Lab 3 - Documentation

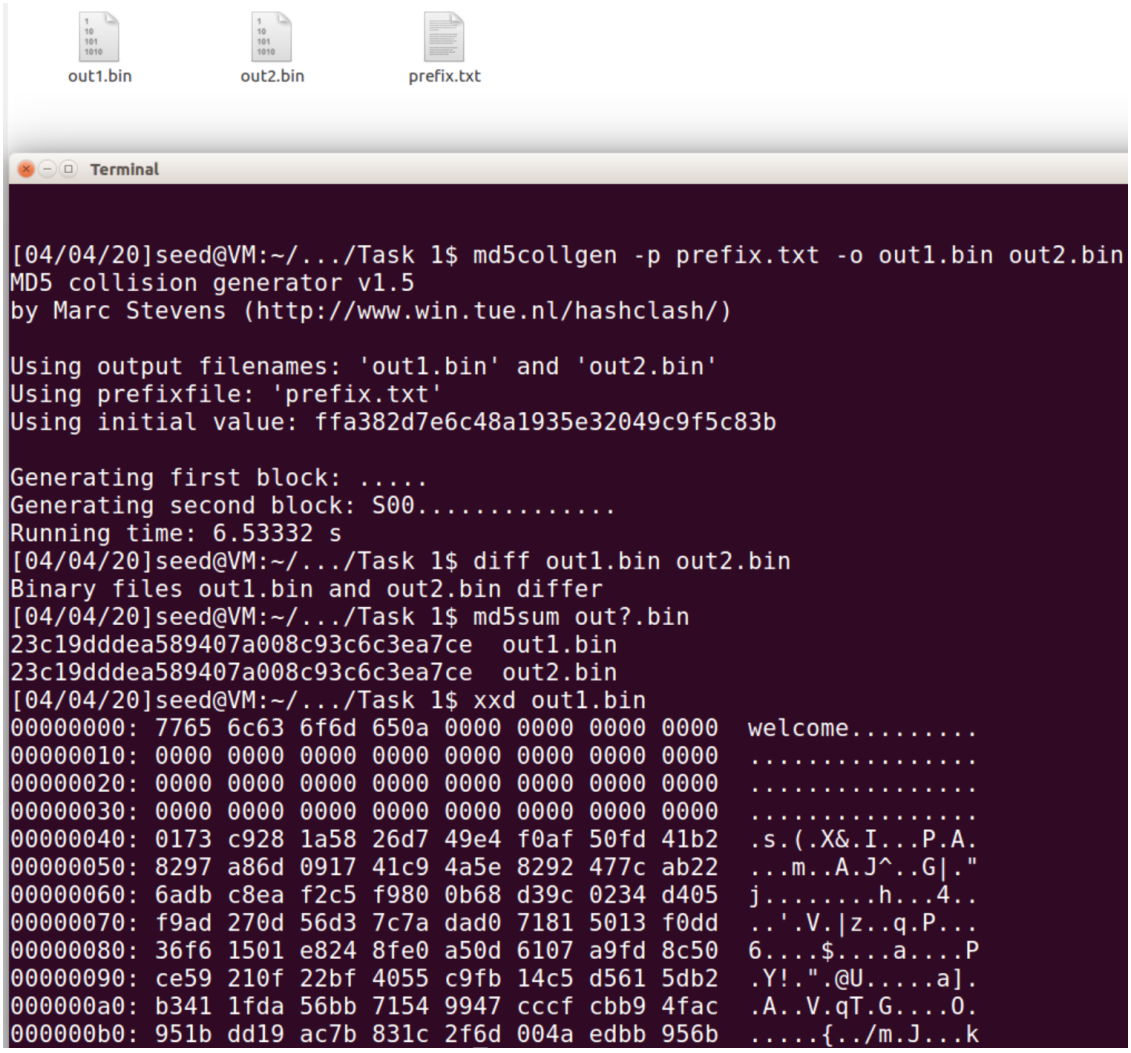
April 4, 2020

1 Lab 3 - MD5 Collision Attack Lab

1.1 ## By: Laura Perda

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

In this task, I will generate two different files with the same MD5 hash values. The beginning parts of these files are required to be the same (share the same prefix), which can be accomplished using the md5collegen program. I created a text file named 'prefix.txt' that contains the word 'welcome'. The picture below shows the following md5collgen command. When using the recommended 'diff' command on 'out1.bin' and 'out2.bin', it shows that the two files are different, but when comparing the hash value through the 'md5sum' command it shows the same.

