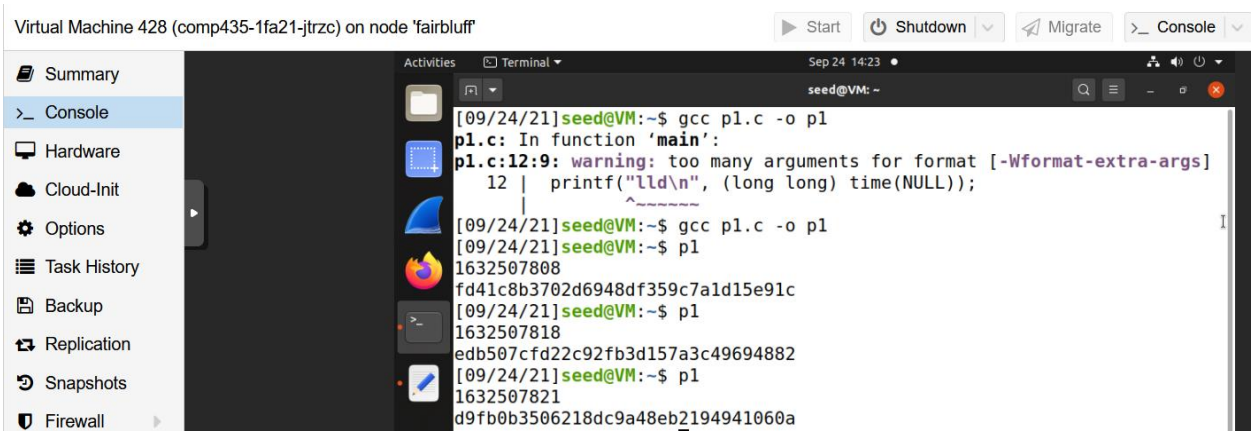
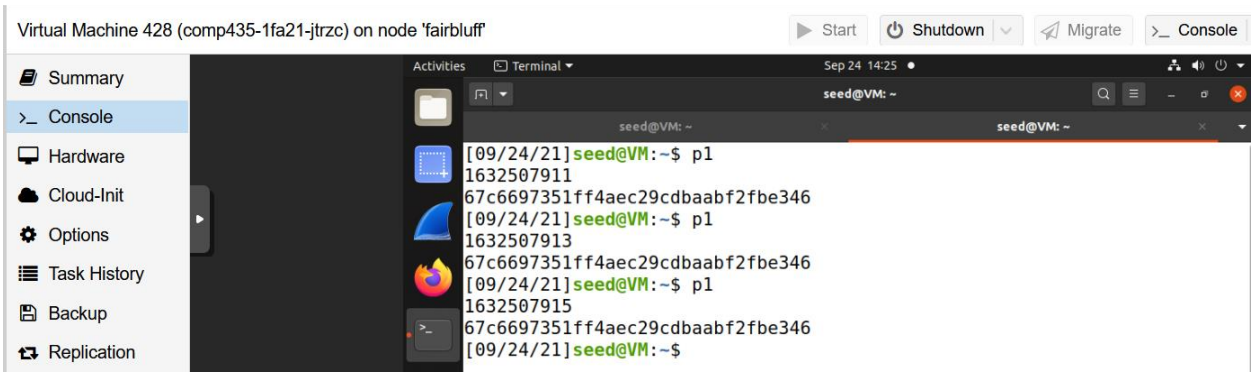
Lab 1 Report
Comp 435
Joshua Trzcienski

Task 1: Generate Encryption Key in a Wrong Way

The purpose of Task 1 is to show how not all random number generation is equivalent. If it is done in a way that can be easily traced backwards, it becomes much easier to decrypt any encryption compared to a method that is more random. Below is the outcome of running the first program with srand(time(NULL)) not being commented out.



Whenever the random function is called, there are two different objects displayed. The first shows the time function value, and the second is the key that is created. In this, each key is very different, but the times are very close together, as they were called one after another quickly.



