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.

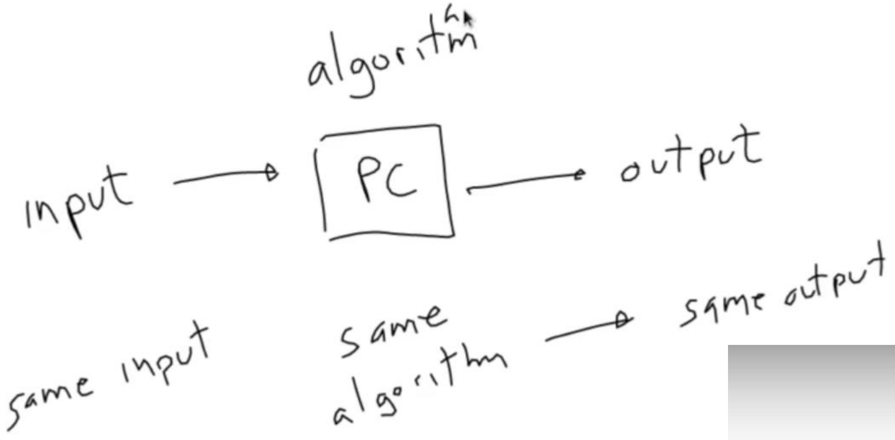
Common Practice Among Developers: Many developers, based on their experience with applications like Monte Carlo simulations, apply similar methods to generate random numbers for security purposes.

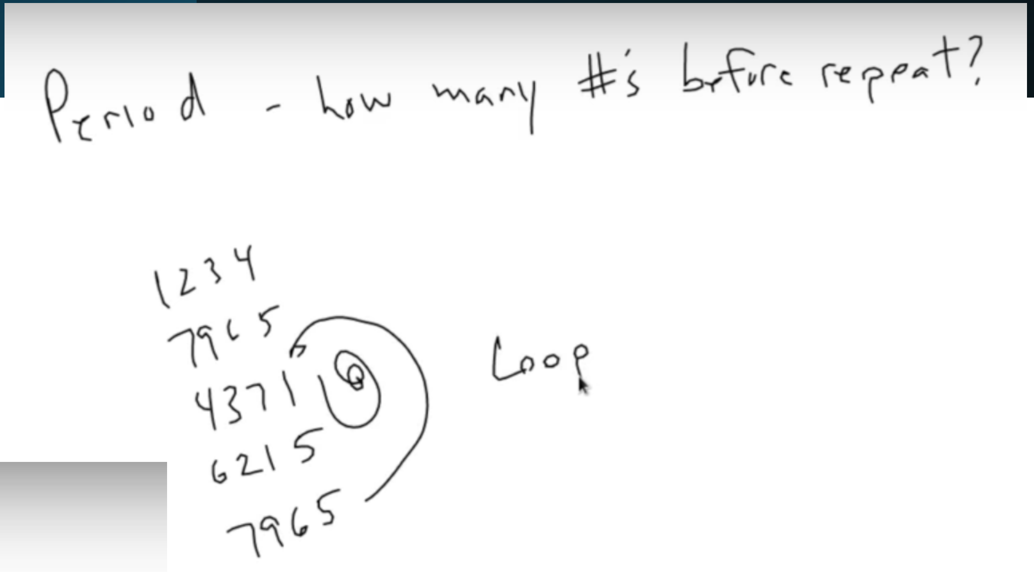
Inadequacy for Encryption Keys: The methods suitable for simulations may not meet the security standards required for encryption keys, potentially compromising security.

