Lab 2 – Pseudo-Random Number Generation Lab (Seed Labs)

Taylor Adams, CS 542 - Cryptography, tkadams1@crimson.ua.edu

***Abstract*—The goal of the lab is to allow the user to gain a greater understanding of pseudo-random number generation. This lab demonstrates the use of secure and non-secure was to generate randomness. This lab also demonstrates why randomness is important in cryptography and gives the students following it a hands on understanding of how to properly generate pseudo random numbers in linux. These pseudo random numbers are incredibly useful in overall computing and are necessary for the security abilities we enjoy today in everyday computing.**

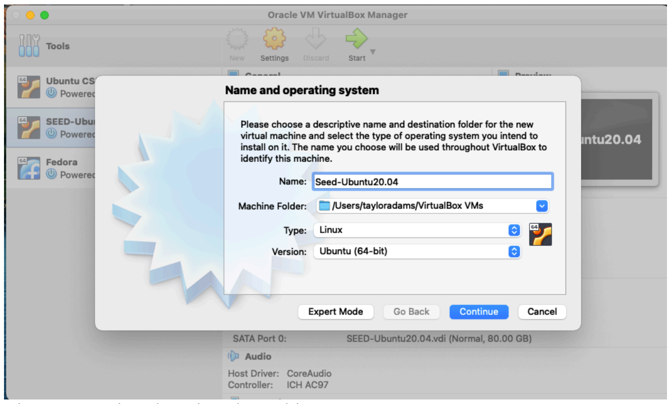
I. Lab Definition

THE goal of this lab is to give students the opportunity to learn about how random number generation works in Linux. It also gives the student the tools necessary to understand how pseudo random numbers can essentially be random enough for use in secure cryptography. This lab shows several popular libraries used in pseudo random number generation and ask the student to perform task with theses libraries. In task 2, the most difficult in this lab the student is also shown how utilizing time in pseudo random number generation can lead to vulnerabilies in key generation if time is utilized. Task 3 has the student understand how true random data can be collected by a computer system. Which in order to get access to truly random data must access some data from outside the computer system itself, such as user input. Task 4 and Task 5 discuss different build in linux libraries /dev/random and /dev/urandom which goes over the pros and cons of using both. Each of these task build on one another allowing students to gain a much greater understanding of pseudo random numbers than they would from just reading a book.

II. Step-By-Step Instructions - Lab Task

A. *Task A – Setting up the Virtual Machine*

In order to begin this lab, I chose to utilize VirtualBox along with the pre-built VM provided at https://seedsecuritylabs.org/labsetup.html [10]. I had previously configured this VM from lab one, below are the steps I took to set up this VM This VM utilizes Ubuntu 20.04. After downloading the SEED-Ubuntu20.04.vdi file, I proceeded to create a new virtual machine to utilize the image with. I followed the setup instructions provided in the above url to complete the setup of the lab. I have frequently utilized virtualbox in the past, so this setup task was no problem at all.



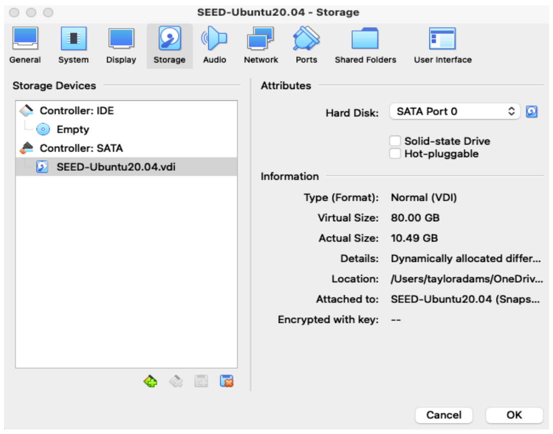
Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: Seed-Ubuntu20.04
Machine Folder: /Users/tayloradams/VirtualBox VMs
Type: Linux
Version: Ubuntu (64-bit)

SATA Port 0: SEED-Ubuntu20.04.vdi (Normal, 80.00 GB)
Audio
Host Driver: CoreAudio
Controller: ICH AC97

Fig. 1. Creating the Virtual Machine



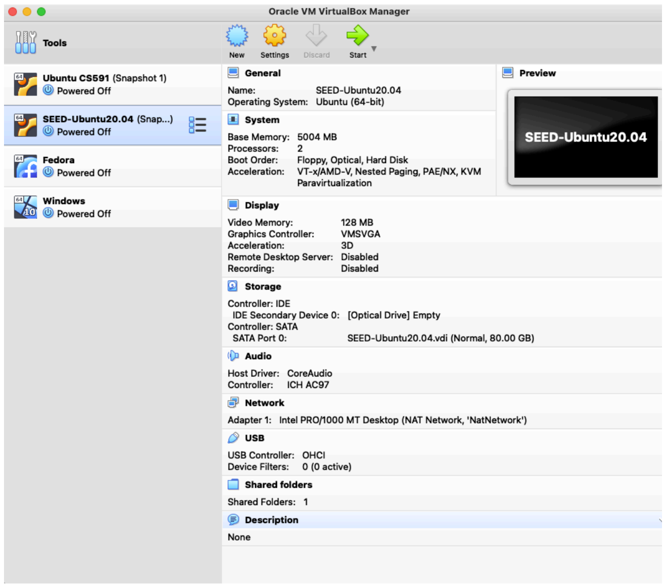
Storage Devices
Controller: IDE
Empty
Controller: SATA
SEED-Ubuntu20.04.vdi

Attributes
Hard Disk: SATA Port 0
Solid-state Drive
Hot-pluggable

Information
Type (Format): Normal (VDI)
Virtual Size: 80.00 GB
Actual Size: 10.49 GB
Details: Dynamically allocated differ...
Location: /Users/tayloradams/OneDriv...
Attached to: SEED-Ubuntu20.04 (Snaps...
Encrypted with key: --

Fig. 1a. Pre-Built VDI file placed in the virtual disk

Manuscript received March 16, 2022.



