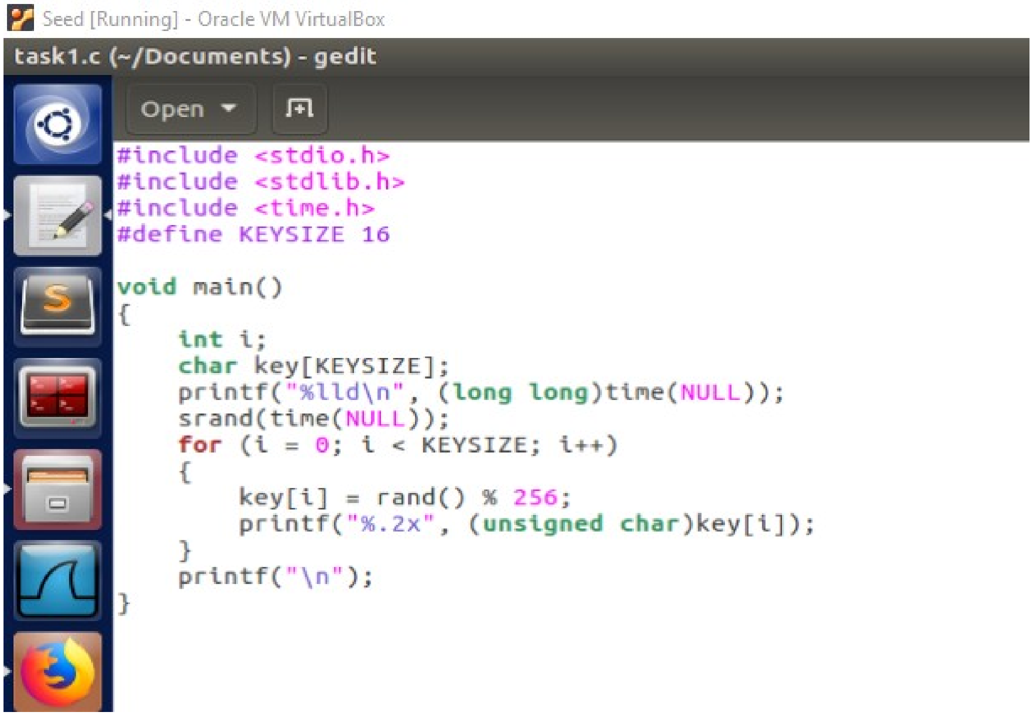
PSEUDO RANDOM NUMBER GENERATION LAB

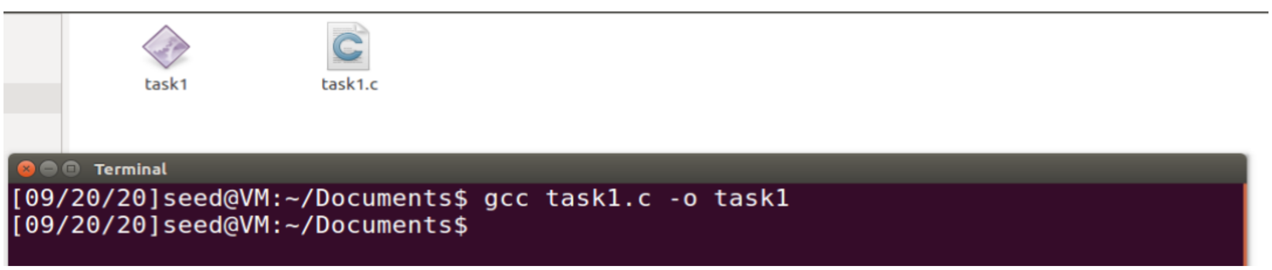
Table of Contents

Task 1



The above code will be used to generate random numbers. The code generates the seed using the srand() function. And takes the time() function as input for seed generation. The time function returns the value of time in seconds from 00:00:00 UTC, January 1, 1970.

Now let us execute this program multiple times and see the results. But before executing we need to compile the c program using the command shown below:



We can see that we get an executable file created in the directory when we run the gcc command.

