

Cryptography - Lab 4 - Pseudorandom numbers

Javier Antxon Garrues Apecechea

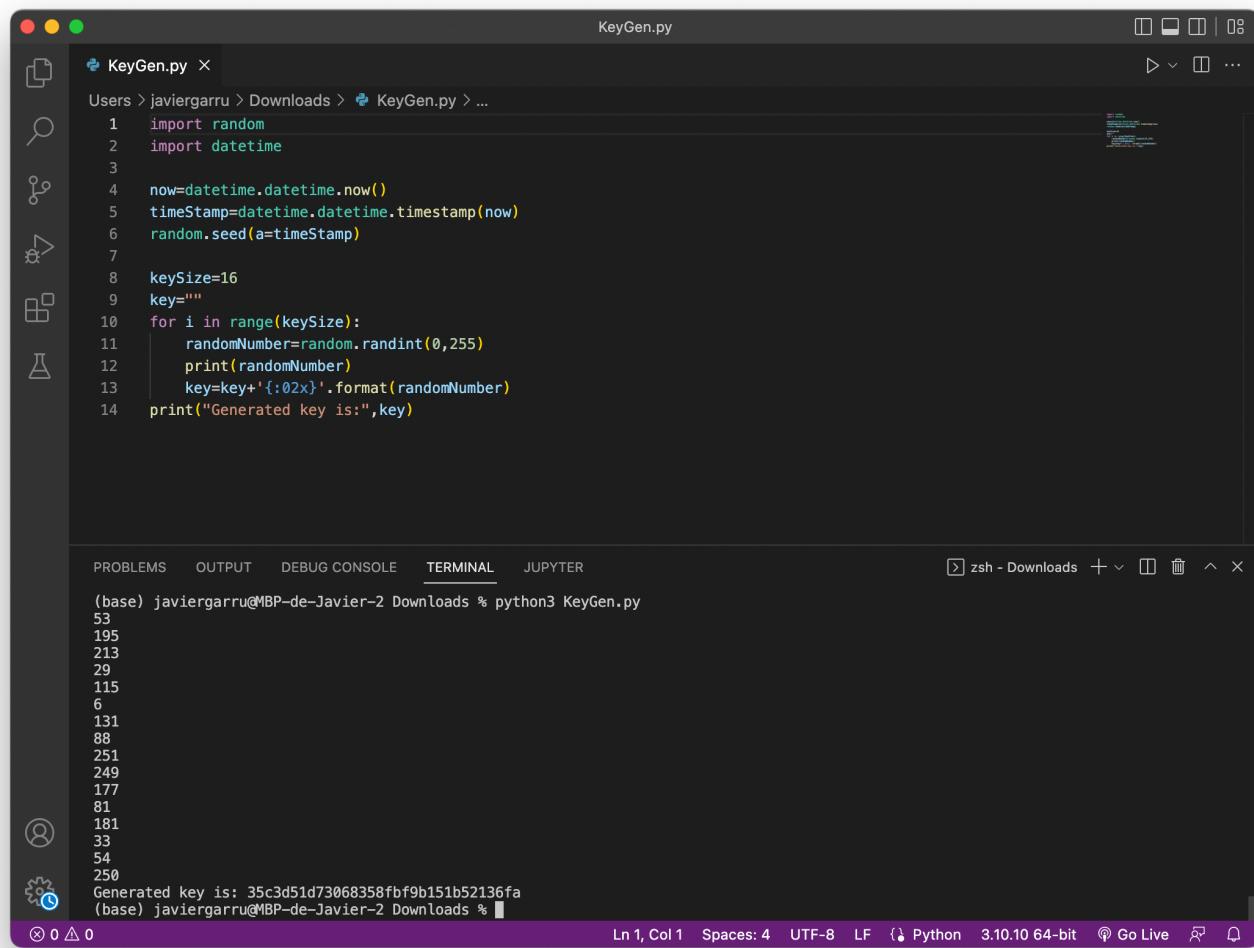
Student ID: 24647808

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:

```
python3 KeyGen.py
```



The screenshot shows a dark-themed instance of Visual Studio Code. In the center, there's a code editor window titled "KeyGen.py" containing the following Python script:

```
import random
import datetime
now=datetime.datetime.now()
timeStamp=datetime.datetime.timestamp(now)
random.seed(a=timeStamp)

keySize=16
key=""
for i in range(keySize):
    randomNumber=random.randint(0,255)
    print(randomNumber)
    key=key+{:02x}.format(randomNumber)
print("Generated key is:",key)
```

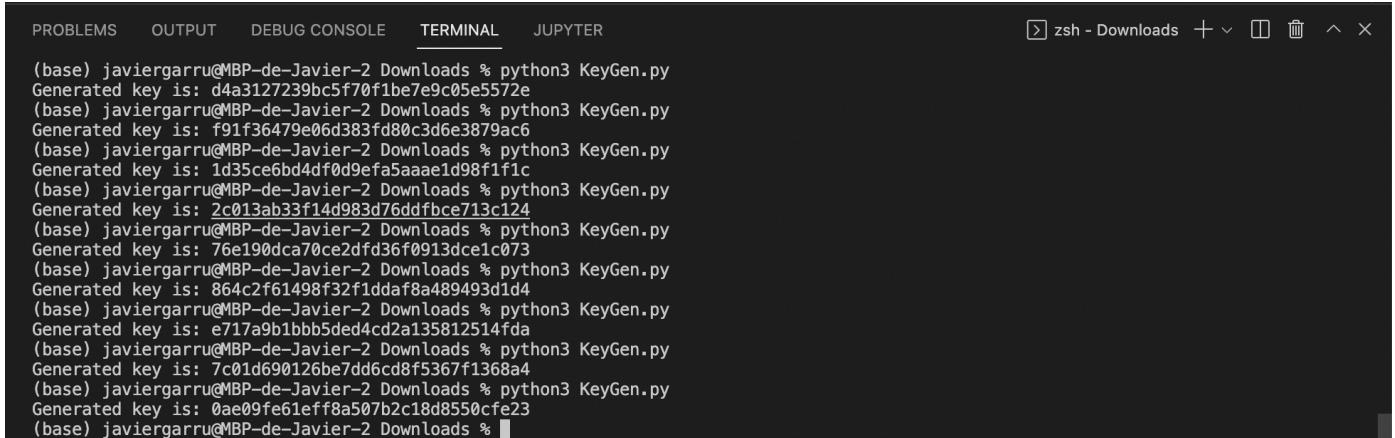
Below the code editor is a terminal window showing the output of running the script:

```
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
53
195
213
29
115
6
131
88
251
249
177
81
181
33
54
250
Generated key is: 35c3d51d73068358fbf9b151b52136fa
(base) javiergarru@MBP-de-Javier-2 Downloads %
```

The status bar at the bottom indicates the file has 14 lines, 0 spaces, and 0 tabs, and is using Python 3.10.10 64-bit.

