Eric Samuel Huddleston

Matthew Frierson

Group 14

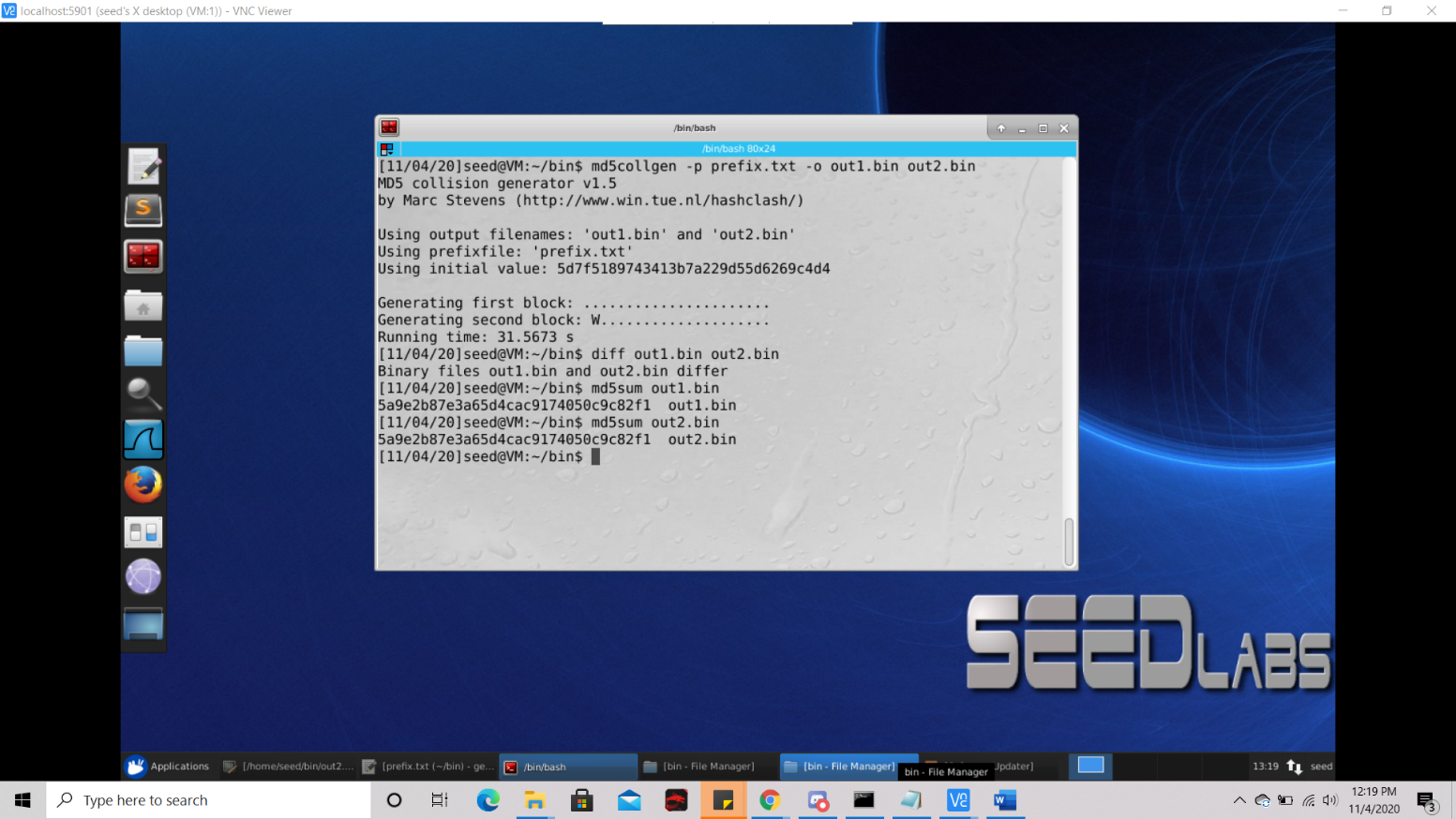
11/5/2020

Section 001

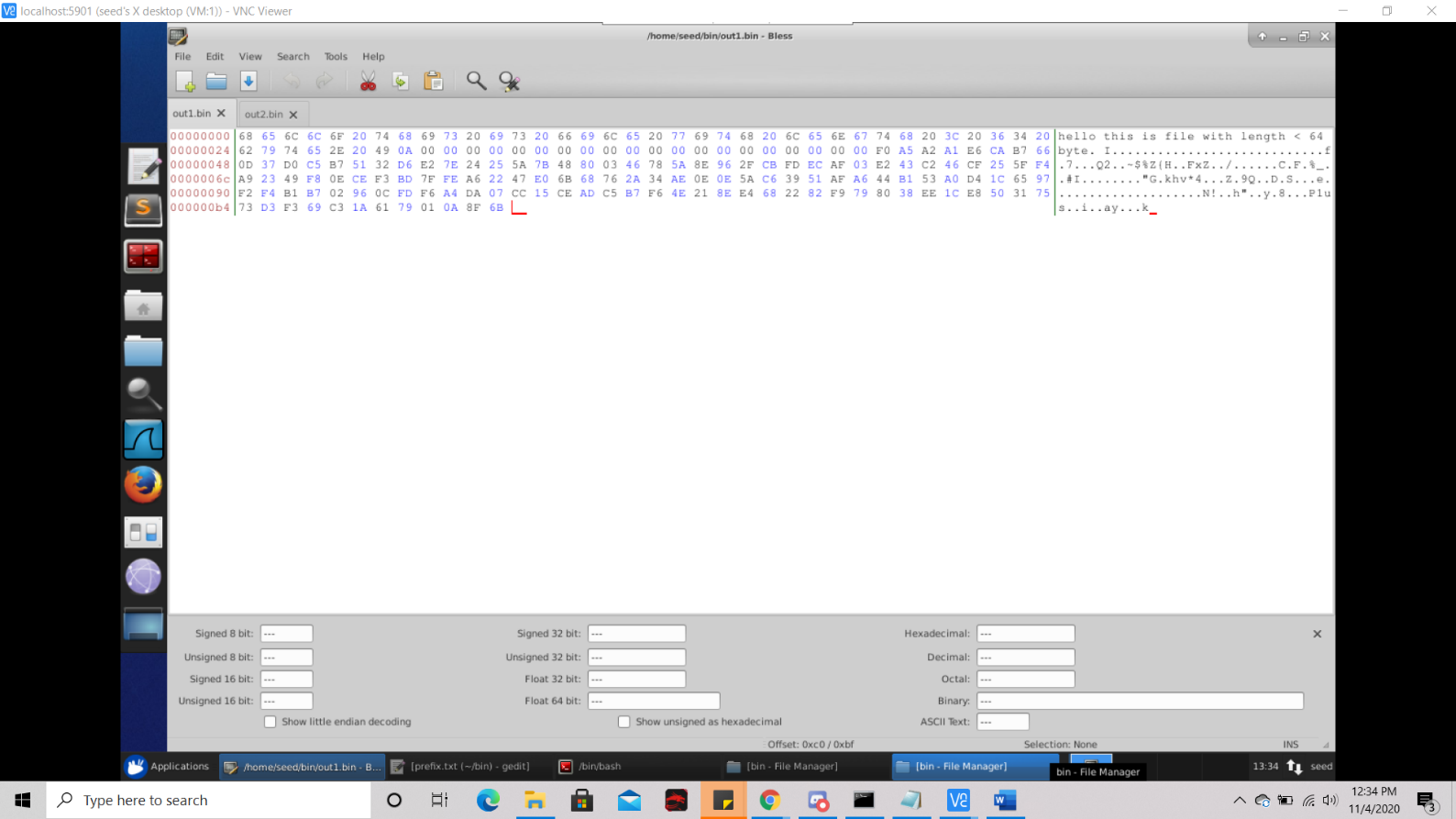
**Task 1:: Generating Two Different Files with the Same MD5 Hash**

In this lab we can use Bless to read and modify the binaries, and we can also use python to pass values inside of the prefix file. For example we could do the command to add 40 A’s to the prefix file ‘echo $(python -c ‘print(“\x41”\*55)’) > prefix.txt’.

Using the tool to create arbitrary prefix file in order to generate two files out1.bin and out2.bin which will have same MD5 hash, and checking if the two output files are the same since the tool generates components Q and P for given prefix text such that hash(prefix + Q) = hash(prefix + P):



Using bless and to view the binary files:



Q1: With prefix file of size 44 bytes we yield out files over twice as large with size 192 bytes. We use the md5collgen tool to yield these output files. We can use ‘diff’ command to verify that these feels are different. The files are then opened in Bless hex editor. Upon looking into the hex editor, we see that all of the bytes of prefix.txt which are short of multiple of 64 are padded with (0A)(00)\* regular expression.

