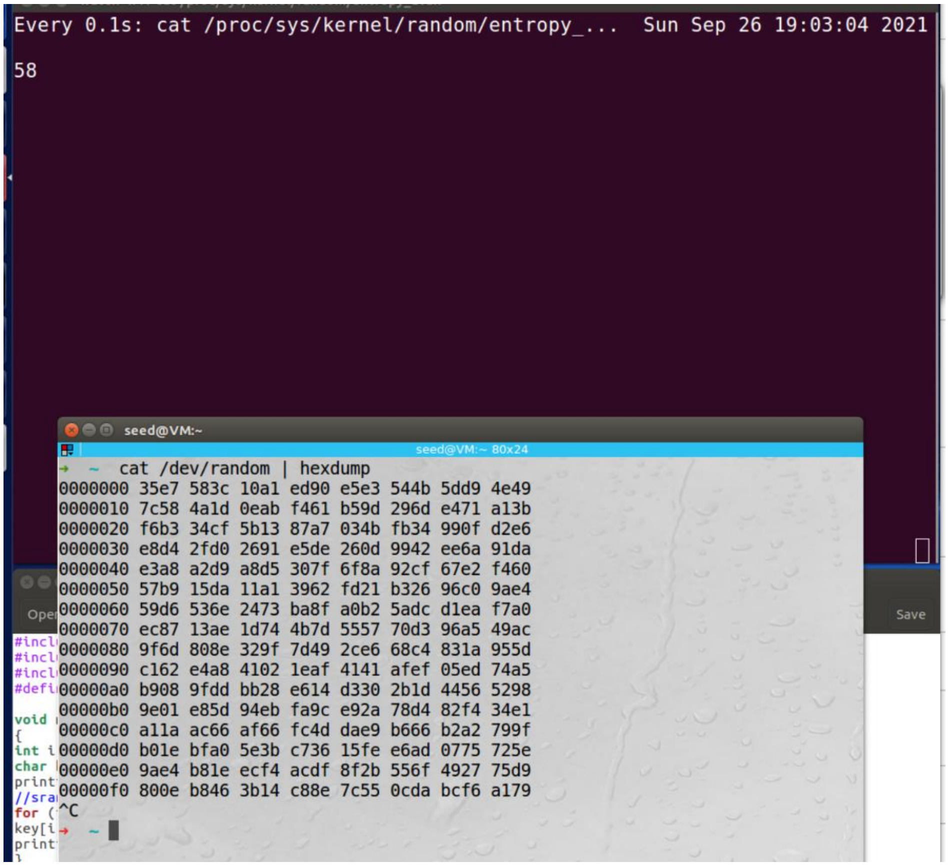
```
Every 0.1s: cat /proc/sys/kernel/random/entropy_...  Sun Sep 26 18:59:21 2021

1936
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_...  Sun Sep 26 18:58:40 2021

1918
```

Task 4) I opened the entropy command in one terminal, then used the hexdump visualization in another. It was interesting to see that the entropy stayed relatively the same when the mouse was not moving. Every few seconds, the hexdump would add a line, and the entropy would go back to zero.



Then I began to move my cursor around, and the entropy greatly increased, just to drop down to zero every time the hexdump produced a new line. Not sure what the mechanism is there.

In response to the question about a DoS attack against a server using /dev/random, it looks like somehow dropping the server's entropy will not allow the random seed to be generated. So that way, you can drop the entropy so low that the server cannot generate any sort of randomness, because the function will block generation.

Task 5) This command using /dev/urandom was much too fast to know if the mouse moving had any effect, but from what I could tell, no it did not.

```
→  Lab1  cat /dev/urandom | hexdump
0000000 ea1e 284b e677 f8e4 8363 50c0 ff42 c426
0000010 2afe ef7d 7420 302c fd8c b87d 2b41 f0eb
0000020 5d3d f068 e1cd f48f d12c 4cab 3167 77f7
0000030 4546 0203 5c26 4f26 7407 9e90 c862 cd08
0000040 e14c 2662 5223 d880 ac5d 5be5 35dc 2881
0000050 5e16 429b c525 f358 5bd1 ed3e 0748 a918
0000060 e189 ffc2 62a6 9092 0486 d169 fc08 7625
0000070 2cff 0e6b ca8f 5a08 0b78 06ca af30 6e1c
0000080 60e8 1757 4d91 ba30 4a08 dd42 af99 73a7
0000090 bbdb d0b9 04d6 04f9 6c62 3040 bf59 9cd3
00000a0 7da7 ae36 4782 acff 6418 e27f 5857 d168
00000b0 49d9 6fbe f48a 9245 973b 4724 495e 6e8a
```

