Random Number Generation Lab

Assignment 3

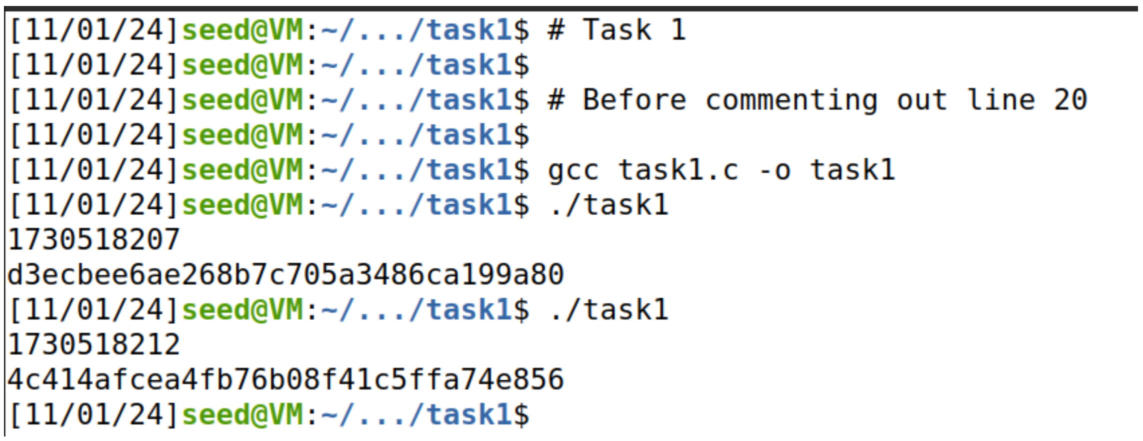
EDMUND LAWERH AMANOR Fundamentals of Info. Security, Fall 2024

This report is a documentation of my observation of the process while I simulated the **random number generation** lab in the SEED virtual machine. I provided screenshots, from the various stages.

I commented out the steps I took and observations made directly in the terminal window, both before and after executing commands to make is easy to follow what was happening.

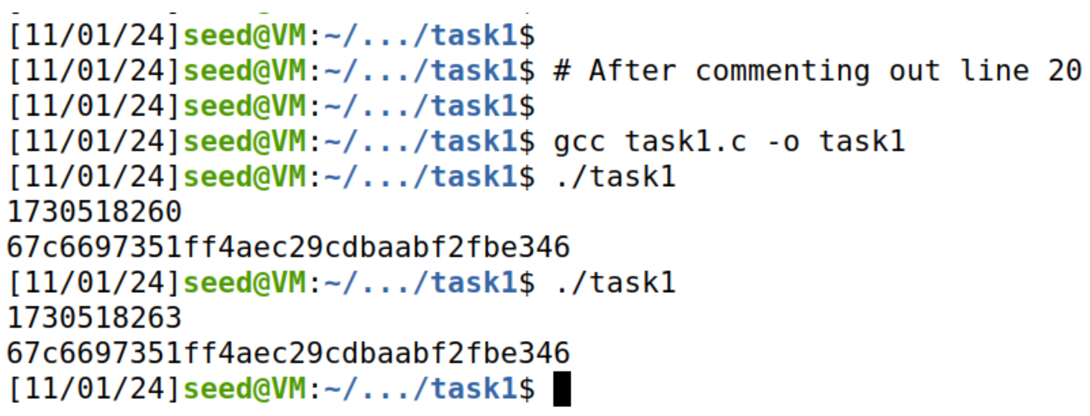
Task 1: Generate Encryption Key in a Wrong Way

- The image below shows the output of running *task*1.*c* before commenting out the *srand*() function call. I observed that when the *srand*() function call is used, the output of the key is really random and unpredictable even after quick successive runs of the program.



```
[11/01/24]seed@VM:~/.../task1$ # Task 1
[11/01/24]seed@VM:~/.../task1$
[11/01/24]seed@VM:~/.../task1$ # Before commenting out line 20
[11/01/24]seed@VM:~/.../task1$
[11/01/24]seed@VM:~/.../task1$ gcc task1.c -o task1
[11/01/24]seed@VM:~/.../task1$ ./task1
1730518207
d3ecbee6ae268b7c705a3486ca199a80
[11/01/24]seed@VM:~/.../task1$ ./task1
1730518212
4c414afcea4fb76b08f41c5ffa74e856
[11/01/24]seed@VM:~/.../task1$
```