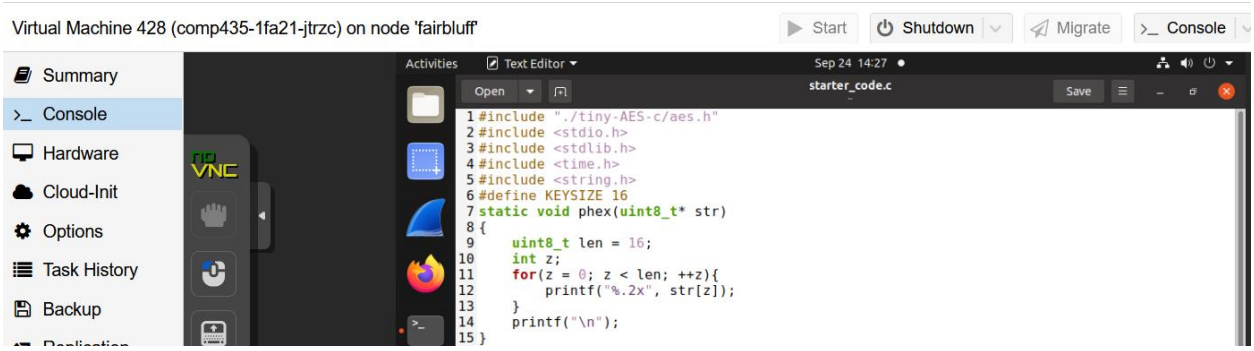
In the above screenshot, this shows what happens when srand(time(NULL)) is commented out. While the time still changes as it should, the keys are exactly the same. In the previous example, the keys were all completely different, but now they are the exact same.

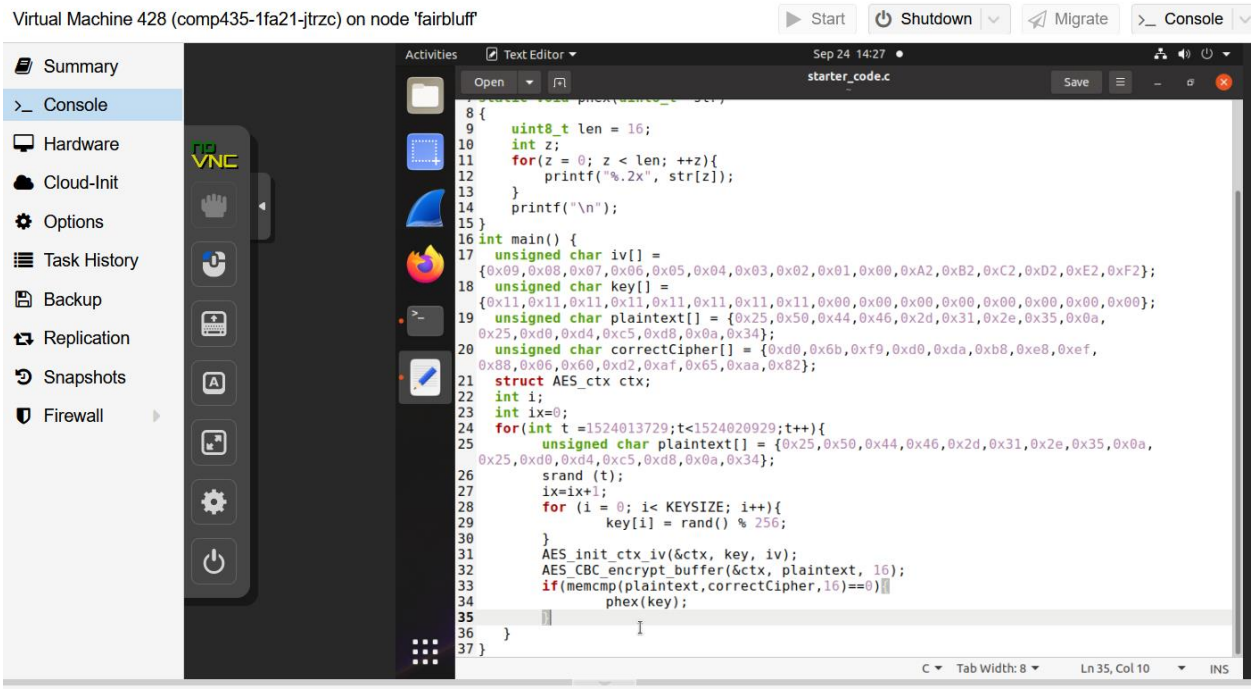
These examples show that what srand() does is essentially prime the rand function to work correctly. If it is not seeded, you are just going to get the same key every time, so there is no randomness at all. Srand() allows the rand() function to generate pseudo random numbers, and without it, using rand() like this would not work to generate unique keys. Time() is taking the amount of seconds since January 1st, 1970, therefore every time this program is used to generate a key, the keys should