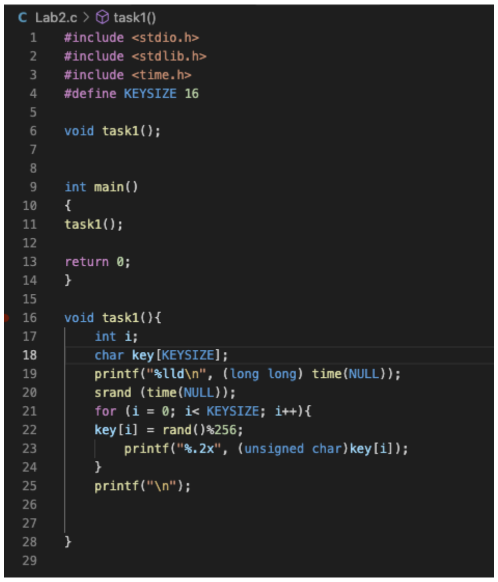
```
Ubuntu CS591 (Snapshot 1)
Powered Off
SEED-Ubuntu20.04 (Snap...)
Powered Off
Fedora
Powered Off
Windows
Powered Off

General
Name:               SEED-Ubuntu20.04
Operating System:   Ubuntu (64-bit)

System
Base Memory:   5004 MB
Processors:    2
Boot Order:    Floppy, Optical, Hard Disk
Acceleration:  VT-x/AMD-V, Nested Paging, PAE/NX, KVM
               Paravirtualization

Display
Video Memory:            128 MB
Graphics Controller:     VMSVGA
Acceleration:            3D
Remote Desktop Server:   Disabled
Recording:               Disabled

Storage
Controller: IDE
 IDE Secondary Device 0:  [Optical Drive] Empty
Controller: SATA
 SATA Port 0:             SEED-Ubuntu20.04.vdi (Normal, 80.00 GB)

Audio
Host Driver:  CoreAudio
Controller:   ICH AC97

Network
Adapter 1:  Intel PRO/1000 MT Desktop (NAT Network, 'NatNetwork')

USB
USB Controller:  OHCI
Device Filters:  0 (0 active)

Shared folders
Shared Folders:  1

Description
None
```

Fig. 1b. SEED Virtual Machine Complete Settings

For task 1, I was able to run the code outside of the vm on my host machine, which I did successfully. However, when I attempted to complete task2 I ran into significant difficulty attempting to run my code on my host machine, so I switched back to the seed labs VM. I installed the openssl library on my host machine by running the command: brew install openssl.

B. *Task 1: Generate Encryption Key in a Wrong Way*

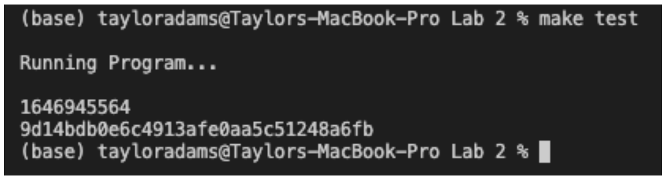
For task 1, I was given a block of code. This block of code generates an encryption key while utilizing the current time (time.h library in c). The time function returns the number of seconds that have passed since the date 01-01-1970.



```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #define KEYSIZE 16
5
6   void task1();
7
8
9   int main()
10  {
11  task1();
12
13  return 0;
14  }
15
16  void task1(){
17      int i;
18      char key[KEYSIZE];
19      printf("%lld\n", (long long) time(NULL));
20      srand (time(NULL));
21      for (i = 0; i< KEYSIZE; i++){
22      key[i] = rand()%256;
23          printf("%.2x", (unsigned char)key[i]);
24      }
25      printf("\n");
26
27
28  }
29
```

Fig. 2. Code Block Provided by seed labs instructions (slightly modified)

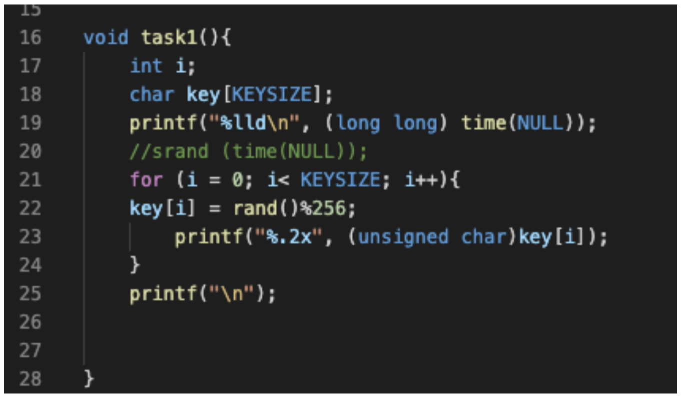
I created a new .c file to utilize for this lab as seen above. Following the lab instructions I ran the task.



```
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646945564
9d14bdb0e6c4913afe0aa5c51248a6fb
(base) tayloradams@Taylors-MacBook-Pro Lab 2 %
```

Fig. 3. Running the task the first time with line 20 uncommented

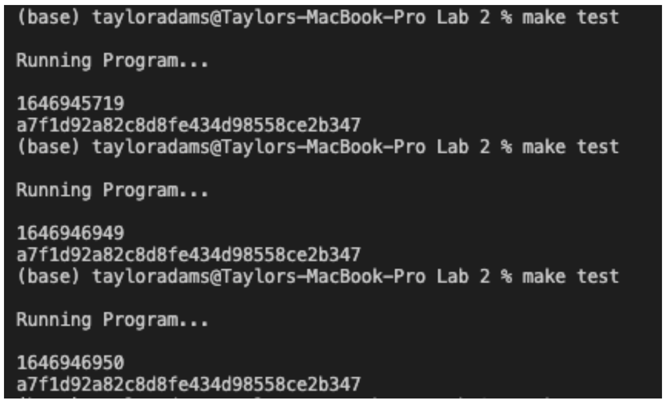
The first line printed is the just the time passed since 1970. The second printed line is the generated key. Each time this task is run, the key changes.



```
16  void task1(){
17      int i;
18      char key[KEYSIZE];
19      printf("%lld\n", (long long) time(NULL));
20      //srand (time(NULL));
21      for (i = 0; i< KEYSIZE; i++){
22      key[i] = rand()%256;
23          printf("%.2x", (unsigned char)key[i]);
24      }
25      printf("\n");
26
27
28  }
```

Fig. 4 Commented out code section

Below is the subsequent execution of the code above.