Now let us view the output of the program

```
[09/20/20]seed@VM:~/Documents$ ./task1
1600612862
a6de6980d6a3fc05f55362cef3629874
[09/20/20]seed@VM:~/Documents$ ./task1
1600612869
30ed3c48e877346bc2373ca8354323ee
[09/20/20]seed@VM:~/Documents$ ./task1
1600612871
fbac9dd05a19ae587c2d9e71db95b589
[09/20/20]seed@VM:~/Documents$ ./task1
1600612873
b7b6ab399e66180084bbd298309aeae6
[09/20/20]seed@VM:~/Documents$ ./task1
1600612873
```

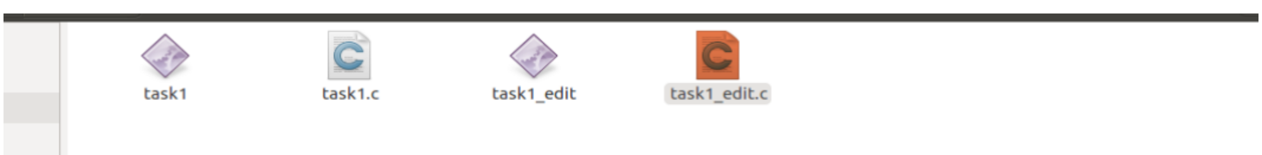
We can see that the random number generated are different because their seed was different.

Now if we comment out the srand() function as shown below we will see some changes to the code's output.

task1.c

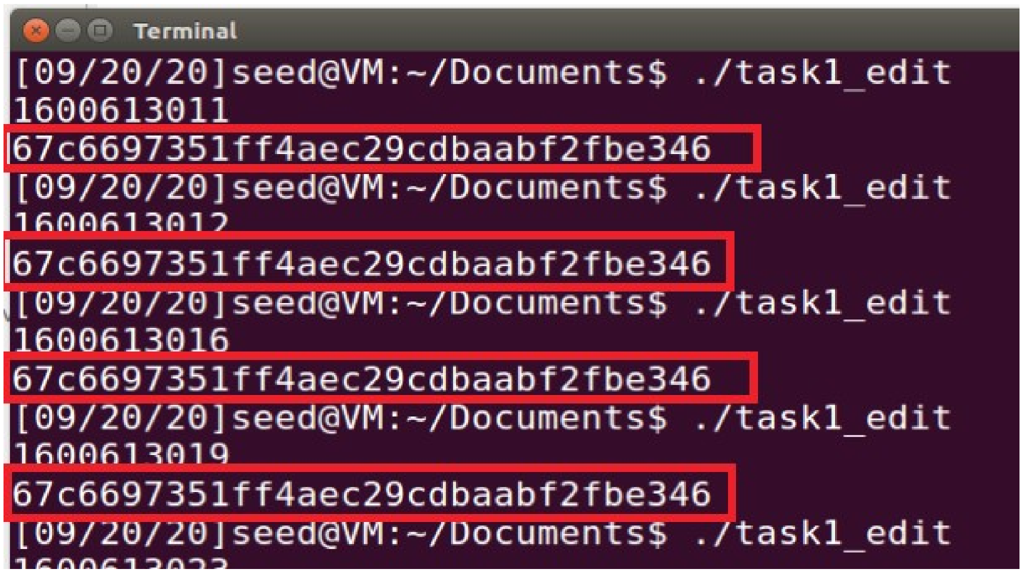
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16

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");
}
```

This is the code we will be using, and we can clearly see above that the srand() function is now commented. Similarly, as we did with the previous code, we need to compile this as well using gcc as shown below:




```
[09/20/20]seed@VM:~/Documents$ gcc task1_edit.c -o task1_edit
[09/20/20]seed@VM:~/Documents$
```

Now let us run is executable file task1_edit and observe the results.



Now the random numbers are same because the seed is not being changed, so each time we will get the same result if we run the program.

Task 2

First we need to find the time in seconds between our target time and the Epoch. We will use the command shown below:

```
Terminal
[09/20/20]seed@VM:~/Documents$ date -d "2018-04-17 23:08:49" +%s
1524020929
[09/20/20]seed@VM:~/Documents$
```

We can see that the value comes out to be 1524020929. Since the activity was mostly done 2 hours prior to this time we will write a program to give the value of all possible keys which we will use to find the real key. The code for this program is shown below:

```
task1.c × | task1_edit.c × | task2_keys.c × | keys.txt ×
#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++) // our 2 hour target window
    {
        srand(t);
        for (i = 0; i < KEYSIZE; i++)
        {
            key[i] = rand() % 256;
            printf("%.2x", (unsigned char)key[i]);
        }
        printf("\n");
    }
}
```

In the for loop we will start from 2 hours prior and reach our target time by adding 1 sec after each iteration. Now when we compile and run this program it will start displaying numerous values on the terminal like below:

```
[09/20/20]seed@VM:~/Documents$ ./task2_keys
0a6226fc01a201b82b7d42caa7de3e05
12d494f3e5506c3fc152d68ae5d35bc8
64b838761768baa431899b84dc5bbed0
fd9b1b3ae04452506a7f269b77d95e8e
9a45a8c0eea61d185e2e896ea1e96167
bd858bd80cb81b68981fc6ab1f6b6ff0
405350cf2bf03e912e03bba28a2a3cc6
66b32d34c8315750343b34fbf329b8b5
794885b8757f4791ee06970ed2c1f92b
c270b9219acd47d50997e8404ef066f3
6203e205ae67a8ae76871c3061f1a8a7
018f4c7bd0b7ba866f27560b599540e5
```

So, it will be better if we redirect the output of the program to a text file which we can easily use as a dictionary when we launch our brute force attack. We will do this by using the command shown below:

```
Terminal
[09/20/20]seed@VM:~/Documents$ ./task2_keys >> keys.txt
[09/20/20]seed@VM:~/Documents$
```

We can see below that our dictionary file is successfully created.

```
task1.c        task1_edit.c        task2_keys.c        keys.txt
0a6226fc01a201b82b7d42caa7de3e05
12d494f3e5506c3fc152d68ae5d35bc8
64b838761768baa431899b84dc5bbed0
fd9b1b3ae04452506a7f269b77d95e8e
9a45a8c0eea61d185e2e896ea1e96167
bd858bd80cb81b68981fc6ab1f6b6ff0
405350cf2bf03e912e03bba28a2a3cc6
66b32d34c8315750343b34fbf329b8b5
794885b8757f4791ee06970ed2c1f92b
c270b9219acd47d50997e8404ef066f3
6203e205ae67a8ae76871c3061f1a8a7
018f4c7bd0b7ba866f27560b599540e5
da0178625826d50ecbcc10361d351a8a
4e1c67274a2434aa9d2461d9db86266d
7ecf3134dd870a2397da6b469802d0dd
68c87b6f9a4df9ae0a1f7d99d2458935
6e7b43cf01ab2d03bb37db5796b4ce72
19640ceb1984f19c9fc8978aea81ea8b
59b7a75473ea9cb18cc76a85209633a6
164838987cba367c53fb418b5e18f9f1
dde0a306a3a564f4cb8ccefbbef11017
691a489da8eb95542c9a32cd177d6398
8c8e9425e5b9799cdfc8c8e76e86fce6
2dca1ff950d5ade1698647d99f94eabc
c6426944a1a230163c50a4ea2c37eba7
81b28317cab6d081ed228f8522102177
5cb1eea488990c612c46558a802e00c9
0671eb7a30349893c7375cfc917e3110
52e34e56fa4845f2a129d2a75fca150f
62458880c6c7008f33e0ed381dc8e55c
5b5295c40428bc28d648cd71b7923387
06d32a82e19cc28c71960e8f39744e0c
918838e1bac38531deb9d6a7b7f2e1e5
34919c03ad3f72d64c5a92b0b49760ac
c99716ae97d3d63913f1246de3a01644
e58cd77285fd7a0f6931d5eec40a3ce9
a8394a1645402532d88dd9dacf43f88b
0b6259165678afa0af8597067f1276b3
```

Now we will write a short python program which contains the plaintext, ciphertext and the initialization vector and will use the keys.txt dictionary file to find out or real key. The code is shown below:

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

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

with open('keys.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(plaintext)
    if guess == ciphertext:
        print("Key Found:", k)
        exit(0)

print("Key Not Found!")
```

The code is very straight forward. We provide all the plaintext, ciphertext and the IV values as given in the question. We provide the key.txt file and also the AES mode which is AES CBC mode. Now when the code runs it will try to create a ciphertext of the provided plaintext and when the new ciphertext matches the original ciphertext our key will be found.