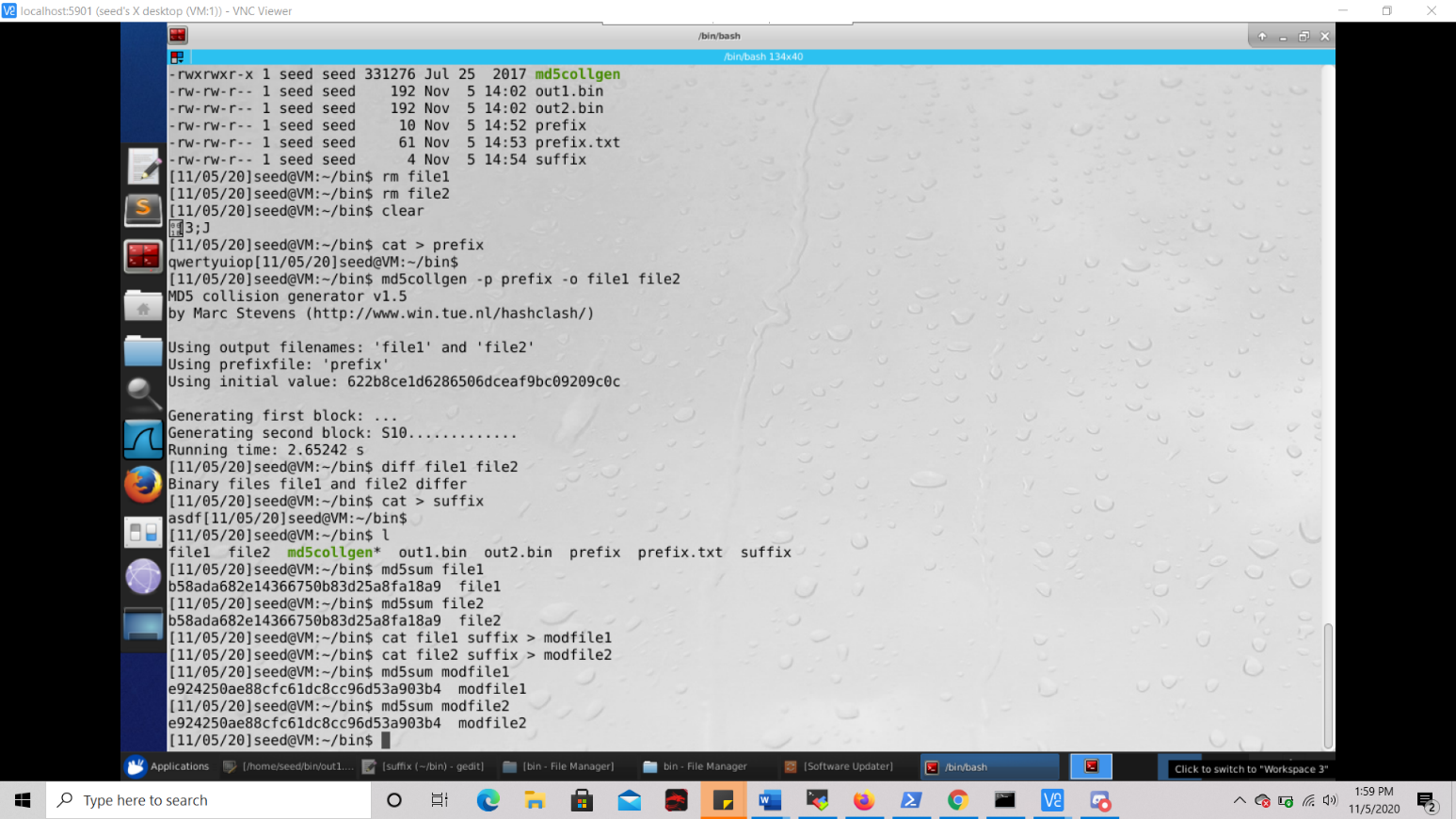
Q2: If the length of prefix. txt is exactly 64 bytes long, it will generate the two output files in 3 seconds less time (28 seconds vs 31 seconds) the tool will still pad the prefix but instead pads the next 64 bytes, specifically from byte offset 40 to byte offset 7F. This is also true for prefix file with a length as a multiple of 64, and thus for any length 64 \* l, the md5collgen will add 64 bytes of passing after it.

Q3: If we issue this command and pipe to the prefix file as so ‘echo $(python -c ‘print(“\x41”\*60)’) > prefix.txt’, it will add 4 bytes of padding as well as data that may differ between the two files.

Additionally we can compare the sha256 sums to help illustrate the difference, for example ‘sha256sum out1.bin and sha256sum out2.bin’, or we can use the ‘xxd’ command but this isn’t necessary since we have the Bless hex editor at out disposal.

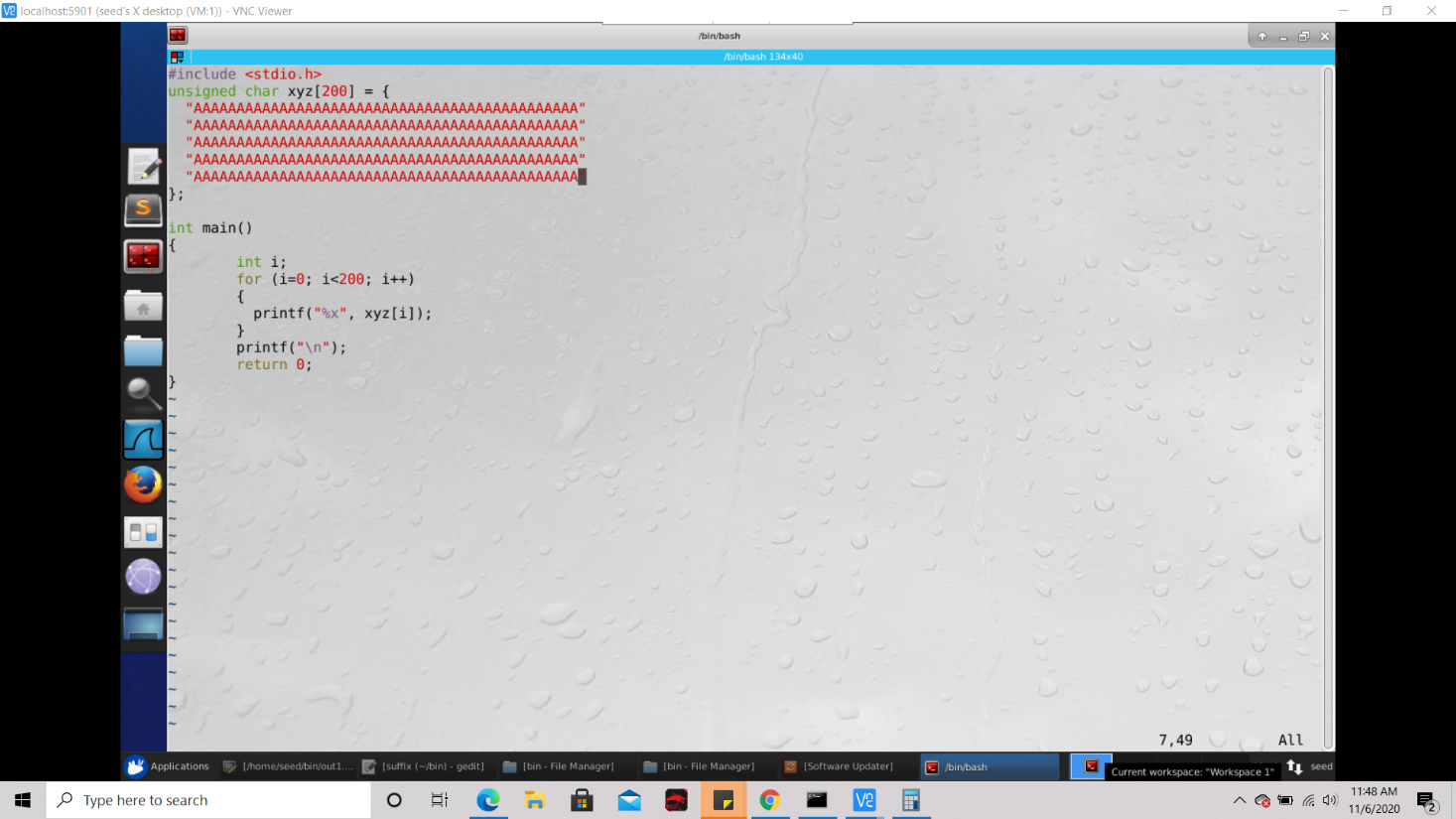
**Task 2:: Understanding MD5’s property**