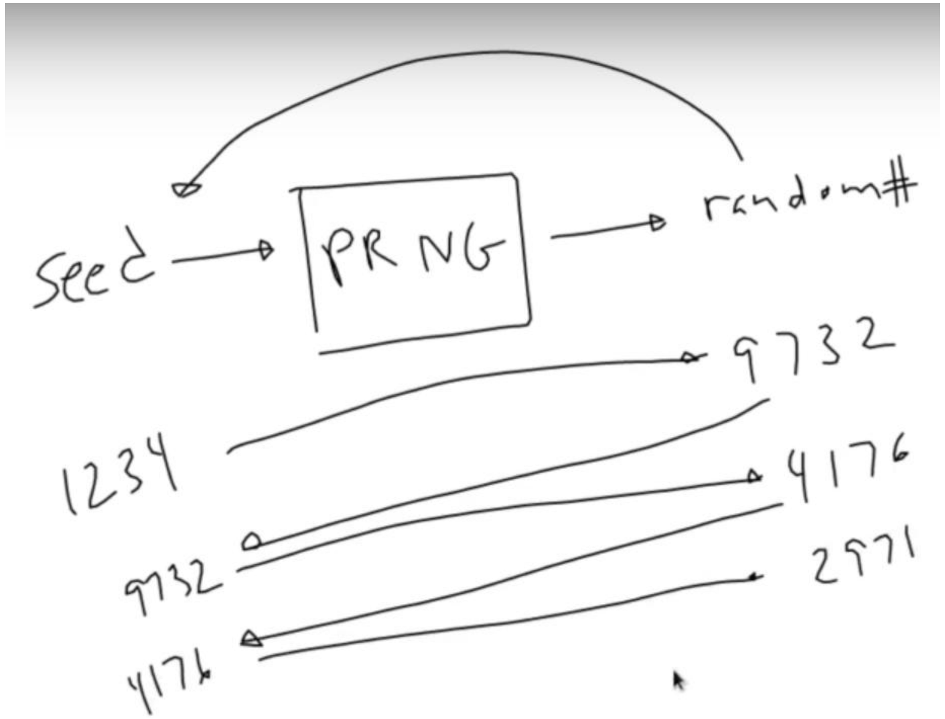
Historical Precedence of Mistakes: Notable mistakes in generating secure random numbers have occurred in well-known products such as Netscape and Kerberos, highlighting the importance of correct implementation.

