Cryptography - Lab 4 - Pseudorandom numbers

Javier Antxon Garrues Apecechea

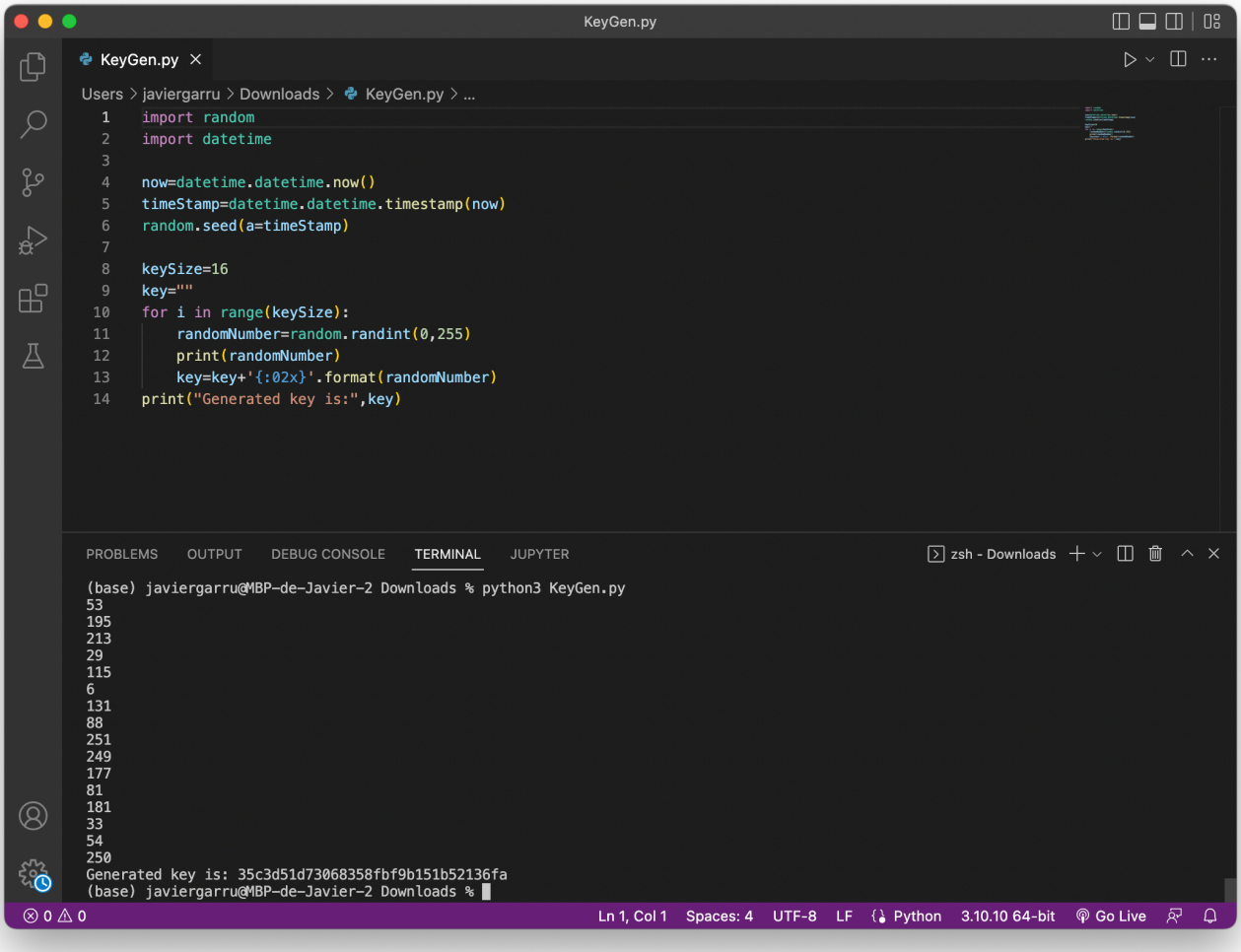
Student ID: 24647808

In this lab we will get familiar with pseudorandom and random number generation.

Part One: generate Random Numbers in a Wrong Way

Executing the unmodified file:

```
pyhton3 KeyGen.py
```



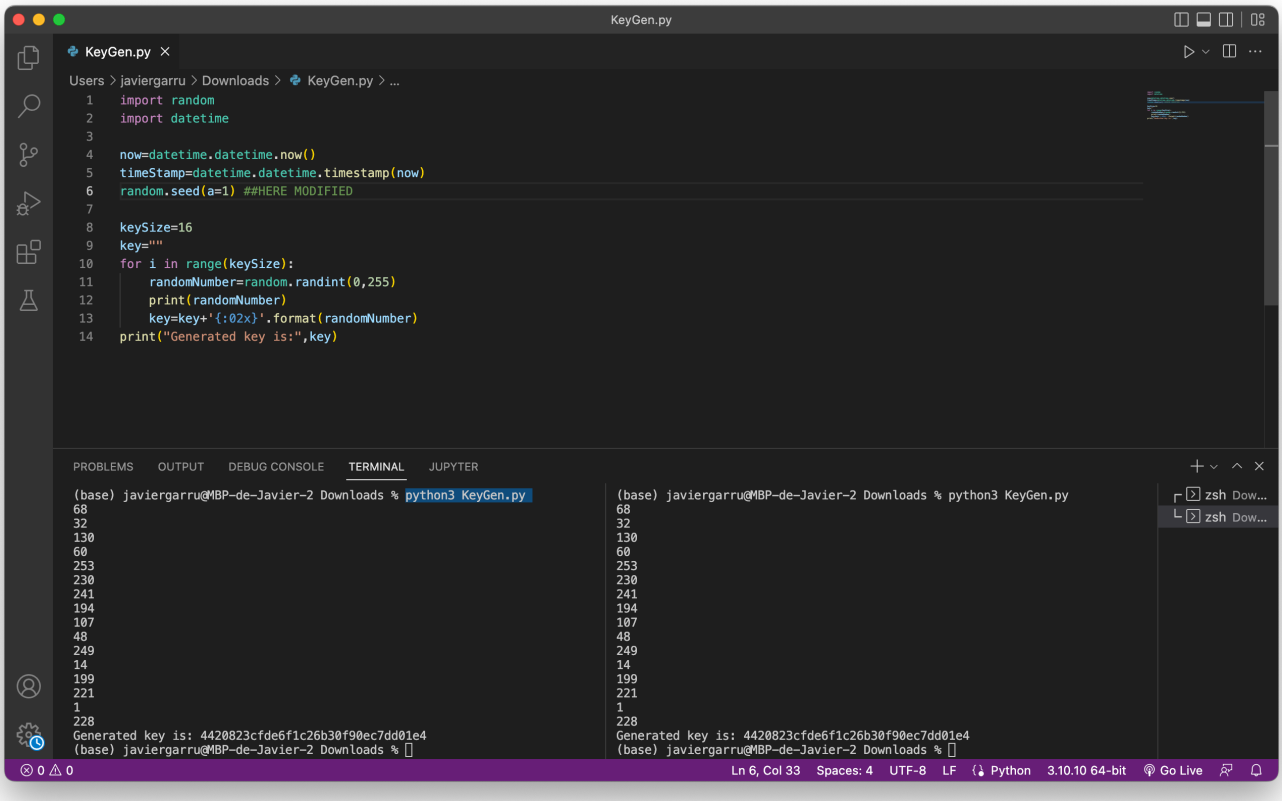
Describe your observations:

After obtaining the actual time and using it as the seed of the *random* number generator, the keySize is defined. Then, a number of random numbers equal to the keySize are generated and concatenated changing their format to a 2 digit hexadecimal value. If the value is lower than 10, the missing value is filled with a zero. For instance, the generated number 6 is transformed to 06 in the key. Executing several times does give the impression of randomness as the results vary significantly. Here I attach some examples of the outputs:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER                    zsh - Downloads