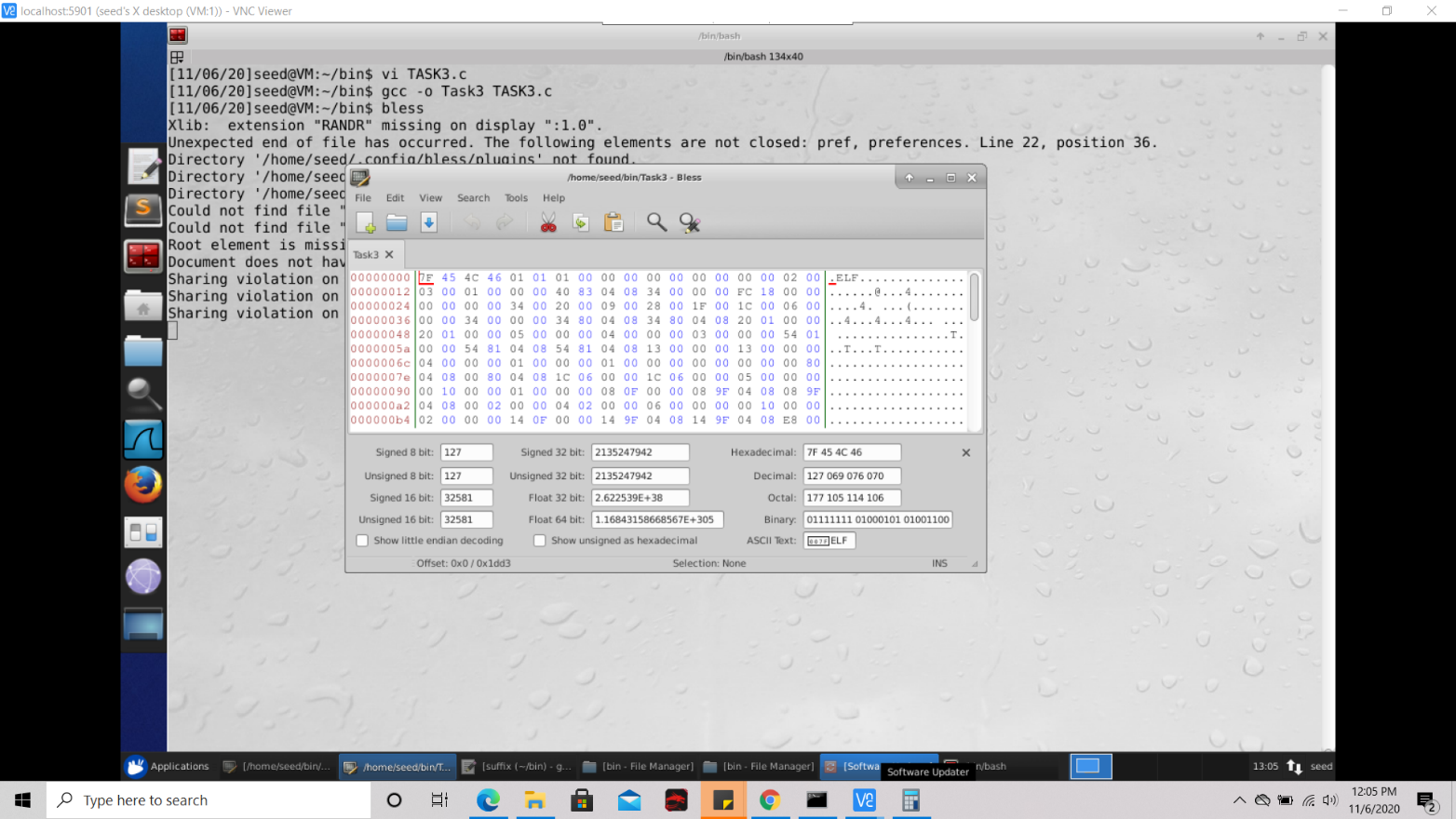
At the higher level, MD5 divides it’s data into 64 bytes blocks then computes the hash in an iterative manner on these blocks. At the core of MD5 is a sort of compression function which yields a 128 bit intermediate hash value. The input for the first of these iterations (IHV0) is fixed. Based on how the MD5 algorithm operates, we can derive a specific property, that given two inputs N and M, if MD5(N) = MD5(M), then for any given input T, MD5(N || T) = MD5(M || T); ∴ adding on a suffix for any two distinct messages which have the same MD5 hash, yields two new longer messages resultant of the concatenation of both the original and the suffix messages, both of each having the same MD5 hash. In order to demonstrate this, we use ‘cat’ command in order to concatenate the content of these files together as follows:

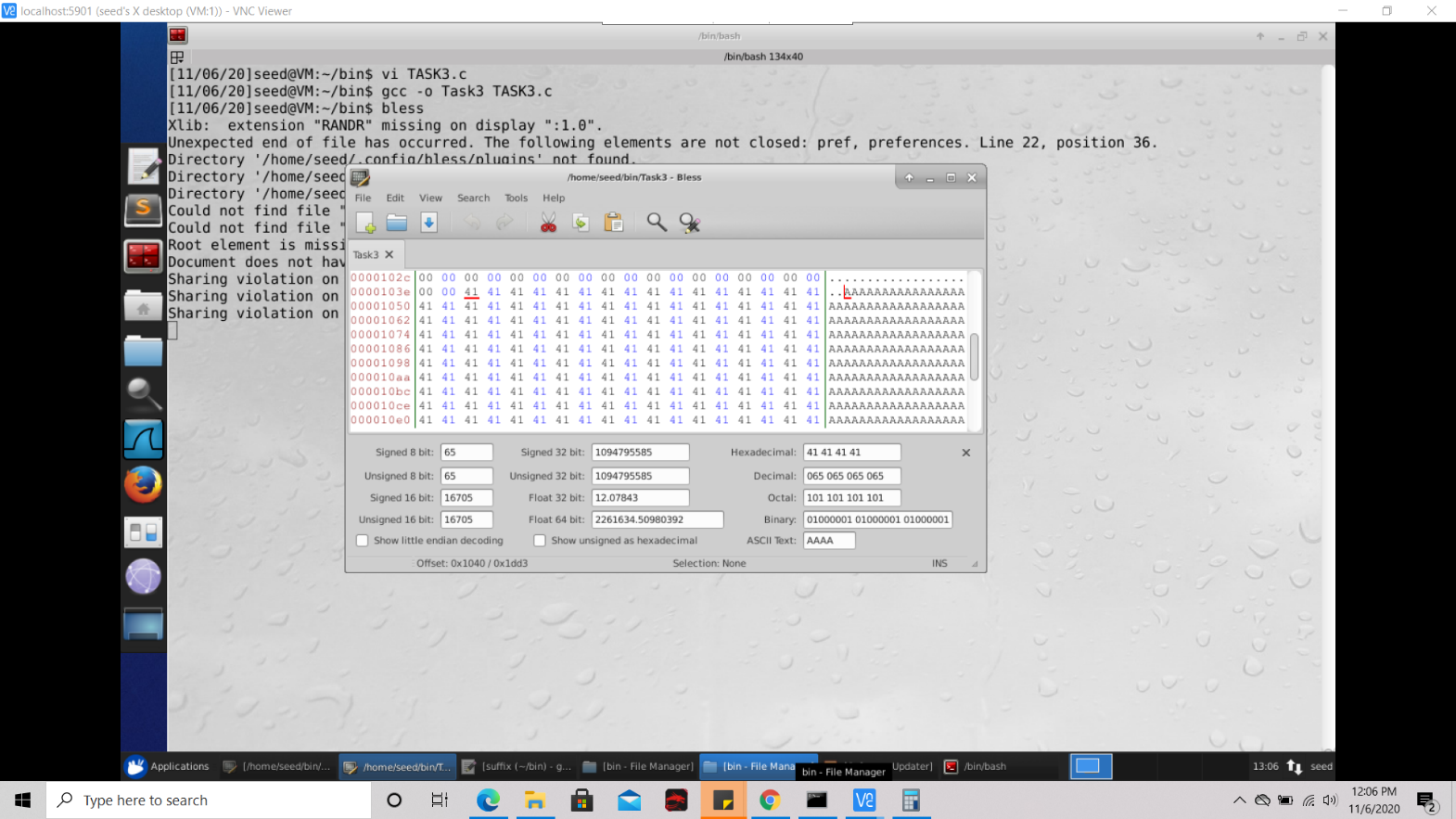


**Task 3: Generating Two Executable Files with the Same MD5 Hash**

Using the given code in C, we are asked to create two different versions of this code such that the difference in them lies in the array contents, but the hash values which correspond to the executables are both the same. As since in the first screenshot below we initially just fill the array contents with just A’s. This makes it rather easy to spot were the array is in the executable after compilation, as seen in the third screenshot.