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



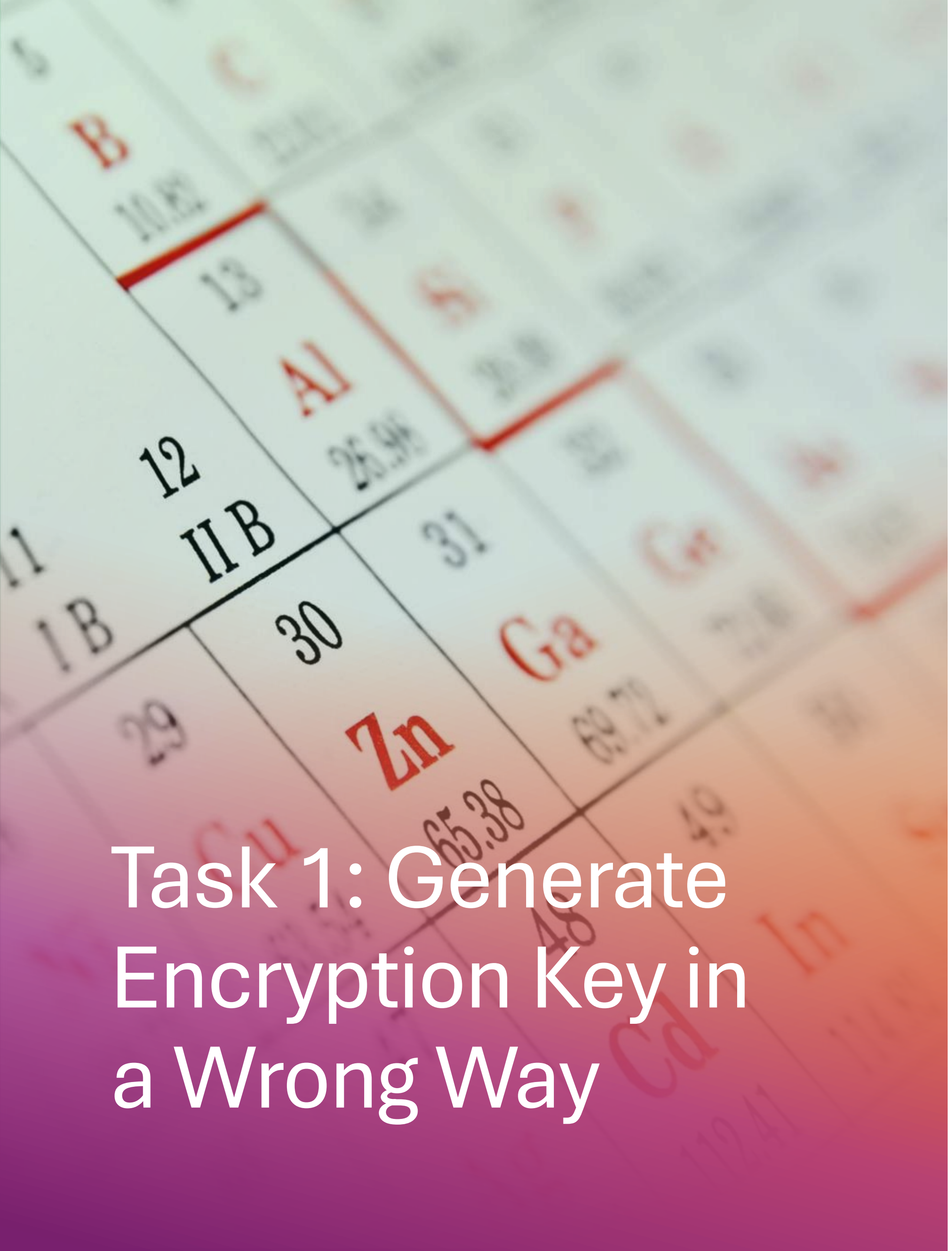




(Peters, 2016a)

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



Task 1: Generate Encryption Key in a Wrong Way

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

- //srand(time(NULL));

Open *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");
}
```

```
[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

```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");
}
```

```
[04/19/24]seed@VM:~$ gcc 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
170ada64731aa63a0c64c5690e6c4d59
[04/19/24]seed@VM:~$ ./task1
1713541789
903bcd4b7cf4bd6cf7b1790d26d893da
```

Task 2: Guessing the Key

April 17, 2018: Alice saves tax return as PDF.

Encrypts file using program-generated key.

Key written in notebook, stored securely.

Bob later accesses encrypted file from Alice's computer.

```
[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
```

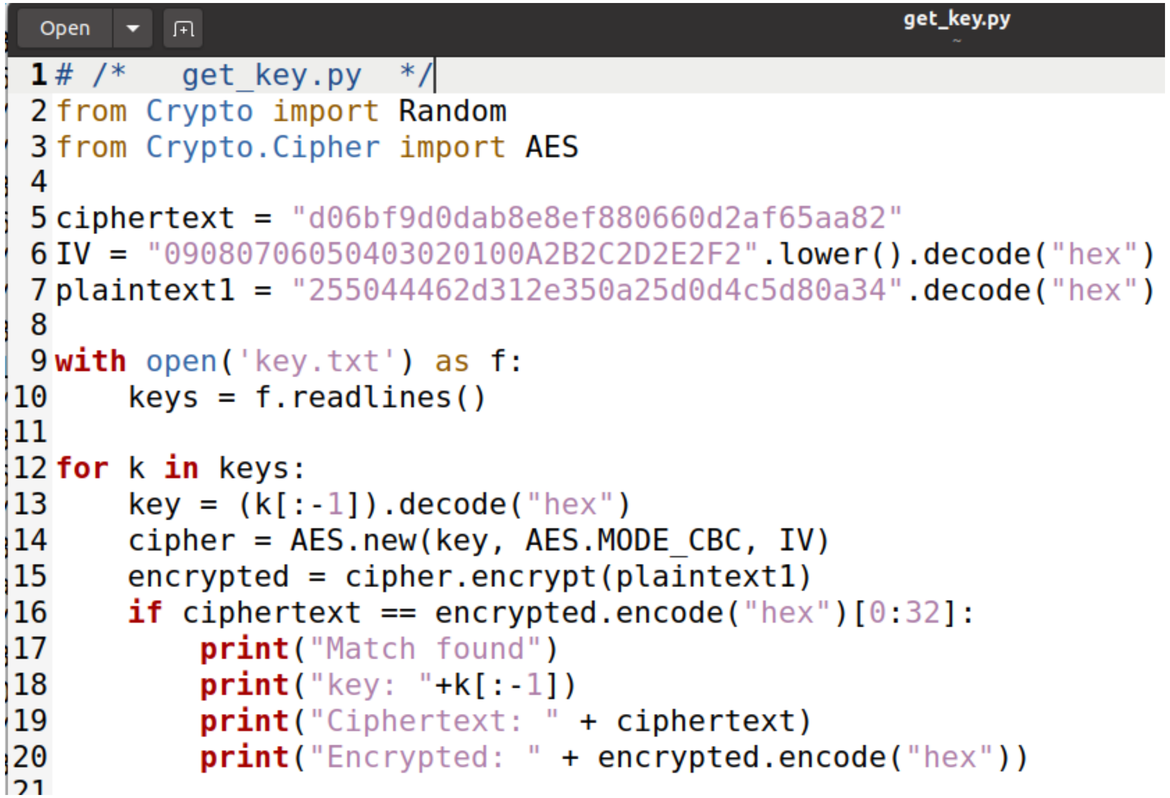
Task 2: Guessing the Key

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 two-hour window before creation.

