**CYBER SECURITY LAB**

**(CMPG769)**

**ASTHA PATEL**

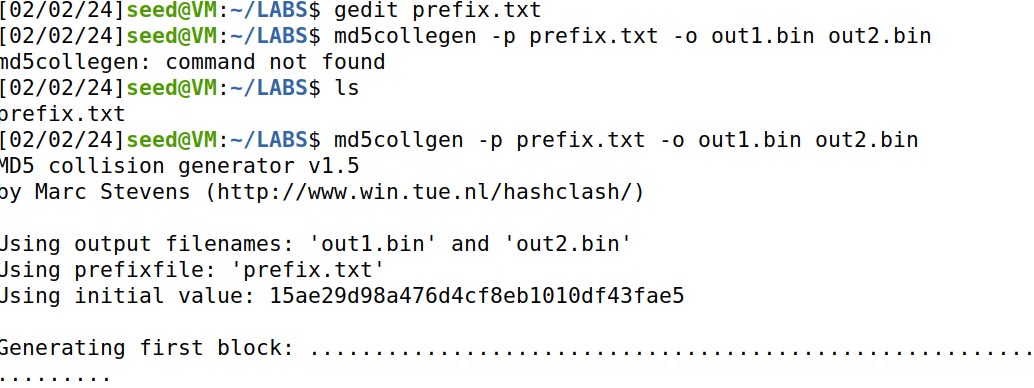
**000558727**

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

This commans will generate 2 binary files:

1. out1.bin
2. out2.bin

md5collegen –p prefix.txt –o out1.bin out2.bin

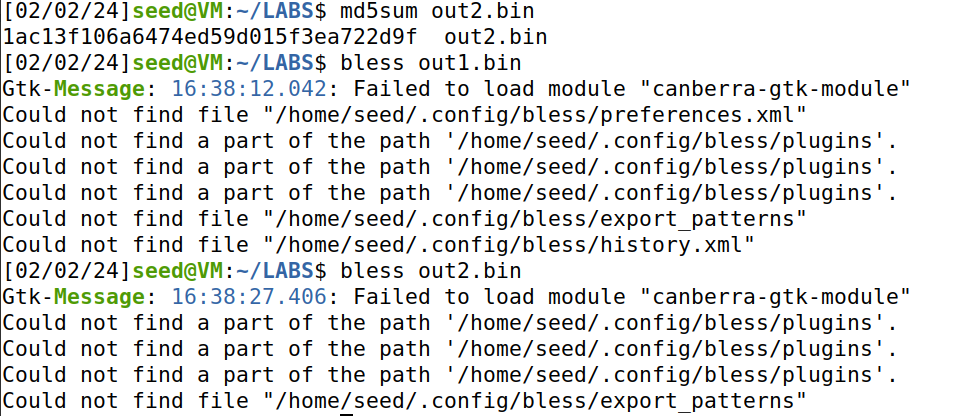


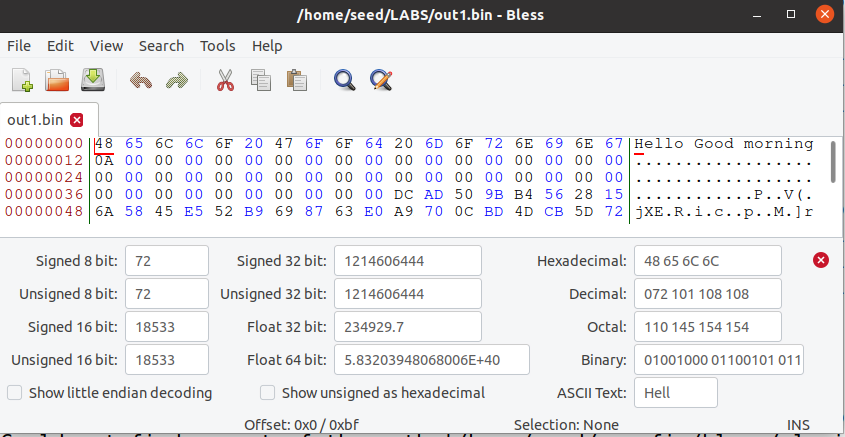
**Que-1: If the length of your prefix file is not multiple of 64, what is going to happen?**