change because the time would be different, and so the seed is constantly changing every time. While this may work, as it will be seen in Task 2, there are some pitfalls to this method.

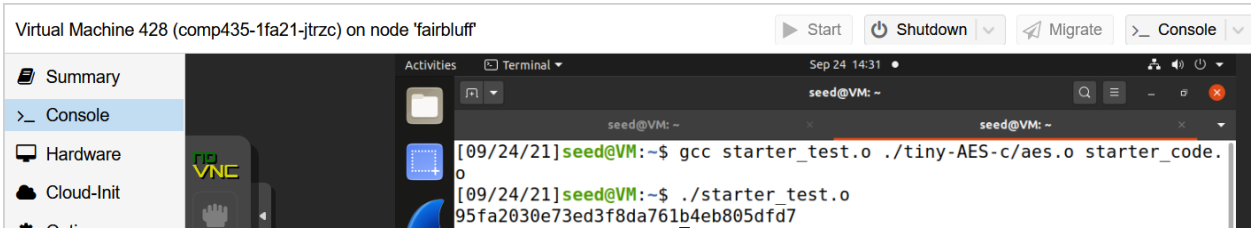
Task 2: Guessing the Key

In Task 1, keys were generated using time() to seed the random number generator to generate the keys. This does work, but it creates some dangers if someone who was trying to decrypt the encryption can figure out the time when it was done, and can figure out some of the plaintext and initial vector. In this example, one block of plaintext, ciphertext, and IV was able to figured out, and the task was to figure out the key to decrypt the rest of the document. Due to the fact that time() was used to seed the random number generator, and there is a two hour window in which Alice probably used the encryption, you can work backwards to figure out the key.





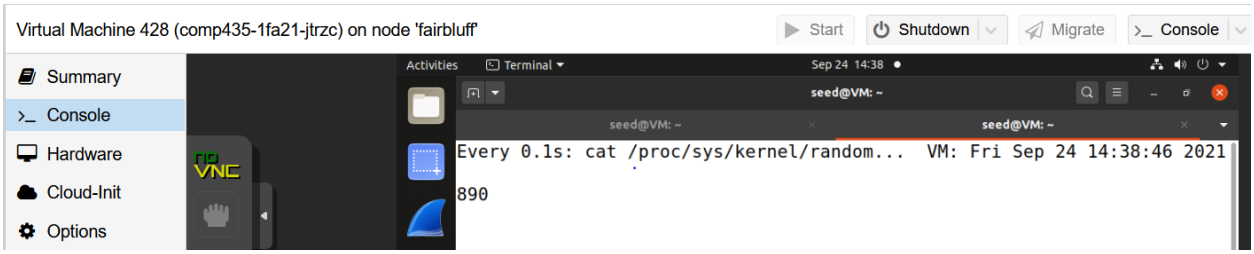
Above is the code used to figure out the key based off of the parameters. Lines 17 through 20 is just filling in the data of the plaintext, and empty key slot, the IV, and the correct ciphertext. In order to figure out the key, first the time interval needs to be determined for the for loop in line 24. Using the date command, the time interval was determined to be from 1524013729 and 1524020929. By looping through this, every single key possible can be generated based off of this period. In order to figure out which key is the correct key, all that needs to be done is take the plaintext and encrypt it with each generated key, giving a ciphertext. If the generated ciphertext is the same as the given ciphertext that was figured out, then that key that was used to encrypt is the correct key. The for loop on line 24 is doing exactly this, and is first generating a key, then using this generated key with the plaintext to create the ciphertext in lines 31 and 32. On line 33, it is checking if the generated ciphertext is equivalent to the real ciphertext, and if it is, it will print out the key used. Below is the result.