Task 2: Guessing the Key

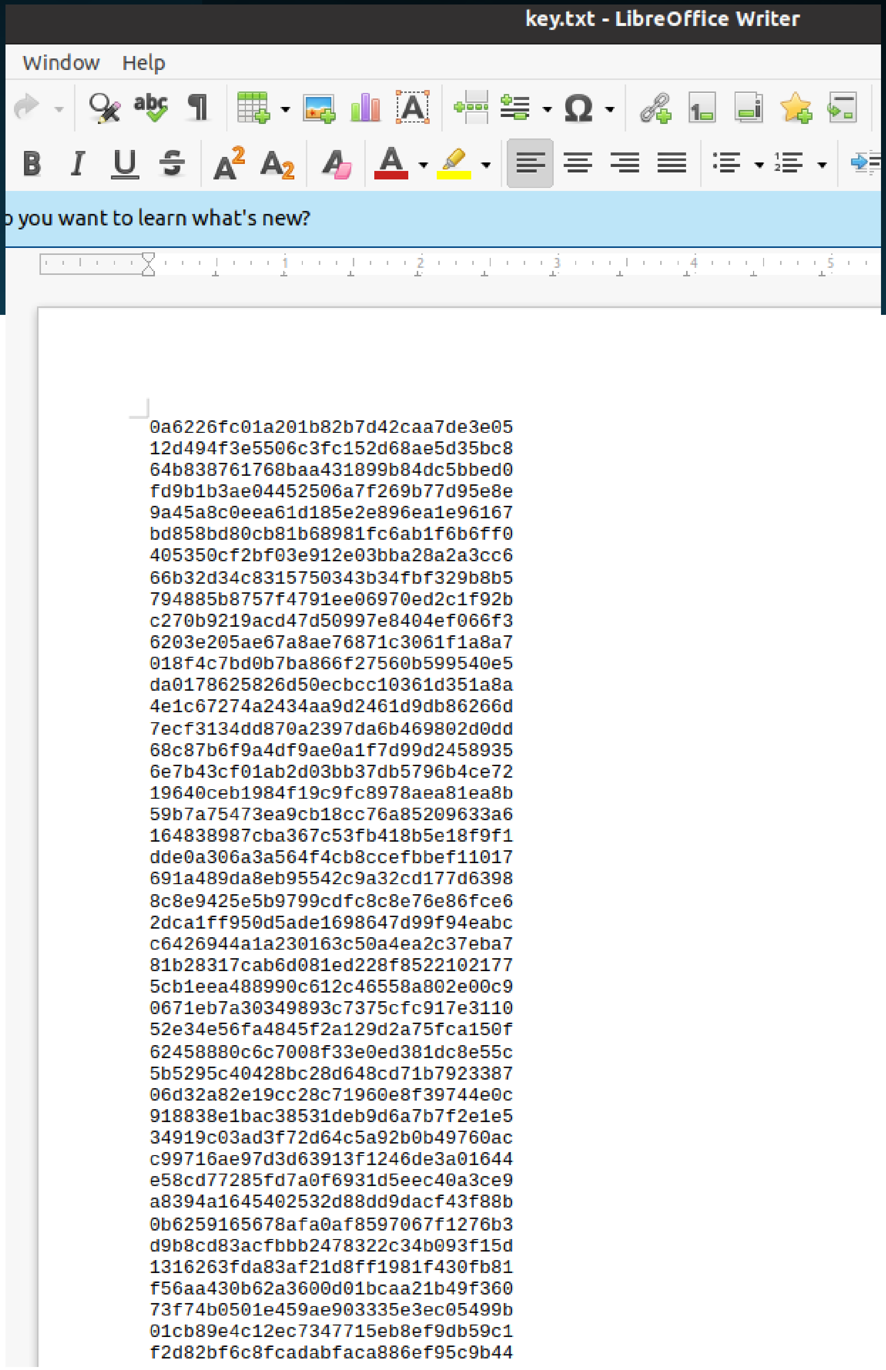


```python
# /*    get_key.py   */
from Crypto import Random
from Crypto.Cipher import AES

ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
IV = "09080706050403020100A2B2C2D2E2F2".lower().decode("hex")
plaintext1 = "255044462d312e350a25d0d4c5d80a34".decode("hex")

with open('key.txt') as f:
    keys = f.readlines()

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"))
```



key.txt - LibreOffice Writer

```
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
```

Task 2: Guessing the Key

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.