Describe your observations:

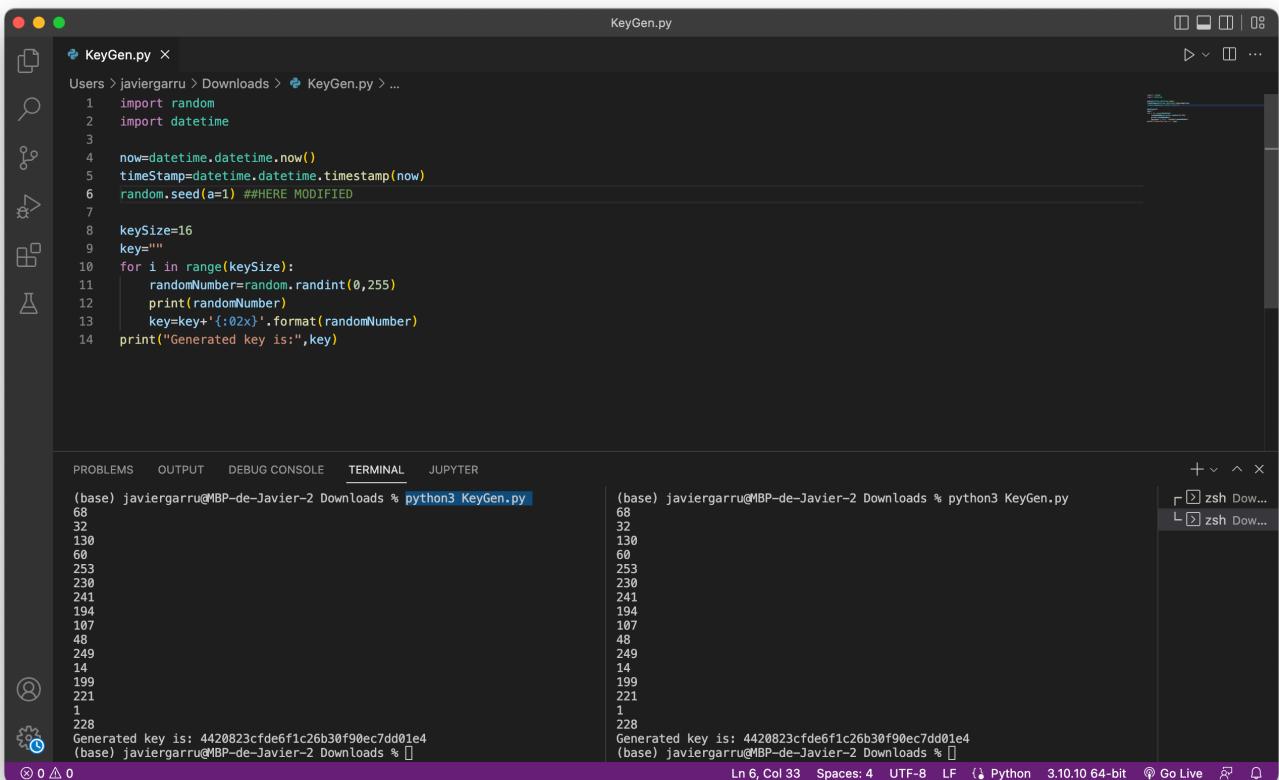
After obtaining the actual time and using it as the seed of the *random* number generator, the keySize is defined. Then, a number of random numbers equal to the keySize are generated and concatenated changing their format to a 2 digit hexadecimal value. If the value is lower than 10, the missing value 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
Generated key is: f91f36479e06d383fd80c3d6e3879ac6
Generated key is: 1d35ce6bd4df0d9efa5aaae1d98f1f1c
Generated key is: 2c013ab33f14d983d76ddfbce713c124
Generated key is: 76e190dca70ce2dfd36f0913dce1c073
Generated key is: 864c2f61498f32f1ddaf8a489493d1d4
Generated key is: e717a9b1bbb5ded4cd2a135812514fda
Generated key is: 7c01d690126be7dd6cd8f5367f1368a4
Generated key is: 0ae09fe61eff8a507b2c18d8550cf23
Generated key is: 1
(base) javiergarru@MBP-de-Javier-2 Downloads %
```

Now, executing the modified file:



```
KeyGen.py
KeyGen.py

Users > javiergarru > Downloads > KeyGen.py > ...
1 import random
2 import datetime
3
4 now=datetime.datetime.now()
5 timestamp=datetime.datetime.timestamp(now)
6 random.seed(a=1) ##HERE MODIFIED
7
8 keySize=16
9 key=""
10 for i in range(keySize):
11     randomNumber=random.randint(0,255)
12     print(randomNumber)
13     key=key+{:02x}.format(randomNumber)
14 print("Generated key is:",key)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER + ▾ ▲ ×
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
68
32
130
60
253
230
241
194
107
48
249
14
199
221
1
228
Generated key is: 4420823cfde6f1c26b30f90ec7dd01e4
(base) javiergarru@MBP-de-Javier-2 Downloads %
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 KeyGen.py
68
32
130
60
253
230
241
194
107
48
249
14
199
221
1
228
Generated key is: 4420823cfde6f1c26b30f90ec7dd01e4
(base) javiergarru@MBP-de-Javier-2 Downloads %
Ln 6, Col 33 Spaces: 4 UTF-8 LF ⚙ Python 3.10.10 64-bit ⚙ Go Live ⚙
```

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 correspondent 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: 0908070605040302010A2B2C2D2E2F2

Ciphertext: 2f15e526b607a73b228648cd206f9c95

Describe your idea to find out the key:

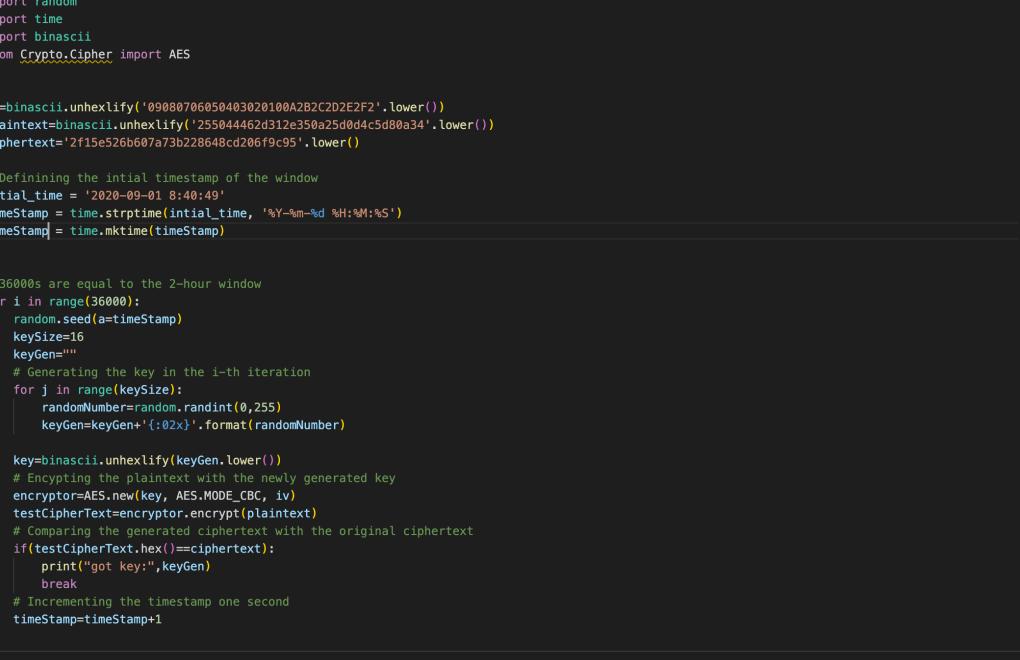
My idea would be iterating on the range of possible timestamps (with a for loop) and generating a different key at each iteration. In the same iteration I would use the aes-128-cb with the obtained key and predefined IV and ciphertext to decrypt the message. At last, I would compare the obtained plaintext 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 possession) 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
```