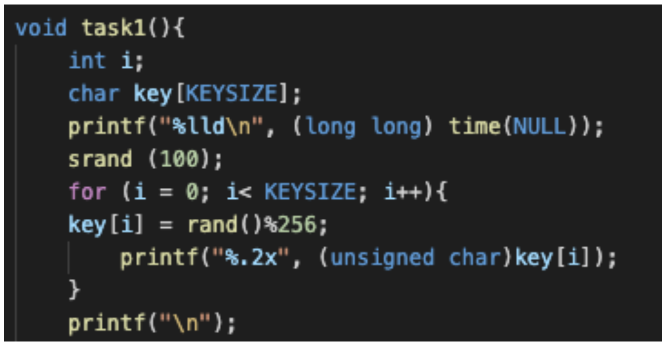


```
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646945719
a7f1d92a82c8d8fe434d98558ce2b347
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646946949
a7f1d92a82c8d8fe434d98558ce2b347
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646946950
a7f1d92a82c8d8fe434d98558ce2b347
```

Fig. 5. Running the task with line 20 commented out

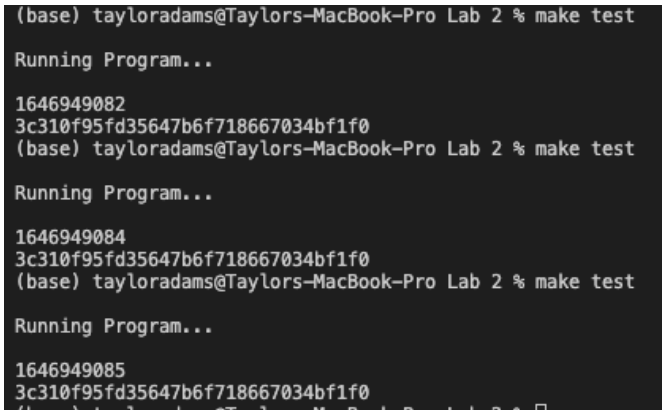
When the task is run with line 20 commented out the same key is generated every time the code is run, even though the time has slightly increased.

Observation on srand() and time() functions:
Srand() is a C function that takes in an unsigned int as a seed variable. Initially I was confused as to why commenting out the srand() function did anything, but after doing some research I found: “If random numbers are generated with rand() without first calling srand(), your program will create the same sequence of numbers each time it runs.”[2]. I did further testing by setting line 20 to srand(100). Each time srand was run with this particular function call the value returned was the exact same as the first run.



```
void task1(){
    int i;
    char key[KEYSIZE];
    printf("%lld\n", (long long) time(NULL));
    srand (100);
    for (i = 0; i< KEYSIZE; i++){
    key[i] = rand()%256;
        printf("%.2x", (unsigned char)key[i]);
    }
    printf("\n");
```

Fig. 6. Passing in “100” to srand()



```
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646949082
3c310f95fd35647b6f718667034bf1f0
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646949084
3c310f95fd35647b6f718667034bf1f0
(base) tayloradams@Taylors-MacBook-Pro Lab 2 % make test

Running Program...

1646949085
3c310f95fd35647b6f718667034bf1f0
```

Fig. 7. Results from passing in “100” to srand()

This clearly shows the srand() function is deterministic and is based on the value passed into it’s function call. The reason it is able to return a different value when the time() function is called is because it is generally running at least a second later than the previous fun, producing a slightly larger time value.

Srand() and rand() are closely related functions which is why commenting out line 20 breaks task1. “Srand() sets the seed which is used by rand to generate “random” numbers. If you don’t call srand before your first call to rand, it’s as if you had called srand(1) to set the seed to one. In short, srand() — Set Seed for rand() Function.” [2].

Time() is a function which returns the number of seconds passed since 01-01-1970 as stated before. Evidence of this exact use case in task 1 was found on GeekforGeeks website, “Standard practice is to use the result of a call to srand(time(0)) as the seed. However, time() returns a time_t value which vary every time and hence the pseudo-random number vary for every program call.” [2]. Time in the case above is used to pass a “new” value into srand() different from any previous value passed into srand().

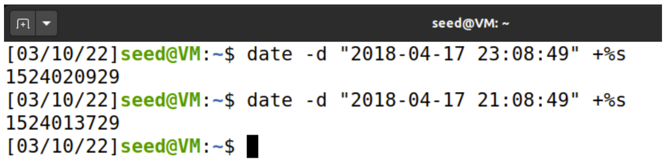
C. *Task 2: Encryption using Different Ciphers and Modes*

In Task 2 I was requested to find the key by utilizing the given variables:

- Plaintext: 255044462d312e350a25d0d4c5d80a34
- Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
- IV: 09080706050403020100A2B2C2D2E2F2DES-ECB

I was also given the information that the encrypted pdf file was encrypted with AES-128-cbc encryption, the key was generated utilizing the same function from task1, and it is likely the key was generated within a 2 hour window before the pdf file was created on "2018-04-17 23:08:49".