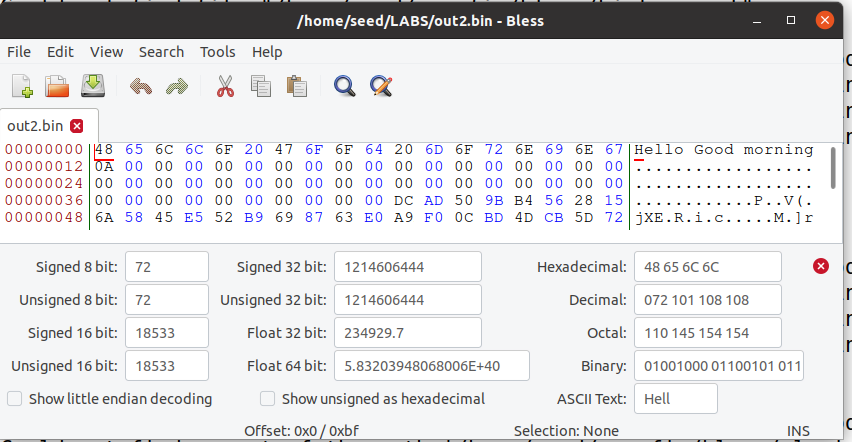
**Ans:**

The length of the prefix file isn't a multiple of 64, which can result in unexpected behaviors and potentially affect the effectiveness of the MD5 collision attack in a lab environment.

A padding adjustment will be made to align the message size to a multiple of 64 bytes if the prefix file's length is not a multiple of 64. The MD5 algorithm cannot function properly without this padding procedure. The message length must be a multiple of 512 bits, and padding is needed to ensure that MD5 examines messages in blocks of 512 bits (64 bytes).







**Que-2: Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.**

**Ans:**

It is dependent on the particular features and implementation of the collision tool if you build a 64-byte prefix file and then use it.

All things considered, collision attacks often use a prefix file to identify two distinct inputs that, when combined with the prefix, hash to the same value. Your initial input data section is fixed to a certain value if you supply a 64-byte prefix file.

After that, the collision tool would likely create more data to add to this prefix in an effort to identify two inputs that, when coupled with the same prefix, give the same hash output. The collision tool's ability to locate such inputs will depend on how successful and efficient it is.

Following are the screenshots that shows what happen if we create a prefix file with exactly 64 bytes and run the collision tool again:

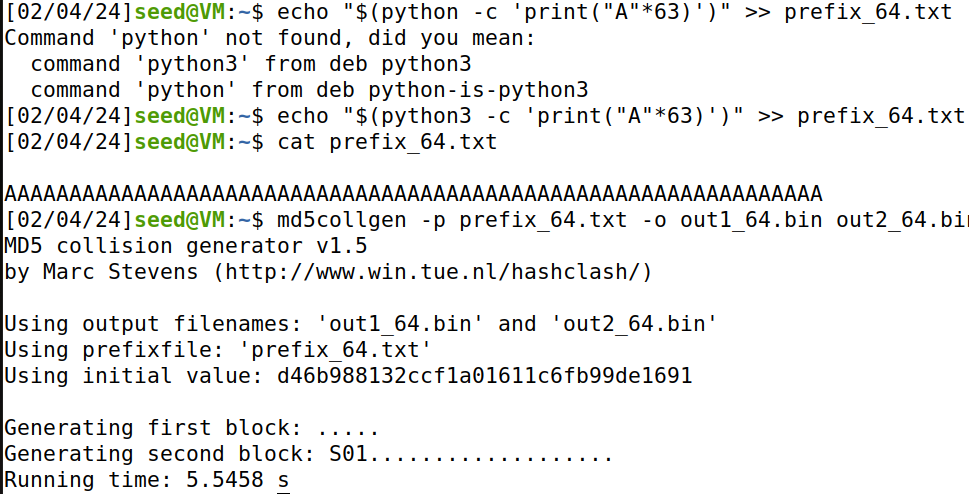
We use the following command which creating exactly 64 bytes prefix file.