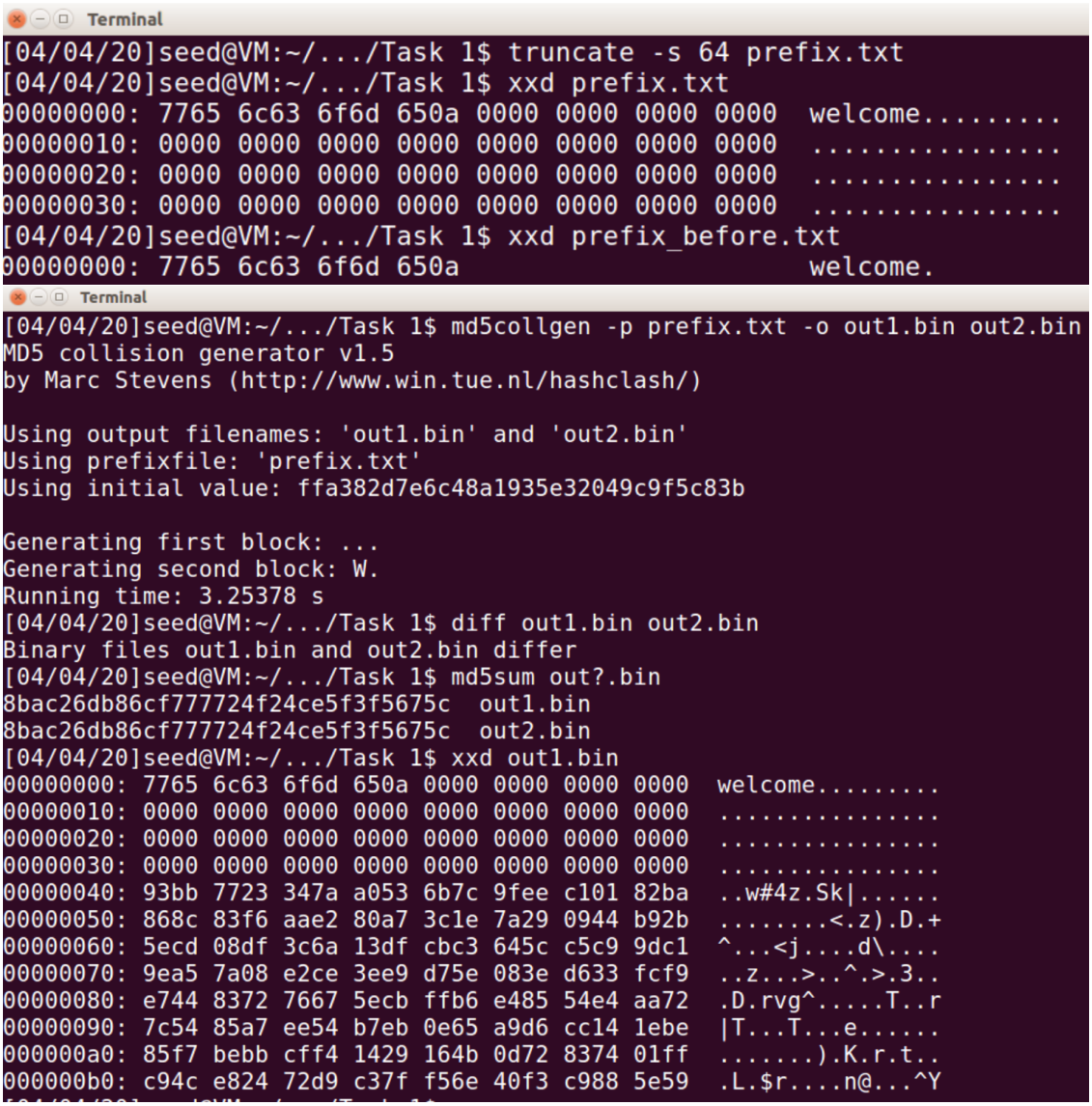


Q1: If the length of your prefix file is not a multiple of 64, what is going to happen?