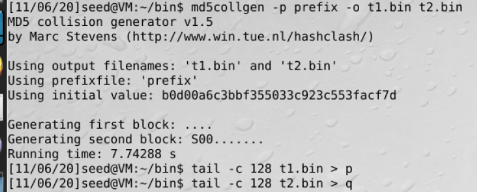
After compiling the program with gcc (screenshot 2) and opening the executable with Bless (screenshot2), we see the byte offset of continuous block of ‘A’s is 1040, or 4160. Now the executable as whole can be divvied up into 3 components. Firstly is the byte offset 0 to x, x to y, and y to the end. The component y to the end will be treated as a suffix, and the component 0 to x will be treated as the prefix. The component x to y is the where the change is necessary, or the variant such that - MD5(prefix || variant1 || suffix) = MD5(prefix || variant2 || suffix). We simply keep the prefix over the byte offset of the first A as well as multiple of 54, and this by choosing byte offset 4160, evrerything of the first 4160 bytes will be the prefix:



Thus the last 4289, prefix || variant2 || suffix, bytes will go into the suffix:



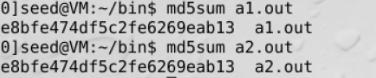
Using the prefix file for md5collgen, we get two files having the same hash called t1.bin and t2.bin; this gives the files ending with byte offset as , therefore we must keep the bytes after from the original binary as the suffix:



Now in order to create the resulting two binaries we concatenate the suffix to both original files:

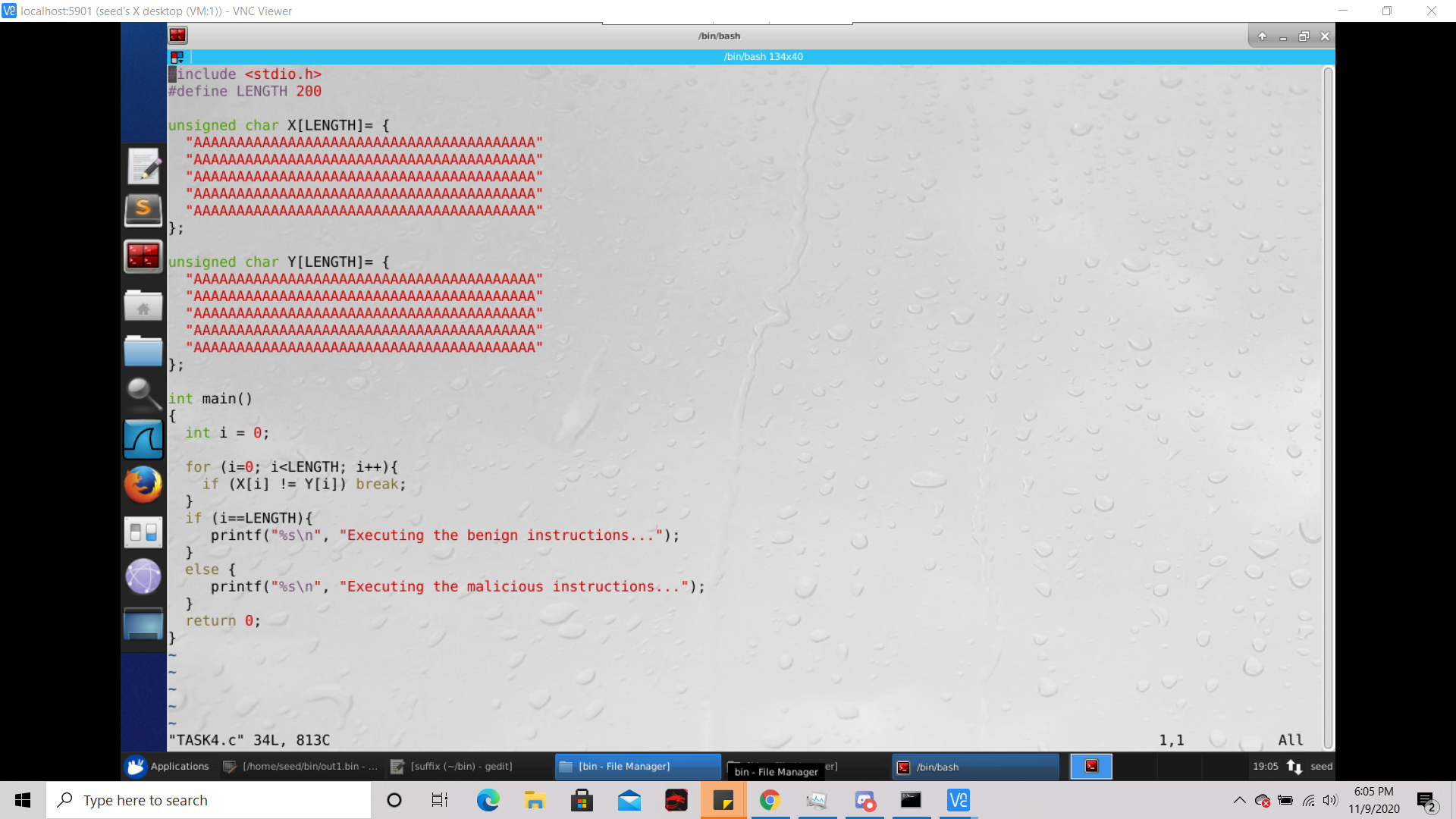


The code files can be turned into executables, which once executed will print out different data. Despite this the md5sums of both of the code files are the same. We can make sure that the data outputted by the code files is different by computing a hash on the files; if the hashes happen to be different then the two files in fact printed different data.