Task 2: Guessing the Key

- 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
```

Open GetKey.py

```python
# /*   GetKey.py {python3} */
from Crypto.Cipher import AES
import binascii

ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
IV = binascii.unhexlify("09080706050403020100A2B2C2D2E2F2".lower())
plaintext1 = binascii.unhexlify("255044462d312e350a25d0d4c5d80a34")

with open('key.txt', 'r') as f:
    keys = f.readlines()

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())
```

Open get_key.py

```python
# /*   get_key.py   */
from Crypto import Random
from Crypto.Cipher import AES

ciphertext = "d06bf9d0dab8e8ef880660d2af65aa82"
IV = "09080706050403020100A2B2C2D2E2F2".lower().decode("hex")
plaintext1 = "255044462d312e350a25d0d4c5d80a34".decode("hex")

with open('key.txt') as f:
    keys = f.readlines()

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

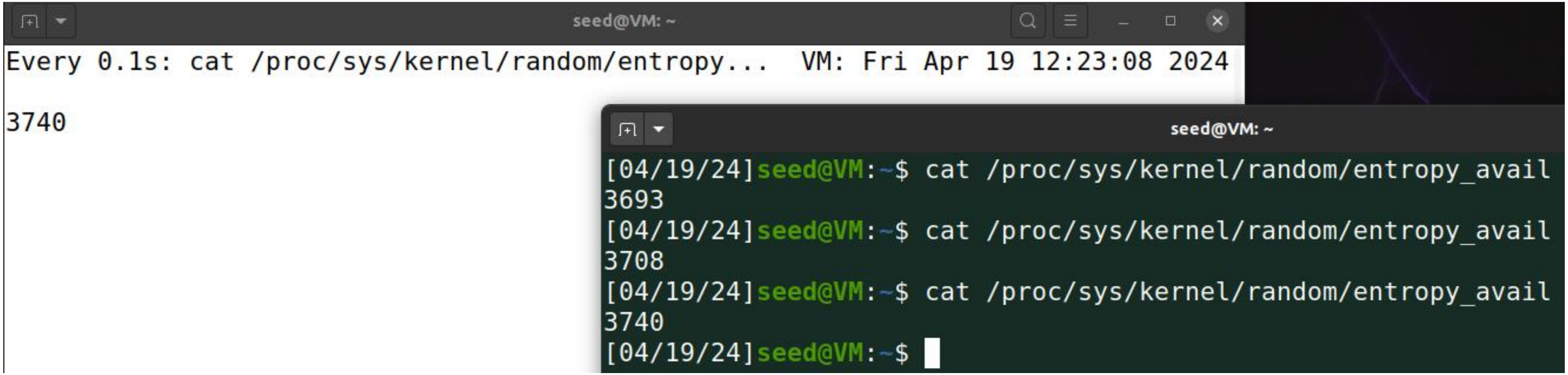
GENERATING RANDOMNESS IN SOFTWARE IS TOUGH.