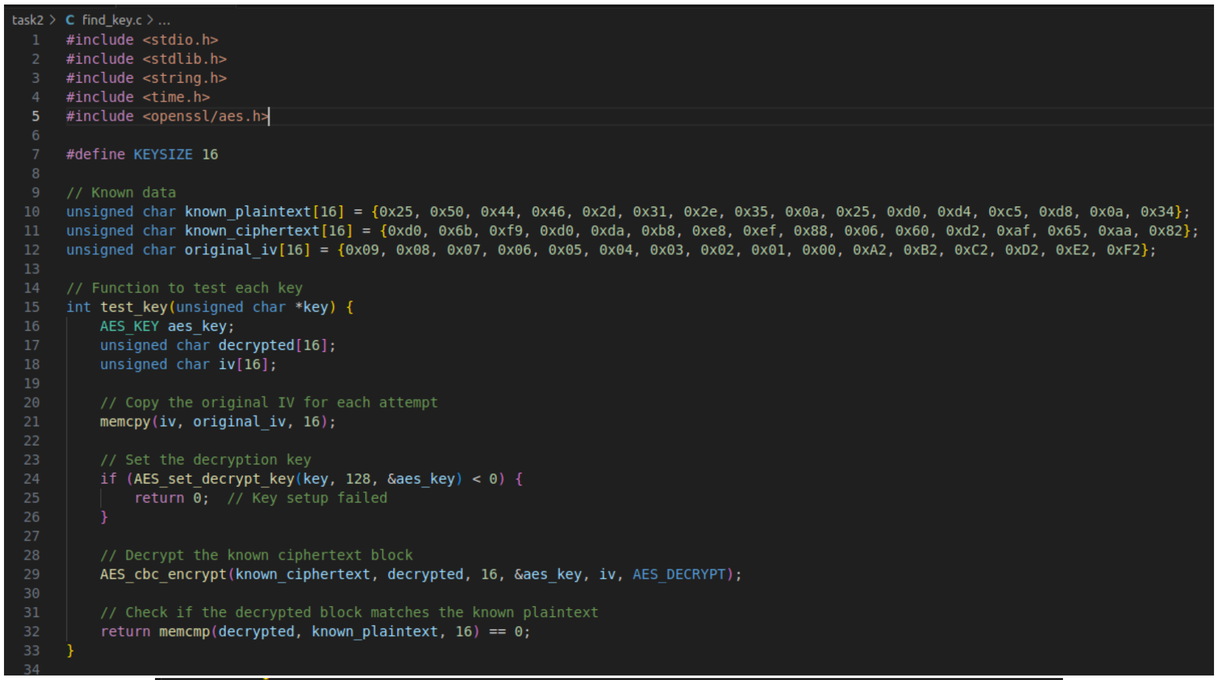
- Next, I show the keys generated when line 20 of the code is commented. I observed that the same predictable key is generated when the *srand*() function call is not used. This is because, every time the program is run, the same sequence of seed values are used hence the predictable keys generated. So the purpose of *srand*() is to initialize the random number generator with a seed. This ensures that the sequence of random numbers generated by *rand*() are different each time the program runs.



```
[11/01/24]seed@VM:~/.../task1$
[11/01/24]seed@VM:~/.../task1$ # After commenting out line 20
[11/01/24]seed@VM:~/.../task1$
[11/01/24]seed@VM:~/.../task1$ gcc task1.c -o task1
[11/01/24]seed@VM:~/.../task1$ ./task1
1730518260
67c6697351ff4aec29cdbaabf2fbe346
[11/01/24]seed@VM:~/.../task1$ ./task1
1730518263
67c6697351ff4aec29cdbaabf2fbe346
[11/01/24]seed@VM:~/.../task1$
```

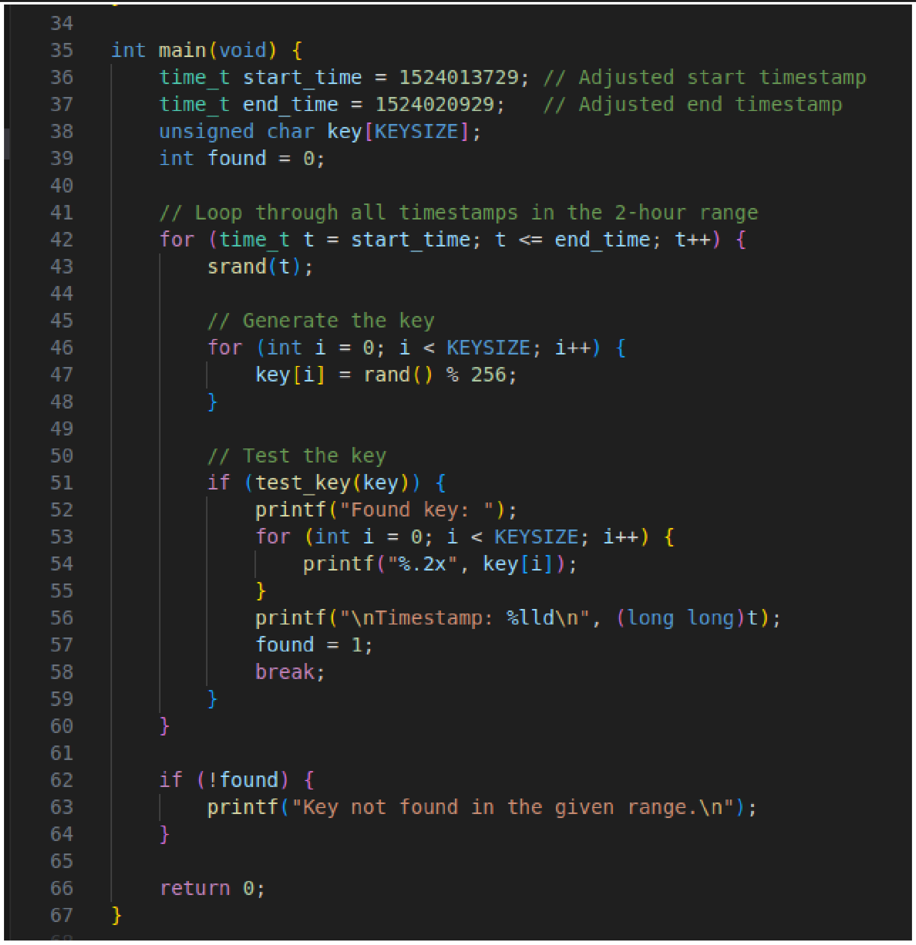
Task 2: Guessing the Key

First, I show the modified version of the code in task1.c which I use to find the key. The image below is a snapshot of the code.