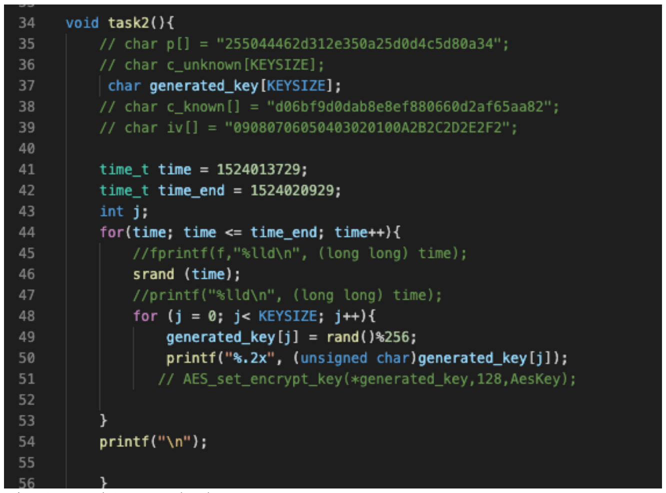
Having this knowledge I began by starting my vm and running the “date -d” command as specified in the lab



```
[03/10/22]seed@VM:~$ date -d "2018-04-17 23:08:49" +%s
1524020929
[03/10/22]seed@VM:~$ date -d "2018-04-17 21:08:49" +%s
1524013729
[03/10/22]seed@VM:~$
```

Fig. 8. Date -d command run to find time in seconds

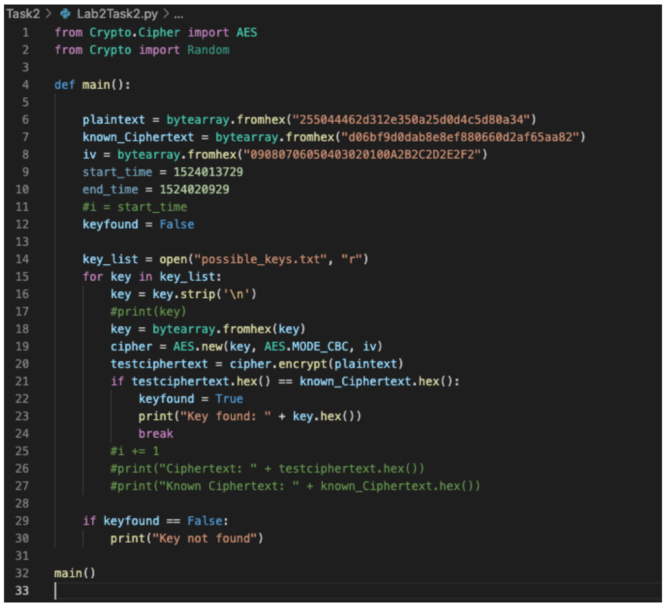
I now know the key was likely generated using a time between 1524013729 and 1524020929 seconds since 01-01-1970. I then wrote the c script for task 2.



```
void task2(){
    // char p[] = "255044462d312e350a25d0d4c5d80a34";
    // char c_unknown[KEYSIZE];
     char generated_key[KEYSIZE];
    // char c_known[] = "d06bf9d0dab8e8ef880660d2af65aa82";
    // char iv[] = "09080706050403020100A2B2C2D2E2F2";

    time_t time = 1524013729;
    time_t time_end = 1524020929;
    int j;
    for(time; time <= time_end; time++){
        //fprintf(f,"%lld\n", (long long) time);
        srand (time);
        //printf("%lld\n", (long long) time);
        for (j = 0; j< KEYSIZE; j++){
            generated_key[j] = rand()%256;
            printf("%.2x", (unsigned char)generated_key[j]);
           // AES_set_encrypt_key(*generated_key,128,AesKey);

    }
    printf("\n");

    }
```

Fig. 9. Task2 C method

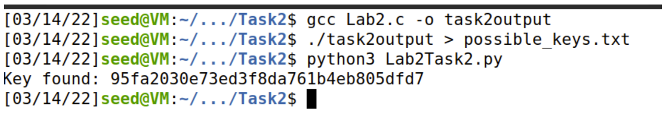
Task2() takes in the assumed time range as a time_t variable. It also takes in a generated key size of a constant 16 bytes as stated by the lab. The code then utilize the same code from task one to print the key generated for a specific time. This code generates about 7000 different keys and outputs them to the console. Initally I attempted to write the entire code in C, however I ran into significant difficulty attempting to utilize the AES library with C. As a compromise I ran the AES code in python after generating the possible keys in C.



```
Task2 > Lab2Task2.py > ...
1   from Crypto.Cipher import AES
2   from Crypto import Random
3
4   def main():
5
6       plaintext = bytearray.fromhex("255044462d312e350a25d0d4c5d80a34")
7       known_Ciphertext = bytearray.fromhex("d06bf9d0dab8e8ef880660d2af65aa82")
8       iv = bytearray.fromhex("09080706050403020100A2B2C2D2E2F2")
9       start_time = 1524013729
10      end_time = 1524020929
11      #i = start_time
12      keyfound = False
13
14      key_list = open("possible_keys.txt", "r")
15      for key in key_list:
16          key = key.strip('\n')
17          #print(key)
18          key = bytearray.fromhex(key)
19          cipher = AES.new(key, AES.MODE_CBC, iv)
20          testciphertext = cipher.encrypt(plaintext)
21          if testciphertext.hex() == known_Ciphertext.hex():
22              keyfound = True
23              print("Key found: " + key.hex())
24              break
25          #i += 1
26          #print("Ciphertext: " + testciphertext.hex())
27          #print("Known Ciphertext: " + known_Ciphertext.hex())
28
29      if keyfound == False:
30          print("Key not found")
31
32  main()
33
```

Fig. 10. Lab2Task2.py

Lab2Task2 takes in the known variables known from the lab: plaintext, ciphertext, and iv. It also takes in the time variables found from running the date -d command. In line 14, it reads the possible_keys.txt file which is generated from running the task2 c file. Then it reads through each key in the possible_keys file. AES.new() is run which generates the cipher with the known values of the key, AES CBC mode, and the iv. This cipher then encrypts the plaintext and stores the value in the tested ciphertext. This ciphertext is then tested to see if it matches the known ciphertext provided by the lab.