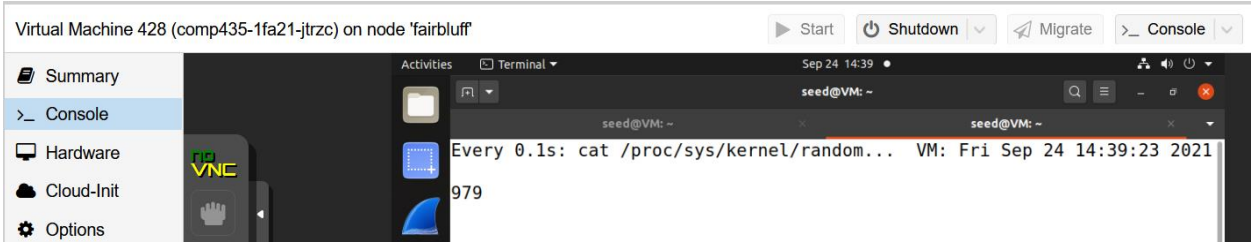
Once the program is run, it runs through everything, and on the last line, display the key that Alice used to encrypt the document.

Key: 95fa2030e73ed3f8da761b4eb805dfd7

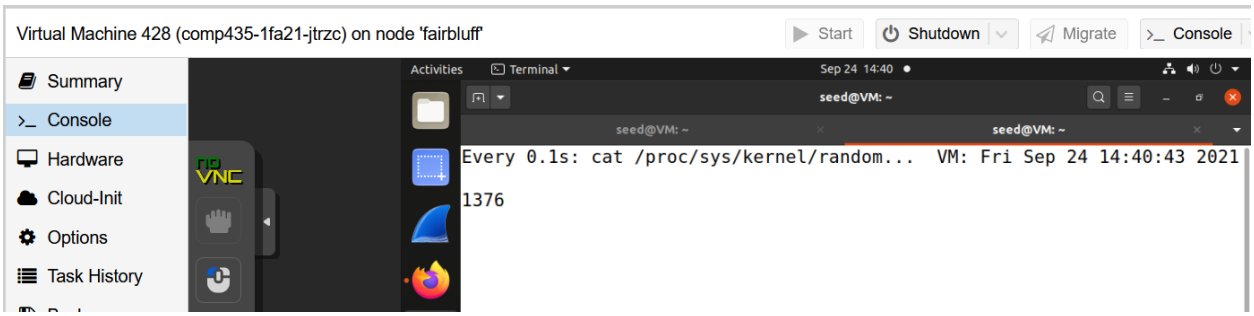
Task 3: Measure the Entropy of Kernel



When the watch command was first run, the entropy output was 890.



After moving the cursor around the screen a couple of times, the entropy was slowly going up. Each movement only added a decent amount to the total entropy, as it increased around 100. Each button click on the mouse, or any button on the keyboard also increased the entropy by 1 each time.