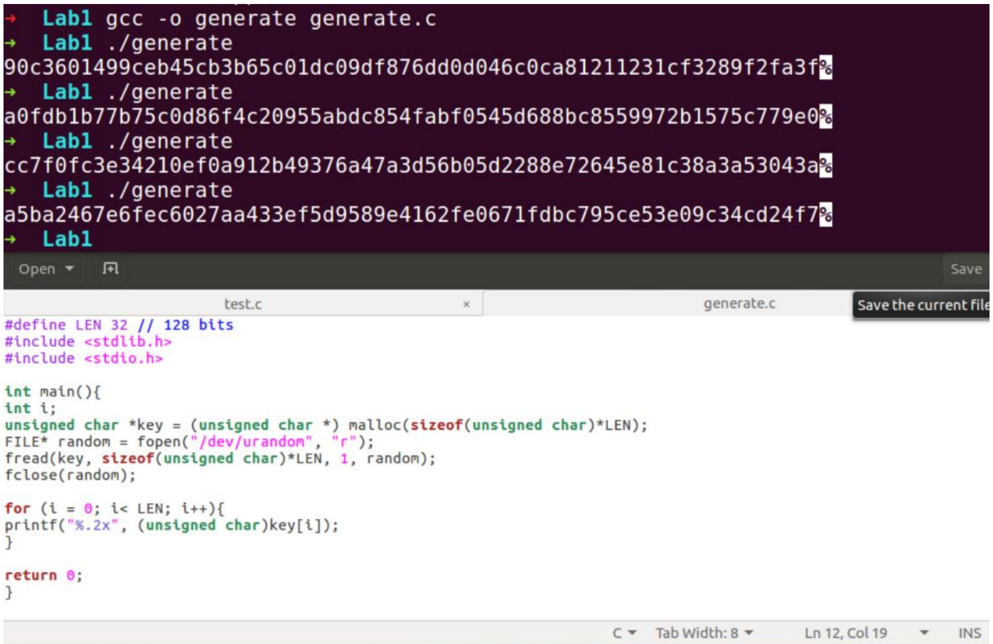
```
→  Lab1  head -c 1M /dev/urandom > output.bin
→  Lab1
→  Lab1 ent output.bin
Entropy = 7.999798 bits per byte.

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

Chi square distribution for 1048576 samples is 294.25, and randomly
would exceed this value 4.60 percent of the times.

Arithmetic mean value of data bytes is 127.5227 (127.5 = random).
Monte Carlo value for Pi is 3.146015724 (error 0.14 percent).
Serial correlation coefficient is -0.000194 (totally uncorrelated = 0.0).
→  Lab1
```

It seems like the quality of these pseudo random numbers is extremely good. It is very close to random. Just from a simple statistics course in undergrad, I know that the Chi-2 distribution of 294.25 is pretty huge, which means there seems to be little to no patterns Also, it says that 127.5 mean is random, whereas this one is 127.5227 mean. That’s really close, so I think the quality is very high.



In the picture to the left, I write a program that defines the LEN as 32, in order to generate a 256 bit key. Then using the given code, I generate a random number using the device in /dev/urandom. The bottom half of the picture is the code I wrote, and the top half is the resulting output. After running this 4 times, I receive 4 very different 256-bit keys. IT is clear that /dev/urandom is better to use due to the fact that it doesn't "block" generation when entropy is low. After re-reading the assignment instructions, I noticed that this wasn't in binary as was required. I adapted my code to print it out in binary, but some of the formatting got messed up. Below is the binary output of running the code one time.

```
→  Lab1 gcc -o generatebin generate.c
→  Lab1 ./generatebin
01001001�[� 10111110�[� 00110010�[� 10111101�[� 10110100�[� 01001110�[�
 11110100�[� 11101110�[� 11100011�[� 00000001�[� 10000000�[� 00110011�[
� 00001101�[� 01010110�[� 01100110�[� 11101111�[� 01101001�[� 00001000�
[� 00111100�[� 01100011�[� 00111101�[� 11010100�[� 00001011�[� 10001111�
�[� 00100001�[� 01001000�[� 10000010�[� 00001001�[� 11011011�[� 10001011
�[� 01100101�[� 11111111�[� %
→  Lab1
```

generate.c (~/Crypto/Lab1) - gedit

```c
#define LEN 32 // 128 bits
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void bin(unsigned n);

int main(){
char binary[256];
int i;
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 (i = 0; i< LEN; i++){
//printf("%.2x", (unsigned char)key[i]);
bin((unsigned char)key[i]);
printf("%s ",binary);
}


return 0;
}


void bin(unsigned n)
{
    unsigned i;
    for (i = 1 << 7; i > 0; i = i / 2)
        (n & i) ? printf("1") : printf("0");
}
```