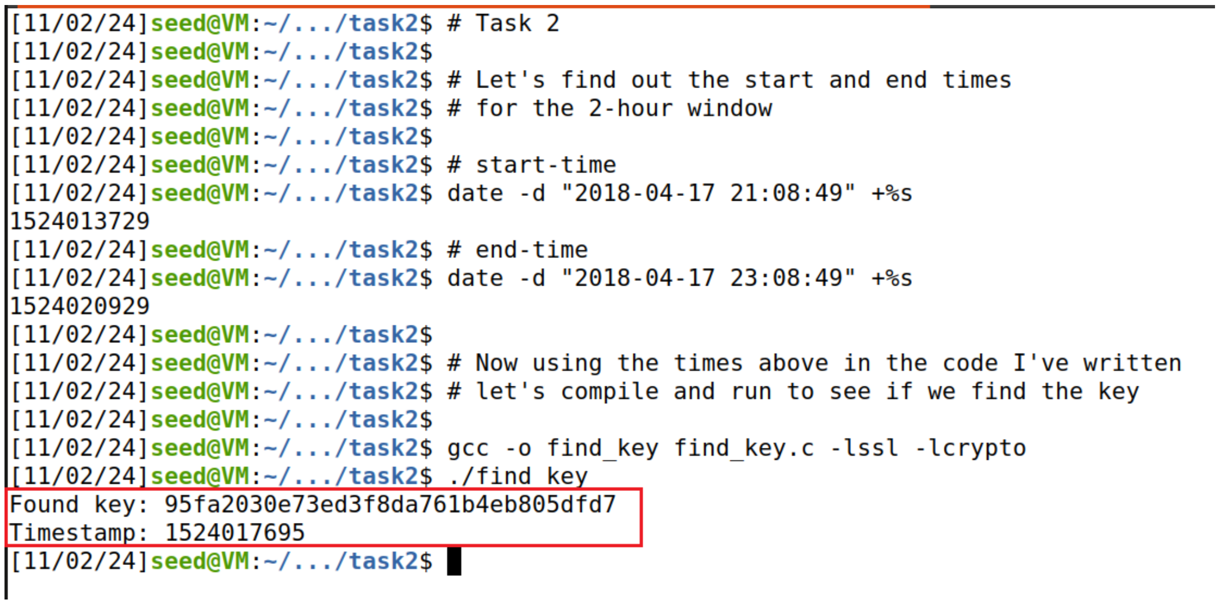
task2 > C find_key.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <openssl/aes.h>

#define KEYSIZE 16

// Known data
unsigned char known_plaintext[16] = {0x25, 0x50, 0x44, 0x46, 0x2d, 0x31, 0x2e, 0x35, 0x0a, 0x25, 0xd0, 0xd4, 0xc5, 0xd8, 0x0a, 0x34};
unsigned char known_ciphertext[16] = {0xd0, 0x6b, 0xf9, 0xd0, 0xda, 0xb8, 0xe8, 0xef, 0x88, 0x06, 0x60, 0xd2, 0xaf, 0x65, 0xaa, 0x82};
unsigned char original_iv[16] = {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0xA2, 0xB2, 0xC2, 0xD2, 0xE2, 0xF2};

// Function to test each key
int test_key(unsigned char *key) {
    AES_KEY aes_key;
    unsigned char decrypted[16];
    unsigned char iv[16];

    // Copy the original IV for each attempt
    memcpy(iv, original_iv, 16);

    // Set the decryption key
    if (AES_set_decrypt_key(key, 128, &aes_key) < 0) {
        return 0;  // Key setup failed
    }

    // Decrypt the known ciphertext block
    AES_cbc_encrypt(known_ciphertext, decrypted, 16, &aes_key, iv, AES_DECRYPT);

    // Check if the decrypted block matches the known plaintext
    return memcmp(decrypted, known_plaintext, 16) == 0;
}
```



```c

int main(void) {
    time_t start_time = 1524013729; // Adjusted start timestamp
    time_t end_time = 1524020929;   // Adjusted end timestamp
    unsigned char key[KEYSIZE];
    int found = 0;

    // Loop through all timestamps in the 2-hour range
    for (time_t t = start_time; t <= end_time; t++) {
        srand(t);

        // Generate the key
        for (int i = 0; i < KEYSIZE; i++) {
            key[i] = rand() % 256;
        }

        // Test the key
        if (test_key(key)) {
            printf("Found key: ");
            for (int i = 0; i < KEYSIZE; i++) {
                printf("%.2x", key[i]);
            }
            printf("\nTimestamp: %lld\n", (long long)t);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("Key not found in the given range.\n");
    }

    return 0;
}
```

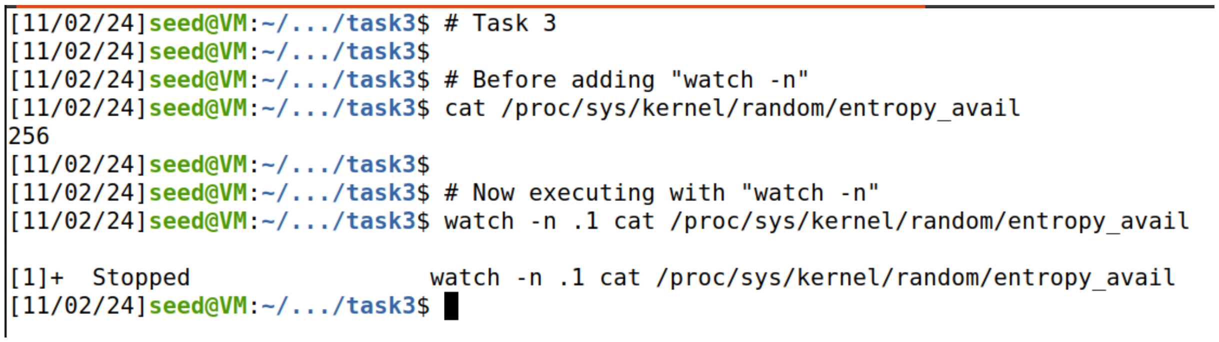
- The next snapshot show the process of code compilation and running to find the key. And it was successfully found and displayed as shown below:



```
[11/02/24]seed@VM:~/.../task2$ # Task 2
[11/02/24]seed@VM:~/.../task2$
[11/02/24]seed@VM:~/.../task2$ # Let's find out the start and end times
[11/02/24]seed@VM:~/.../task2$ # for the 2-hour window
[11/02/24]seed@VM:~/.../task2$
[11/02/24]seed@VM:~/.../task2$ # start-time
[11/02/24]seed@VM:~/.../task2$ date -d "2018-04-17 21:08:49" +%s
1524013729
[11/02/24]seed@VM:~/.../task2$ # end-time
[11/02/24]seed@VM:~/.../task2$ date -d "2018-04-17 23:08:49" +%s
1524020929
[11/02/24]seed@VM:~/.../task2$
[11/02/24]seed@VM:~/.../task2$ # Now using the times above in the code I've written
[11/02/24]seed@VM:~/.../task2$ # let's compile and run to see if we find the key
[11/02/24]seed@VM:~/.../task2$
[11/02/24]seed@VM:~/.../task2$ gcc -o find_key find_key.c -lssl -lcrypto
[11/02/24]seed@VM:~/.../task2$ ./find_key
Found key: 95fa2030e73ed3f8da761b4eb805dfd7
Timestamp: 1524017695
[11/02/24]seed@VM:~/.../task2$
```

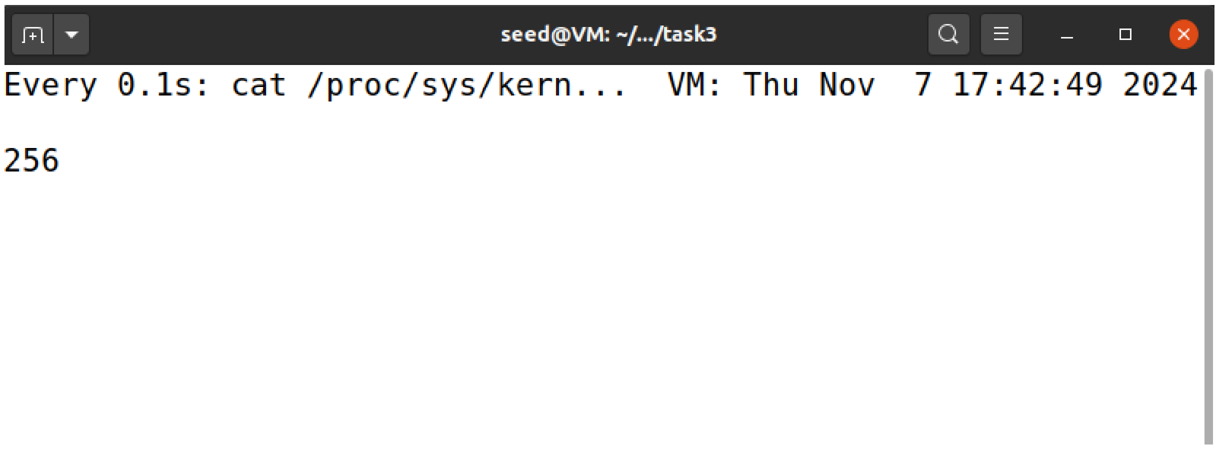
Task 3: Measure the Entropy of Kernel

In this task, the expectation was to observe how the entropy of the system changes when enough randomness is present through activities like mouse clicks and keystrokes from the keyboard. The snapshot below shows that the entropy of the system is 256, before running the same command with "watch".



```
[11/02/24]seed@VM:~/.../task3$ # Task 3
[11/02/24]seed@VM:~/.../task3$
[11/02/24]seed@VM:~/.../task3$ # Before adding "watch -n"
[11/02/24]seed@VM:~/.../task3$ cat /proc/sys/kernel/random/entropy_avail
256
[11/02/24]seed@VM:~/.../task3$
[11/02/24]seed@VM:~/.../task3$ # Now executing with "watch -n"
[11/02/24]seed@VM:~/.../task3$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail

[1]+  Stopped                 watch -n .1 cat /proc/sys/kernel/random/entropy_avail
[11/02/24]seed@VM:~/.../task3$
```

- The next snapshot shows my observation when I run with the watch command, and tried to introduce some randomness through mouse movements and keyboard strokes.

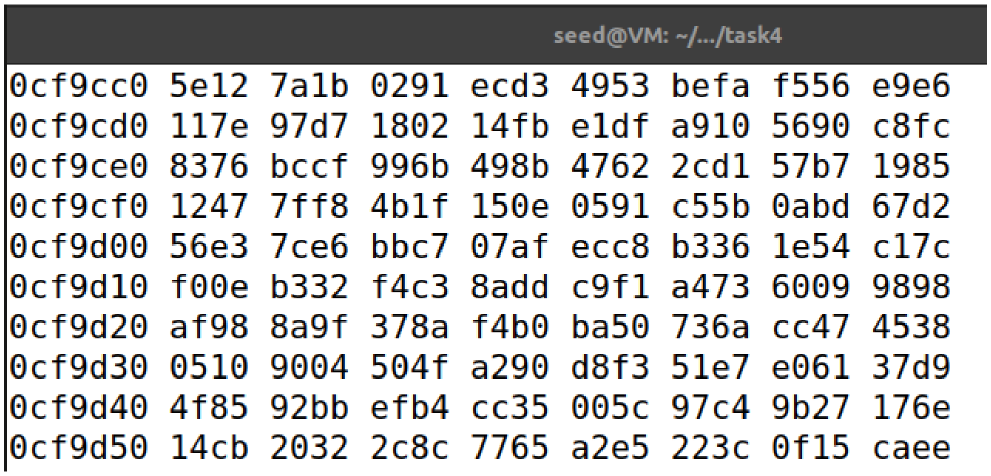