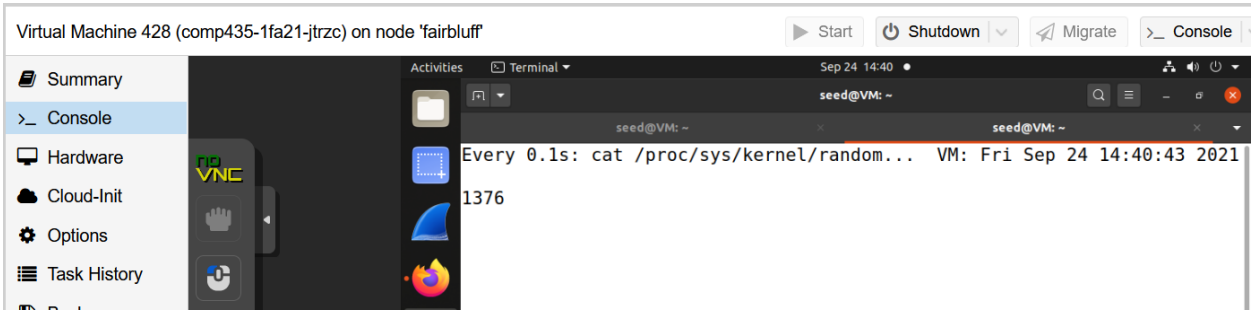


When testing what opening a browser, between the time the browser was clicked to have it open, and it loading, the entropy had increased by around 25, and by the time everything had loaded it had increased even more to around 100, but it seems like most of this was likely just moving the mouse around.

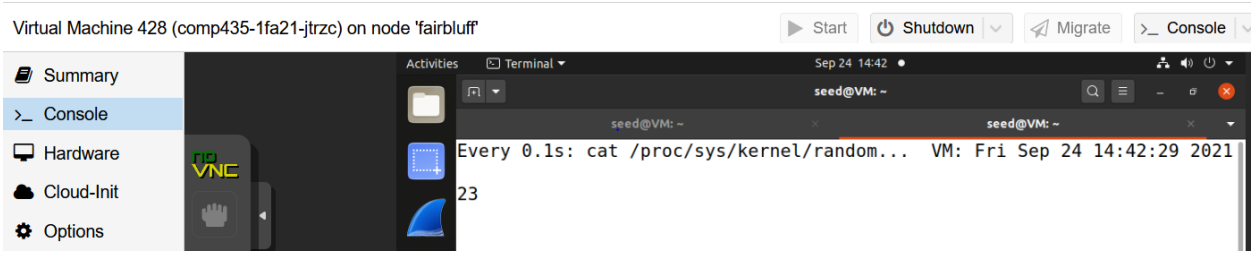
A video from YouTube was opened and played in the background, and while this was happening, the entropy was increasing relatively fast compared to some of the other ways to increase it like button clicks. As long as the video was playing in was increase a couple times per second. It seems like more complex process like playing the video on YouTube increased the entropy much more compared to any sort of button click. The mouse movement did also add a fair amount as well especially compared to the button clicks, but not quite as much as playing a video.

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

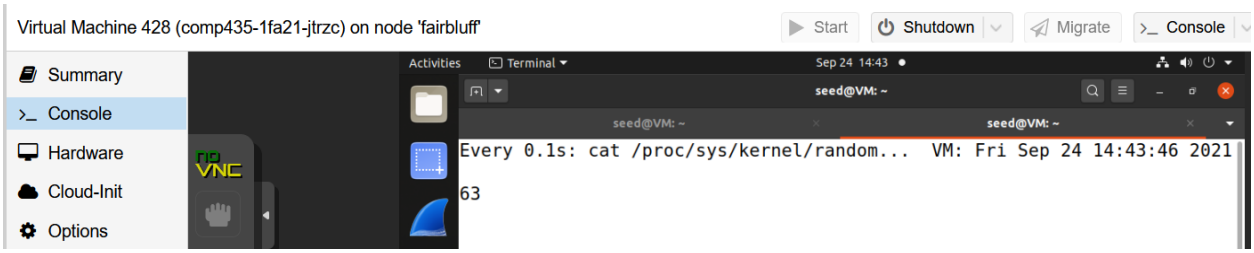
Below is the baseline entropy before hexdump is used

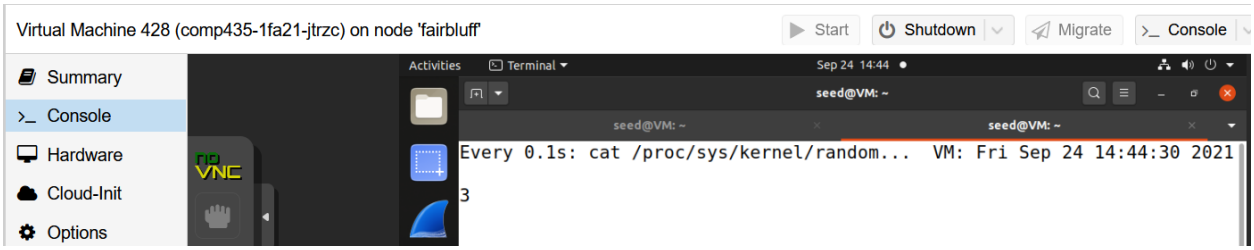


The entropy is 1376, but as soon as hexdump is called, it is reset all the way back down to 0.