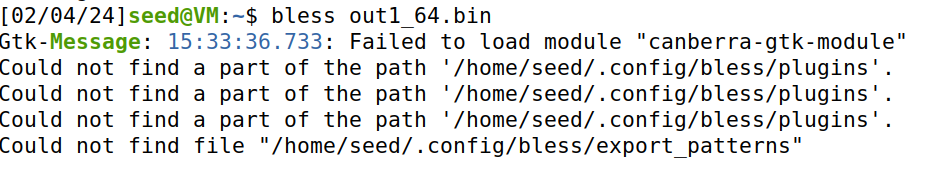
echo “$(python3 –c ‘print(“A”\*63)’)” >> prefix\_64.txt

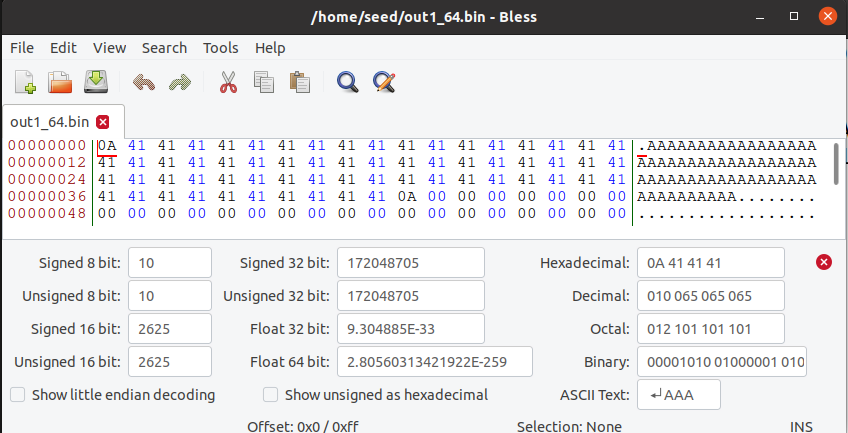
After that we will generate two different files with the same MD5 hash values.

md5collgen -p prefix\_64.txt -o out1\_64.bin out2\_64.bin

For checking output we will use: bless out1\_64.bin command which will return image and in which

****

****

****

**Que-3: Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different.**

**Ans:**

The MD5 hashing algorithm's colliding files are created using the MD5CollGen program. The files are different even when it produces two output files with the same MD5 hash. Still, only a subset of the files, usually referred to as the "suffix" are different between the two colliding files.

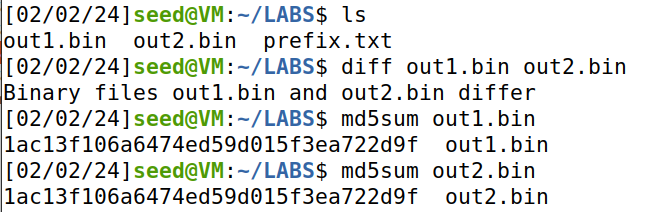
Both files have the same prefix and the first portion of the suffix (often referred to as the "fixed block"). The section that varies comes after the fixed block.

The fixed block for the MD5CollGen tool is normally 64 bytes in size. In order to cause the collision, the final 64 bytes (128 bytes total - 64 bytes fixed block) in each file are produced differently.

Analyzing the unique MD5CollGen result for those two colliding files would be necessary to pinpoint the precise bytes that differ between the two files. Since the variation that causes the collision is introduced in the suffix region of the files, every colliding pair produced by MD5CollGen would have a unique difference in this area.

We use the following command for checking either 2 binary files are different or same:

diff out1.bin out2.bin



**Task-2: Understanding MD5’s Property**

First we will create two files: file1 and file2

Now, if we want to check what is there inside this files then we will use cat command to see.

And then we will concatinate these two files and store it in another file named file3 by using following command:

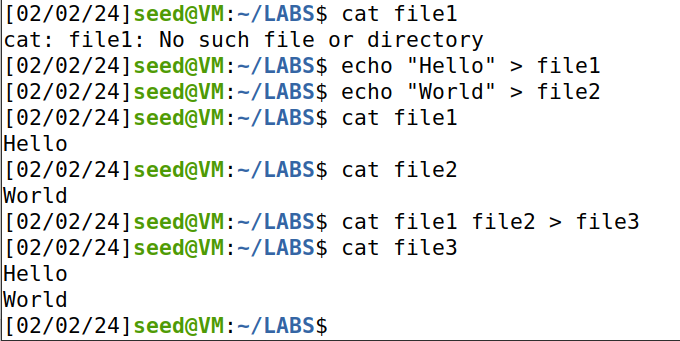
cat file1 file2 > file3

Now all data from file1 and file2 will concatinate and return

**Hello**

**World**

As an output.