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

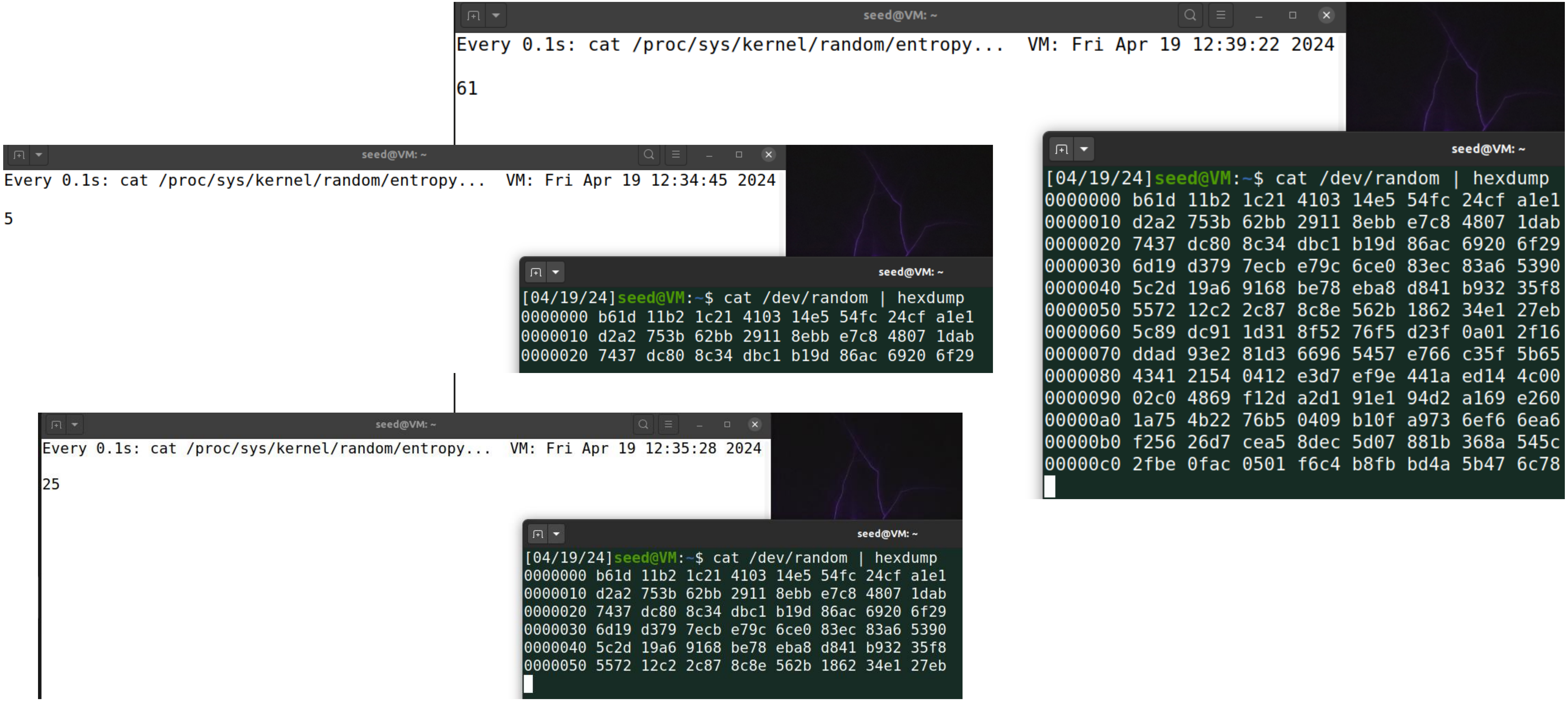
Task 4: Get Pseudo Random Numbers from **/dev/random**

- 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.