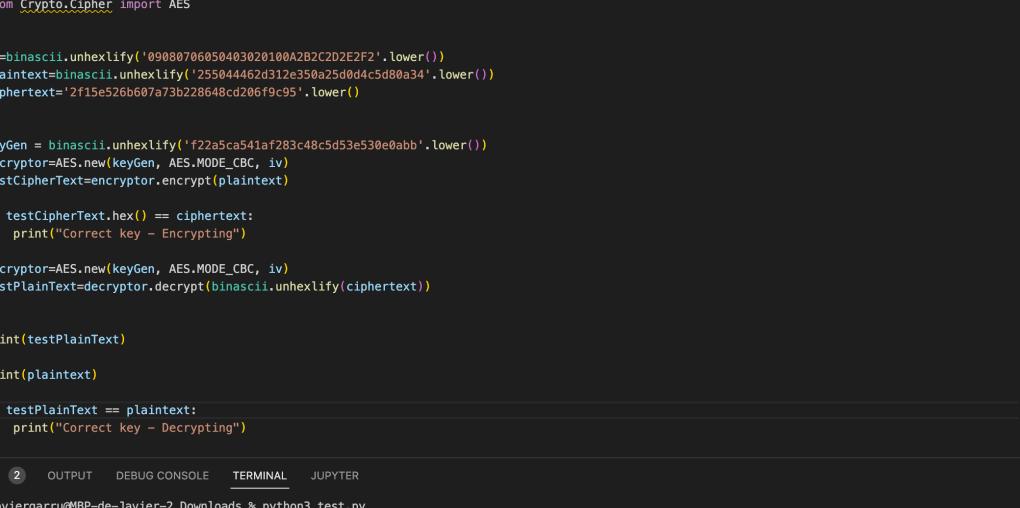
Then, we edit the skeleton code:



```
AttackSkeleton.py
Users > javiergarru > Downloads > AttackSkeleton.py > ...
1 import random
2 import time
3 import binascii
4 from Crypto.Cipher import AES
5
6
7 iv=binascii.unhexlify('09080706050403020100A2B2C2D2E2F2'.lower())
8 plaintext=binascii.unhexlify('255044462d312e350a25d0d4c5d80a34'.lower())
9 ciphertext='2f15e526607a73b228648cd206f9c95'.lower()
10
11 # Definining the initial timestamp of the window
12 initial_time = '2020-09-01 8:40:49'
13 timeStamp = time.strptime(initial_time, '%Y-%m-%d %H:%M:%S')
14 timeStamp = time.mktime(timeStamp)
15
16
17 # 3600s are equal to the 2-hour window
18 for i in range(3600):
19     random.seed(a=timeStamp)
20     keySize=16
21     keyGen=""
22     # Generating the key in the i-th iteration
23     for j in range(keySize):
24         randomNumber=random.randint(0,255)
25         keyGen=keyGen+{:02x}.format(randomNumber)
26
27     key=binascii.unhexlify(keyGen.lower())
28     # Encrypting the plaintext with the newly generated key
29     encryptor=AES.new(key, AES.MODE_CBC, iv)
30     testCipherText=encryptor.encrypt(plaintext)
31     # Comparing the generated ciphertext with the original ciphertext
32     if(testCipherText.hex()==ciphertext):
33         print("got key:",keyGen)
34         break
35     # Incrementing the timestamp one second
36     timeStamp=timeStamp+1

(base) javiergarru@MBP-de-Javier-2 Downloads % python3 AttackSkeleton.py
got key: f2a5ca541af283c48c5d53e50eabb
(base) javiergarru@MBP-de-Javier-2 Downloads %
```

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



```
test.py
KeyGen.py AttackSkeleton.py test.py 1 x
Users > javiergarru > Downloads > test.py > ...
1 import binascii
2 from Crypto.Cipher import AES
3
4
5 iv=binascii.unhexlify('00080706050403020100A2B2C2D2E2F2'.lower())
6 plaintext=binascii.unhexlify('255044462d312e350a25d0d4c5d80a34'.lower())
7 ciphertext='2f15e526b607a73b228648cd206f9c95'.lower()
8
9
10 keyGen = binascii.unhexlify('f22a5ca541af283c48c5d53e530e0abb'.lower())
11 encryptor=AES.new(keyGen, AES.MODE_CBC, iv)
12 testCipherText=encryptor.encrypt(plaintext)
13
14 if testCipherText.hex() == ciphertext:
15     print("Correct key - Encrypting")
16
17 decryptor=AES.new(keyGen, AES.MODE_CBC, iv)
18 testPlainText=decryptor.decrypt(binascii.unhexlify(ciphertext))
19
20
21 print(testPlainText)
22
23 print(plaintext)
24
25 if testPlainText == plaintext:
26     print("Correct key - Decrypting")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

(base) javiergarru@MBP-de-Javier-2 Downloads % python3 test.py
Correct key - Encrypting
b'PDF-1.5\x01\x04\xC5\xD8\n4'
b'PDF-1.5\x01\x04\xC5\xD8\n4'
Correct key - Decrypting
(base) javiergarru@MBP-de-Javier-2 Downloads %

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

The screenshot shows a code editor window with several tabs open. The active tab is 'AttackSkeleton.py'. The code in the editor is as follows:

```
AttackSkeleton.py
Users > javiergarru > Downloads > AttackSkeleton.py > ...
12 initial_time = 2020-09-01 0:40:49
13 timeStamp = time.strptime(initial_time, '%Y-%m-%d %H:%M:%S')
14 timeStamp = time mktime(timeStamp)
15
16
17 # 3600s are equal to the 2-hour window
18 for i in range(3600):
19     random.seed(a=timeStamp)
20     keySize=16
21     keyGen=""
22     # Generating the key in the i-th iteration
23     for j in range(keySize):
24         randomNumber=random.randint(0,255)
25         keyGen=keyGen+'{:02x}'.format(randomNumber)
26
27     key=keyGen.encode('utf-8')
28     # Encrypting the plaintext with the newly generated key
29     encryptor=AES.new(key, AES.MODE_CBC, iv)
30     testCipherText=encryptor.encrypt(plaintext)
31     # Comparing the generated ciphertext with the original ciphertext
32     if(testCipherText.hex()==ciphertext):
33         print("got key:",keyGen)
34         break
35
36     timeStamp=timeStamp+1
37
38 date_time = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(timeStamp))
39
40 print(date_time)
```

Below the code editor, there is a terminal window showing the execution of the script:

```
(base) javiergarru@MBP-de-Javier-2 Downloads % python3 AttackSkeleton.py
got key: f22a5ca41af283c48c5d53e530e0abb
2020-09-01 10:40:00
(base) javiergarru@MBP-de-Javier-2 Downloads %
```

The terminal also shows some background processes in the taskbar.

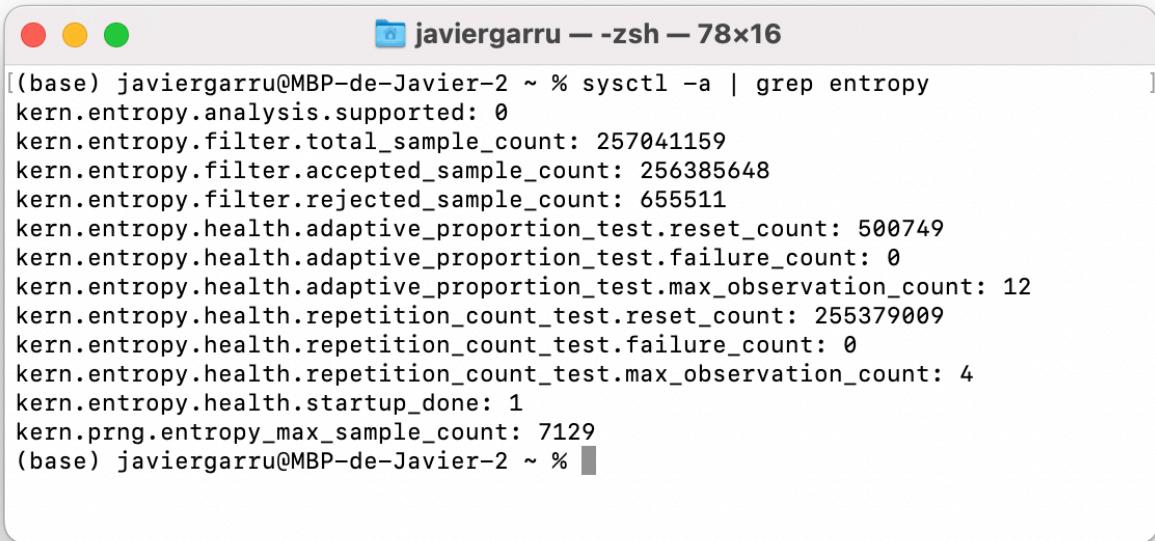
Part Three: Measure the Entropy of Kernel

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
```



```
[base] javiergarru@MBP-de-Javier-2 ~ % sysctl -a | grep entropy
kern.entropy.analysis.supported: 0
kern.entropy.filter.total_sample_count: 257041159
kern.entropy.filter.accepted_sample_count: 256385648
kern.entropy.filter.rejected_sample_count: 655511
kern.entropy.health.adaptive_proportion_test.reset_count: 500749
kern.entropy.health.adaptive_proportion_test.failure_count: 0
kern.entropy.health.adaptive_proportion_test.max_observation_count: 12
kern.entropy.health.repetition_count_test.reset_count: 255379009
kern.entropy.health.repetition_count_test.failure_count: 0
kern.entropy.health.repetition_count_test.max_observation_count: 4
kern.entropy.health.startup_done: 1
kern.prng.entropy_max_sample_count: 7129
(base) javiergarru@MBP-de-Javier-2 ~ %
```

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:

```

javiergarru -- zsh -- 145x51
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
kern.entropy.filter.total_sample_count: 257744314
kern.entropy.filter.total_sample_count: 257744359
kern.entropy.filter.total_sample_count: 257744397
kern.entropy.filter.total_sample_count: 257744431
kern.entropy.filter.total_sample_count: 257744474
kern.entropy.filter.total_sample_count: 257744520
kern.entropy.filter.total_sample_count: 257744554
kern.entropy.filter.total_sample_count: 257744599
kern.entropy.filter.total_sample_count: 257744698
kern.entropy.filter.total_sample_count: 257744743
kern.entropy.filter.total_sample_count: 257744781
kern.entropy.filter.total_sample_count: 257744986
kern.entropy.filter.total_sample_count: 257745042
kern.entropy.filter.total_sample_count: 257745192
kern.entropy.filter.total_sample_count: 257745344
kern.entropy.filter.total_sample_count: 257745524
kern.entropy.filter.total_sample_count: 257745599
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
kern.entropy.filter.total_sample_count: 257746673
kern.entropy.filter.total_sample_count: 257746861
kern.entropy.filter.total_sample_count: 257747039
kern.entropy.filter.total_sample_count: 257747233
kern.entropy.filter.total_sample_count: 257747423
kern.entropy.filter.total_sample_count: 257747600
kern.entropy.filter.total_sample_count: 257747770
kern.entropy.filter.total_sample_count: 257747943
kern.entropy.filter.total_sample_count: 257748119
kern.entropy.filter.total_sample_count: 257748307
kern.entropy.filter.total_sample_count: 257748502
kern.entropy.filter.total_sample_count: 257748710
kern.entropy.filter.total_sample_count: 257748889
kern.entropy.filter.total_sample_count: 257749040
kern.entropy.filter.total_sample_count: 257749257
kern.entropy.filter.total_sample_count: 257749478
kern.entropy.filter.total_sample_count: 257749508
kern.entropy.filter.total_sample_count: 257749555
kern.entropy.filter.total_sample_count: 257749586
kern.entropy.filter.total_sample_count: 257749616
kern.entropy.filter.total_sample_count: 257749661
kern.entropy.filter.total_sample_count: 257749710
kern.entropy.filter.total_sample_count: 257749756
^C
(base) javiergarru@MBP-de-Javier-2 ~ %

```

((((())))

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 time and type in one of them while the other one executes the loop:

```

(jbase) javiergarru@MBP-de-Javier-2 ~ % while true; do sysctl -a | grep kern.entropy.filter.total_sample_count; sleep 0.1;done;
kern.entropy.filter.total_sample_count: 258296882
kern.entropy.filter.total_sample_count: 258296938
kern.entropy.filter.total_sample_count: 258296988
kern.entropy.filter.total_sample_count: 258297089
kern.entropy.filter.total_sample_count: 258297243
kern.entropy.filter.total_sample_count: 258297400
kern.entropy.filter.total_sample_count: 258297558
kern.entropy.filter.total_sample_count: 258297682
kern.entropy.filter.total_sample_count: 258297826
kern.entropy.filter.total_sample_count: 258297902
kern.entropy.filter.total_sample_count: 258297949
kern.entropy.filter.total_sample_count: 258297991
kern.entropy.filter.total_sample_count: 258298027
kern.entropy.filter.total_sample_count: 258298058
kern.entropy.filter.total_sample_count: 258298106
kern.entropy.filter.total_sample_count: 258298139
kern.entropy.filter.total_sample_count: 258298281
kern.entropy.filter.total_sample_count: 258298245
kern.entropy.filter.total_sample_count: 258298288
kern.entropy.filter.total_sample_count: 258298382
kern.entropy.filter.total_sample_count: 258298570
kern.entropy.filter.total_sample_count: 258298736
kern.entropy.filter.total_sample_count: 258298877
kern.entropy.filter.total_sample_count: 258299034
kern.entropy.filter.total_sample_count: 258299165
kern.entropy.filter.total_sample_count: 258299284
kern.entropy.filter.total_sample_count: 258299452
kern.entropy.filter.total_sample_count: 258299598
kern.entropy.filter.total_sample_count: 258299679
kern.entropy.filter.total_sample_count: 258299781
kern.entropy.filter.total_sample_count: 258299921
kern.entropy.filter.total_sample_count: 258300076
kern.entropy.filter.total_sample_count: 258300221
kern.entropy.filter.total_sample_count: 258300298
kern.entropy.filter.total_sample_count: 258300333
kern.entropy.filter.total_sample_count: 258300425
kern.entropy.filter.total_sample_count: 258300579
kern.entropy.filter.total_sample_count: 258300727
kern.entropy.filter.total_sample_count: 258300871
kern.entropy.filter.total_sample_count: 258301016
kern.entropy.filter.total_sample_count: 258301065
kern.entropy.filter.total_sample_count: 258301100
kern.entropy.filter.total_sample_count: 258301158
kern.entropy.filter.total_sample_count: 258301205
kern.entropy.filter.total_sample_count: 258301240
kern.entropy.filter.total_sample_count: 258301276
kern.entropy.filter.total_sample_count: 258301320
kern.entropy.filter.total_sample_count: 258301369
kern.entropy.filter.total_sample_count: 258301415
^C
(base) javiergarru@MBP-de-Javier-2 ~ %

```

```

(jbase) javiergarru@MBP-de-Javier-2 ~ % ndfjbnagerojgbaseojfnerojdvnnewfaojnreojnerawpfoneraoerbnvopjaervnprenbap'jrEBn'o