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

First, we will create task3.c file with the following command:

sudo gedit task3.c

And then paste following program and save it as c program:

#include <stdio.h>

unsigned char xyz[400] = {"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};

int main()

{

int i;

for (i=0; i<400; i++)

{

printf("%x", xyz[i]);

} printf("\n");

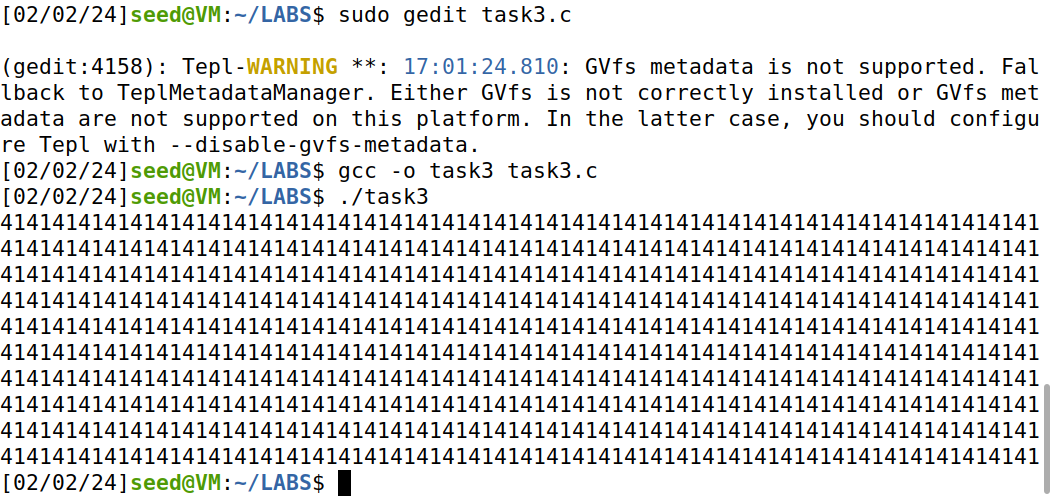
}

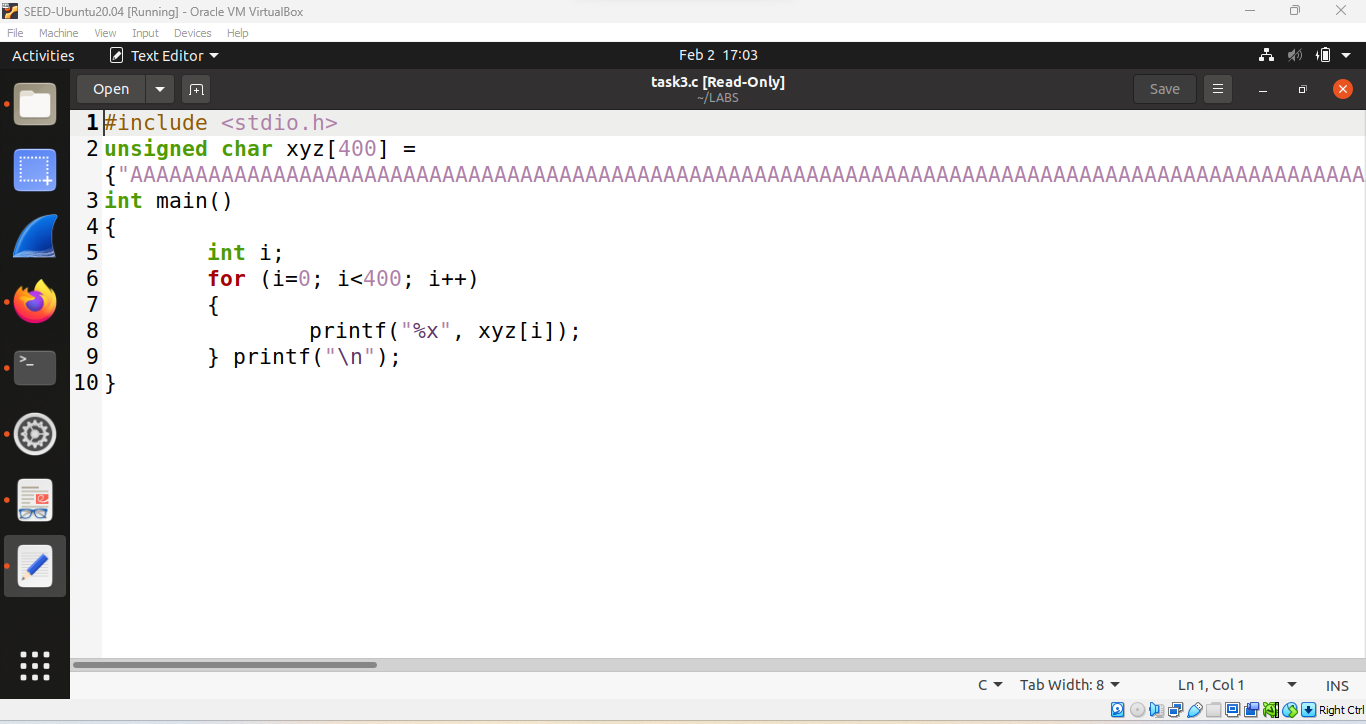
And then to see the result use following commands:

gcc –o task3 task3.c

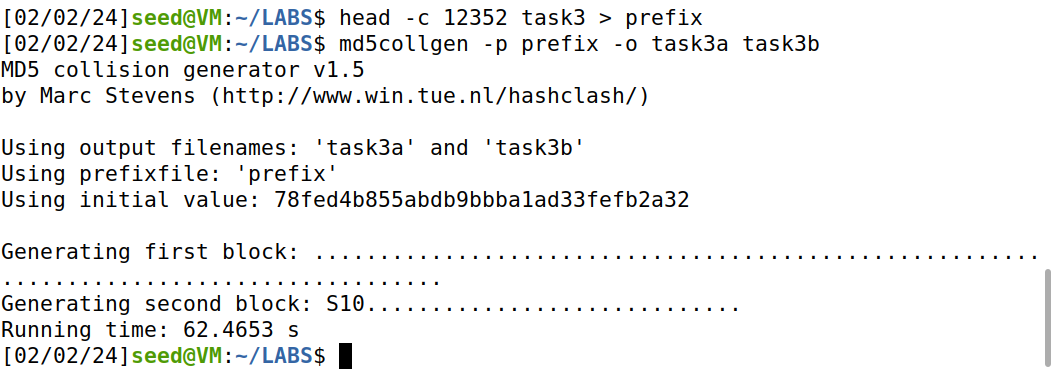
./task3

Following is the result:





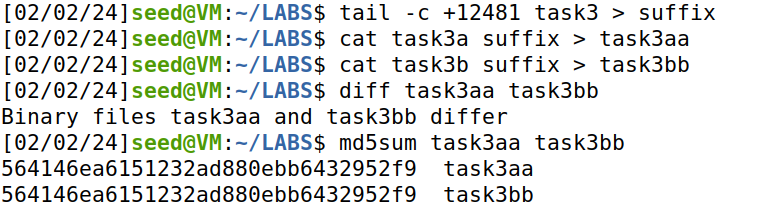
Now, we will create one file named task3 that needs to be multiple of 64 bytes and here we are taking 12352 which is divisable by 64. Then we will craete two md5 files task3a and task3b.

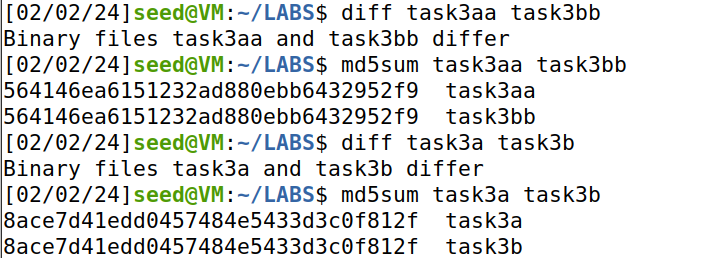


Then we will add 128 in 12352 which would be 12481 from where our suffix will start. So we will generate our suffix using that number:

tail –c +12481 task3 > suffix

Whatever we got from task3a and task3b, we will store suffix part in task3aa and task3bb respectively and we will check they are same or not.





