Pseudo Random Number Generation Lab

Karen Chang
IT513 Information System Security
California Lutheran University



AGENDA

- Overview
- Lab Tasks
 - Task 1: Generate Encryption Key in a Wrong Way
 - Task 2: Guessing the Key
 - Task 3: Measure the Entropy of Kernel
 - Task 4: Get Pseudo Random Numbers from /dev/random
 - Task 5: Get Random Numbers from /dev/urandom
- Example
- Reference



Overview



Importance of Randomness in Security Software: Random numbers are crucial for security software, especially for generating encryption keys internally, to ensure unpredictability and maintain encryption integrity.



Common Practice Among Developers: Many developers, based on their experience with applications like Monte Carlo simulations, apply similar methods to generate random numbers for security purposes.



Inadequacy for Encryption Keys: The methods suitable for simulations may not meet the security standards required for encryption keys, potentially compromising security.

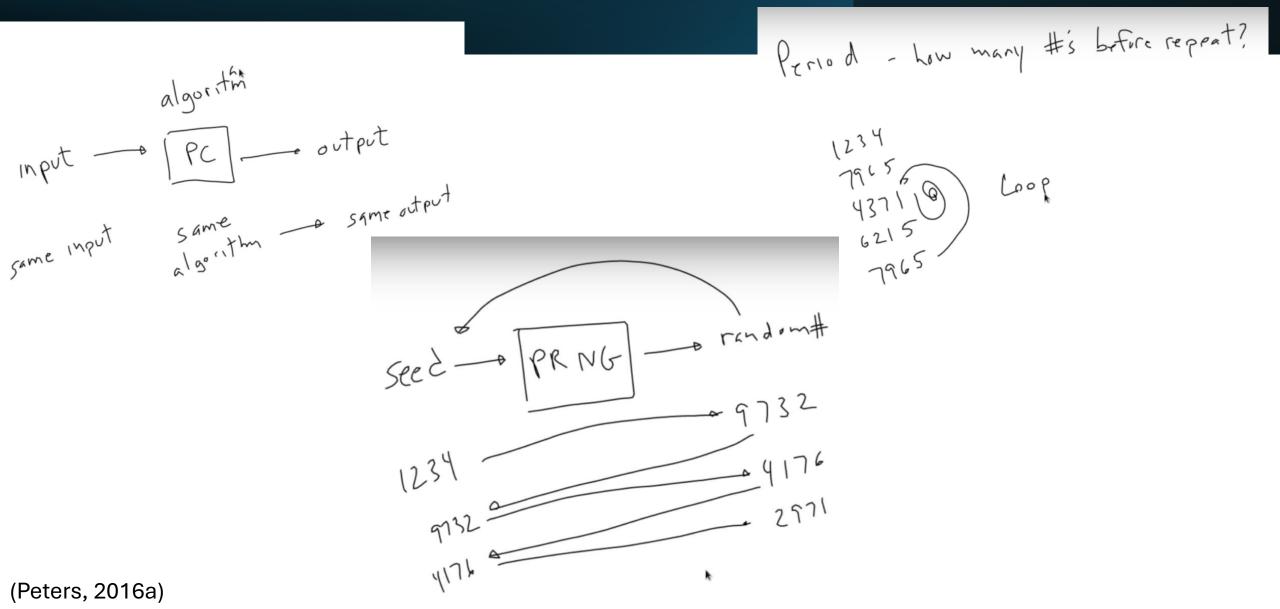


Historical Precedence of Mistakes: Notable mistakes in generating secure random numbers have occurred in well-known products such as Netscape and Kerberos, highlighting the importance of correct implementation.



Educational Focus: Students will learn about the shortcomings of typical random number generation methods for security purposes and will be taught a standard method to generate pseudo-random numbers that are suitable for security applications.