The number here shows 23 due to it increasing with mouse movement, but once it was first called the number was reset down to 0.



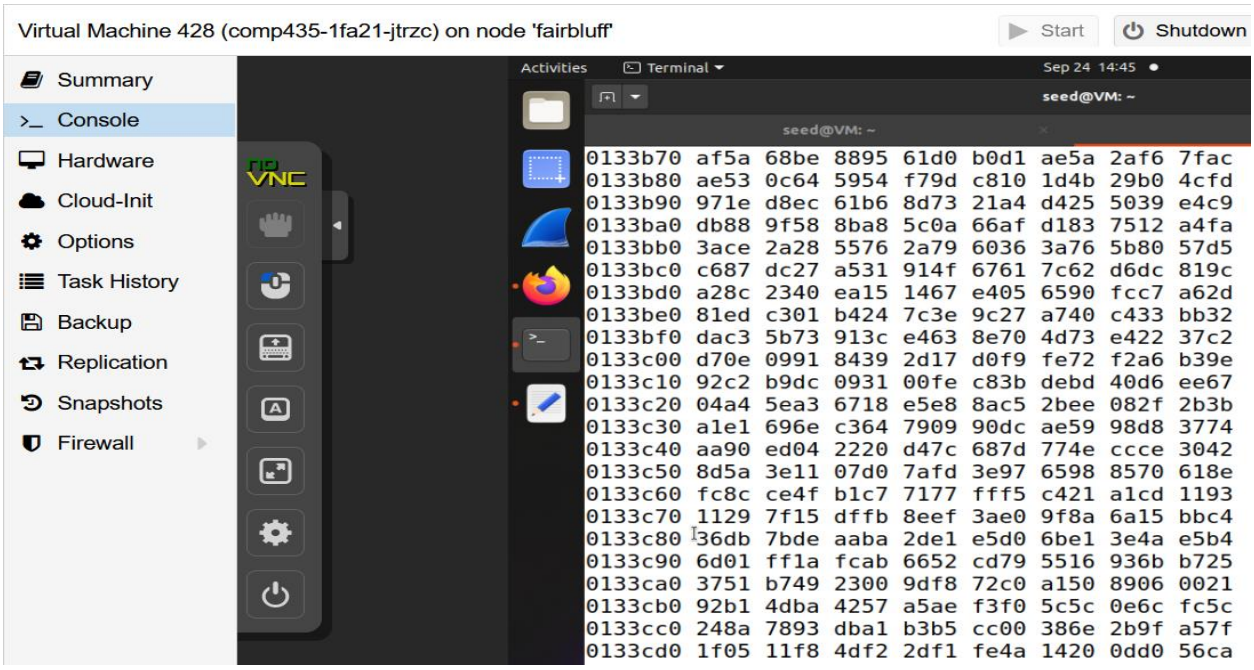


What is different from before is now, once it gets to 63, instead of going higher to 64 and beyond, it is reset back to 0. Now it is just in a constant cycle between 0 and 63.