```
Every 0.1s: cat /proc/sys/kern...   VM: Thu Nov  7 17:42:49 2024

256
```

- **Observation**: Even though I did a lot of mouse movements and provided a lot of keyboard strokes, the entropy of the system remained at 256, which is not what we ideally expected. However, this is a result of a patch that has been introduced in the recent Linux systems that capped the entropy at 256. Hence my mouse movements and keyboard strokes didn't causes any entropy changes as expected.

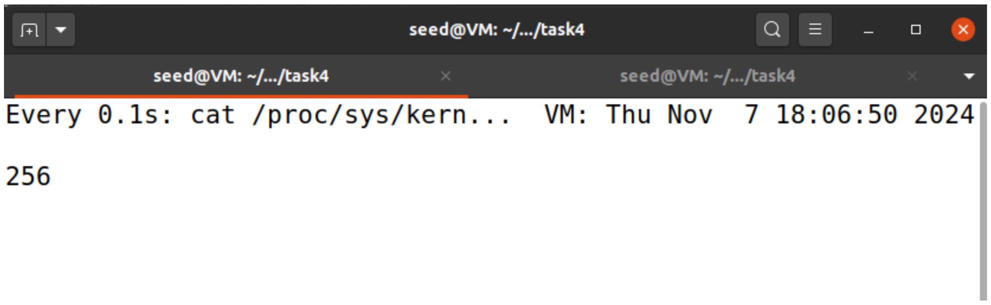
Task 4: Get Pseudo Random Numbers from /dev/random

The goal here is to observe pseudo random number from */dev/random*. So still with the command from task 3 that enabled us to observe the entropy still running, I observed how the entropy of the system is affected when random numbers are generated using */dev/random*. The image below is a snapshot of running the $\$cat/dev/random|hexdump$ to generate random numbers.



```
0cf9cc0 5e12 7a1b 0291 ecd3 4953 befa f556 e9e6
0cf9cd0 117e 97d7 1802 14fb e1df a910 5690 c8fc
0cf9ce0 8376 bccf 996b 498b 4762 2cd1 57b7 1985
0cf9cf0 1247 7ff8 4b1f 150e 0591 c55b 0abd 67d2
0cf9d00 56e3 7ce6 bbc7 07af ecc8 b336 1e54 c17c
0cf9d10 f00e b332 f4c3 8add c9f1 a473 6009 9898
0cf9d20 af98 8a9f 378a f4b0 ba50 736a cc47 4538
0cf9d30 0510 9004 504f a290 d8f3 51e7 e061 37d9
0cf9d40 4f85 92bb efb4 cc35 005c 97c4 9b27 176e
0cf9d50 14cb 2032 2c8c 7765 a2e5 223c 0f15 caee
```

- While generating the random numbers using the */dev/random* command, I also run the "kernel entropy command" to observe the how the generation of random numbers affected the entropy of the system. The image below shows the terminal output.



```
Every 0.1s: cat /proc/sys/kern...  VM: Thu Nov  7 18:06:50 2024

256
```

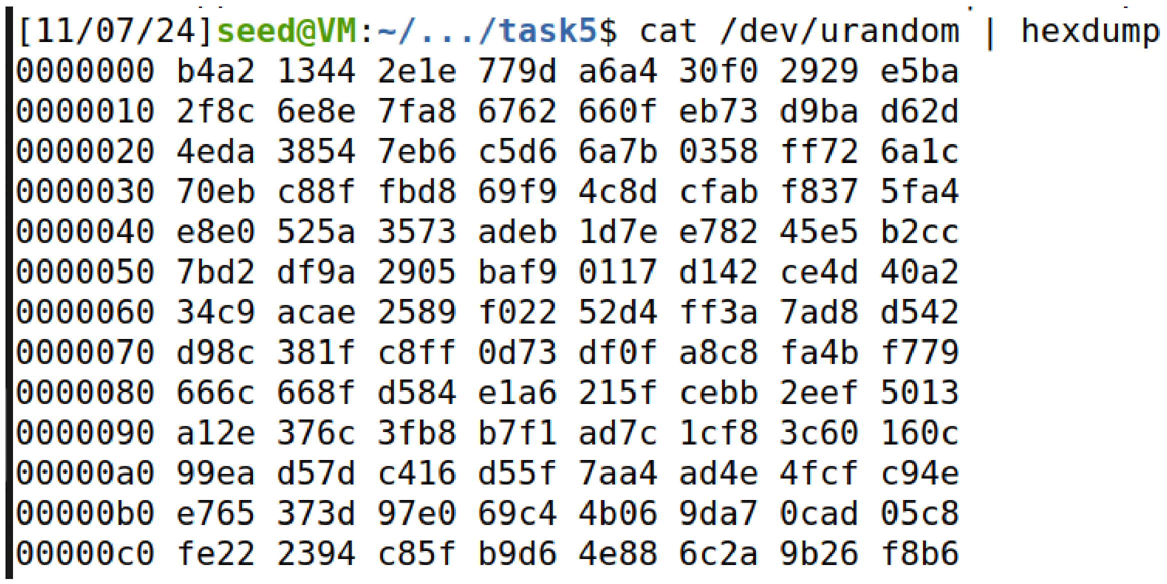
- **Expectation**: I expected to see the entropy of the system to decrease while generating the random numbers using the */dev/random* command. This is expected because /dev/random is a blocking device, meaning it will stop outputting data when the entropy pool is exhausted until more randomness is gathered from system events (e.g., mouse movements, keyboard strokes, etc.).

- **Observation**: Contrary to the expectation above, I observed that the entropy of my system remained at 256 without any changes. And this observation was made both when I didn't move my mouse and when I moved my mouse. And the reason for this observation is due to the patch that has been introduced in recent Linux systems as explained for task 3 above.

- **DoS Attack Explanation**: If a server depends on /dev/random for generating cryptographic or session keys, it could face delays when the entropy is low. So I could exploit this by repeatedly requesting random data or running processes that drain the entropy pool faster than it can be replenished. This would prevent the server from generating the required random data promptly, potentially leading to a Denial-of-Service (DoS) scenario where the server's performance is hindered, or it becomes temporarily unavailable due to the lack of sufficient entropy.

Task 5: Get Random Numbers from /dev/urandom

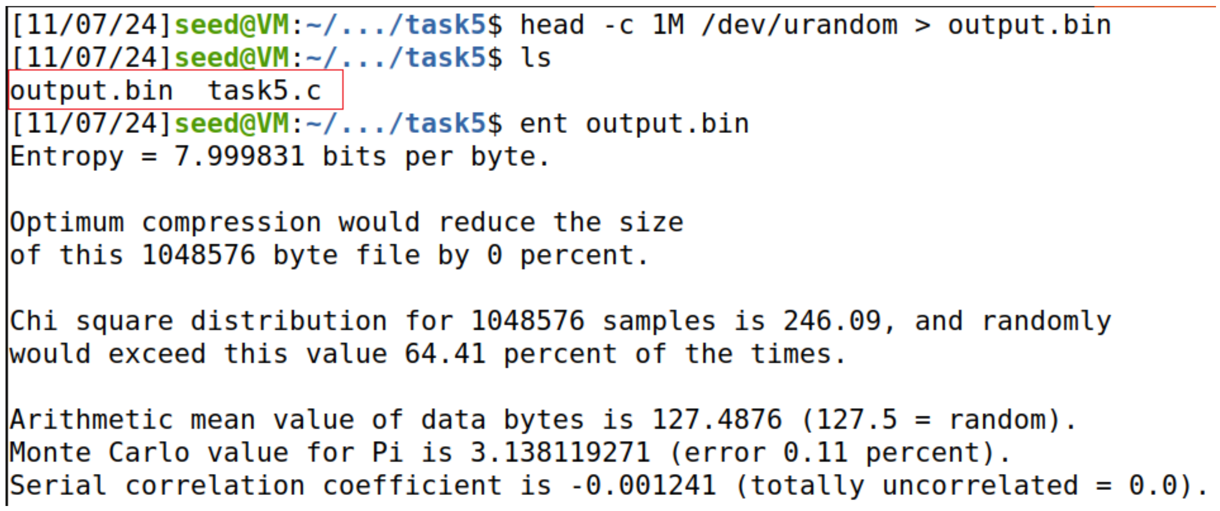
Since /dev/urandom is a non-blocking device, the goal is to observe how it differs from /dev/random. The image below shows the use of /dev/urandom to generate random numbers.