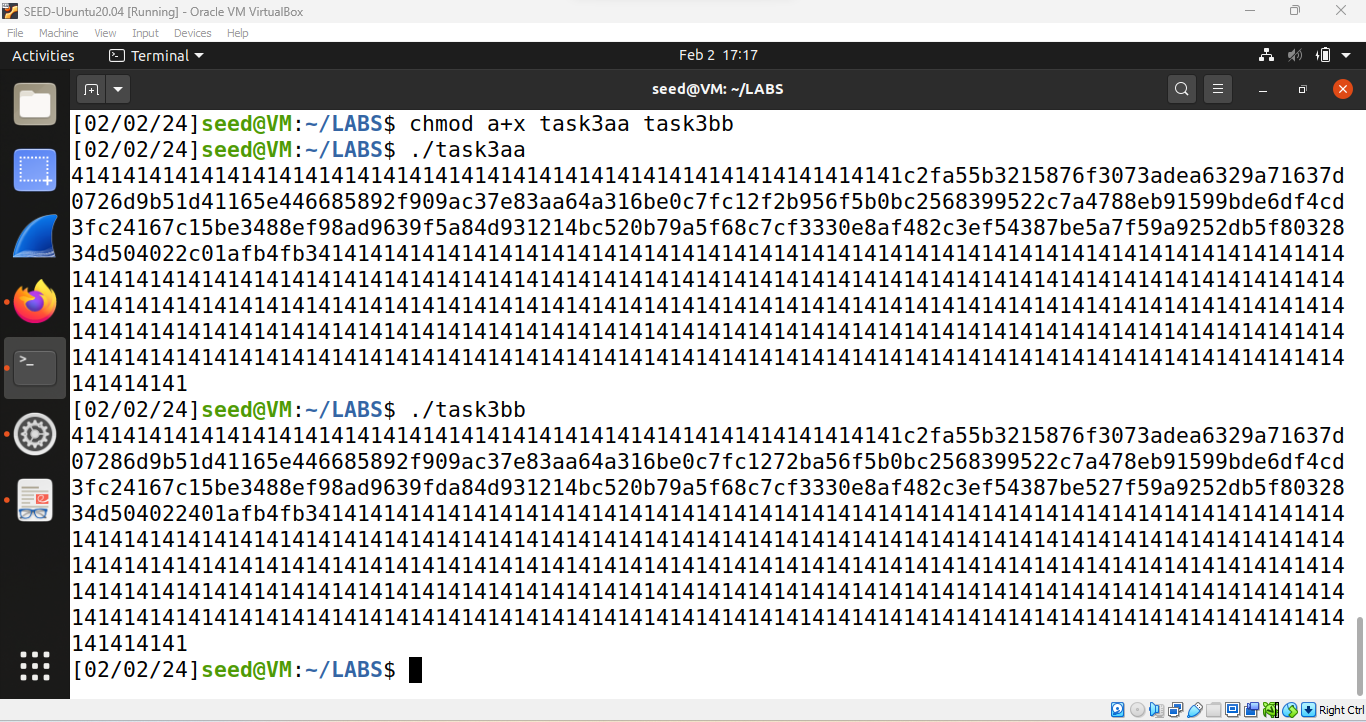
Now we will add prefix, 128 byte region and suffix by following command:

chmod a+x task3aa task3bb

Then check what is there inside:

./task3aa

./task3bb



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

In this task we have to create a file in which we have to pass 2 variables each with 400 A’s and then we will write whole code to see it works proper or not.

Now first, we will create one c file named task4.c and then we will put the following code inside this file:

#include <stdio.h>

unsigned char X[400] = {"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};

unsigned char Y[400] = {"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};

int a = 0;

int main()

{ int i;

for (i=0; i<400; i++)

{

if (X[i] == Y[i])

continue;

else

{

a=1;

break;