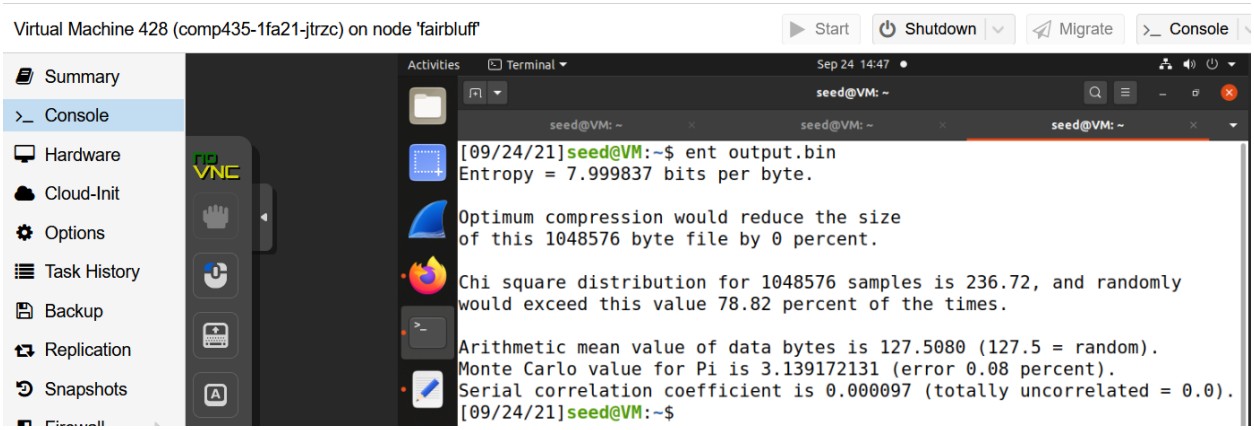
DOS Attack: One of the downfalls of /dev/random is that if there is no entropy, it will block output. Each output also decreases entropy as well. So one potential way would be to in someway make it so the server could not generate entropy, because then it could not run /dev/random, and you would be denied access.

Task 5: Get Random Numbers from /dev/urandom

After runnning cat /dev/urandom | hexdump, the output was a very fast list of random numbers being printed. Moving the mouse seemed to have no effect at all, and a sample of the output can be seen below.

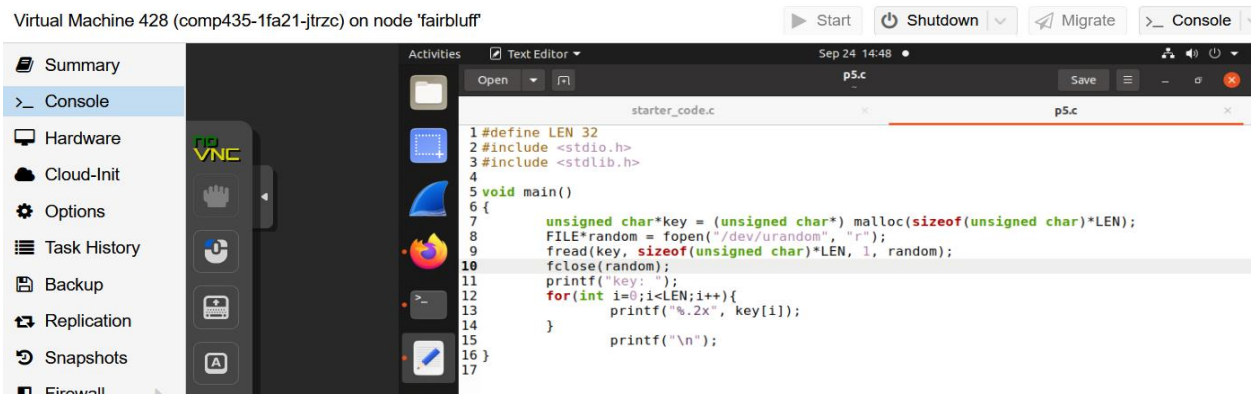


Next, these random numbers will be stored in a file where it will be analyzed using ent. Below is the summary given.



The two statistics that are relevant here are the arithmetic mean value and the serial correlation. The arithmetic mean value was 127.5080, where 127.5 is random, meaning that the data set was pretty much as close to random as you can get according to this statistic. The next statistic is the serial correlation coefficient and that was .000097, where 0 is completely unrelated, which this statistic also supports the above one, as this is saying this data set is very random as well. According to these statistics, these generated numbers are pretty good random numbers.

The last step is to generate a 256-bit key using /dev/urandom. Below is a screenshot of the code used.



First an array called key is created based off of the size of the key that is defined, and in this case LEN=32. The file random uses /dev/urandom like the above examples, and reads the input to put into the key. After this the key is then printed to the terminal as seen in the examples below.