Pseudo Random Number Generators Part I



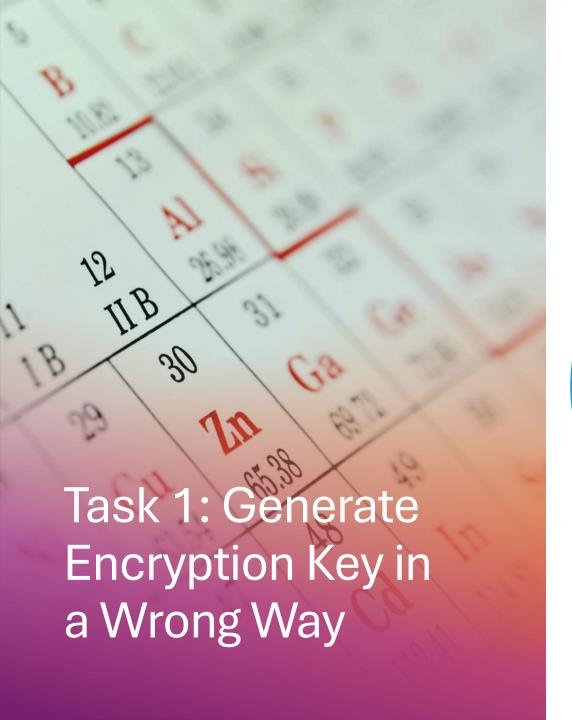
Lab Tasks

Task 1: Generate Encryption Key in a Wrong Way

// Kill event

is0biect

- Task 2: Guessing the Key
- Task 3: Measure the Entropy of Kernel
- Task 4: Get Pseudo Random Numbers from /dev/random
- Task 5: Get Random Numbers from /dev/urandom



To generate good pseudo random numbers, we need to start with something that is random; otherwise, the outcome will be quite predictable.

The library function time() returns the time as the number of seconds since the Epoch, 1970-01-0100:00:00 +0000 (UTC).

Task 1: Generate Encryption Key in a Wrong Way

*task1.c

//srand(time(NULL));

```
Open ▼ 升
1#include <stdio.h>
2#include <stdlib.h>
3#include <time.h>
4#define KEYSIZE 16
5 void main()
6 {
      int i;
      char kev[KEYSIZE]:
      printf("%lld\n", (long long)time(NULL));
10
     //srand(time(NULL));
11
      for (i = 0; i < KEYSIZE; i++)
12
13
          key[i] = rand() % 256;
14
          printf("%.2x", (unsigned char)key[i]);
15
16
      printf("\n");
17 }
```

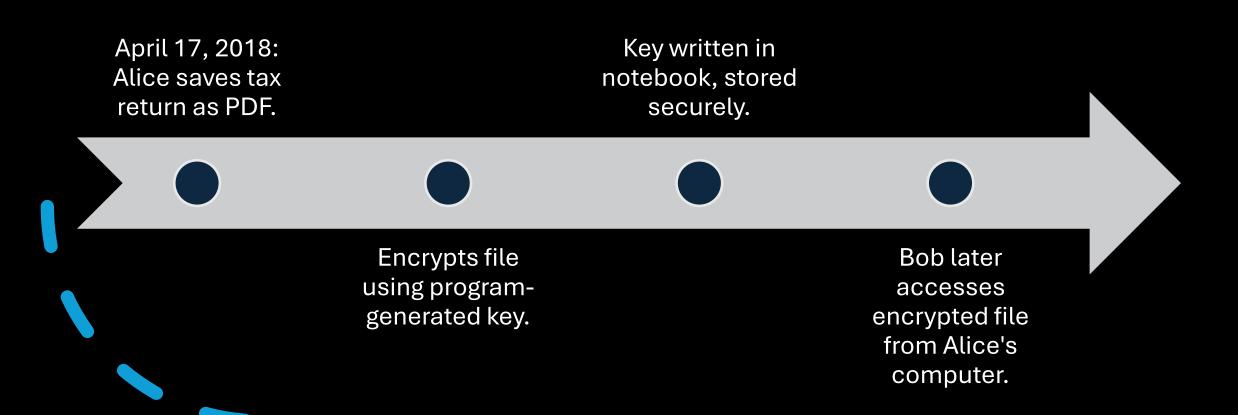
```
[04/19/24]seed@VM:~$ gcc task1.c -o task1
[04/19/24] seed@VM:~$ ./task1
1713541643
67c6697351ff4aec29cdbaabf2fbe346
[04/19/24]seed@VM:~$ ./task1
1713541647
67c6697351ff4aec29cdbaabf2fbe346
[04/19/24]seed@VM:~$ ./task1
1713541651
67c6697351ff4aec29cdbaabf2fbe346
[04/19/24]seed@VM:~$ ./task1
1713541652
67c6697351ff4aec29cdbaabf2fbe346
[04/19/24]seed@VM:~$ ./task1
1713541655
67c6697351ff4aec29cdbaabf2fbe346
[04/19/24]seed@VM:~$ ./task1
1713541657
67c6697351ff4aec29cdbaabf2fbe346
[04/19/24]seed@VM:~$ ./task1
1713541659
67c6697351ff4aec29cdbaabf2fbe346
```