```

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 rhythm. We modify the original command to:

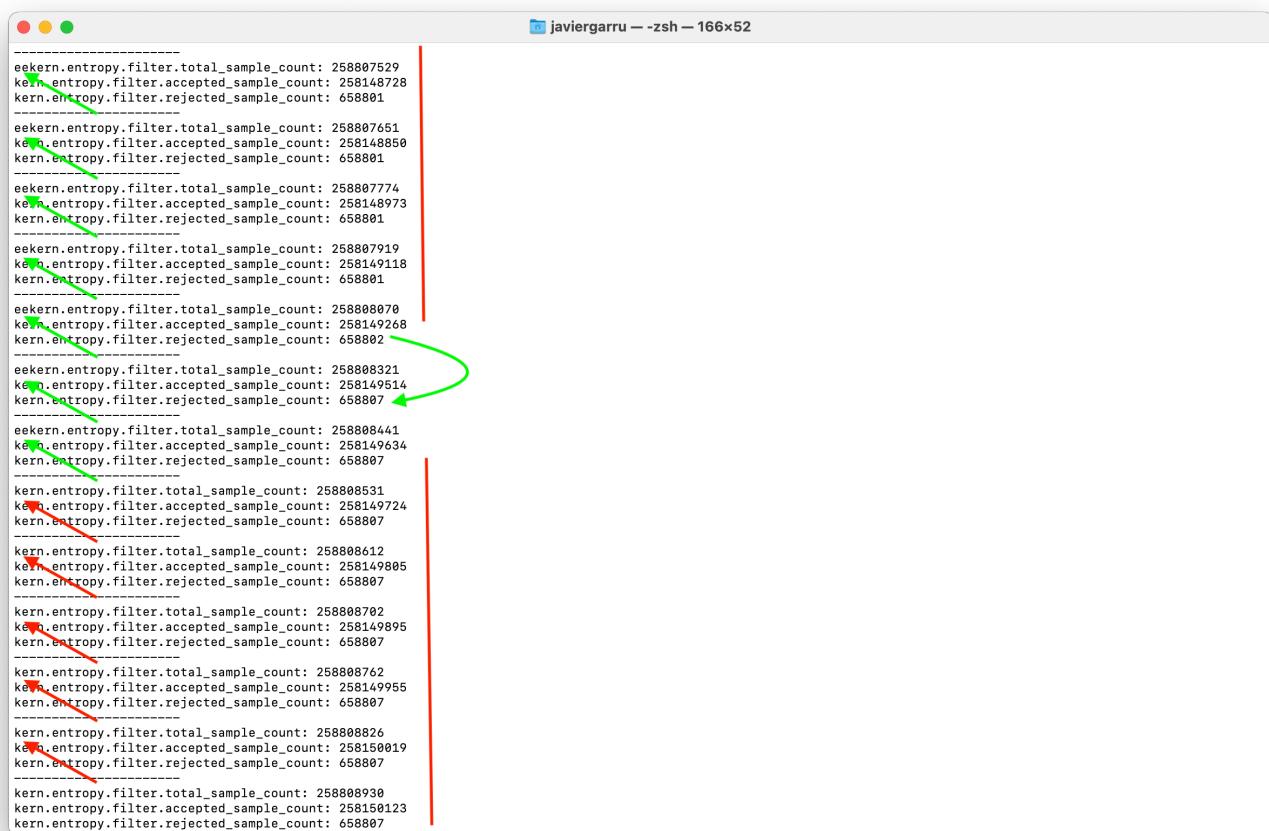
```

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.



```
javiergarru -- zsh -- 166x52
eekern.entropy.filter.total_sample_count: 258807529
kern.entropy.filter.accepted_sample_count: 258148728
kern.entropy.filter.rejected_sample_count: 658801

eekern.entropy.filter.total_sample_count: 258807651
kern.entropy.filter.accepted_sample_count: 258148850
kern.entropy.filter.rejected_sample_count: 658801

eekern.entropy.filter.total_sample_count: 258807774
kern.entropy.filter.accepted_sample_count: 258148973
kern.entropy.filter.rejected_sample_count: 658801

eekern.entropy.filter.total_sample_count: 258807919
kern.entropy.filter.accepted_sample_count: 258149118
kern.entropy.filter.rejected_sample_count: 658801

eekern.entropy.filter.total_sample_count: 258808070
kern.entropy.filter.accepted_sample_count: 258149268
kern.entropy.filter.rejected_sample_count: 658802

eekern.entropy.filter.total_sample_count: 258808321
kern.entropy.filter.accepted_sample_count: 258149514
kern.entropy.filter.rejected_sample_count: 658807

eekern.entropy.filter.total_sample_count: 258808441
kern.entropy.filter.accepted_sample_count: 258149634
kern.entropy.filter.rejected_sample_count: 658807

kern.entropy.filter.total_sample_count: 258808531
kern.entropy.filter.accepted_sample_count: 258149724
kern.entropy.filter.rejected_sample_count: 658807

kern.entropy.filter.total_sample_count: 258808612
kern.entropy.filter.accepted_sample_count: 258149805
kern.entropy.filter.rejected_sample_count: 658807

kern.entropy.filter.total_sample_count: 258808702
kern.entropy.filter.accepted_sample_count: 258149895
kern.entropy.filter.rejected_sample_count: 658807

kern.entropy.filter.total_sample_count: 258808762
kern.entropy.filter.accepted_sample_count: 258149955
kern.entropy.filter.rejected_sample_count: 658807

kern.entropy.filter.total_sample_count: 258808826
kern.entropy.filter.accepted_sample_count: 258150019
kern.entropy.filter.rejected_sample_count: 658807

kern.entropy.filter.total_sample_count: 258808930
kern.entropy.filter.accepted_sample_count: 258150123
kern.entropy.filter.rejected_sample_count: 658807
```

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

```

javiergarru@MBP-de-Javier-2 ~ % while true; do sysctl -a | grep kern.entropy.filter;echo "-----"; sleep 1;done;
kern.entropy.filter.total_sample_count: 258829045
kern.entropy.filter.accepted_sample_count: 258170215
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829157
kern.entropy.filter.accepted_sample_count: 258170327
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829243
kern.entropy.filter.accepted_sample_count: 258170413
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829353
kern.entropy.filter.accepted_sample_count: 258170523
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829424
kern.entropy.filter.accepted_sample_count: 258170594
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829493
kern.entropy.filter.accepted_sample_count: 258170663
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829549
kern.entropy.filter.accepted_sample_count: 258170719
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829624
kern.entropy.filter.accepted_sample_count: 258170794
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829735
kern.entropy.filter.accepted_sample_count: 258170905
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829800
kern.entropy.filter.accepted_sample_count: 258170970
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829866
kern.entropy.filter.accepted_sample_count: 258171036
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258829922
kern.entropy.filter.accepted_sample_count: 258171092
kern.entropy.filter.rejected_sample_count: 658830 -----  

kern.entropy.filter.total_sample_count: 258830025
kern.entropy.filter.accepted_sample_count: 258171195
kern.entropy.filter.rejected_sample_count: 658830 -----  


```

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 rhythmic, 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
```

```

javiergarru@MBP-de-Javier-2 ~ % head -c 16 /dev/random | hexdump
00000000 c081 5e41 bc9d dce7 3b12 651e 985b 7e24
0000010
javiergarru@MBP-de-Javier-2 ~ % head -c 16 /dev/random | hexdump
00000000 1af8 420a f16d 7029 668d 1959 b9d8 ffc1
0000010
javiergarru@MBP-de-Javier-2 ~ % head -c 16 /dev/random | hexdump
00000000 b8c7 06e1 14c0 1b10 effc 927a e27d aa15
0000010
javiergarru@MBP-de-Javier-2 ~ %

```

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 measurements 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:

xabbu

ago. '22

This happened because of changes in Linux 5.18 to RNG's crypto and backports to older LTS Kernels. Basically the `entropy_avail` number is now meaningless. But I don't have any real inside in how it's now working.