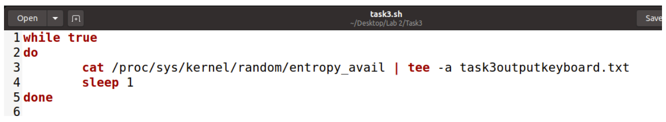
```
[03/14/22]seed@VM:~/.../Task2$ gcc Lab2.c -o task2output
[03/14/22]seed@VM:~/.../Task2$ ./task2output > possible_keys.txt
[03/14/22]seed@VM:~/.../Task2$ python3 Lab2Task2.py
Key found: 95fa2030e73ed3f8da761b4eb805dfd7
[03/14/22]seed@VM:~/.../Task2$ 
```

Fig. 11. Running task 2 and finding the key used for the encryption

Running the code for task 2 I found the key to be: 95fa2030e73ed3f8da761b4eb805dfd7

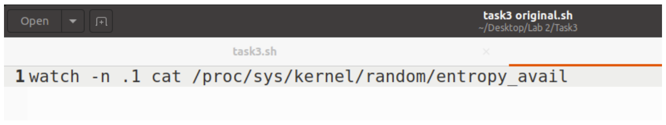
D. *Task 3: Measure the Entropy of Kernel*

For this task I was asked to measure the randomness of the kernel by running the commands: "cat /proc/sys/kernel/random/entropy_avail" and "watch -n .1 cat /proc/sys/kernel/random/entropy_avail". The former command is the latter, but only runs once. The later command runs every second to display the current entropy of the kernel. I ran a slightly modified version of the command to capture the output to a file as well as display the text in the console using the "tee" command



```
while true
do
        cat /proc/sys/kernel/random/entropy_avail | tee -a task3outputkeyboard.txt
        sleep 1
done

```

Fig. 12. Task3 Script with entropy command and text output



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

Fig. 13. Task3_original.sh shell script

Utilizing the above script generated the following results:

The keyboard clicks appeared to generate significant entropy at first, but the entropy increase seemed increase seemed to taper off after a while. Mouse movements and clicks generated some entropy, but the increase was minimal. Going to a website did increase the entropy, a significant amount. However, I had difficulty measuring the entropy increases when visiting a website due to my VM lagging significantly. (My host machine only has a dual core CPU.)

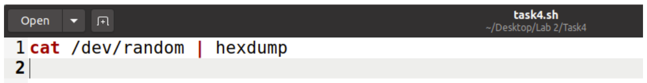
Entropy outputs from task3:
Keyboard: 1880 >2514
Mouse: 3754 > 3805
Website: 1064 > 2195

The entire entropy output files can be found in the task3 folder.

E. *Task 4: Get Pseudo Random Numbers from /dev/random*

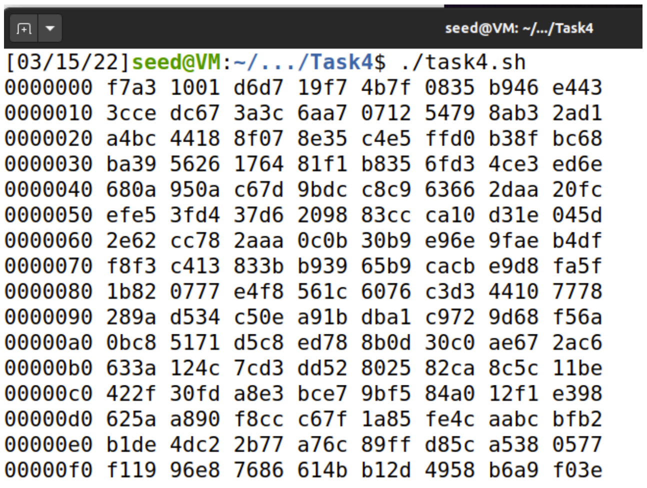
The goal of this task is to understand how padding works with different modes. Padding is used when the file to be encrypted does not have zero remainder byte size when divided by the number of bytes in a block for the encryption mode. This means padding will need to be used to fill the remaining space for the encryption to work.

I repeated the step from the previous task of creating a shell script to generate keys for each of the encryption modes. AES was chosen as the cipher since it is the most popular cipher in use today. I then made another shell script 'Task4Enc.sh' which creates 3 text files containing 5, 10, and 16 bytes. This script then encrypts the 3 text files using each of the encryption modes: CBC, ECB, CFB, OFB.



```
cat /dev/random | hexdump