We get the following result when we run the code:

```
[09/20/20]seed@VM:~/Documents$ python3 task2_find_key.py
Key Found: 95fa2030e73ed3f8da761b4eb805dfd7
```

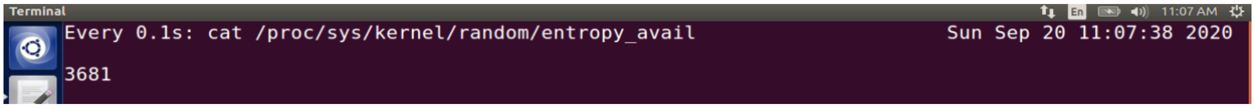
We can see that the key is found successfully.

Task 3

For this task we analyze the available entropy the system has using the command below:

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

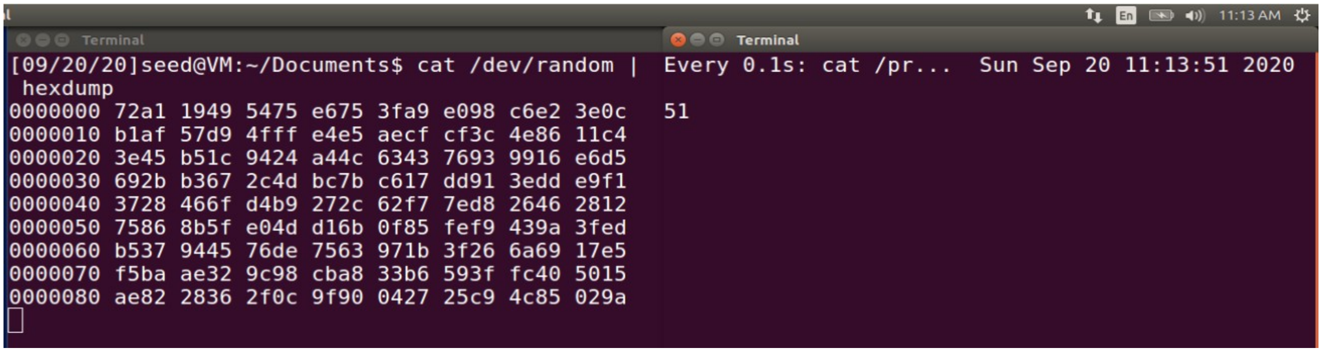
The watch command will execute the cat statement after every 0.1 second so that we can monitor the randomness entropy to the system.



If we do not move the mouse or do other activities this value changes very slowly. But if we move and click the mouse, or do other tasks like read files, browse the internet etc. This value is changed very quickly. The value changes most quickly when we click and move the mouse simultaneously.

Task 4

Now we will monitor the /dev/random device and how it generates the random number and we will also monitor its effect on the available entropy file. To do this we will open both 2 terminals side by side and view the results as shown below:



Whenever we do some random action like clicking or moving the mouse the entropy increases. And after some time, a new line of random number appears in the /dev/random and causes the entropy to decrease. So by this behavior we can say that the /dev/random uses the available entropy to generate new random numbers.

QUESTION: If a server uses /dev/random to generate the random session key with a client. Please describe how you can launch a DenialOfService (DOS) attack on such a server.

Since the session IDs are generated using /dev/random an attacker can launch an attack where they request multiple sessions IDs from the server. Now since the values in /dev/random are limited so if an attacker can get the server to send them all the session IDs then legitimate new users would not be able to use that web application because server would be unbale to provide them with a session ID. And this issue of availability will cause a Denial-Of-Service (DOS) attack on the server.