```
[11/07/24]seed@VM:~/.../task5$ cat /dev/urandom | hexdump
0000000 b4a2 1344 2e1e 779d a6a4 30f0 2929 e5ba
0000010 2f8c 6e8e 7fa8 6762 660f eb73 d9ba d62d
0000020 4eda 3854 7eb6 c5d6 6a7b 0358 ff72 6a1c
0000030 70eb c88f fbd8 69f9 4c8d cfab f837 5fa4
0000040 e8e0 525a 3573 adeb 1d7e e782 45e5 b2cc
0000050 7bd2 df9a 2905 baf9 0117 d142 ce4d 40a2
0000060 34c9 acae 2589 f022 52d4 ff3a 7ad8 d542
0000070 d98c 381f c8ff 0d73 df0f a8c8 fa4b f779
0000080 666c 668f d584 e1a6 215f cebb 2eef 5013
0000090 a12e 376c 3fb8 b7f1 ad7c 1cf8 3c60 160c
00000a0 99ea d57d c416 d55f 7aa4 ad4e 4fcf c94e
00000b0 e765 373d 97e0 69c4 4b06 9da7 0cad 05c8
00000c0 fe22 2394 c85f b9d6 4e88 6c2a 9b26 f8b6
```

- Just like the tasks performed before this, the linux system patch makes it impossible to oberserve any significant changes on the entropy when /dev/urandom is used to generate random numbers. Though I expected the behavior of /dev/urandom to be slightly different from that of /dev/random, where, the random number generation does not pause due to insufficient entropy. The entropy remained the same irrespective of whether I move the mouse or not.

Analysis using ent tool



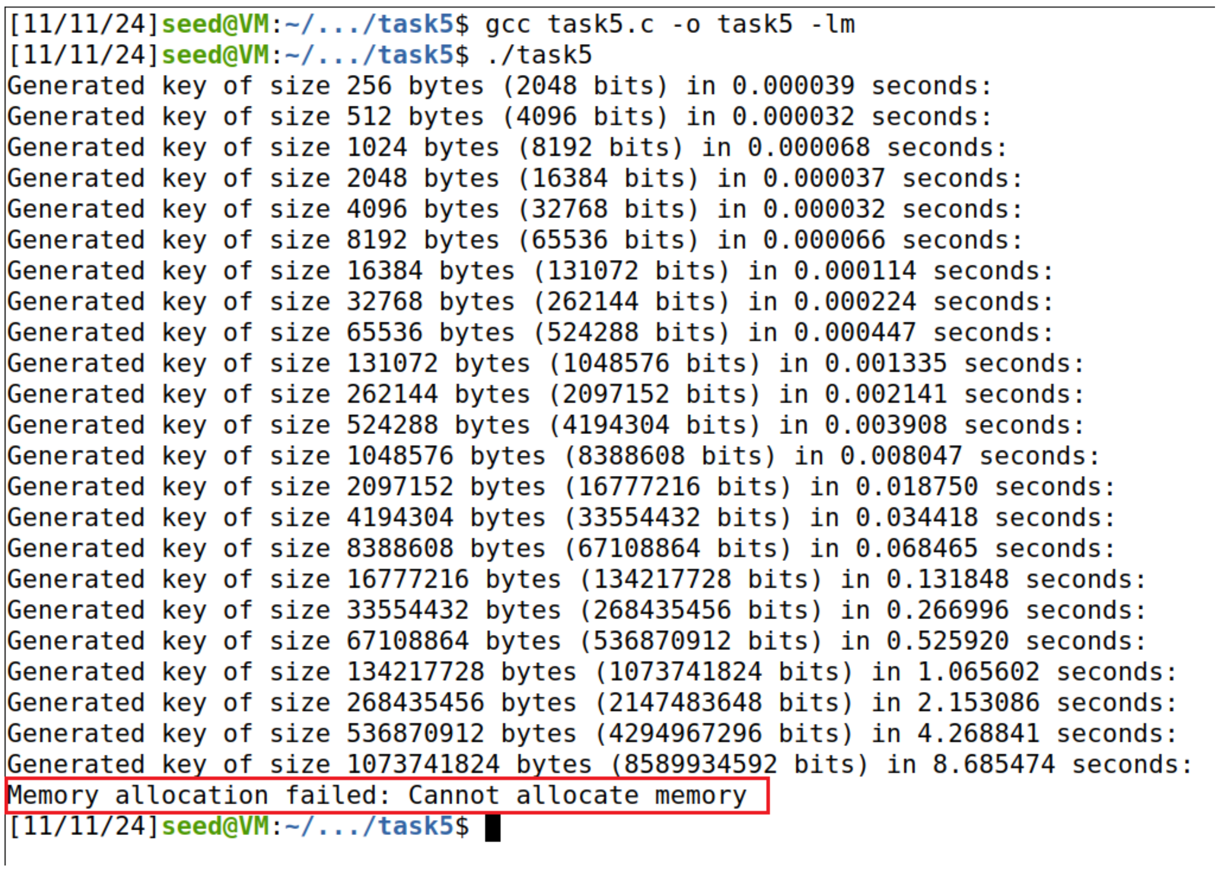
```
[11/07/24]seed@VM:~/.../task5$ head -c 1M /dev/urandom > output.bin
[11/07/24]seed@VM:~/.../task5$ ls
output.bin  task5.c
[11/07/24]seed@VM:~/.../task5$ ent output.bin
Entropy = 7.999831 bits per byte.