```

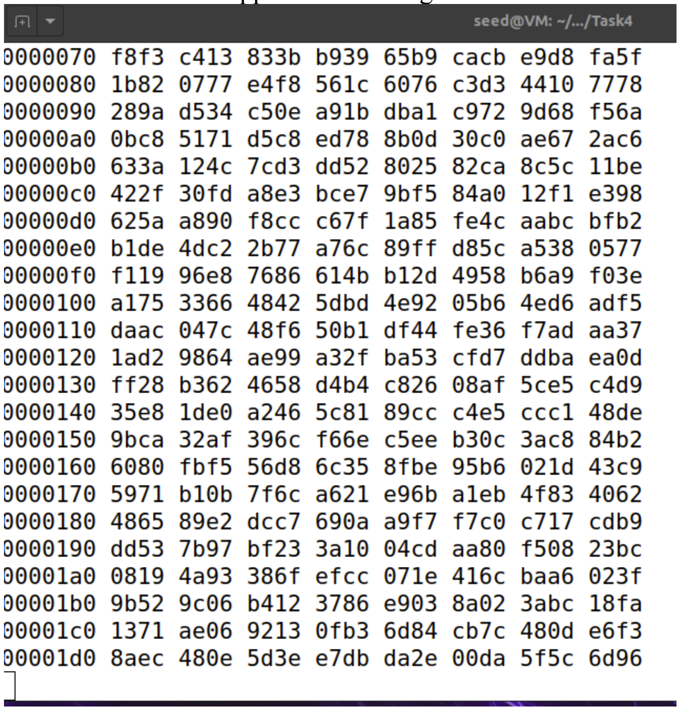
Fig. 14. Task4.sh



```
seed@VM: ~/.../Task4
[03/15/22]seed@VM:~/.../Task4$ ./task4.sh
0000000 f7a3 1001 d6d7 19f7 4b7f 0835 b946 e443
0000010 3cce dc67 3a3c 6aa7 0712 5479 8ab3 2ad1
0000020 a4bc 4418 8f07 8e35 c4e5 ffd0 b38f bc68
0000030 ba39 5626 1764 81f1 b835 6fd3 4ce3 ed6e
0000040 680a 950a c67d 9bdc c8c9 6366 2daa 20fc
0000050 efe5 3fd4 37d6 2098 83cc ca10 d31e 045d
0000060 2e62 cc78 2aaa 0c0b 30b9 e96e 9fae b4df
0000070 f8f3 c413 833b b939 65b9 cacb e9d8 fa5f
0000080 1b82 0777 e4f8 561c 6076 c3d3 4410 7778
0000090 289a d534 c50e a91b dba1 c972 9d68 f56a
00000a0 0bc8 5171 d5c8 ed78 8b0d 30c0 ae67 2ac6
00000b0 633a 124c 7cd3 dd52 8025 82ca 8c5c 11be
00000c0 422f 30fd a8e3 bce7 9bf5 84a0 12f1 e398
00000d0 625a a890 f8cc c67f 1a85 fe4c aabc bfb2
00000e0 b1de 4dc2 2b77 a76c 89ff d85c a538 0577
00000f0 f119 96e8 7686 614b b12d 4958 b6a9 f03e
```

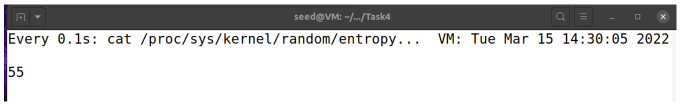
Fig. 15. Task 4 Hexdumped data

After running task4.sh the kernel entropy dropped to around zero. It would steadly increase as I moved my mouse, typed or did any kind of action in the VM. Once the system entrophy reached around 60 the /dev/random would unblock and use the data in system entropy to create a new hexdumped line which would be appended to the figure 15 above.



```
seed@VM: ~/.../Task4
0000070 f8f3 c413 833b b939 65b9 cacb e9d8 fa5f
0000080 1b82 0777 e4f8 561c 6076 c3d3 4410 7778
0000090 289a d534 c50e a91b dba1 c972 9d68 f56a
00000a0 0bc8 5171 d5c8 ed78 8b0d 30c0 ae67 2ac6
00000b0 633a 124c 7cd3 dd52 8025 82ca 8c5c 11be
00000c0 422f 30fd a8e3 bce7 9bf5 84a0 12f1 e398
00000d0 625a a890 f8cc c67f 1a85 fe4c aabc bfb2
00000e0 b1de 4dc2 2b77 a76c 89ff d85c a538 0577
00000f0 f119 96e8 7686 614b b12d 4958 b6a9 f03e
0000100 a175 3366 4842 5dbd 4e92 05b6 4ed6 adf5
0000110 daac 047c 48f6 50b1 df44 fe36 f7ad aa37
0000120 1ad2 9864 ae99 a32f ba53 cfd7 ddba ea0d
0000130 ff28 b362 4658 d4b4 c826 08af 5ce5 c4d9
0000140 35e8 1de0 a246 5c81 89cc c4e5 ccc1 48de
0000150 9bca 32af 396c f66e c5ee b30c 3ac8 84b2
0000160 6080 fbf5 56d8 6c35 8fbe 95b6 021d 43c9
0000170 5971 b10b 7f6c a621 e96b a1eb 4f83 4062
0000180 4865 89e2 dcc7 690a a9f7 f7c0 c717 cdb9
0000190 dd53 7b97 bf23 3a10 04cd aa80 f508 23bc
00001a0 0819 4a93 386f efcc 071e 416c baa6 023f
00001b0 9b52 9c06 b412 3786 e903 8a02 3abc 18fa
00001c0 1371 ae06 9213 0fb3 6d84 cb7c 480d e6f3
00001d0 8aec 480e 5d3e e7db da2e 00da 5f5c 6d96
```

Fig. 16. Task 4 appended Hexdumped data after running for a few minutes



```
seed@VM: ~/.../Task4
Every 0.1s: cat /proc/sys/kernel/random/entropy...   VM: Tue Mar 15 14:30:05 2022

55
```

Fig. 17. Task 4 running watch -n .1 cat /proc/sys/kernel/random/entropy_avail

The above entropy number generally hits 60 and resets to zero when /dev/random is running and becomes unblocked.

Task 4 Question:

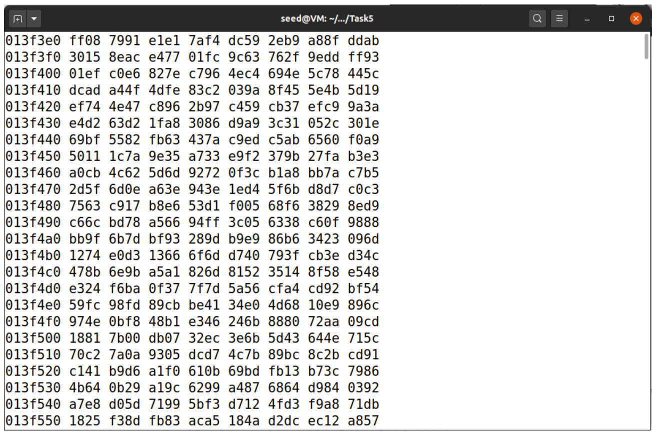
If a server uses /dev/random to generate the random session key with a client a DOS attack could cripple the server. If many clients attempt to connect at the same time (like in a DOS attack) the server will likely run out of random bits to generate the session key which will lead /dev/random to block on the server. This would cause all incoming additional connections to have to wait until enough random bits could be generated to satisfy the client demand on the server.