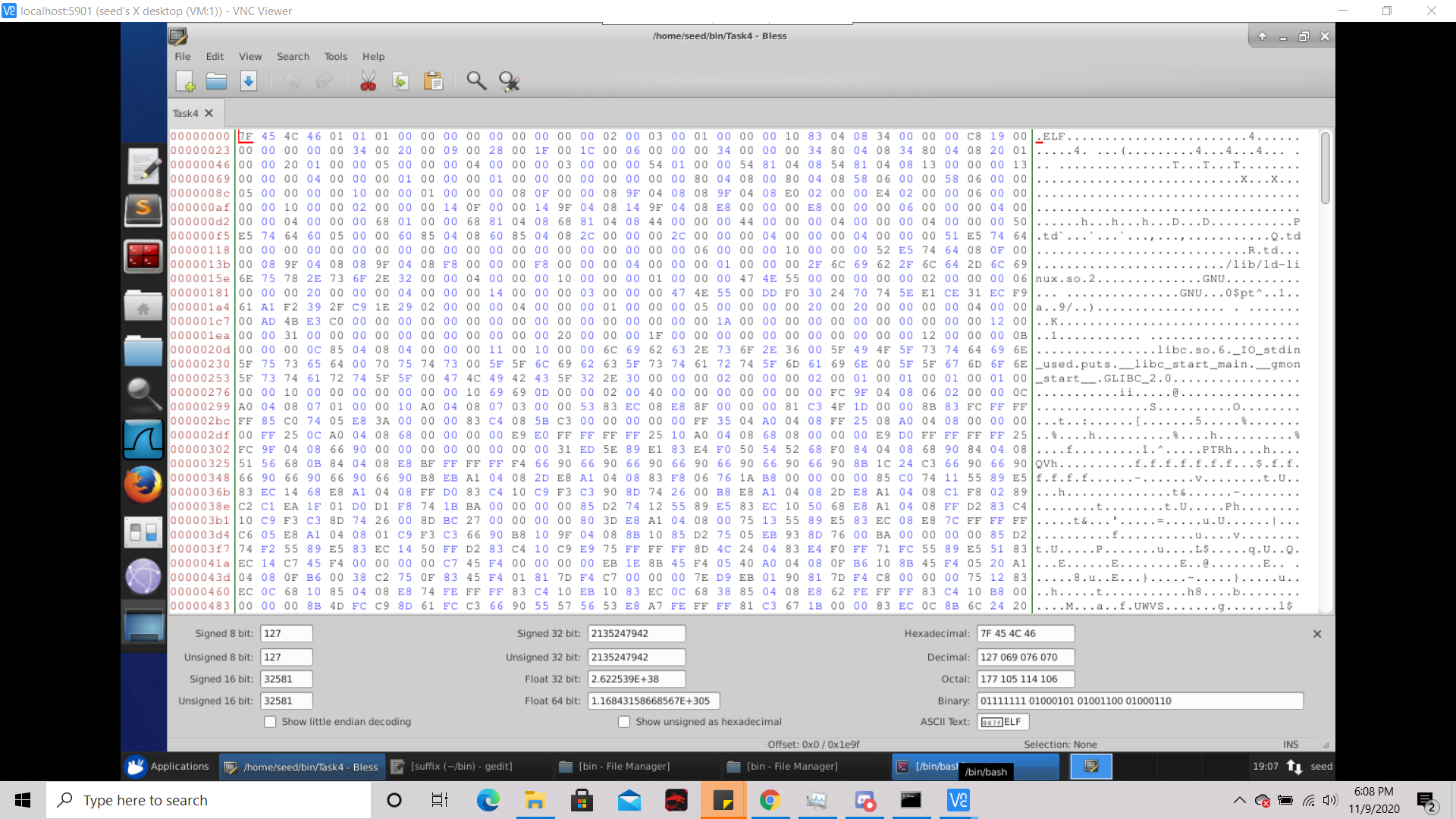


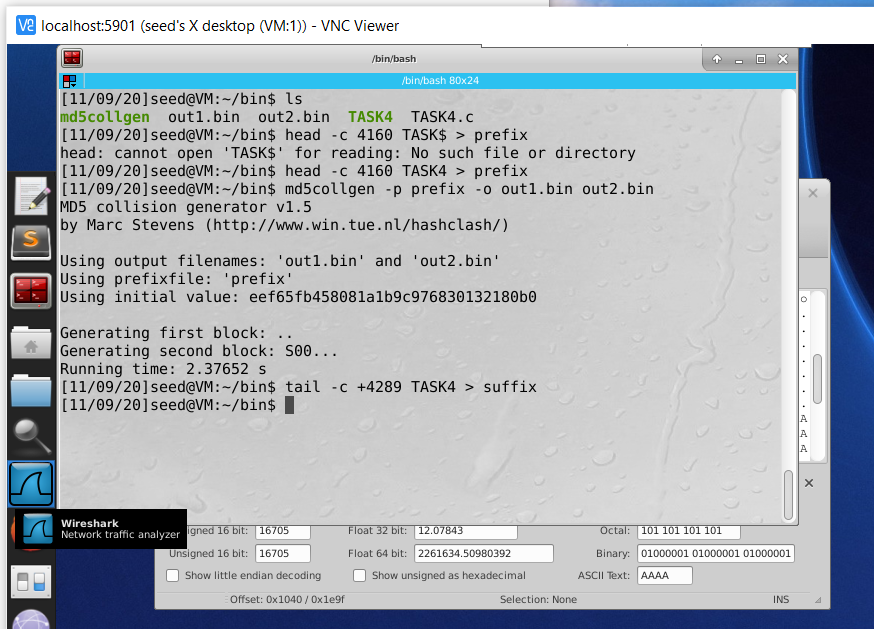
**Task 4: Making the Two Programs Behave Differently**

Firstly we create a program called TASK4.c, which has two components, a benign component and a malicious component. If the factor that determines whether the benign code is executed or the malicious one is something that could be exploited, then the attacker could use that to write a program which can pass all of the verification checks and still manage to run the malicious code. The idea behind said factor is maintain it in two arrays. If the contents of the arrays are the same, then we execute the benign code, otherwise we execute the malicious code. Thus, we can write the program as follows, populating each array with 200 ‘A’s:

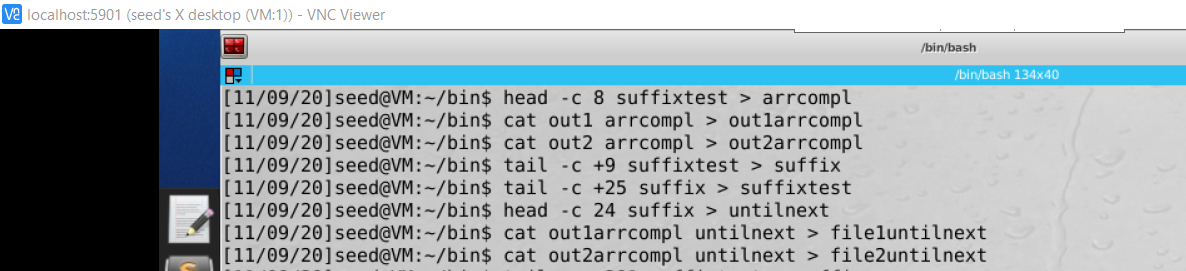


We then compile this code to yield an executable, which I name as Task4. We then set the prefix using the command ‘head’, which includes all of the bytes prior to the start of the first array. Then we generate two files using the prefix, yielding out1, and out2, having all except the final 8 elements in the first array. Then we add all the after the 4353rd one in Task4 to suffixtest. We can determine this value by looking at the Bless hex editor (seen below).

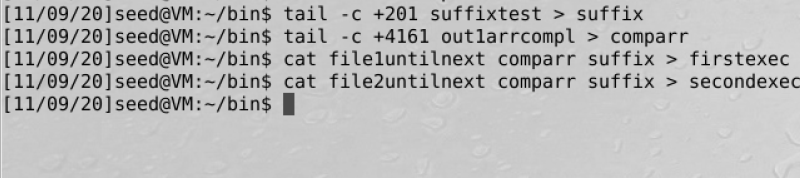




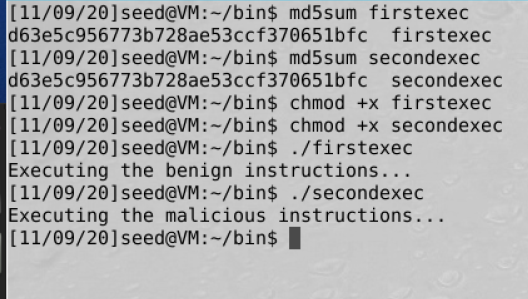
Now we complete the array in the out1 and out2 files by adding the first 8 bytes of suffix test yielding the files out1arrcompl and out2arrcompl. Now the suffix file is created which contains all of the bytes after the 8th byte within ‘suffix test’. Then we add the bytes that are between the end of the first array and the start of the second array, and make a file which I dubbed ‘untilnext’. Then we stored the bytes starting with the second array in suffix to suffixtest. Then we add the bytes to out1arrcompl and out2arrcompl to yield ‘file1untilnext’ and ‘file2untilnext’.



Now the two result files are also the two separate=part executable files which have their contents going up to the start of the second array. To ensure the attack is successful, one file needs to print the “benign code’ while the other prints the ‘malicious code’. In order to do this contents of the second must be equal to one of the generated arrays. So we put the bytes after the second array in suffixtest into suffix. Then we copy the first array from out1arrcompl to comparr. Then the comparr file can be appended to file1untilnext and file2untilnext along with the suffix in order to give the final executables firstexec and secondexec.



Then we make the two final files executable with chmod and calculate the md5 hash sum. Executing them both yields desirable results, as seen below:



This demonstrates how exactly the vulnerability of md5 collision could potentially be exploited by an attacker.