The changes can be found here.

https://kernelnewbies.org/Linux_5.18#Security 53

If you search with "linux 5.18 entropy_avail" you might find more information on the internet.

2 Respuestas ▾

✓ Solución



⌚ Haveged is broken no idea for how long ago it broke?

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:

The screenshot shows a terminal window titled "Ubuntu 64-bit Arm Server 17.10 2". The window has a dark purple header bar with standard window controls. The main area is white and contains a terminal session. The session starts with the BusyBox built-in shell prompt (~ #). It then shows four consecutive runs of the command "cat /proc/sys/kernel/random/entropy_avail", which outputs values 297, 312, 325, and 338 respectively. Finally, it shows the command "head -c 16 /dev/random" being run, with the output "S +FH-*~ #".

```
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
S +FH-*~ #
```

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

```
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
dsjbjvsbvbsvosubnsdvosdbojsbos█
```

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:

The screenshot shows a terminal window with a dark purple header bar. The title bar reads "Ubuntu 64-bit Arm Server 17.10 2". Below the title bar is a light gray status bar with icons for volume, battery, signal strength, and a message: "To release your mouse press: Control-⌘". The main terminal area has a white background and displays a BusyBox shell session. The session starts with the prompt "BusyBox v1.22.1 (Ubuntu 1:1.22.0-19ubuntu2) built-in shell (ash)". It then shows several commands being run, each followed by its output count:

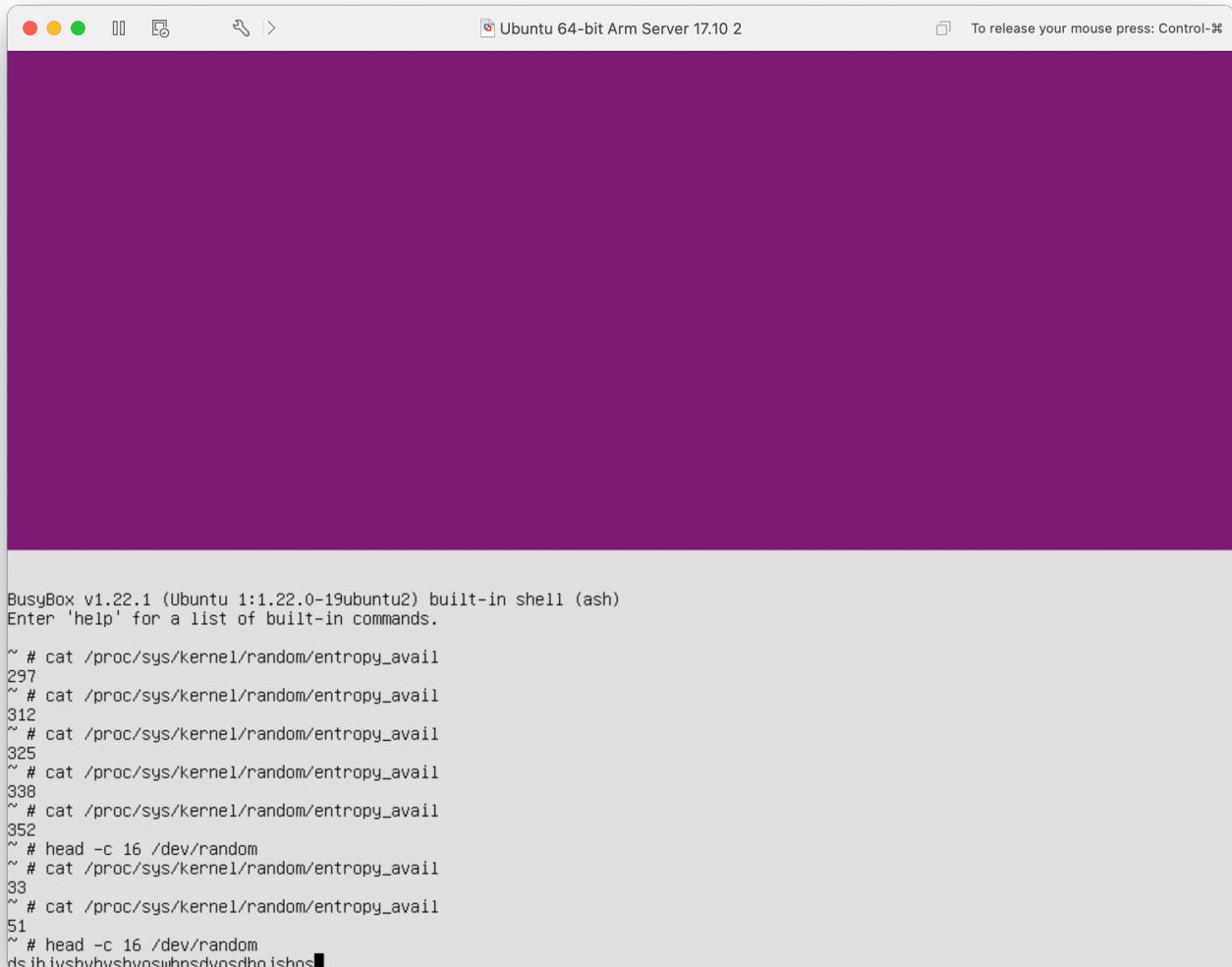
```
BusyBox v1.22.1 (Ubuntu 1:1.22.0-19ubuntu2) built-in shell (ash)
Enter 'help' for a list of built-in commands.

~ # 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
```