A: In the picture above, I also decided to check the hex dump of 'out1.bin' which show it was padded with zeroes to bring it up to 64 bytes and then followed by random data being appended to that. The hex dump for 'out2.bin' has the same 64 initial bytes, but differs slightly when it comes to the random data appended at the end of the file.

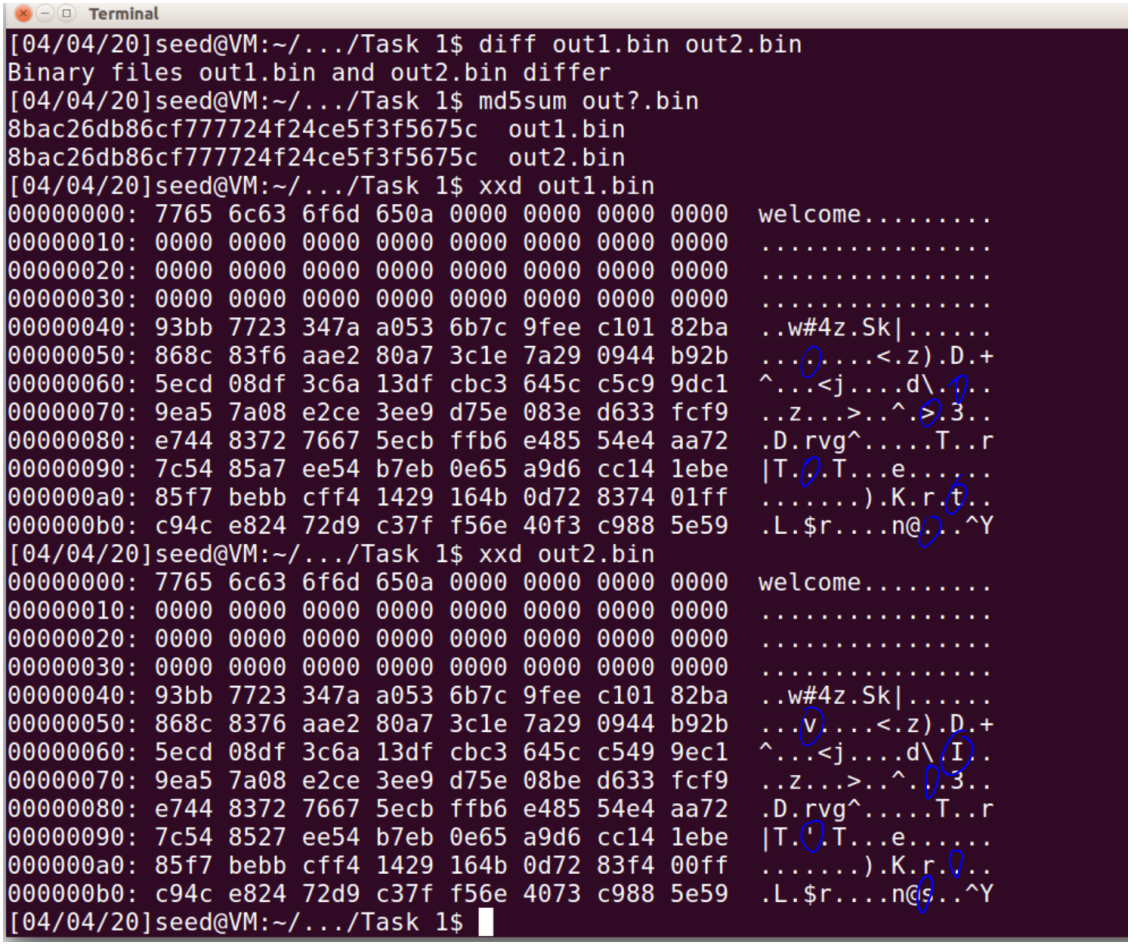
Q2: Create a prefix file with exactly 64 bytes & run the collision tool again to see what happens.