(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: d4a3127239bc5f70f1be7e9c05e5572e
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: f91f36479e06d383fd80c3d6e3879ac6
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: 1d35ce6bd4df0d9efa5aaae1d98f1f1c
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: 2c013ab33f14d983d76ddfbce713c124
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: 76e190dca70ce2dfd36f0913dce1c073
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: 864c2f61498f32f1ddaf8a489493d1d4
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: e717a9b1bbb5ded4cd2a135812514fda
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: 7c01d690126be7dd6cd8f5367f1368a4
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
Generated key is: 0ae09fe61eff8a507b2c18d8550cfe23
(base) javiergarru@MBP-de-Javier-2 Downloads %
```

Now, executing the modified file:



Describe your observations:

Now, the **seed** has been modified and left **constant** (a=1). This poses a big problem as subsequent independent executions of the same program generated the same sequence of numbers. This can be clearly seen in the above image where the generated keys are the same due to the generated values being also equal.

What is the purpose of the random.seed() function:

The random.seed() function is used to **initialize** the **vector** that will be used in the Mersenne Twister, which is the underlying algorithm of the random function. This algorithm uses bit shifts and XORs to modify the state of the vector and using a bit mixing function forms a value from the vector's bits. Then the process is repeated by shifting and XORing the vector. Using the same seed would clearly generate the same extracted values as the sequential states of the vector will be identical in the correpondent steps of different executions. When using the timestamp, we ensure that the seed will be different from the previous ones and the generated numbers too. The period of the Mersenne Twister is 2^19937-1 which indicates that it would take a **extremely** long time to generate the same sequence when using the timestamp as seed.

Part Two: guessing the key

Bob has access to an encrypted file whose key has been generated with the file KeyGen.py. He also has the following information:

Aprox. time of encryption: 2020-09-01 10:40:49 (two hour window before this time) -> timestamp in [2020-09-01 10:40:49, 2020-09-01 8:40:49]

Algorithm: aes-128-cb

First 16 bytes (1 block) of plaintext: 255044462d312e350a25d0d4c5d80a34

IV: 090807060504030201 0A2B2C2D2E2F2

Ciphertext: 2f15e526b607a73b228648cd206f9c95

Describe your idea to find out the key:

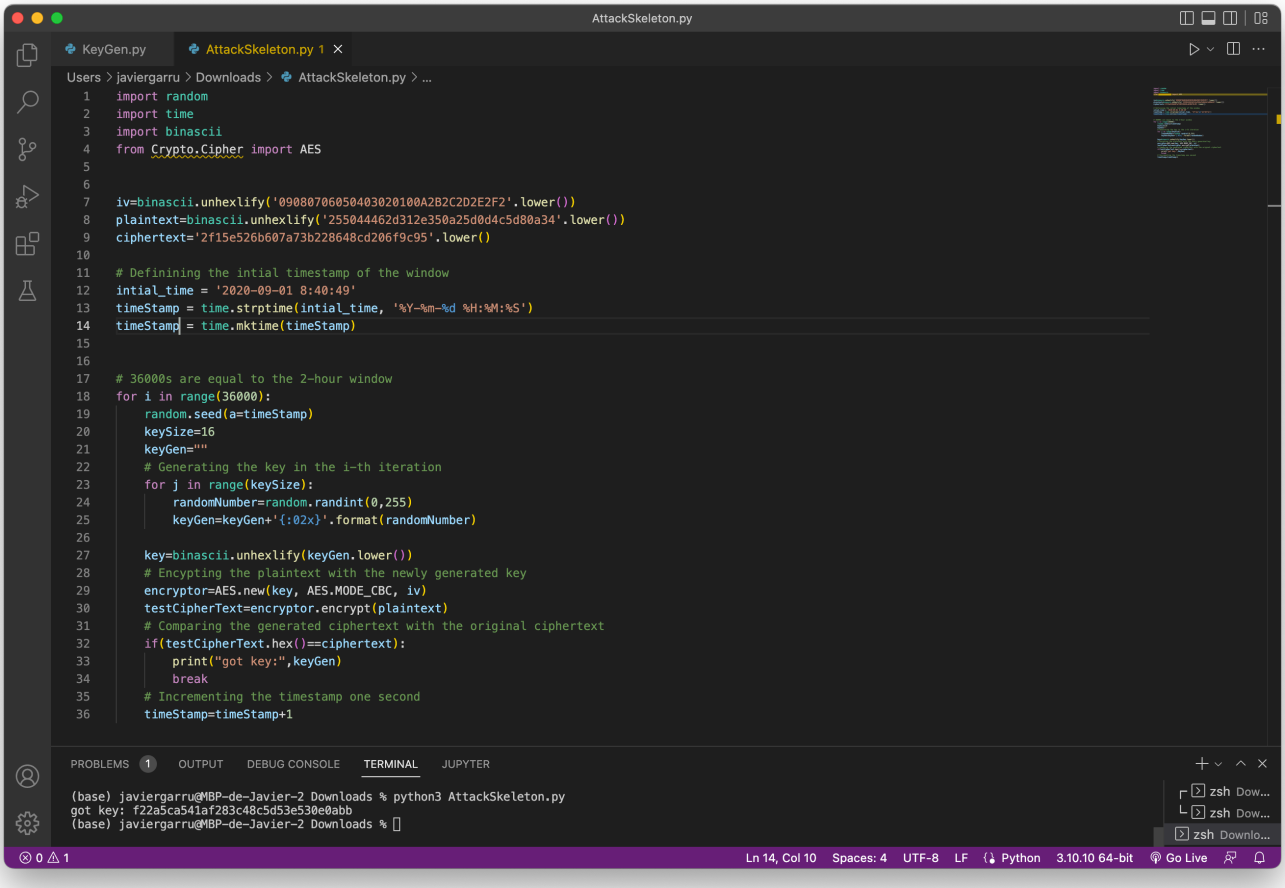
My idea would be iterating on the range of possible timestamps (with a for loop) and generating a different key at each iteration. In the same iteration I would use the aes-128-cb with the obtained key and predefined IV and ciphertext to decrypt the message. At last, I would compare the obtained plaintext to the given plaintext and stop when a match occurs. I will be using the fact that the same seed generates the same key. Which means that if we find out when the key was generated we will have enough information (added to the one already in our possesion) to decrypt the ciphertext. We are also using the fact that same key and iv produce the same ciphertext when encrypting the same plaintext.

Have a try based on the code snippets and find out the key:

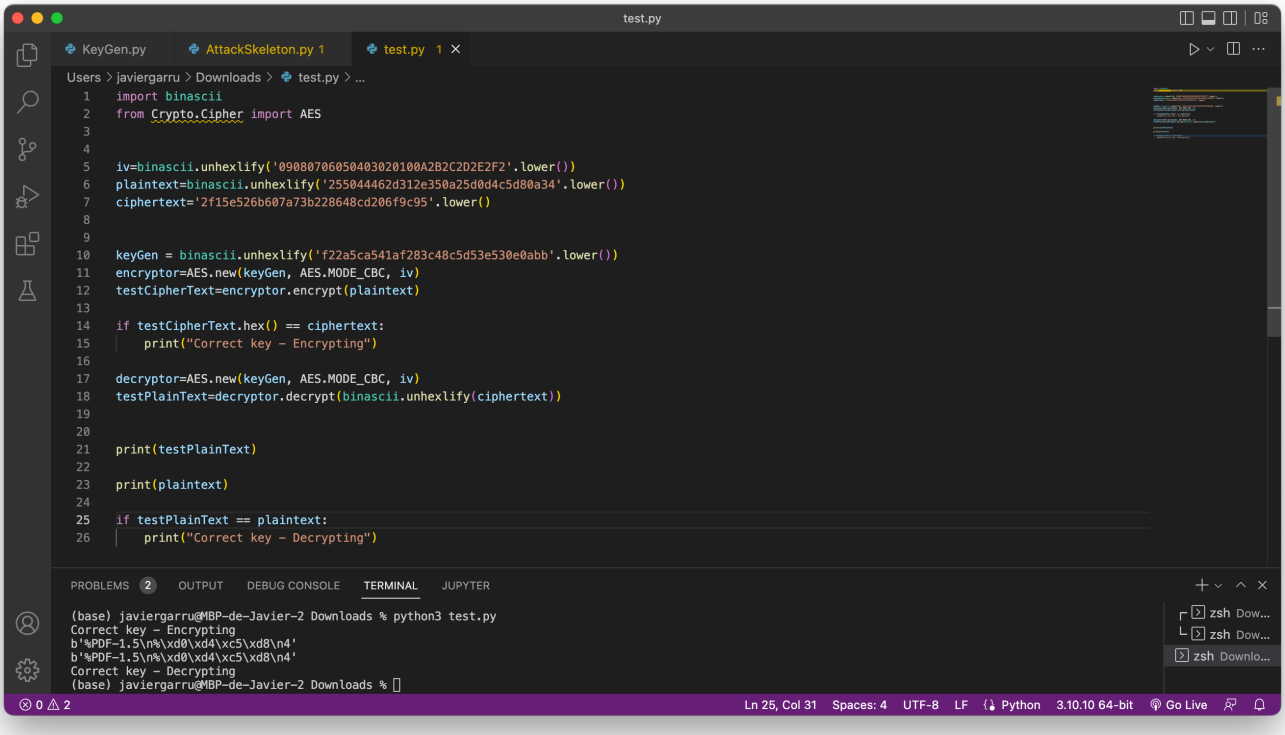
First we download the module:

```
sudo pip3 install pycrypto
```

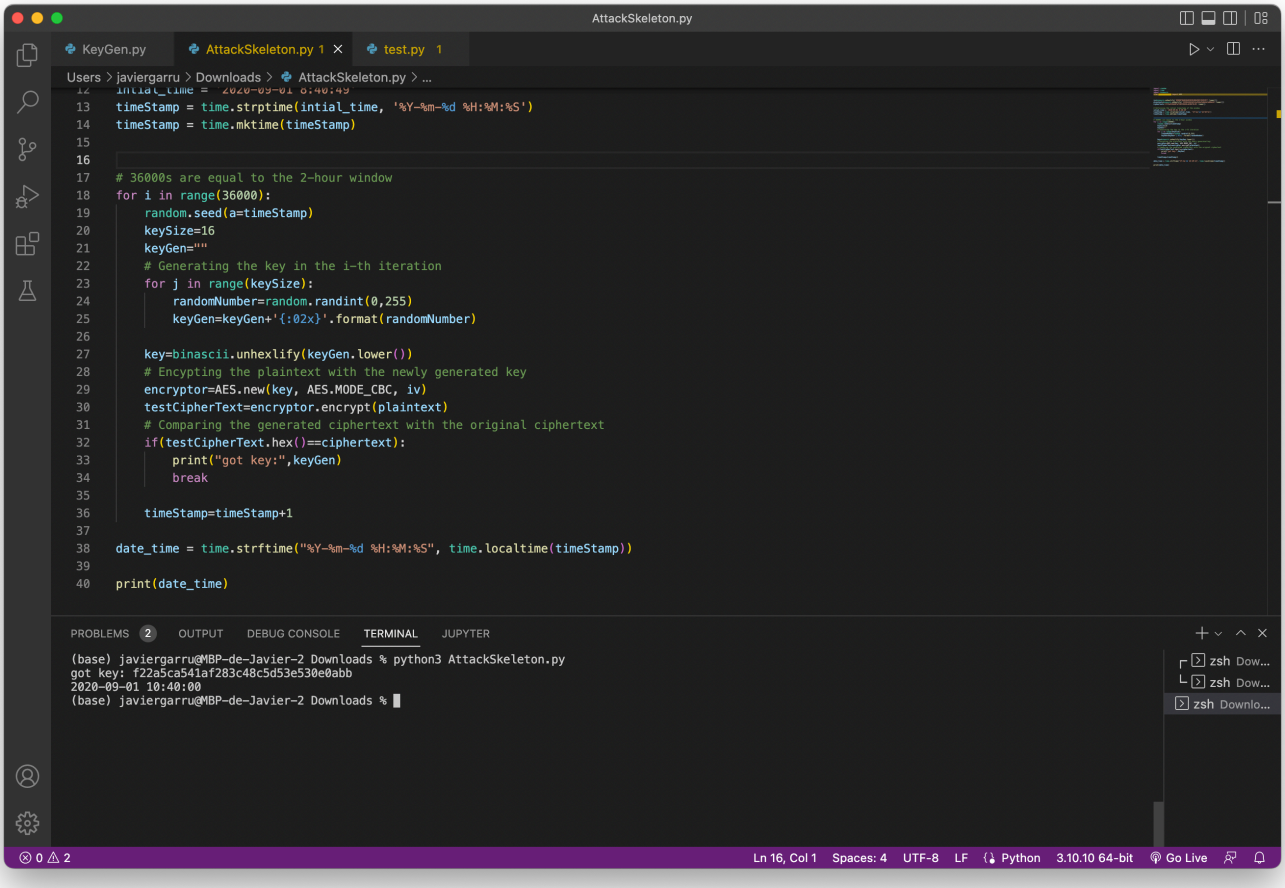
Then, we edit the skeleton code:



The obtained key is : `f22a5ca541af283c48c5d53e530e0abb`. We can make a small program to check that this is indeed like this:



We can also modify the original program to obtain the exact encyption time:



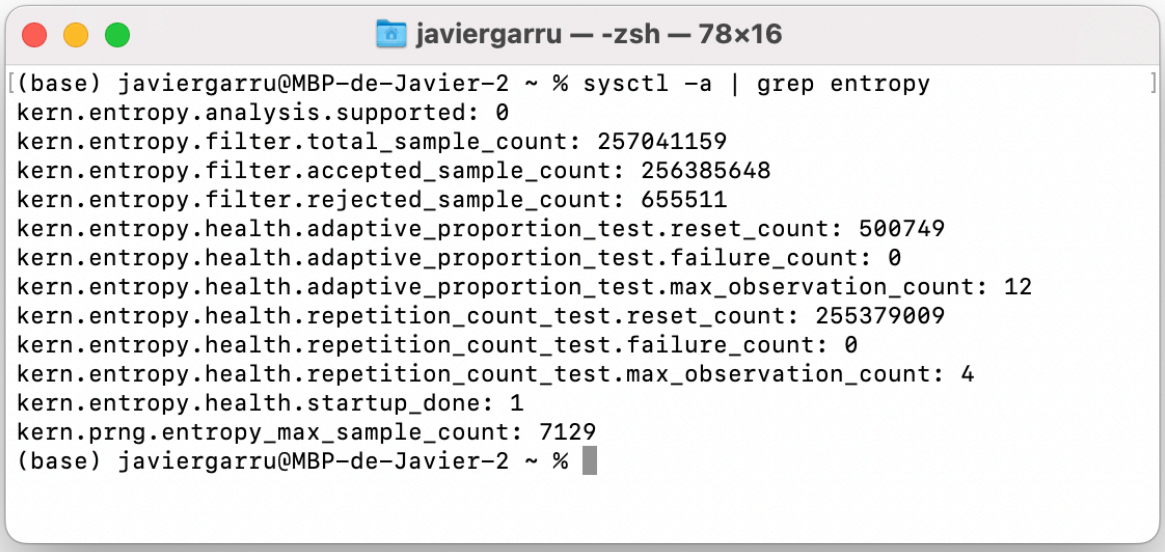
Part Three: Measure the Entropy of Kernel

The best way of generating **good** pseudorandom numbers is by starting by something that is random. Software isn't good doing so and computers rely on hardware to do it. Some sources of this randomness are:

- Keystrokes
- Mouse movement
- Interruptions
- Finishing time of block device requests

All this randomness is measured using entropy, which is the number of bits of random numbers that the system has at a precise instant. We can have a look at the current entropy of the system's kernel in MacOS:

```
sysctl -a | grep entropy
```



We are interested in the top three lines:

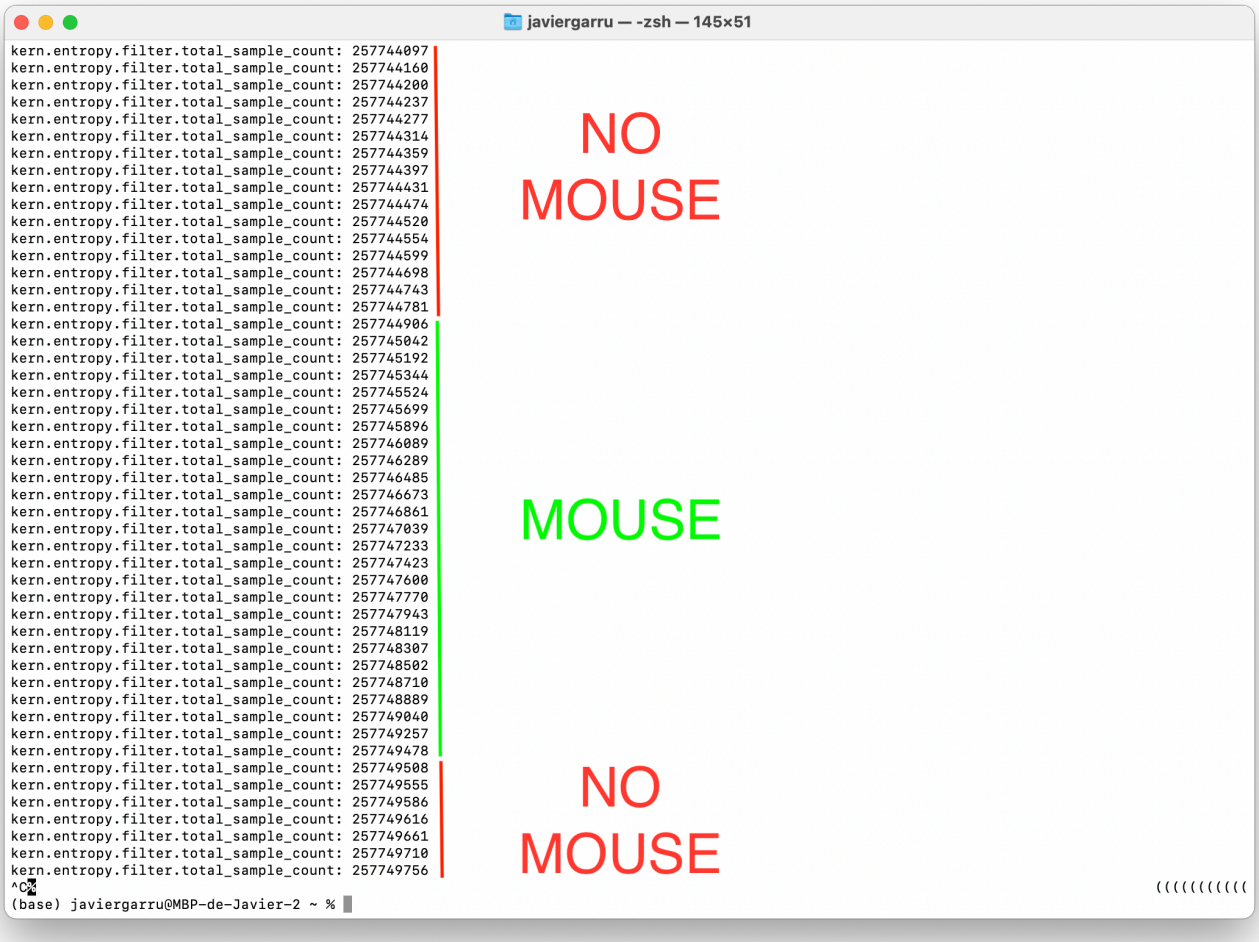
- kern.entropy.filter.total_sample_count
- kern.entropy.filter.accepted_sample_count
- kern.entropy.filter.rejected_sample_count

The sum of the bottom two equals the top one. The reason why some of the bits of the entropy count of the kernel are rejected is because they could be biased or predictable and therefore of poor quality.

In macOS the following command can be used to repeatedly execute the previous command:

```
while true; do sysctl -a | grep kern.entropy.filter.total_sample_count; sleep 0.1;done;
```

Here is the obtained output:



While the numbers are quite similar as it is and absolute could the areas where the mouse wasn't being used and the areas where it was are highlighted. We can zoom in and see how the order of the changes between iterations is almost double or even triple when the mouse is being moved. This peculiarity can be seen in the third last digit starting from the left:

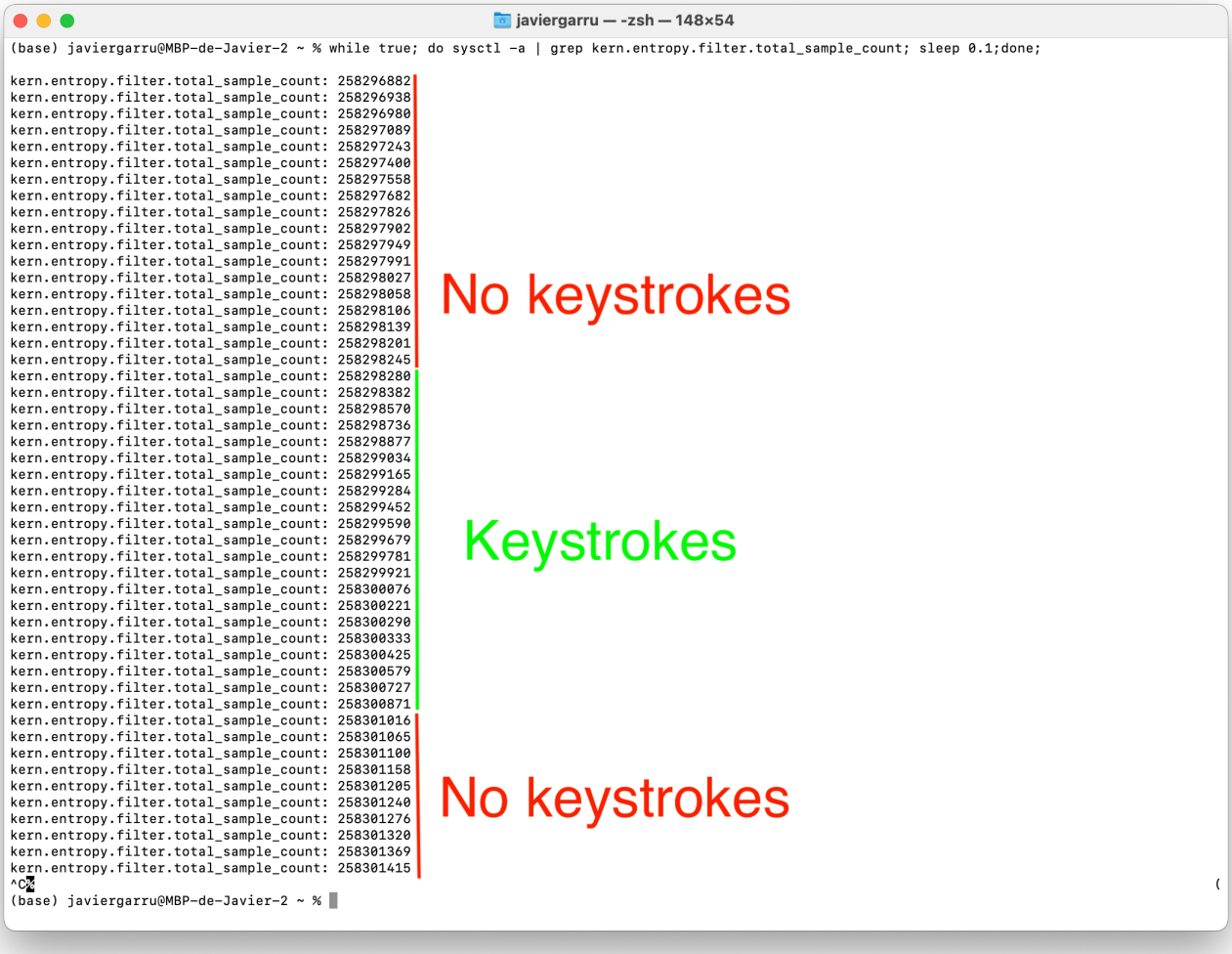
- Iterations extracted when the mouse was **still**:

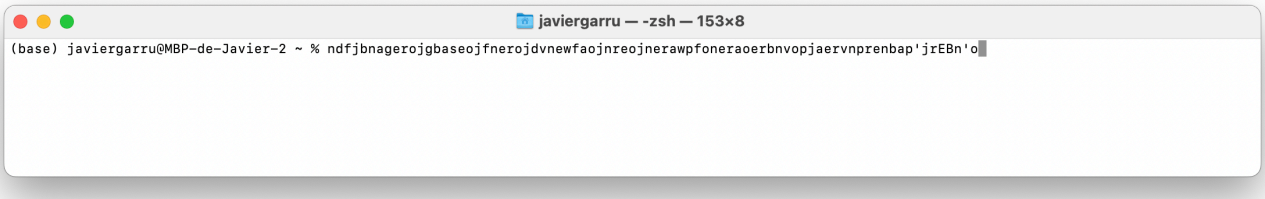
```
kern.entropy.filter.total_sample_count: 257744097
kern.entropy.filter.total_sample_count: 257744160
kern.entropy.filter.total_sample_count: 257744200
kern.entropy.filter.total_sample_count: 257744237
kern.entropy.filter.total_sample_count: 257744277
```

- Iterations extracted when the mouse was **moving**:

```
kern.entropy.filter.total_sample_count: 257745699
kern.entropy.filter.total_sample_count: 257745896
kern.entropy.filter.total_sample_count: 257746089
kern.entropy.filter.total_sample_count: 257746289
kern.entropy.filter.total_sample_count: 257746485
```

Let's now have a look to what happens when we type in something. We will open two terminal windows at the same type and type in one of them while the other one executes the loop:



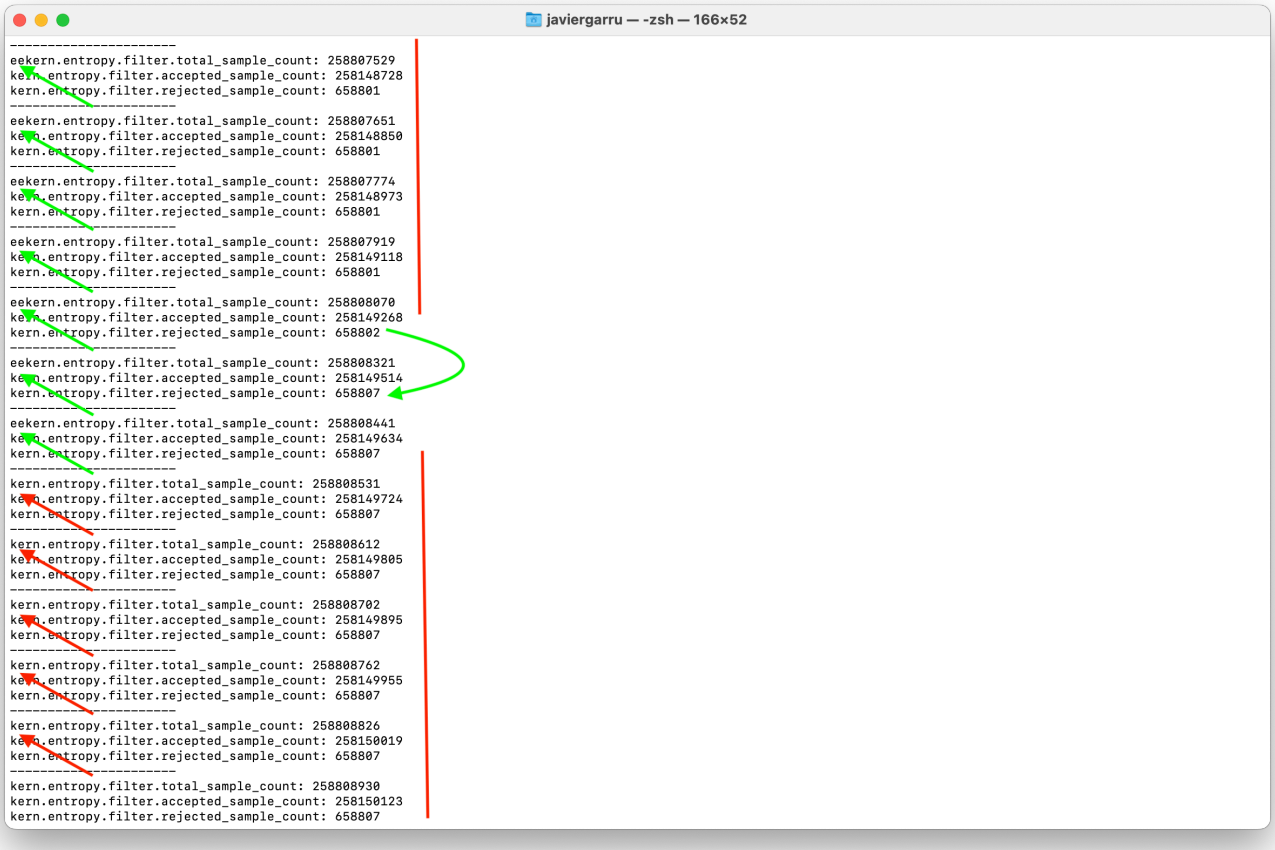


Again it can be seen how the third digit from the left increases more rapidly when the keyboard is being used. This experiment makes me to wonder what would happen if the key being pressed is repeatedly the same and with a constant rithm. We modify the original command to:

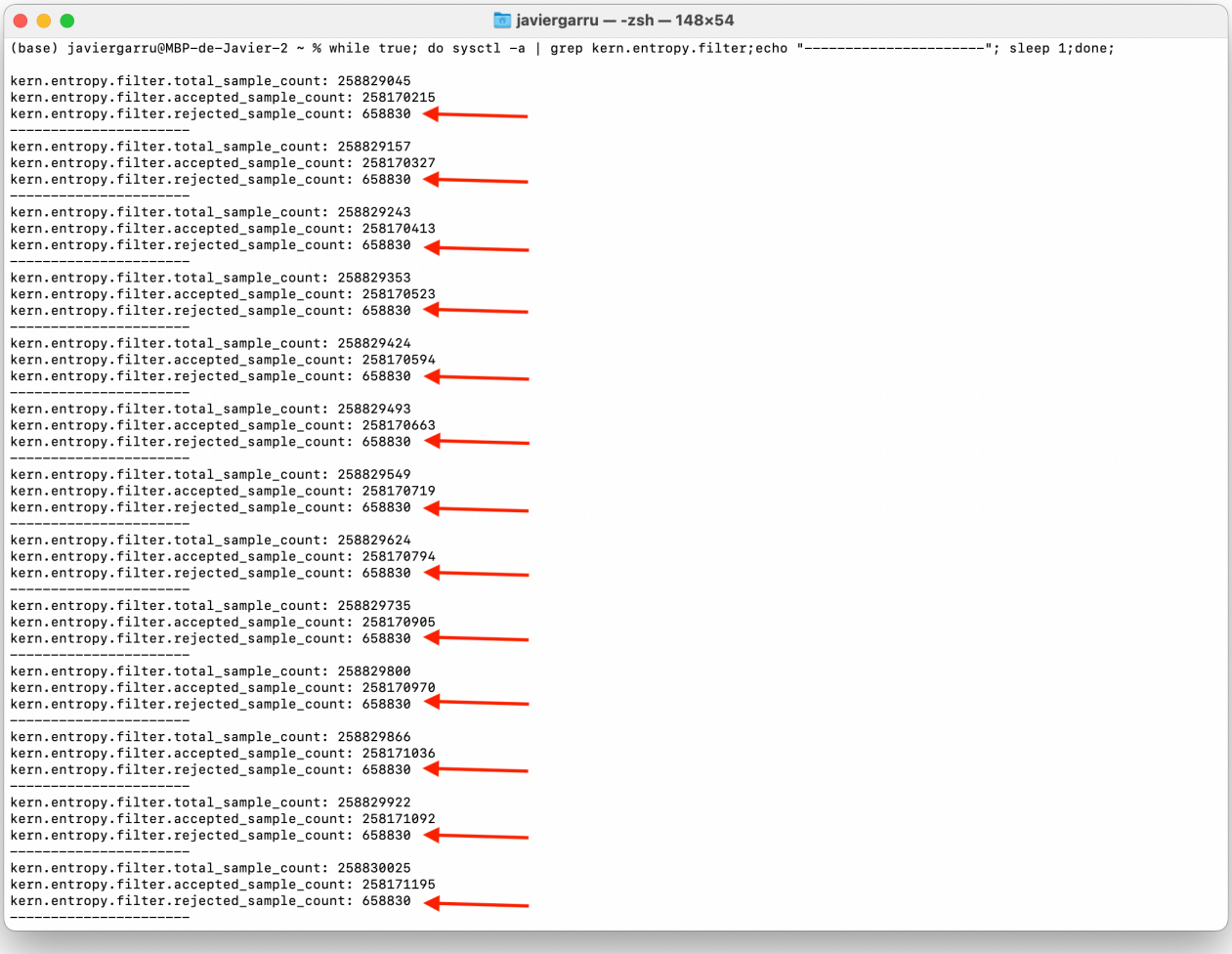
```
while true; do sysctl -a | grep kern.entropy.filter;echo "----------------------";
sleep 1;done;
```

This way we ensure that we can tell the difference between executions and print the three types of bits in the pool.

In this case I will write "ee" (constant value) every time before the loop is executed. Therefore every 1 second (constant pace). As we can see in the following screenshot the value of `kern.entropy.filter.rejected_sample_count` stays constant during the initial iterations, but after a certain number of them the rejected sample count increases significantly.



Here we can see another screenshot of an execution where no keystrokes nor movement of the mouse is executed:



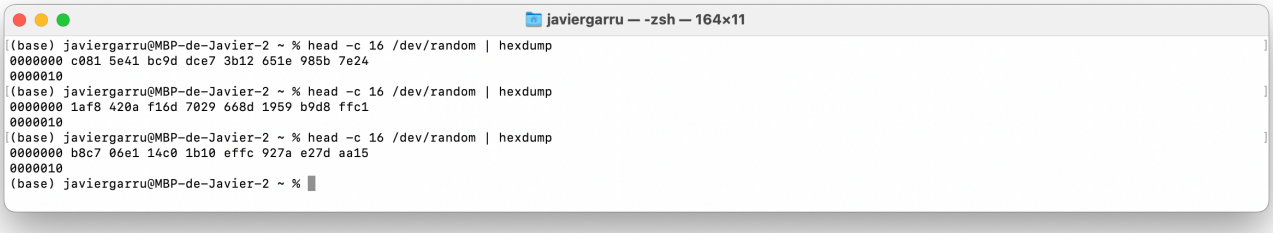
As it is clear from the results, the mouse movement and keyboard keystrokes generate entropy bits that are situated in the entropy pool. We have also seen how when the execution is constant and rithmic, the bits end up being discarded into the rejected pool due to the absence of randomness.

In the following section we would see how this also occurs in linux.

Part Four: Get Random Numbers from /dev/random

We extract 16 random bytes from /dev/random:

```
head -c 16 /dev/random | hexdump
```

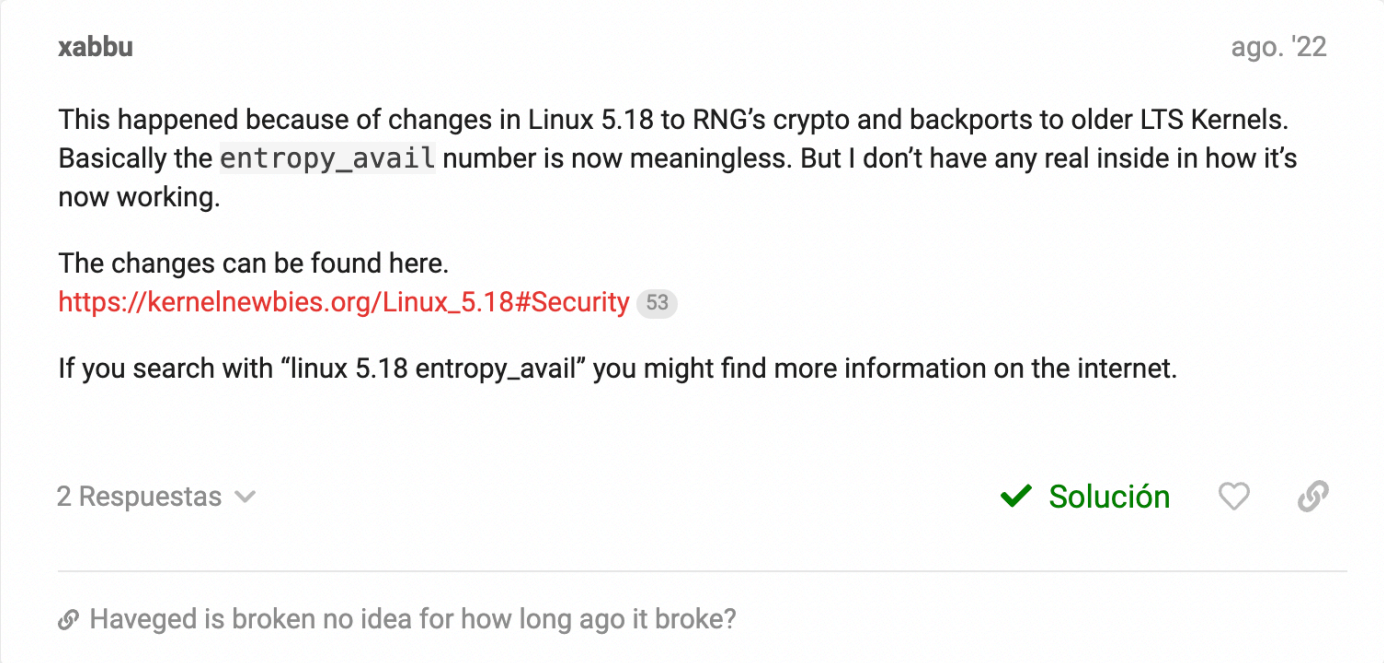


As we can see each execution returns different set of values and seem to be completely random. It is not recommended (and almost impossible) to block /dev/random in Mac. We can see in their **documentation** that the system **won't block** when the **entropy** pool is **low**: *Additional entropy is fed to the generator regularly by the SecurityServer daemon from random jitter mea surements of the kernel.* The random generation of numbers is done through an algorithm that obviously uses entropy but the system isn't as direct as in Linux where the values are directly fed into the entropy pool and extracted from there. It could be said that MacOS has something closer to /dev/urandom because it will never block.

Let's do the following tasks from a virtual machine of Ubuntu. After generating some entropy, we show the first 16 bytes and then proceed to empty the pool:

```
head -c 16 /dev/random | hexdump
```

NOTE: Since an update in 2022 the *entropy_avail* is useless as the entropy **pool size** is set to **256**. Some forums on the topic confirm so:

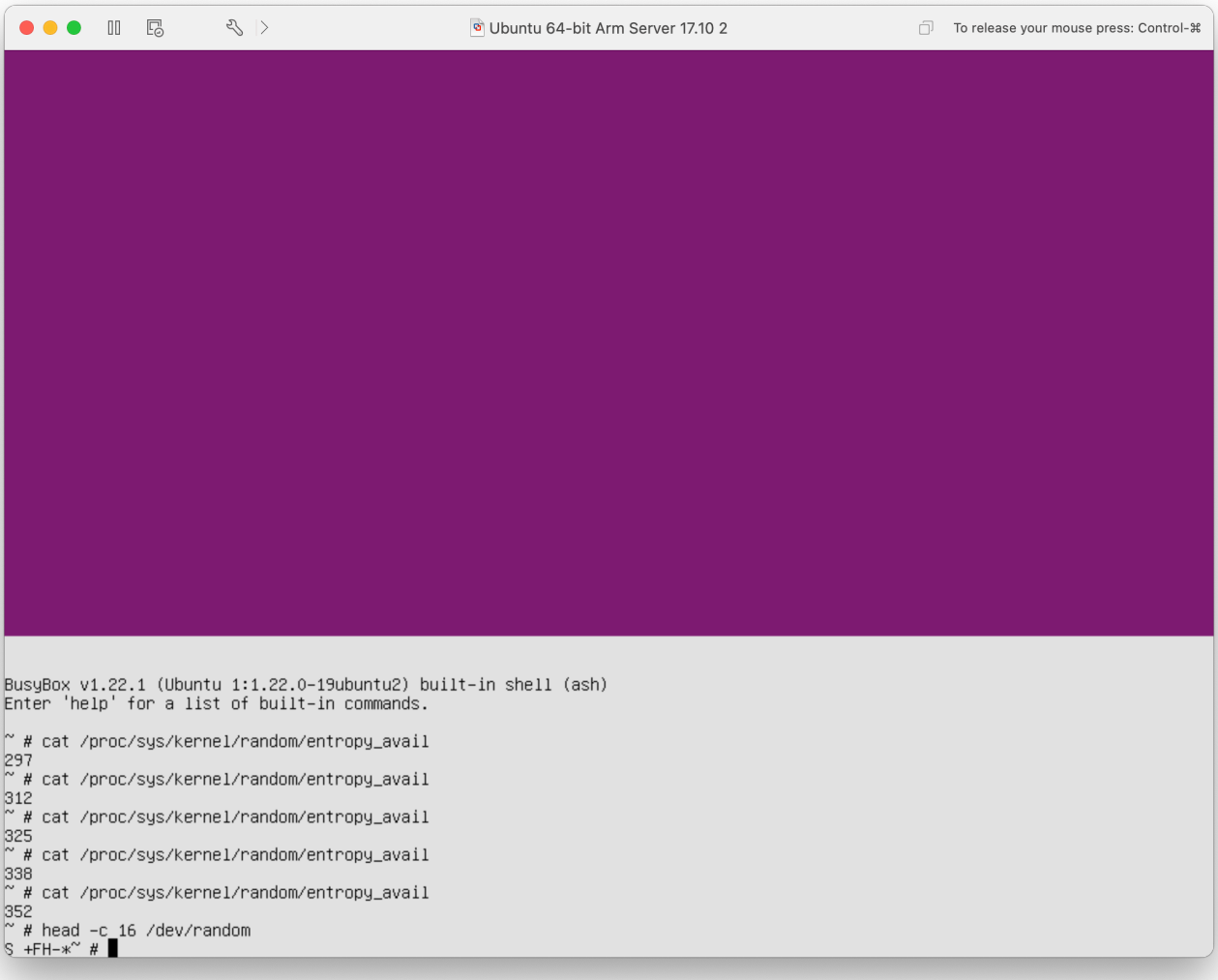


For the following task I needed to download another ubuntu server (older version) different from the one being used in the rest of the sections. Let's proceed with the older version.

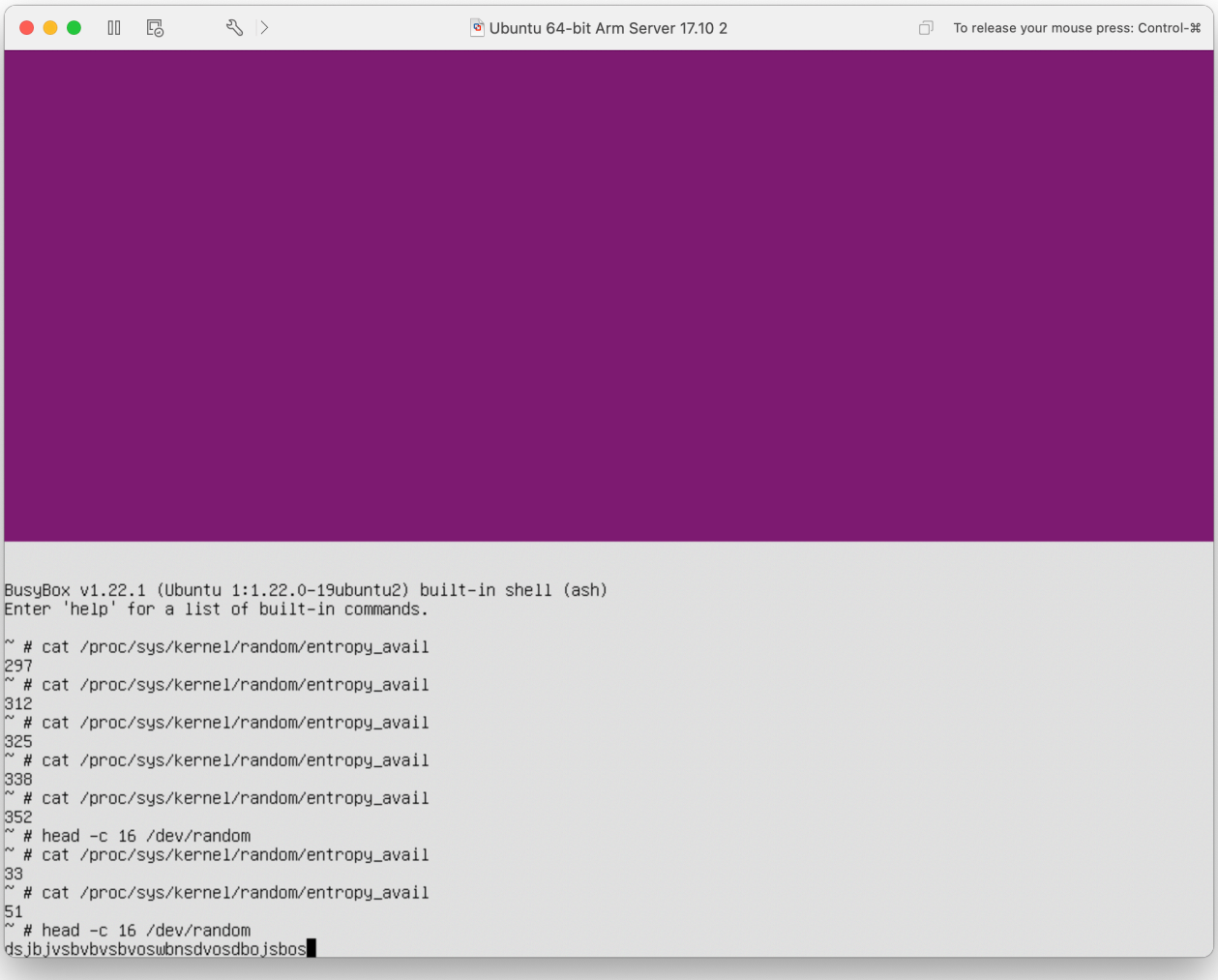
First we use the following command several times to check the entropy available. It seems like executing the command generates entropy due to internal processes:

```
cat /proc/sys/kernel/random/entropy_avail
```

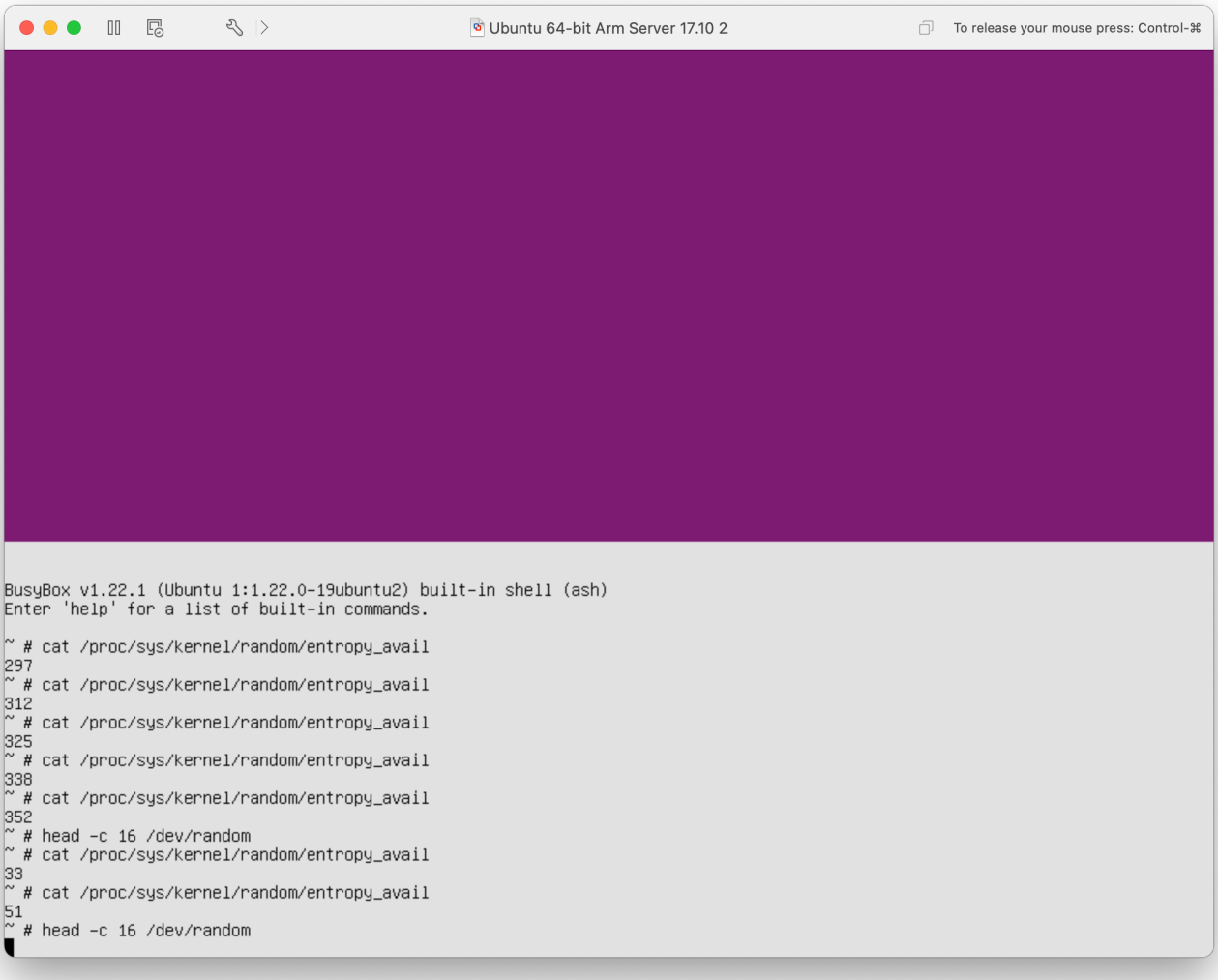
Then, we take 16 bytes and see how that affects the total of bits in the pool. The output bytes are at the bottom of the screenshot:

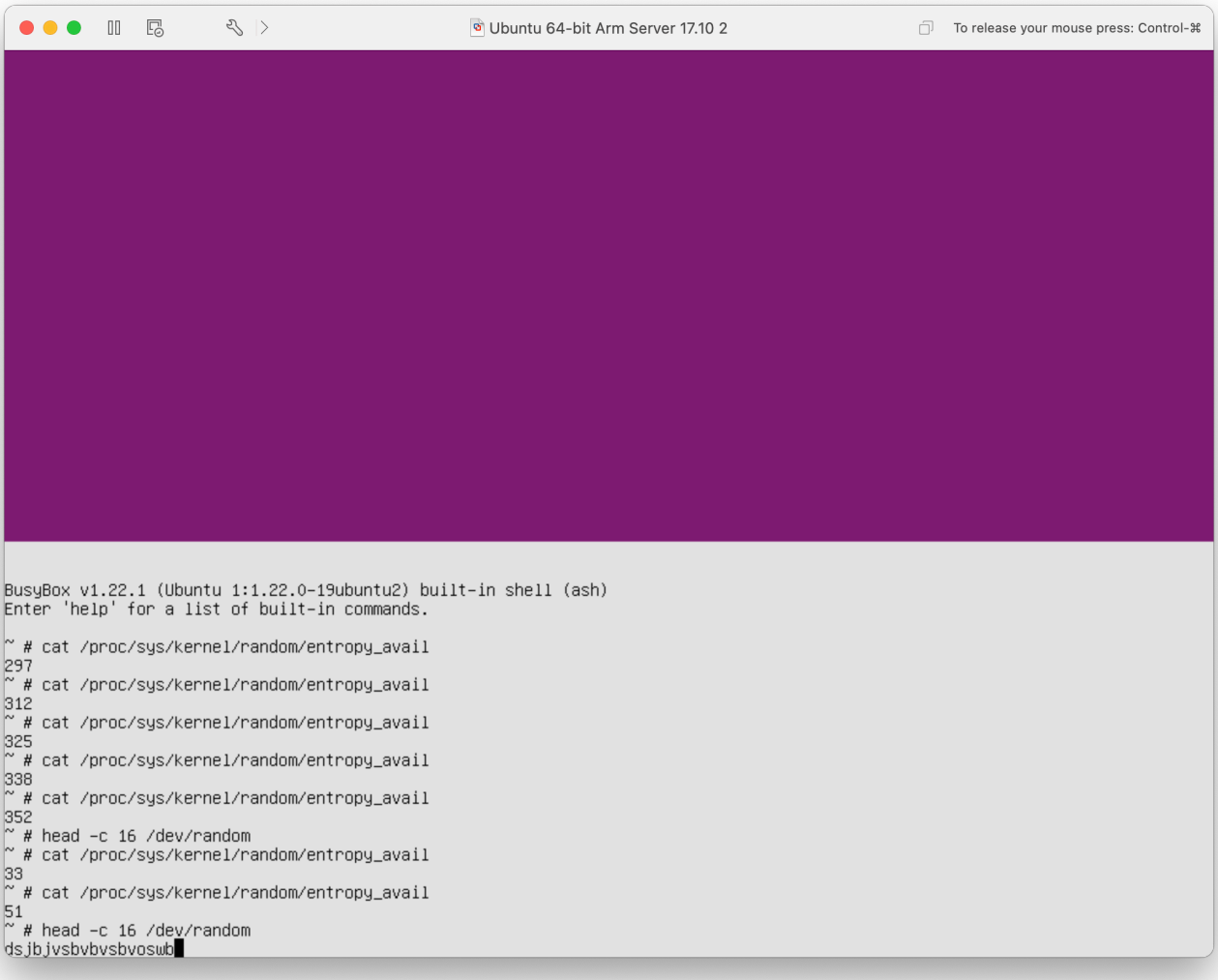


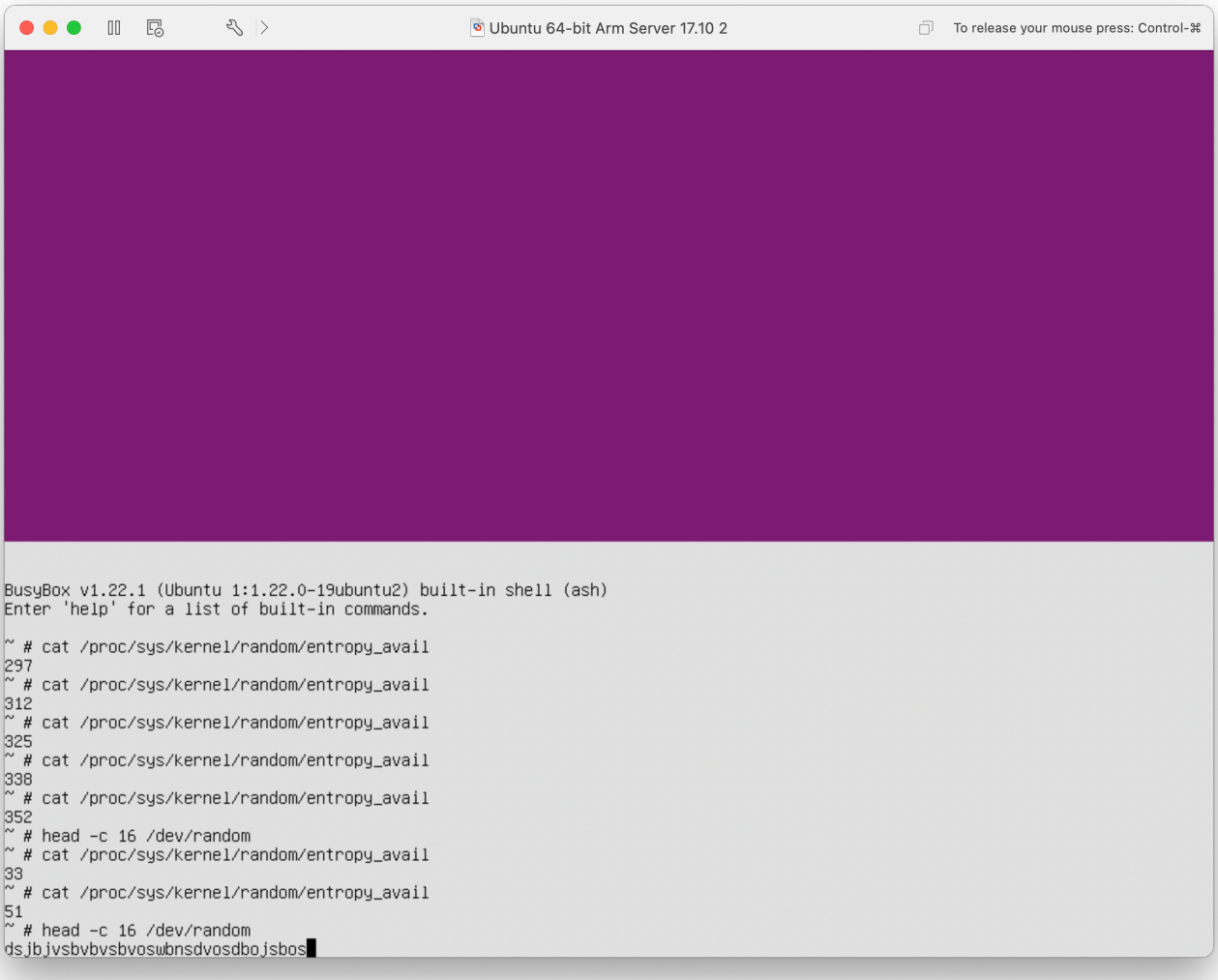
As we can see in the following screenshot the entropy bits have reduced considerably:



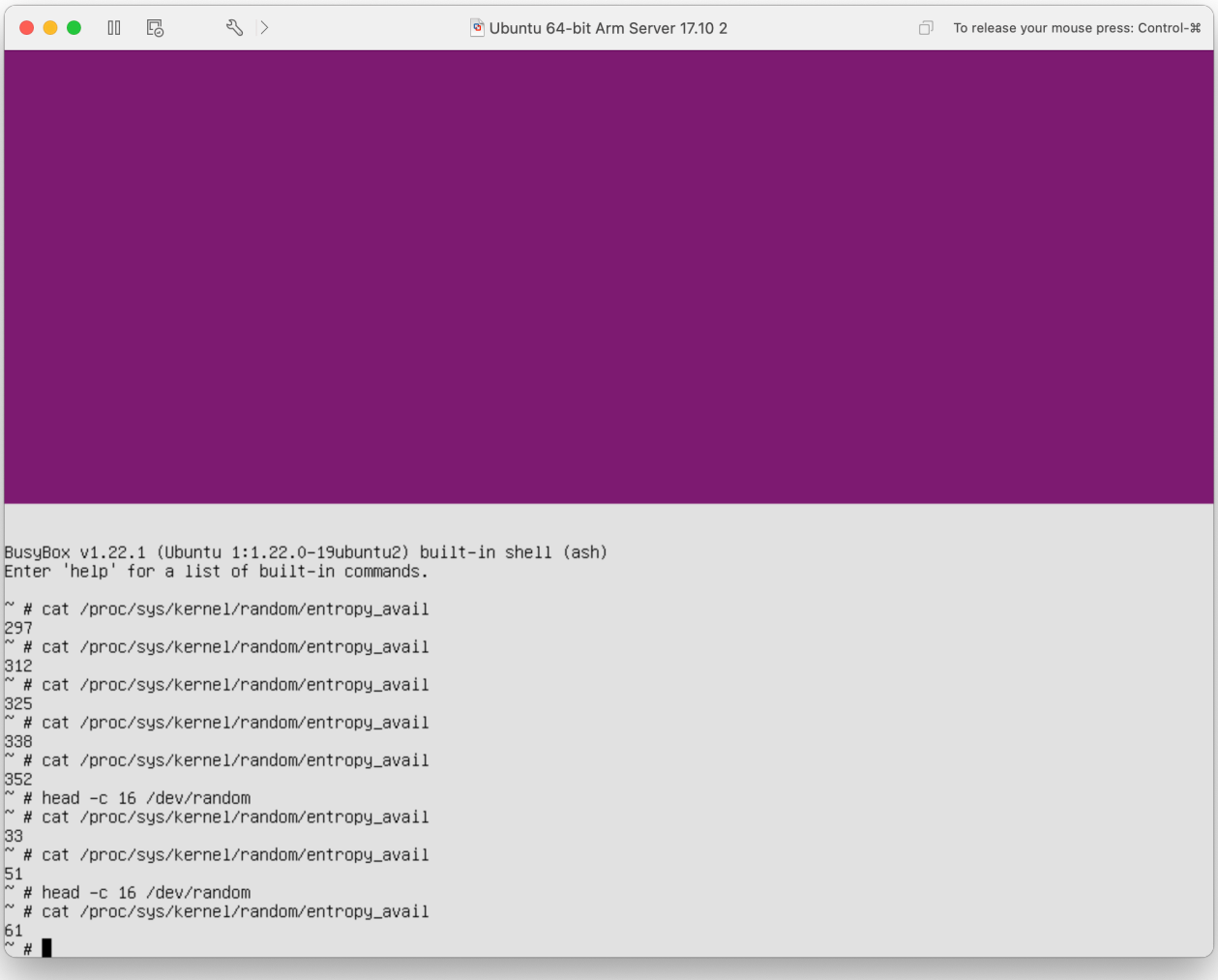
Then we do it again and the system blocks. To unblock it, and considering there is no mouse interaction in ubuntu server, we type random letters until it unlocks:


Ubuntu 64-bit Arm Server 17.10 2
To release your mouse press: Control-⌘
BusyBox v1.22.1 (Ubuntu 1:1.22.0-19ubuntu2) built-in shell (ash)
Enter 'help' for a list of built-in commands.
~ # cat /proc/sys/kernel/random/entropy_avail
297
~ # cat /proc/sys/kernel/random/entropy_avail
312
~ # cat /proc/sys/kernel/random/entropy_avail
325
~ # cat /proc/sys/kernel/random/entropy_avail
338
~ # cat /proc/sys/kernel/random/entropy_avail
352
~ # head -c 16 /dev/random
~ # cat /proc/sys/kernel/random/entropy_avail
33
~ # cat /proc/sys/kernel/random/entropy_avail
51
~ # head -c 16 /dev/random