Task 1: Generate Encryption Key in a Wrong Way

srand(time(NULL));

```
task1.c
 Open ▼ 🗐
 1#include <stdio.h>
 2 #include <stdlib.h>
 3#include <time.h>
 4#define KEYSIZE 16
 5 void main()
 6 {
      int i;
      char key[KEYSIZE];
      printf("%lld\n", (long long)time(NULL));
10
      srand(time(NULL));
11
      for (i = 0; i < KEYSIZE; i++)
12
13
           kev[i] = rand() % 256;
14
           printf("%.2x", (unsigned char)key[i]);
15
16
      printf("\n");
17 }
```

```
[04/19/24]seed@VM:~$ qcc task1.c -o task1
[04/19/24] seed@VM:~$ ./task1
1713541777
8371f872494b6ff27a27ba6cef7d09a2
[04/19/24] seed@VM:~$ ./task1
1713541778
113664e5675ef9b47c73a177576abd62
[04/19/24] seed@VM:~$ ./task1
1713541779
dbea00dd048ffa464a31c302588f0017
[04/19/24]seed@VM:~$ ./task1
1713541780
03903f5da23a85582d9a60c0f537c1f4
[04/19/24]seed@VM:~$ ./task1
1713541781
4baa482111403e24628174e7d9efdc1f
[04/19/24]seed@VM:~$ ./task1
1713541788
l170ada64731aa63a0c64c5690e6c4d59
[04/19/24] seed@VM:~$ ./task1
1713541789
903bcd4b7cf4bd6cf7b1790d26d893da
```



```
[04/19/24]seed@VM:~$ date -d "2018-04-17 21:08:49" +%s 1524013729 [04/19/24]seed@VM:~$ date -d "2018-04-17 23:08:49" +%s 1524020929
```

Bob suspects
Alice's key is
generated by a
program.



Timestamp of file creation: "2018-04-17 23:08:49".



Bob guesses key generated within twohour window before creation.



```
get_key.py
 Open ▼ 🗐
         get key.py */
 2 from Crypto import Random
 3 from Crypto.Cipher import AES
 5 ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
 6 IV = "09080706050403020100A2B2C2D2E2F2".lower().decode("hex")
 7 plaintext1 = "255044462d312e350a25d0d4c5d80a34".decode("hex")
 9 with open('key.txt') as f:
      keys = f.readlines()
11
12 for k in keys:
13
      key = (k[:-1]).decode("hex")
14
      cipher = AES.new(key, AES.MODE CBC, IV)
15
      encrypted = cipher.encrypt(plaintext1)
16
      if ciphertext == encrypted.encode("hex")[0:32]:
          print("Match found")
18
          print("key: "+k[:-1])
19
          print("Ciphertext: " + ciphertext)
20
           print("Encrypted: " + encrypted.encode("hex"))
```



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 d9b8cd83acfbbb2478322c34b093f15d 1316263fda83af21d8ff1981f430fb81 f56aa430b62a3600d01bcaa21b49f360 73f74b0501e459ae903335e3ec05499b 01cb89e4c12ec7347715eb8ef9db59c1 f2d82bf6c8fcadabfaca886ef95c9b44

PDF files start with a version header.

Common version is PDF-1.5, first 8 bytes.

Next 8 bytes are predictable.

Bob has 16 bytes plaintext due to header.

Encrypted with aes-128-cbc, so one plaintext block is known.