The screenshot shows a terminal window with the title "Ubuntu 64-bit Arm Server 17.10 2". The window has a dark purple header bar and a light gray body. The terminal is running a BusyBox shell, indicated by the prompt "BusyBox v1.22.1 (Ubuntu 1:1.22.0-19ubuntu2) built-in shell (ash)". The user has entered several commands to check the entropy available in /dev/random:

```
BusyBox v1.22.1 (Ubuntu 1:1.22.0-19ubuntu2) built-in shell (ash)
Enter 'help' for a list of built-in commands.

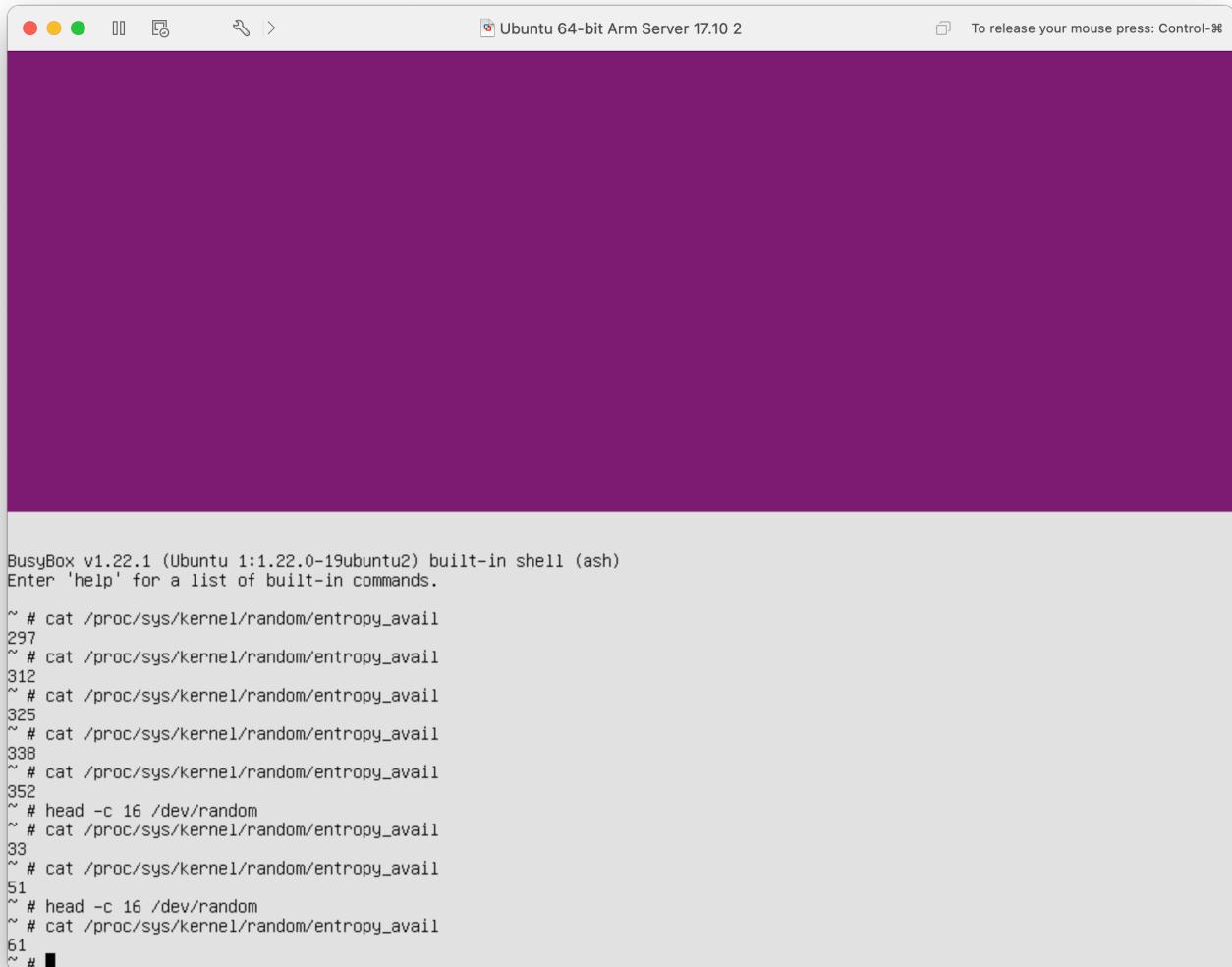
~ # cat /proc/sys/kernel/random/entropy_avail
297
~ # cat /proc/sys/kernel/random/entropy_avail
312
~ # cat /proc/sys/kernel/random/entropy_avail
325
~ # cat /proc/sys/kernel/random/entropy_avail
338
~ # cat /proc/sys/kernel/random/entropy_avail
352
~ # head -c 16 /dev/random
~ # cat /proc/sys/kernel/random/entropy_avail
33
~ # cat /proc/sys/kernel/random/entropy_avail
51
~ # head -c 16 /dev/random
dsjbjvsbvbsvbsub█
```



```
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
dsjbjvsbvbsvsvosubnsdvosdbojsbos
```

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



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.