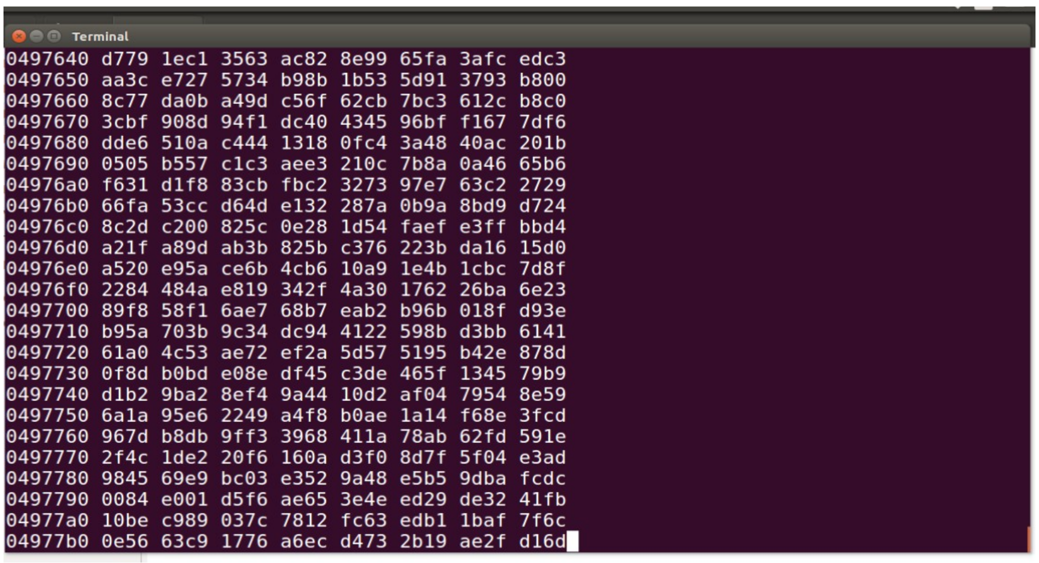
Task 5

In this task we will observe the behaviour of /dev/urandom to generate random numbers.

First, we run the command shown below:

cat /dev/urandom | hexdump

We will get the results as shown below:

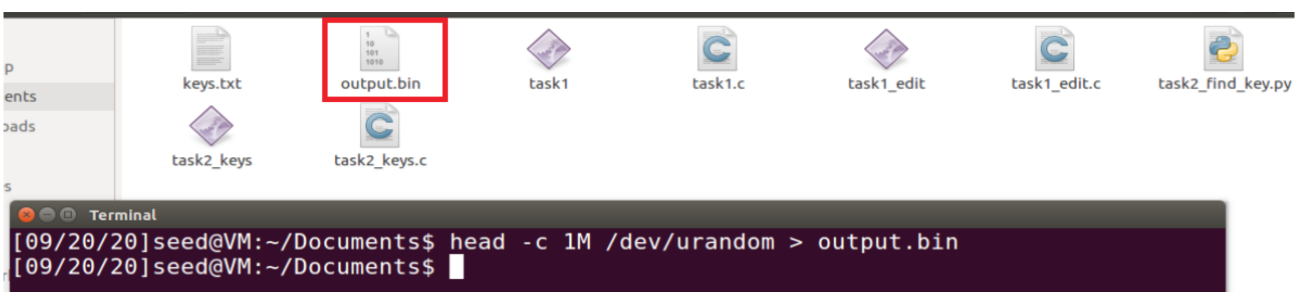


Moving or stopping the mouse does not have any affect on the random numbers being generated and they will keep on generating regardless of what we do.

Now we will run entropy tests on the random numbers created using /dev/urandom. For this we will first store some output values in a file using the command below:

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

The command will store the top 1MB of the results and save it in a file name output.bin as shown below



Now we will use the command below to study the entropy results of /dev/urandom

ent output.bin

We get the following results as shown below

```
[09/20/20]seed@VM:~/Documents$ ent output.bin
Entropy = 7.999834 bits per byte.

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

Chi square distribution for 1048576 samples is 241.68, and randomly
would exceed this value 71.59 percent of the times.

Arithmetic mean value of data bytes is 127.4762 (127.5 = random).
Monte Carlo value for Pi is 3.142445154 (error 0.03 percent).
Serial correlation coefficient is -0.000659 (totally uncorrelated = 0.0).
[09/20/20]seed@VM:~/Documents$
```

By looking at the entropy of the results in significant meaning that the file has good randomization and that the compression of the data would be unlikely to reduce its size.

For the last part of this task we will be using /dev/urandom to generate a 256 bits key, using the code shown below:

```c
#include <stdio.h>
#include <stdlib.h>
#define LEN 32 // since now we are generating a 256 bits value

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

Here we have used length as 32 bytes as we need to create a 256-bit key, because 32*8=256.

Now we just need to compile the code using gcc and execute it, and we will get out required key as output which is shown below:

```
[09/20/20]seed@VM:~/Documents$ gcc task5.c -o task5
[09/20/20]seed@VM:~/Documents$
```

```
[09/20/20]seed@VM:~/Documents$ ./task5
key = 6214d6b5c9bc31300daacc6b529f4ce232741a41d9490b651987199b145d7899
```