- Moreover, Bob also knows the Initial Vector (IV) from the encrypted file (IV is never encrypted). Here is what Bob knows:
- Plaintext: 255044462d312e350a25d0d4c5d80a34
- Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
- IV: 09080706050403020100A2B2C2D2E2F2

```
[04/19/24]seed@VM:~$ python3 GetKey.py
Match found
key: 95fa2030e73ed3f8da761b4eb805dfd7
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
Encrypted: d06bf9d0dab8e8ef880660d2af65aa82
```

```
1# /* GetKey.py {python3} */
2 from Crypto.Cipher import AES
3 import binascii
5 ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
6 IV = binascii.unhexlify("09080706050403020100A2B2C2D2E2F2".lower())
7 plaintext1 = binascii.unhexlify("255044462d312e350a25d0d4c5d80a34")
9 with open('key.txt', 'r') as f:
      kevs = f.readlines()
12 for k in keys:
      key = binascii.unhexlify(k.strip())
      cipher = AES.new(key, AES.MODE CBC, IV)
      encrypted = cipher.encrypt(plaintext1)
      if ciphertext == encrypted.hex()[0:32]:
          print("Match found")
          print("key: "+k.strip())
          print("Ciphertext: " + ciphertext)
          print("Encrypted: " + encrypted.hex())
```

```
get_key.py
 Open ▼ 🕕
         get key.py */
2 from Crypto import Random
3 from Crypto.Cipher import AES
5 ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
6 IV = "09080706050403020100A2B2C2D2E2F2".lower().decode("hex")
7 plaintext1 = "255044462d312e350a25d0d4c5d80a34".decode("hex")
8
9 with open('key.txt') as f:
      keys = f.readlines()
12 for k in keys:
      key = (k[:-1]).decode("hex")
      cipher = AES.new(key, AES.MODE CBC, IV)
      encrypted = cipher.encrypt(plaintext1)
      if ciphertext == encrypted.encode("hex")[0:32]:
          print("Match found")
          print("key: "+k[:-1])
          print("Ciphertext: " + ciphertext)
          print("Encrypted: " + encrypted.encode("hex"))
```

Task 3: Measure the Entropy of Kernel



GENERATING RANDOMNESS IN SOFTWARE IS TOUGH.



SYSTEMS RELY ON PHYSICAL SOURCES FOR RANDOMNESS.



LINUX GATHERS RANDOMNESS FROM VARIOUS PHYSICAL RESOURCES.

```
void add_keyboard_randomness(unsigned char scancode);
void add_mouse_randomness(__u32 mouse_data);
void add_interrupt_randomness(int irq);
void add_blkdev_randomness(int major);
```

Task 3: Measure the Entropy of Kernel

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

cat /proc/sys/kernel/random/entropy_avail

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

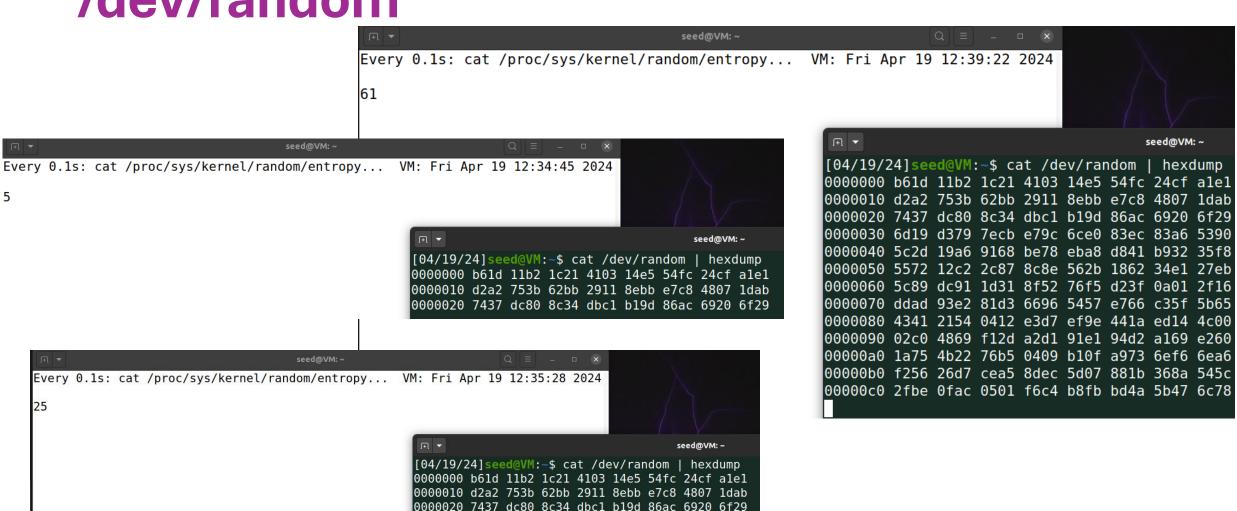
Linux collects random data into pool from physical resources.