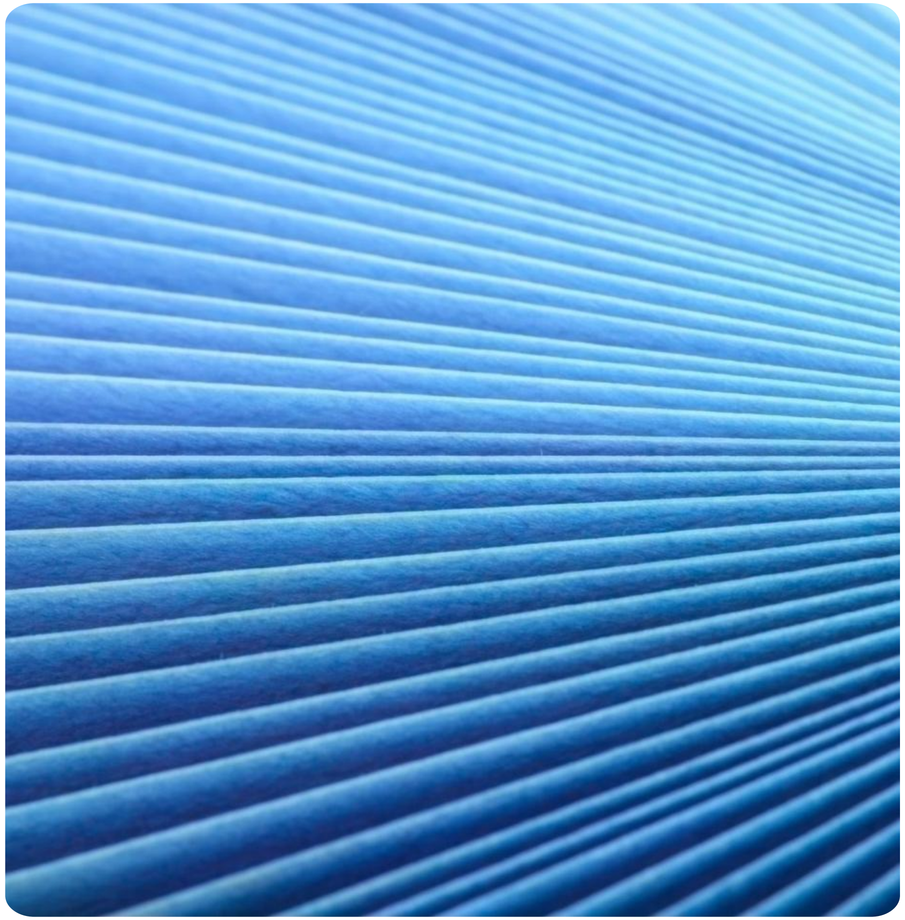
cat /dev/random | hexdump

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

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



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.

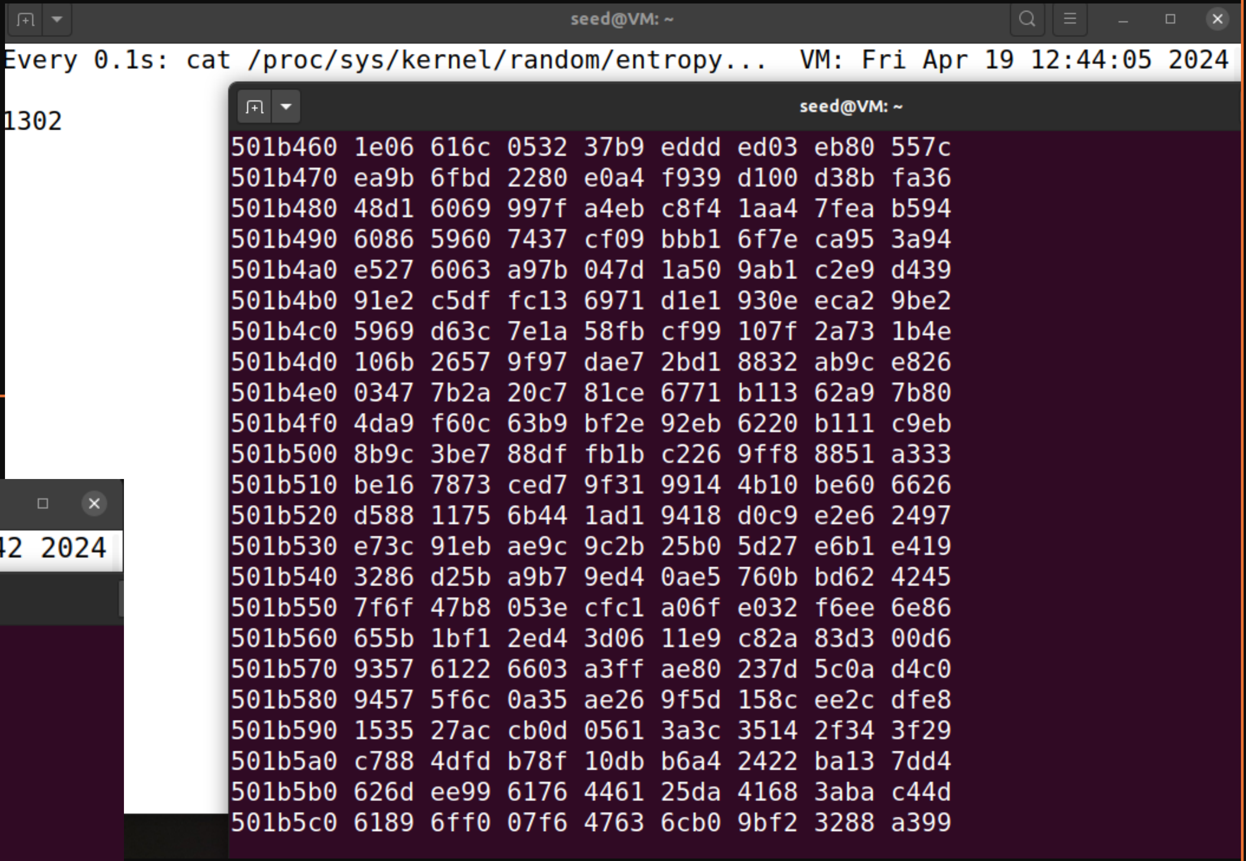
/dev/random pauses if entropy is low, /dev/urandom keeps generating.