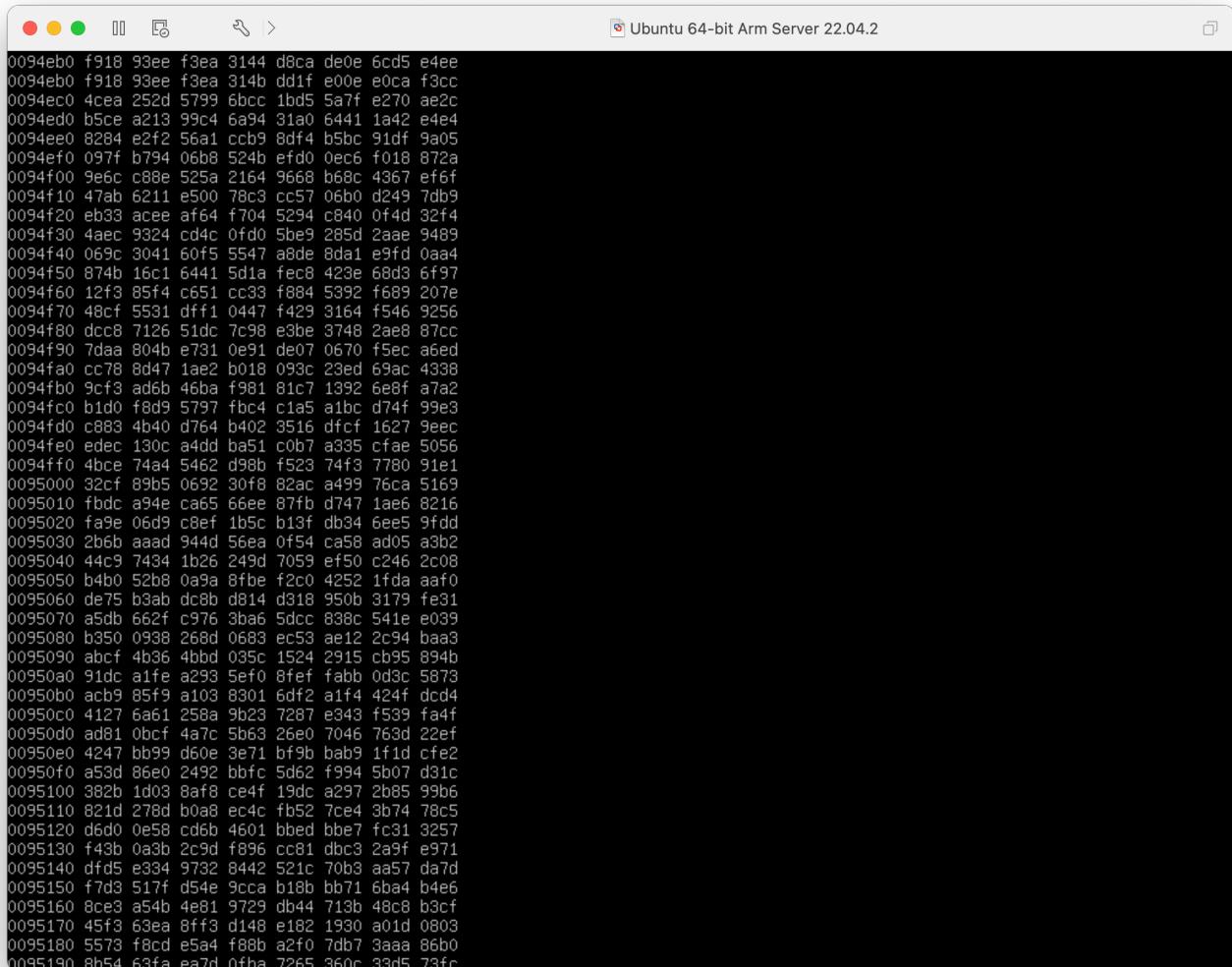
~ # cat /proc/sys/kernel/random/entropy_avail
297
~ # cat /proc/sys/kernel/random/entropy_avail
312
~ # cat /proc/sys/kernel/random/entropy_avail
325
~ # cat /proc/sys/kernel/random/entropy_avail
338
~ # cat /proc/sys/kernel/random/entropy_avail
352
~ # head -c 16 /dev/random
~ # cat /proc/sys/kernel/random/entropy_avail
33
~ # cat /proc/sys/kernel/random/entropy_avail
51
~ # head -c 16 /dev/random
~ # cat /proc/sys/kernel/random/entropy_avail
61
~ #
```

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
```

```
jgarru@myserver:~$ head -c 10M /dev/urandom | hexdump -
```



```
0094eb0 f918 93ee f3ea 3144 d8ca de0e 6cd5 e4ee
0094eb0 f918 93ee f3ea 314b dd1f e00e e0ca f3cc
0094ec0 4cea 252d 5799 6bcc 1bd5 5a7f e270 ae2c
0094edo b5ce a213 99c4 6a94 31a0 6441 1a42 e4e4
0094eed0 8284 e2f2 56a1 ccb9 8df4 b5bc 91df 9a05
0094ef0 097f b794 06b8 524b ef00 0ec6 f018 872a
0094f00 9e6c c88e 525a 2164 9668 b68c 4367 ef6f
0094f10 47ab 6211 e500 78c3 cc57 06b0 d249 7db9
0094f20 eb33 acee af64 f704 5294 c840 0f4d 32f4
0094f30 4aec 9324 cd4c 0fd0 5be9 285d 2aae 9489
0094f40 069c 3041 60f5 5547 a8de 8d1e e9fd 0aa4
0094f50 874b 16c1 6441 5d1a fec8 423e 68d3 6f97
0094f60 12f3 85f4 c651 cc33 f884 5392 f689 207e
0094f70 48cf 5531 dff1 0447 f429 3164 f546 9256
0094f80 dcc8 7126 51dc 7c98 e3be 3748 2ae8 87cc
0094f90 7daa 804b e731 0e91 de07 0670 f5ec a6ed
0094fa0 cc78 8d47 1ae2 b018 093c 23ed 69ac 4338
0094fb0 9cf3 ad6b 46ba f981 81c7 1392 6e8f a7a2
0094fc0 b1d0 f8d9 5797 fbc4 c1a5 a1bc d74f 99e3
0094fd0 c883 4b40 d764 b402 3516 dfcf 1627 9eec
0094fe0 edec 130c a4dd ba51 cob7 a335 cfae 5056
0094ff0 4bce 74a4 5462 d98b f523 74f3 7780 91e1
0095000 32cf 89b5 0692 30f8 82aa a499 76ca 5169
0095010 fbdc a94e ca65 66ee 87fb d747 1ae6 8216
0095020 fa9c 06d9 c8ef 1b5c b13f db34 6ee5 9fdd
0095030 2b6b aaad 944d 56ea 0f54 ca58 ad05 a3b2
0095040 44c9 7434 1b26 249d 7059 ef50 c246 2c08
0095050 b4b0 52b8 0a9a 8fbe f2c0 4252 1fda aaf0
0095060 de75 b3ab dc8b d814 d318 950b 3179 fe31
0095070 a5db 662f c976 3ba6 5dcc 838c 541e e039
0095080 b350 0938 268d 0683 ec53 ae12 2c94 baa3
0095090 abcf 4b36 4bbd 035c 1524 2915 cb95 894b
00950a0 91dc a1fe a293 5ef0 8fef fabb 0d3c 5873
00950b0 acb9 85f9 a103 8301 60f2 a1f4 424f dcd4
00950c0 4127 6a61 258a 9b23 7287 e343 f589 fa4f
00950d0 ad81 0bcf 4a7c 5b63 26e0 7046 763d 22ef
00950e0 4247 bb99 d60e 3e71 bf9b bab9 1f1d cfe2
00950f0 a53d 86e0 2492 bbf5 5d62 f994 5b07 d31c
0095100 382b 1d03 8af8 ce4f 19dc a297 2b85 99b6
0095110 821d 278d b0a8 ec4c fb52 7ce4 3b74 78c5
0095120 d6d0 0e58 cd6b 4601 bbed bbe7 fc31 3257
0095130 f43b 0a3b 2c9d f896 cc81 dbc3 2a9f e971
0095140 dfdf5 e334 9732 8442 521c 70b3 aa57 da7d
0095150 f7d3 517f d54e 9cca b18b bb71 6ba4 b4e6
0095160 8ce3 a54b 4e81 9729 db44 713b 48c8 b3cf
0095170 45f3 63ea 8ff3 d148 e182 1930 a01d 0803
0095180 5573 f8cd e5a4 f88b a2f0 7db7 3aaa 86b0
0095190 8b54 63fa ea7d 0fba 7265 360c 33d5 73fc
```

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:

```
Ubuntu 64-bit Arm Server 22.04.2
ent(1)                                         ent(1)

NAME
    ent - pseudorandom number sequence test

SYNOPSIS
    ent [options] [file]

DESCRIPTION
    ENT Logo

    ent performs a variety of tests on the stream of bytes in file (or standard input if no file is specified) and produces output on standard output; for example:

    Entropy = 7.980627 bits per character.

    Optimum compression would reduce the size
    of this 51768 character file by 0 percent.

    Chi square distribution for 51768 samples is 1542.26, and randomly
    would exceed this value 0.01 percent of the times.

    Arithmetic mean value of data bytes is 125.93 (127.5 = random).
    Monte Carlo value for Pi is 3.169834647 (error 0.90 percent).
    Serial correlation coefficient is 0.004249 (totally uncorrelated = 0.0).

    The values calculated are as follows:

ENTROPY
    The information density of the contents of the file, expressed as a number of bits per character. The results above,
    which resulted from processing an image file compressed with JPEG, indicate that the file is extremely dense in in-
    formation—essentially random. Hence, compression of the file is unlikely to reduce its size. By contrast, the C
    source code of the program has entropy of about 4.9 bits per character, indicating that optimal compression of the
    file would reduce its size by 38%. [Hamming, pp. 104-108]

CHI-SQUARE TEST
    The chi-square test is the most commonly used test for the randomness of data, and is extremely sensitive to errors
    in pseudorandom sequence generators. The chi-square distribution is calculated for the stream of bytes in the file
    and expressed as an absolute number and a percentage which indicates how frequently a truly random sequence would ex-
    ceed the value calculated. We interpret the percentage as the degree to which the sequence tested is suspected of being
    non-random. If the percentage is greater than 99% or less than 1%, the sequence is almost certainly not random.
    If the percentage is between 99% and 95% or between 1% and 5%, the sequence is suspect. Percentages between 90% and
    95% and 5% and 10% indicate the sequence is "almost suspect". Note that our JPEG file, while very dense in informa-
    tion, is far from random as revealed by the chi-square test.

    Applying this test to the output of various pseudorandom sequence generators is interesting. The low-order 8 bits re-
    turned by the standard Unix rand(1) function, for example, yields:
Manual page ent(1) line 1 (press h for help or q to quit)
```

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
```

```
jgarru@myserver:~$ head -c 1M /dev/urandom > output.bin
jgarru@myserver:~$ ent output.bin
Entropy = 7.999860 bits per byte.

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

Chi square distribution for 1048576 samples is 203.51, and randomly
would exceed this value 99.23 percent of the times.

Arithmetic mean value of data bytes is 127.4767 (127.5 = random).
Monte Carlo value for Pi is 3.139172131 (error 0.08 percent).
Serial correlation coefficient is 0.000218 (totally uncorrelated = 0.0).
jgarru@myserver:~$ _
```

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 distribution. 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.