Two devices: /dev/random (blocking) and /dev/urandom.

/dev/random decreases entropy with each use.

Blocks until enough randomness is gained.

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



0000030 6d19 d379 7ecb e79c 6ce0 83ec 83a6 5390 0000040 5c2d 19a6 9168 be78 eba8 d841 b932 35f8 0000050 5572 12c2 2c87 8c8e 562b 1862 34e1 27eb



cat /dev/urandom | hexdump watch -n .1 cat /proc/sys/kernel/random/entropy_avail

Task 5: Get Random Numbers from /dev/urandom



Linux offers /dev/urandom for non-blocking access to random pool.



/dev/random and /dev/urandom use pool data for pseudo-random numbers.



/dev/random pauses if entropy is low, /dev/urandom keeps generating.



Pool data acts as "seed" for generating pseudo-random numbers.

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

1302

```
Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Fri Apr 19 12:43:42 2024
               seed@VM: ~
999
               2baee90 449a a3e9 fa3f 5171 b3d6 7d73 8784 9035
               2baeea0 f5bb ae08 69f3 decd 93fb 6e33 166a a8b3
               2baeeb0 c906 9f08 137b 7916 e10a 3550 0ebb 655b
              <u>2baeec0 431d 3a11 ae29 1ff0 fc33 06de 5a1d e249</u>
              2baeed0 4f0d 0bc2 7c90 0bc3 531b aa00 0f66 2466
              2baeee0 6ee7 61a4 375c bf3c 1462 3f9e 20b3 f7fb
               2baeef0 33f5 9031 ceac a142 f0cf 2e30 7a0a 89e5
              2baef00 db8e c2c0 68bf 1c37 6439 b305 fca6 c0f2
               2baef10 e188 8992 5dfe 1607 899e 6fa6 04e9 de52
              2baef20 974b d82d 0b1a 7bdc e3f8 1070 7dd2 be04
               2baef30 2867 6929 1205 b470 8296 0f0c b6f0 0e99
              2baef40 ea0c 1566 5dca e4a6 bf35 1a6f 73cf aef4
               2baef50 a143 71b4 bd25 680c 84e1 42e0 3610 939b
              2baef60 15f8 1b5c 4236 7f06 676c 9db1 49dc b80d
               2baef70 d25b 6342 832e 356e 8f0d 8a6a 21bb b9c6
              2baef80 7626 507c 4f90 a017 b269 f0f3 3014 8b2e
               2baef90 425d 6963 e979 d74c d5a8 e720 ef43 5686
              2baefa0 4866 4792 2a70 89e3 2c4d fad9 2eea 01bb
               2baefb0 c3c2 b4ba 630d c170 e979 899c 8afe bd48
              2baefc0 2039 f327 eeff 383d 8649 022d 8dbe 4fb4
               2baefd0 00e9 4bdf 278b cddc fff1 26f5 0ad6 f5d0
              2baefe0 3caf 0969 0dcf 2d8e 73ff 020d 30c9 1698
              2baeff0 61ad 195a c6c4 9e1f b27d 7be7 5c92 e7be
```