Ubuntu 64-bit Arm Server 17.10 2
To release your mouse press: Control-⌘
BusyBox v1.22.1 (Ubuntu 1:1.22.0-19ubuntu2) built-in shell (ash)
Enter 'help' for a list of built-in commands.
~ # cat /proc/sys/kernel/random/entropy_avail
297
~ # cat /proc/sys/kernel/random/entropy_avail
312
~ # cat /proc/sys/kernel/random/entropy_avail
325
~ # cat /proc/sys/kernel/random/entropy_avail
338
~ # cat /proc/sys/kernel/random/entropy_avail
352
~ # head -c 16 /dev/random
~ # cat /proc/sys/kernel/random/entropy_avail
33
~ # cat /proc/sys/kernel/random/entropy_avail
51
~ # head -c 16 /dev/random
dsjbjvsbvbvsbvoswb



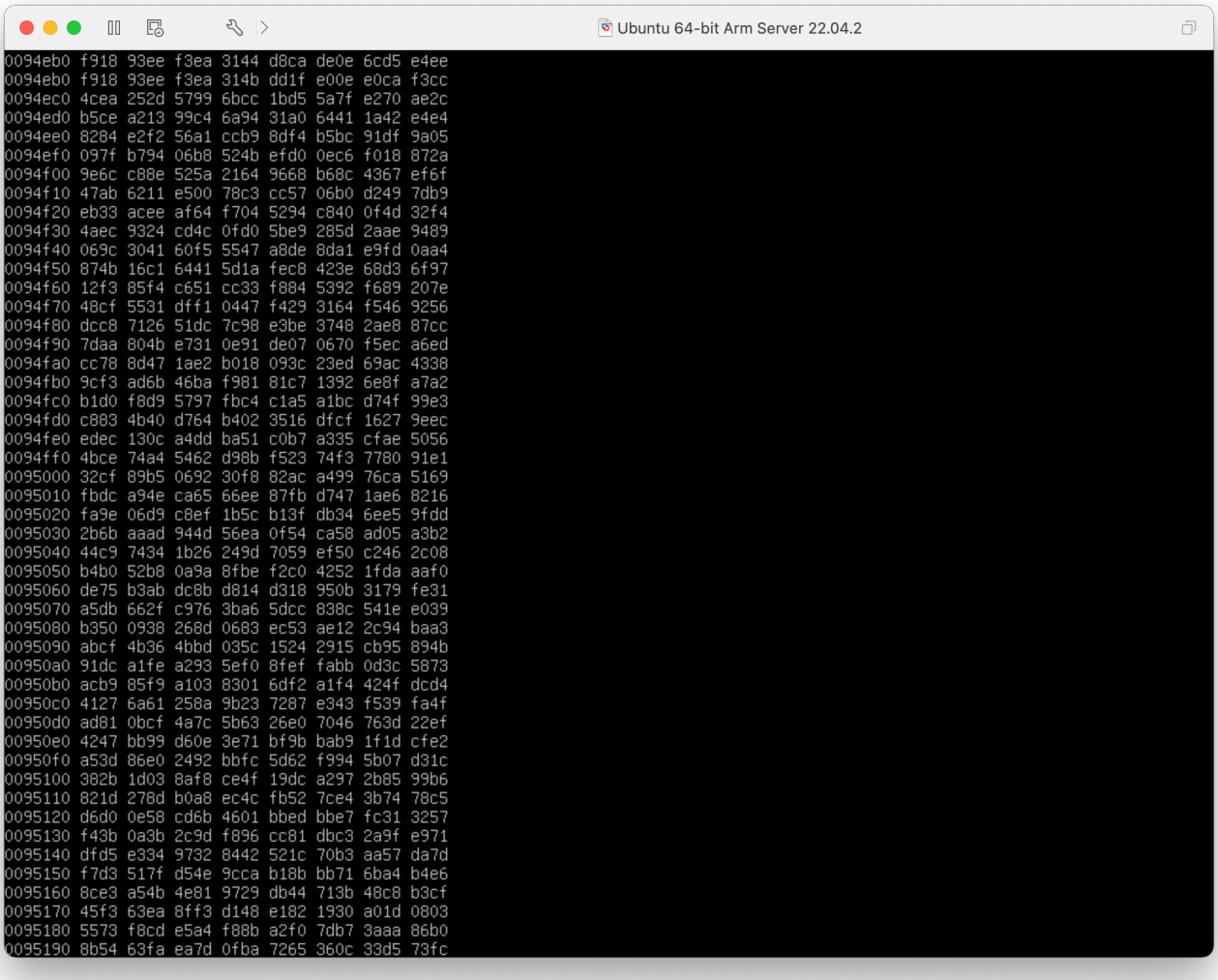
When it unlocks, it shows the extracted value (which cannot be seen in the screenshots as it dissapears when other command is typed in this version). We can visualized the pool again to see how many bits are left after adding entropy and extracting the 16 bytes that caused the system to block:



Part Five: Get Random Numbers from /dev/urandom

In /dev/urandom we will find another source of random numbers but the difference with /dev/random is that /dev/urandom won't block. We can extract 10MB of data from this source and see how it won't block:

```
head -c 10M /dev/urandom | hexdump
```

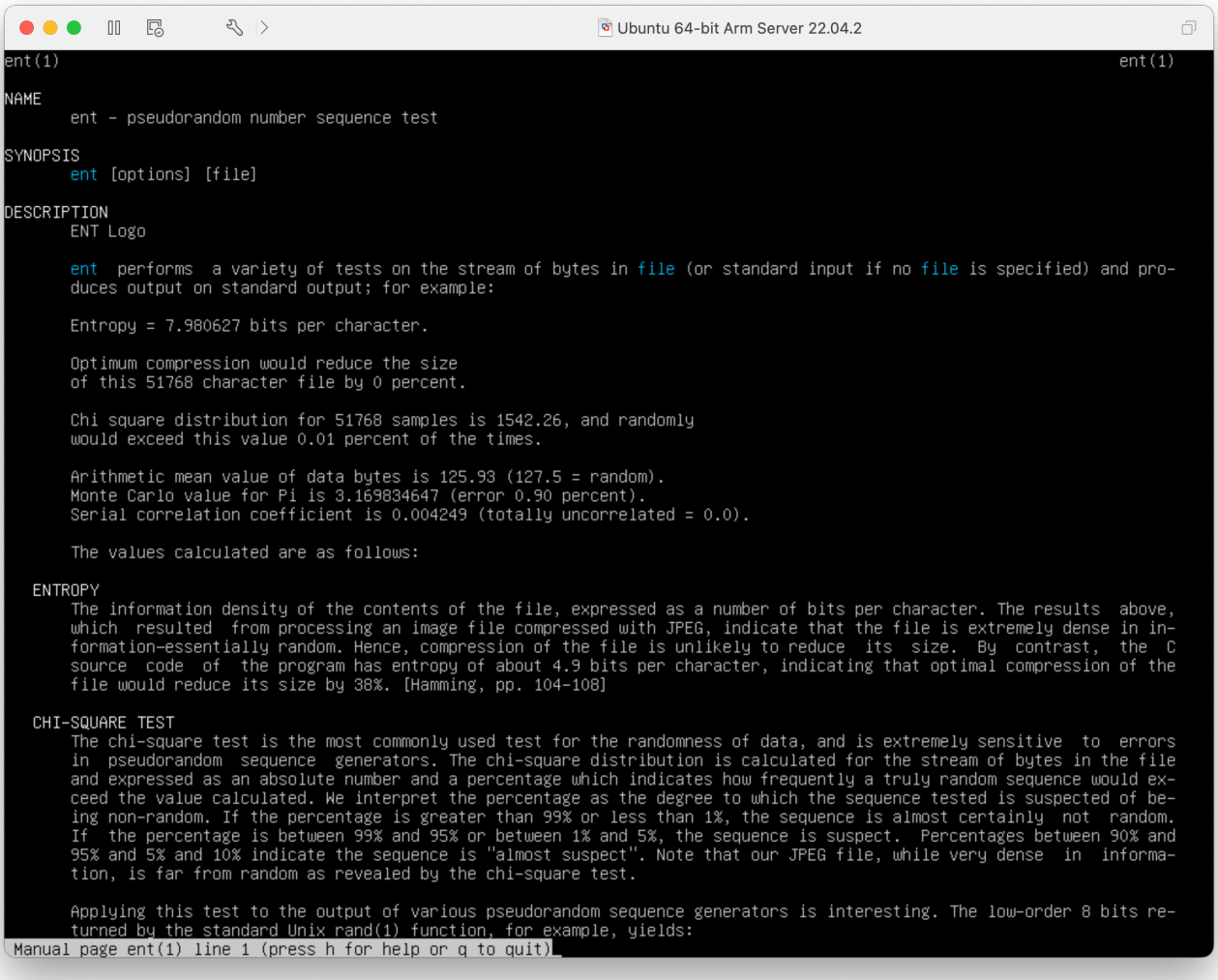


The command takes several seconds to execute and print all the values but after doing it a few times the system doesn't block as urandom keeps generating numbers.

Now, we proceed to download **ent**:

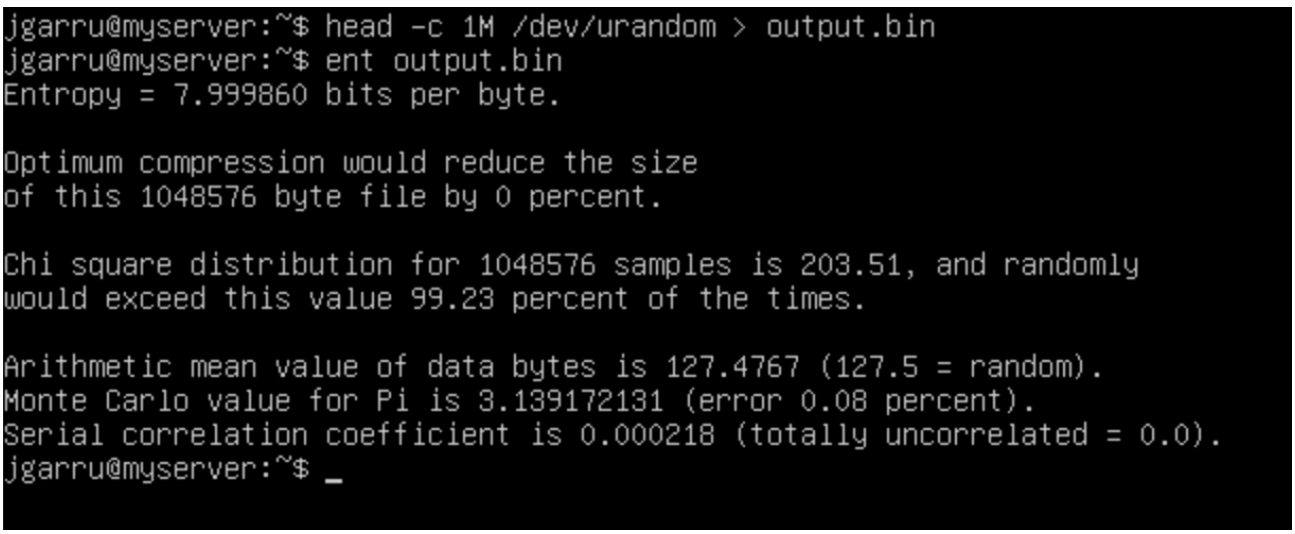
```
sudo apt install ent
```

We can have a look at its manueal to learn about its options:



Now, we extract 1 MB from /dev/urandom and put it in a file output.bin. Then we use ent to analyze the values in ent:

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



The values obtained from /dev/urandom are definitely well randomized as the output of the ent command indicates so. First, the **chi square** distribution is 203.51 and, if the values were random, this value would be exceeded 99.23 percent of the times. This percentage is significantly close to 100% which indicates that the values are significantly random. Furthermore the **mean** of the data is 127.47 which indicates that the values are uniformly distributed. Someone could ask why 127.5 indicates random, and the reason is because 127.5 is the average of 0-255 which is the range of values. The expected value being really close to 127.5 indicates uniform distirbution. The **Monte Carlo** value for **Pi** is also close to the actual value (3.139172131 ~ 3.14.16...) and that is a really good indication. At last the correlation coefficient is nearly zero which indicates that the samples are almost independent which is what we are looking for.

In conclusion, the results of the test are really positive and indicate that the values are indeed random.

Lab summary:

In this lab we have learned several different things. First, we have understood the dangers of generating pseudorandom numbers using the local time as seed. Then, we have seen how to generate entropy and how does MacOS deal with the generated bits. Following this, the differences between /dev/random and /dev/urandom have been outlined and random numbers have been extracted from both. The fact that /dev/random could (and did) block the system wasn't good as it could lead to denial of service attacks. We have learnt that new linux versions have reduced the size of the entropy pool and have methods to generate entropy in /dev/random similar to /dev/urandom. At last, the randomness of the numbers extracted from urandom has been checked using ent.