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

Chi square distribution for 1048576 samples is 246.09, and randomly
would exceed this value 64.41 percent of the times.

Arithmetic mean value of data bytes is 127.4876 (127.5 = random).
Monte Carlo value for Pi is 3.138119271 (error 0.11 percent).
Serial correlation coefficient is -0.001241 (totally uncorrelated = 0.0).
```

- The image above shows the analysis on the quality of the random number generator.
- Entropy close to 8.0 bits per byte indicates that each byte is highly random, as 8 bits per byte represents the maximum randomness (each byte could represent any value between 0 and 255 uniformly). This value suggests that the data generated has excellent randomness.

- Since the file cannot be compressed further, this indicates that the data does not have patterns or redundancy, supporting its randomness.
- The chi-square test checks for the distribution of byte values. A percentage near 50% suggests that the distribution of byte values closely resembles a uniform distribution. In this case, 64.41% indicates that the distribution is within an acceptable range of randomness, with no significant deviation from what is expected for random data.
- The mean value is very close to the ideal mean of 127.5, which is expected if byte values are uniformly distributed between 0 and 255. This further supports the data's randomness.
- The Monte Carlo method is used as a way to test randomness. The result is close to the actual value of π (3.14159265), with a small error of 0.11%. This is within acceptable limits, indicating the data behaves as expected for random input.
- The serial correlation measures the correlation between consecutive bytes. A value near 0.0 indicates that there is no correlation, meaning that each byte is independent of the previous one. The value here is very close to 0, indicating minimal correlation and supporting the randomness of the data.

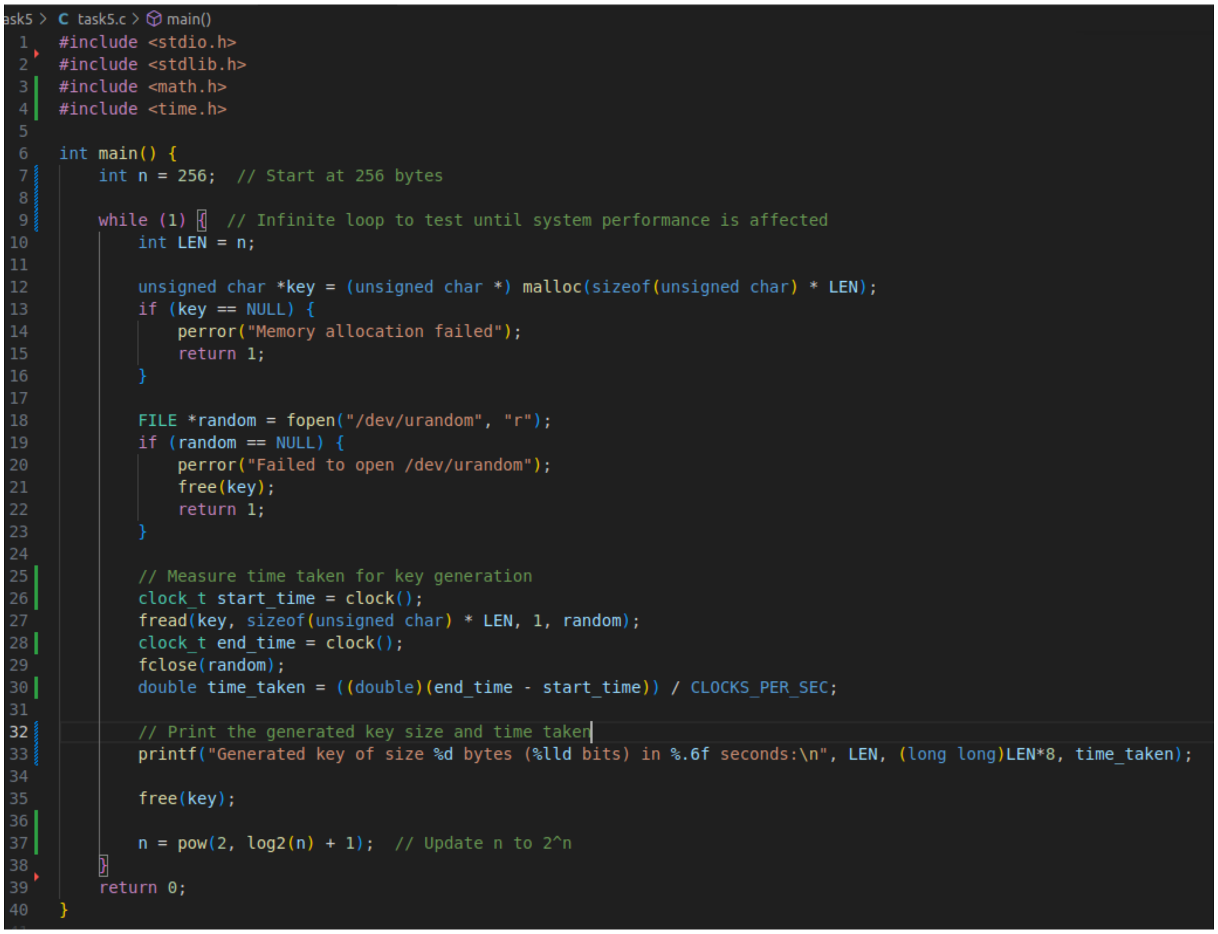
Observing Limitation on Key Size

Here, the goal is to observe if there is a limitation on the key size when using /dev/urandom to generate keys. The image below shows the compilation and running of the modified code sample to generate the keys.