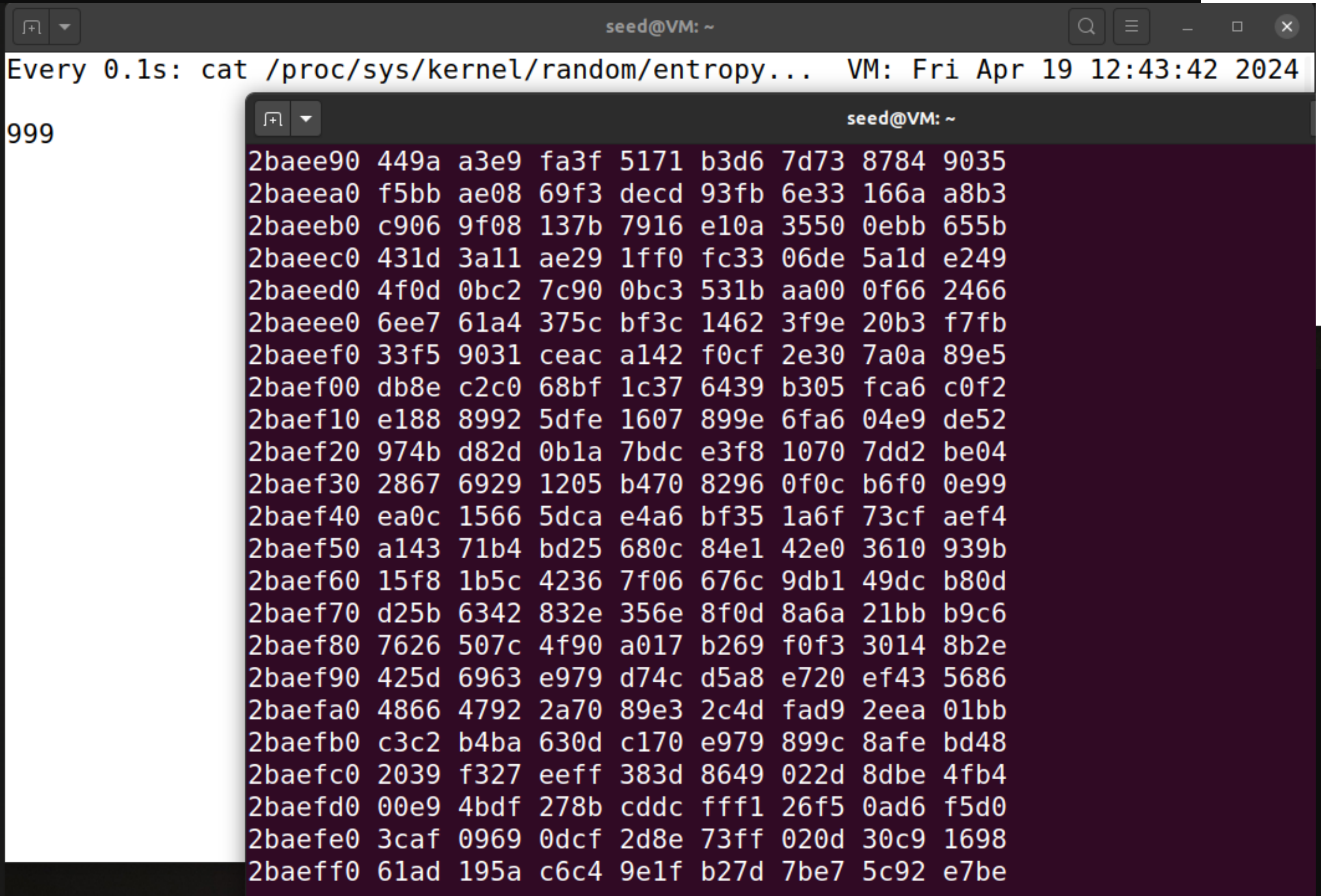
Pool data acts as "seed" for generating pseudo-random numbers.

cat /dev/urandom | hexdump

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

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





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).
```

Open task5.c

```c
/*   task5.c   */
#include <stdio.h>
#include <stdlib.h>
#define LEN 32 //  256  bits

void main()
{

    int i;
    unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);
    FILE *random = fopen("/dev/urandom", "r");
    for (i = 0; i < LEN; i++)
    {
        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
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

Q&A