The question stated above is actually relatively similar to running task 4 in that our system is constantly running /dev/random, but after all the available random numbers are used it blocks until there are enough random numbers to generate the next hexdump.

F. *Task 5: Get Random Numbers from /dev/urandom*

In task 5, I was asked to generate random numbers using /dev/urandom. This function is different from /dev/random in that while both rely on the pseudo random number pool urandom does not block and will continue generating new numbers from a "seed" value.

For the first part of task 5 (labeled task 5-1) I was asked to run the command cat /dev/urandom | hexdump and describe whether moving the mouse had any effect on the outcome.

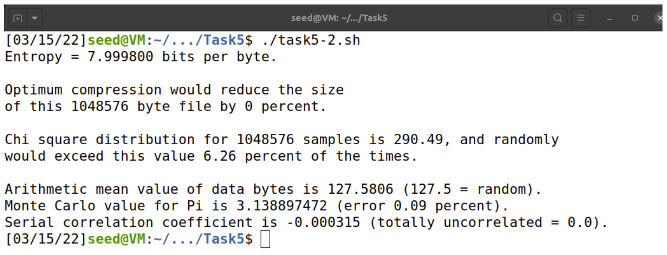


```
seed@VM: ~/.../Task5
013f3e0 ff08 7991 e1e1 7af4 dc59 2eb9 a88f ddab
013f3f0 3015 8eac e477 01fc 9c63 762f 9edd ff93
013f400 01ef c0e6 827e c796 4ec4 694e 5c78 445c
013f410 dcad a44f 4dfe 83c2 039a 8f45 5e4b 5d19
013f420 ef74 4e47 c896 2b97 c459 cb37 efc9 9a3a
013f430 e4d2 63d2 1fa8 3086 d9a9 3c31 052c 301e
013f440 69bf 5582 fb63 437a c9ed c5ab 6560 f0a9
013f450 5011 1c7a 9e35 a733 e9f2 379b 27fa b3e3
013f460 a0cb 4c62 5d6d 9272 0f3c b1a8 bb7a c7b5
013f470 2d5f 6d0e a63e 943e 1ed4 5f6b d8d7 c0c3
013f480 7563 c917 b8e6 53d1 f005 68f6 3829 8ed9
013f490 c66c bd78 a566 94ff 3c05 6338 c60f 9888
013f4a0 bb9f 6b7d bf93 289d b9e9 86b6 3423 096d
013f4b0 1274 e0d3 1366 6f6d d740 793f cb3e d34c
013f4c0 478b 6e9b a5a1 826d 8152 3514 8f58 e548
013f4d0 e324 f6ba 0f37 7f7d 5a56 cfa4 cd92 bf54
013f4e0 59fc 98fd 89cb be41 34e0 4d68 10e9 896c
013f4f0 974e 0bf8 48b1 e346 246b 8880 72aa 09cd
013f500 1881 7b00 db07 32ec 3e6b 5d43 644e 715c
013f510 70c2 7a0a 9305 dcd7 4c7b 89bc 8c2b cd91
013f520 c141 b9d6 a1f0 610b 69bd fb13 b73c 7986
013f530 4b64 0b29 a19c 6299 a487 6864 d984 0392
013f540 a7e8 d05d 7199 5bf3 d712 4fd3 f9a8 71db
013f550 1825 f38d fb83 aca5 184a d2dc ec12 a857
```

Fig. 18. Task5-1.sh running

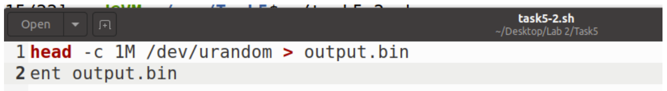
When I ran task 5-1 I noticed the hexdump data seemed to stall briefly when I moved my mouse. I am assuming this is because /dev/urandom is combining the randomness generated from my mouse movements with the seed data it is using to generate its output causing the brief lag. When I leave the VM alone and allow task5-1.sh to run on its own, there does not appear to be any lag or stuttering.

In Task 5-2 I was asked to utilize the ent tool to measure the quality of the random number. The script provided first ran the /dev/urandom command, generating 1 MB of data, and then stored the output to output.bin binary file. Below is my output from running this command.



```
[03/15/22]seed@VM:~/.../Task5$ ./task5-2.sh
Entropy = 7.999800 bits per byte.

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

Chi square distribution for 1048576 samples is 290.49, and randomly
would exceed this value 6.26 percent of the times.

Arithmetic mean value of data bytes is 127.5806 (127.5 = random).
Monte Carlo value for Pi is 3.138897472 (error 0.09 percent).
Serial correlation coefficient is -0.000315 (totally uncorrelated = 0.0).
[03/15/22]seed@VM:~/.../Task5$
```

Fig. 19. Task5-2 output



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

Fig. 20. Task5-2.sh script

My outcome from running task 5-2 can be seen in figure 19. In order to determine if my returned values were a good source of entropy I search online and found a man page which described the ent library function in detail[3]. (The url can be found here: http://www.linuxcertif.com/man/1/ent/

After reviewing the above ent documentation I came to the follow conclusions about my data:

Entropy: The entropy of my generated data was calculated to be 7.99800 bits per byte. This essentially means my data is nearly completely random.

Optimum Compression Reduction: I received the output that my data would be able to be compressed by 0 percent. This test further validates my assumption that the output of the command 1M /dev/urandom is actually random enough to be useful as pseudo random numbers.

Chi-Square Test: "The chi-square test is the most commonly used test for the randomness of data, and is extremely sensitive to errors in pseudorandom sequence generators."[3]. "We interpret the percentage as the degree to which the sequence tested is suspected of being non-random. If the percentage is greater than 99% or less than 1%, the sequence is almost certainly not random. If the percentage is between 99% and 95% or between 1% and 5%, the sequence is suspect. Percentages between 90% and 95% and 5% and 10% indicate the sequence is "almost suspect"." [3]. The above information from my source provided me with the ability to understand the Chi-square test. The distribution of the chi squared test of my data was 290.49 for 1048576 samples. This value would randomly be exceeded 6.26 percent of the time. This means my data would be put in the category of "almost suspect" as noted by my source.