}

}

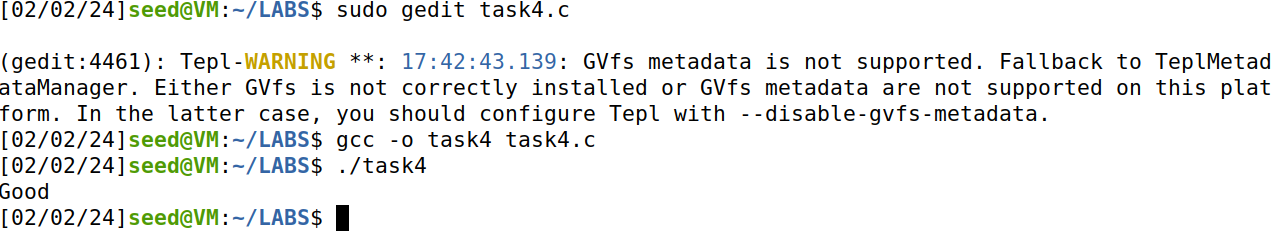
if(a==0)

printf("Good\n");

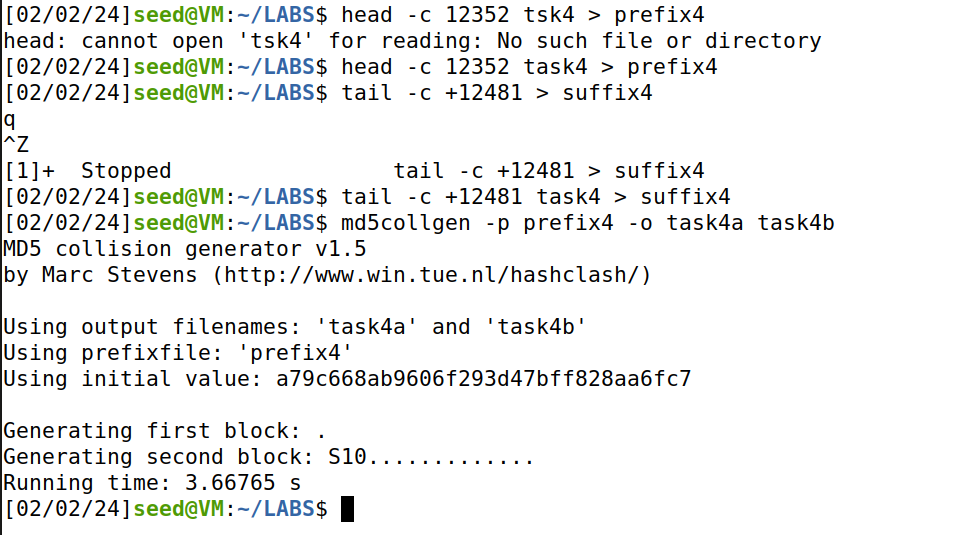
else

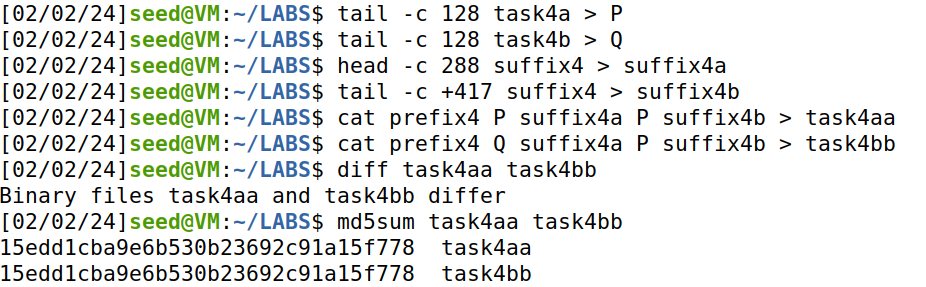
printf("Bad\n");

}

****

Now we will create prefix4 and suffix as we did in the task-3 but here we take 12352 value for prefix and 12481 for prefix as we add 128 in the prefix value. Then we will store those values in task4aa and task4bb.

****

****

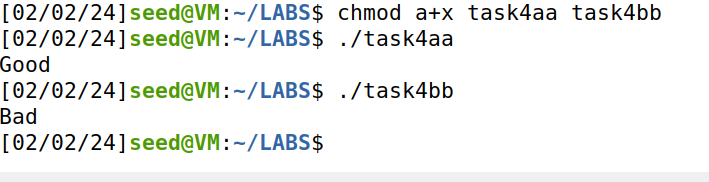
Now we will add prefix, 128 byte region and suffix by following command:

chmod a+x task4aa task4bb

Then check what is there inside:

./task4aa

./task4bb

****