seed@VM:~

Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Fri Apr 19 12:44:05 2024

```
seed@VM: ~
501b460 1e06 616c 0532 37b9 eddd ed03 eb80 557c
|501b470 ea9b 6fbd 2280 e0a4 f939 d100 d38b fa36
501b480 48d1 6069 997f a4eb c8f4 1aa4 7fea b594
501b490 6086 5960 7437 cf09 bbb1 6f7e ca95 3a94
501b4a0 e527 6063 a97b 047d 1a50 9ab1 c2e9 d439
501b4b0 91e2 c5df fc13 6971 d1e1 930e eca2 9be2
501b4c0 5969 d63c 7e1a 58fb cf99 107f 2a73 1b4e
501b4d0 106b 2657 9f97 dae7 2bd1 8832 ab9c e826
501b4e0 0347 7b2a 20c7 81ce 6771 b1<mark>1</mark>3 62a9 7b80
501b4f0 4da9 f60c 63b9 bf2e 92eb 6220 b111 c9eb
501b500 8b9c 3be7 88df fb1b c226 9ff8 8851 a333
501b510 be16 7873 ced7 9f31 9914 4b10 be60 6626
501b520 d588 1175 6b44 1ad1 9418 d0c9 e2e6 2497
|501b530 e73c 91eb ae9c 9c2b 25b0 5d27 e6b1 e419
501b540 3286 d25b a9b7 9ed4 0ae5 760b bd62 4245
501b550 7f6f 47b8 053e cfc1 a06f e032 f6ee 6e86
501b560 655b 1bf1 2ed4 3d06 11e9 c82a 83d3 00d6
501b570 9357 6122 6603 a3ff ae80 237d 5c0a d4c0
501b580 9457 5f6c 0a35 ae26 9f5d 158c ee2c dfe8
501b590 1535 27ac cb0d 0561 3a3c 3514 2f34 3f29
501b5a0 c788 4dfd b78f 10db b6a4 2422 ba13 7dd4
501b5b0 626d ee99 6176 4461 25da 4168 3aba c44d
501b5c0 6189 6ff0 07f6 4763 6cb0 9bf2 3288 a399
```

Measure the quality of the random number

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

```
[04/19/24]seed@VM:~$ head -c 1M /dev/urandom > output.bin
[04/19/24]seed@VM:~$ ent output.bin
Entropy = 7.999830 bits per byte.

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

Chi square distribution for 1048576 samples is 247.34, and randomly
would exceed this value 62.29 percent of the times.

Arithmetic mean value of data bytes is 127.4146 (127.5 = random).
Monte Carlo value for Pi is 3.144779758 (error 0.10 percent).
Serial correlation coefficient is -0.001256 (totally uncorrelated = 0.0).
```

```
task5.c
       task5.c */
 2 #include <stdio.h>
 3#include <stdlib.h>
 4 #define LEN 32 // 256 bits
 6 void main()
       int i;
10
11
       FILE *random = fopen("/dev/urandom", "r");
12
13
       for (i = 0; i < LEN; i++)
14
15
16
17
18
           fread(key, sizeof(unsigned char) * LEN, 1, random);
           printf("%.2x", *key);
       printf("\n");
       fclose(random);
```

```
[04/19/24]seed@VM:~$ gcc task5.c -o task5
                                                            [04/19/24]seed@VM:~$ ./task5
                                                            fc6fa74d095e7313ab60783e57fa2014ffce40ba9990961bc967064762002291
                                                            [04/19/24]seed@VM:~$ ./task5
                                                            f9c9132c81968b72a4ca988fcebe6804b971af4180cd80f95e2af95120a6f997
                                                            [04/19/24]seed@VM:~$ ./task5
unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char) * LEN); 7fac318c3087e4d6c0001ddfd8c9b554e38e2a49024bac2de5803c77a113e291
                                                            [04/19/24]seed@VM:~$ ./task5
                                                            21ecb5dd7fdf20e53478f51ad83dec023377158919439774501d976341f9f848
                                                            [04/19/24] seed@VM:~$ ./task5
                                                            efba86768cf712ac602247a09de92c86a7f3785117133e4255f184b2d2f441fc
                                                             [04/19/24]seed@VM:~$
```

The "Real" Random Number ©

gcc task5.c -o task5. /task5

Reference

- Guo, J. (2022, October 14). Seed-lab: pseudo random number generation lab. Jiaming Guo's Blog. https://gls.show/p/c8db7269/
- Peters, K. (2016a). Coding math: Episode 51 pseudo random number generators part I. YouTube.
 - https://www.youtube.com/watch?v=4sYawx70iP4
- Peters, K. (2016b). Coding math: Episode 52 pseudo random number generators, part II. YouTube.
 - https://www.youtube.com/watch?v=RbwNVzhNwJ8
- SKPrimin. (2022). *Pseudo Random Number Generation Lab*. cnblog.
 - https://www.cnblogs.com/skprimin/p/16062870.html