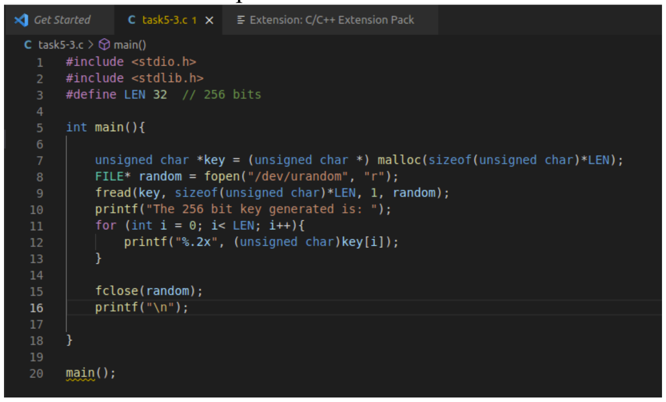
Arithmetic mean value: The arithmetic mean value sums all the bytes and divides that sum by the file length. According to the ent man page I good random value would be around 127.5. The value return from my data was 127.5806 meaning my result is random.

Monte Carlo value: The value for pi from my data was 3.138897472 which gives an error rate of 0.09 percent against the true value of pi. The closer the value is to the real value of pi represents the more random your data is: "the value will approach the correct value of Pi if the sequence is close to random." [3]. My resulting Monte Carlo value demonstrates my data is random.

Serial Correlation Coefficient: "This quantity measures the extent to which each byte in the file depends upon the previous byte." [3]. Serial Correlation Coefficient basically measures how much the current byte depends on the byte that came right before it. This returns a value from 0-1 (positive or negative) with 0 having no correlation and 1 being completely correlated. The value from my data was -.000315 meaning the data was totally uncorrelated.

For task 5-2 my generated data was random enough to be used for cryptographic purposes.

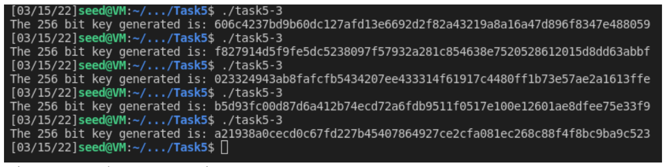
For task 5-3 I was asked to take the given code snippet and modify it to generate a 256 bit encryption key using /dev/urandom since in task 5-2 I determined urandom was basically just as good as /dev/random for pseudo random number generation. /dev/urandom also comes without the drawback of limited the limited throughput /dev/random has. For this task I installed Visual Studio Code in my VM to make modifying the given code easier. I also installed the C/C++ Extension pack for VS code.



```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);
    printf("The 256 bit key generated is: ");
    for (int i = 0; i< LEN; i++){
        printf("%.2x", (unsigned char)key[i]);
    }

    fclose(random);
    printf("\n");

}

main();
```

Fig. 21. Task5-3.c code to generate 256 bit encryption key with urandom

When writing this code I was unfamiliar with the fread() function. I looked up an article from geekforgeeks.com[4] which allowed me to gain a greater understanding of the function. After this I function call I used a modified code snippet from task 1 to print off each of the key characters. The Length variable was also changed to 32 to generate a 256 bit encryption key instead of a 128 bit encryption key. After building task 5-3, running it generated the following output seen in figure 22.



```
[03/15/22]seed@VM:~/.../Task5$ ./task5-3
The 256 bit key generated is: 606c4237bd9b60dc127afd13e6692d2f82a43219a8a16a47d896f8347e488059
[03/15/22]seed@VM:~/.../Task5$ ./task5-3
The 256 bit key generated is: f827914d5f9fe5dc5238097f57932a281c854638e7520528612015d8dd63abbf
[03/15/22]seed@VM:~/.../Task5$ ./task5-3
The 256 bit key generated is: 023324943ab8fafcfb5434207ee433314f61917c4480ff1b73e57ae2a1613ffe
[03/15/22]seed@VM:~/.../Task5$ ./task5-3
The 256 bit key generated is: b5d93fc00d87d6a412b74ecd72a6fdb9511f0517e100e12601ae8dfee75e33f9
[03/15/22]seed@VM:~/.../Task5$ ./task5-3
The 256 bit key generated is: a21938a0cecd0c67fd227b45407864927ce2cfa081ec268c88f4f8bc9ba9c523
[03/15/22]seed@VM:~/.../Task5$
```

Fig. 22. Task5-3.c running

Each time task5-3.c is run a different encryption key is generated utilizing the pseudo random data seed output from /dev/urandom.

III. Conclusion

I was able to learn a lot from this lab that I did not know previously. As mentioned previously my understanding of pseudorandom number generation before this lab was not very good. By completing this lab, I now have a much better grasp pseudo random number generation and understand that it can be complemented with real random numbers to increase the entropy of the kernel. This allows for cryptographic use of pseudo random numbers.

References

[1] "C library function - srand()." https://www.tutorialspoint.com/c_standard_library/c_function_srand.htm (accessed Mar. 10, 2022).
[2] "rand() and srand() in C/C++," GeeksforGeeks, Jun. 23, 2017. https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/ (accessed Mar. 10, 2022).
[3] "Linux Certif - Man ent(1)." http://www.linuxcertif.com/man/1/ent/ (accessed Mar. 15, 2022).
[4] "fread() function in C++," GeeksforGeeks, Aug. 18, 2018. https://www.geeksforgeeks.org/fread-function-in-c/ (accessed Mar. 15, 2022).
[5] "SEED Project." https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_Random_Number/ (accessed Mar. 15, 2022).