```
[11/11/24]seed@VM:~/.../task5$ gcc task5.c -o task5 -lm
[11/11/24]seed@VM:~/.../task5$ ./task5
Generated key of size 256 bytes (2048 bits) in 0.000039 seconds:
Generated key of size 512 bytes (4096 bits) in 0.000032 seconds:
Generated key of size 1024 bytes (8192 bits) in 0.000068 seconds:
Generated key of size 2048 bytes (16384 bits) in 0.000037 seconds:
Generated key of size 4096 bytes (32768 bits) in 0.000032 seconds:
Generated key of size 8192 bytes (65536 bits) in 0.000066 seconds:
Generated key of size 16384 bytes (131072 bits) in 0.000114 seconds:
Generated key of size 32768 bytes (262144 bits) in 0.000224 seconds:
Generated key of size 65536 bytes (524288 bits) in 0.000447 seconds:
Generated key of size 131072 bytes (1048576 bits) in 0.001335 seconds:
Generated key of size 262144 bytes (2097152 bits) in 0.002141 seconds:
Generated key of size 524288 bytes (4194304 bits) in 0.003908 seconds:
Generated key of size 1048576 bytes (8388608 bits) in 0.008047 seconds:
Generated key of size 2097152 bytes (16777216 bits) in 0.018750 seconds:
Generated key of size 4194304 bytes (33554432 bits) in 0.034418 seconds:
Generated key of size 8388608 bytes (67108864 bits) in 0.068465 seconds:
Generated key of size 16777216 bytes (134217728 bits) in 0.131848 seconds:
Generated key of size 33554432 bytes (268435456 bits) in 0.266996 seconds:
Generated key of size 67108864 bytes (536870912 bits) in 0.525920 seconds:
Generated key of size 134217728 bytes (1073741824 bits) in 1.065602 seconds:
Generated key of size 268435456 bytes (2147483648 bits) in 2.153086 seconds:
Generated key of size 536870912 bytes (4294967296 bits) in 4.268841 seconds:
Generated key of size 1073741824 bytes (8589934592 bits) in 8.685474 seconds:
Memory allocation failed: Cannot allocate memory
[11/11/24]seed@VM:~/.../task5$
```

- The snapshot below is a snapshot of the modified code sample.



```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main() {
    int n = 256;  // Start at 256 bytes

    while (1) {  // Infinite loop to test until system performance is affected
        int LEN = n;

        unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char) * LEN);
        if (key == NULL) {
            perror("Memory allocation failed");
            return 1;
        }

        FILE *random = fopen("/dev/urandom", "r");
        if (random == NULL) {
            perror("Failed to open /dev/urandom");
            free(key);
            return 1;
        }

        // Measure time taken for key generation
        clock_t start_time = clock();
        fread(key, sizeof(unsigned char) * LEN, 1, random);
        clock_t end_time = clock();
        fclose(random);
        double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

        // Print the generated key size and time taken
        printf("Generated key of size %d bytes (%lld bits) in %.6f seconds:\n", LEN, (long long)LEN*8, time_taken);

        free(key);

        n = pow(2, log2(n) + 1);  // Update n to 2^n
    }
    return 0;
}
```

Observation:

- At 128 MB (134217728 bytes), key generation takes about 1 second.
- At 1 GB (1073741824 bytes), it takes over 8 seconds.
- This confirms that both memory usage and time taken increase significantly with each doubling of the key size.
- The system fails to allocate memory when attempting to allocate 2 GB or larger, and that is when my program exits. If I didn't have the error checking implemented, that probably would have caused my system to slow down or something instead of the code exiting at the return statement.
- So it's safe to say that my system can tolerate only up to 1GB key generation without any impact on performance.