A: For this question, I truncated the prefix.txt file to be the size of 64 bytes thru the 'truncate' command. The results shown in the pictures below show that the outcomes were not padded this time. The only change is the type of random data appended to the prefix.



Q3: Are the data (128 bytes) genereated by md5collgen completely different for the two output files? Please identify all the bytes that are different.

A: The picture belows shows that there are a total of 6 different bytes from the random appended data that are different in the hex dumps.

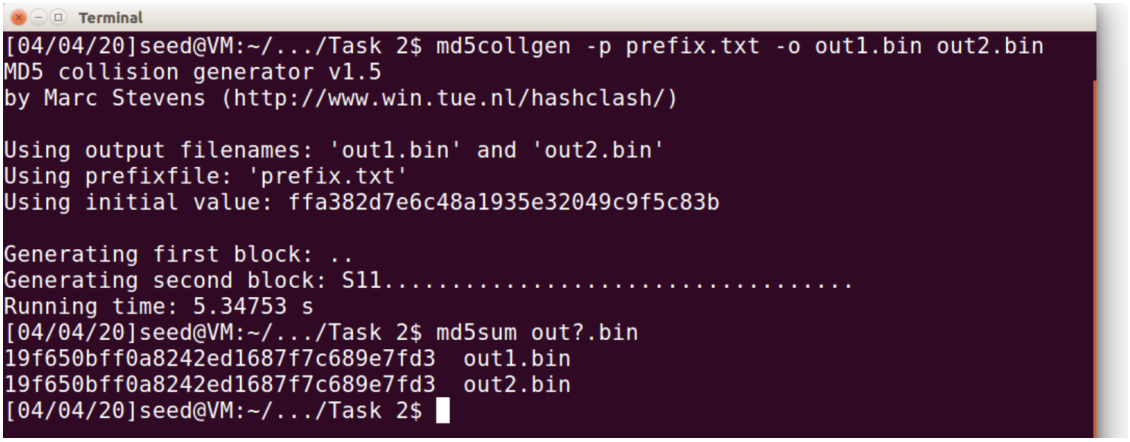


1.3 Task 2 - Understanding MD5's Property

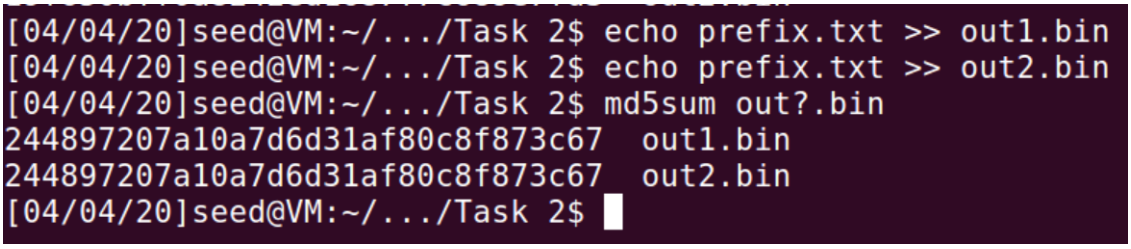
In this task, we will try to understand the MD5 algorithm and undertand the importance of these properties. From what we have seen, MD5 divides the input data into blocks of 64 bytes and then computes the hash over these blocks. The core of the MD5 alg. is the compression function, which takes 2 inputs, a 64 byte block and the outcome of the previous iteration. It then produces a 128 bit IHV, which stands for "Intermediate Hash Value", which is fed to the next iteraiton.

We have to design a experiment on the simple fact that if inputs M and N have the same hash, adding the suffix T to them will result in two outputs that have the same hash value. This was already mostly accomplished in Task 1. Thus, the steps in Task 1 will repeated and then show an extra step in which we append some suffix and redo the hash.

Below shows that the M & N have the same hash value.



The picture below shows that after appending the prefix.txt file to the end of both .bin files, the 'md5sum' command shows that the hash value remains the same.

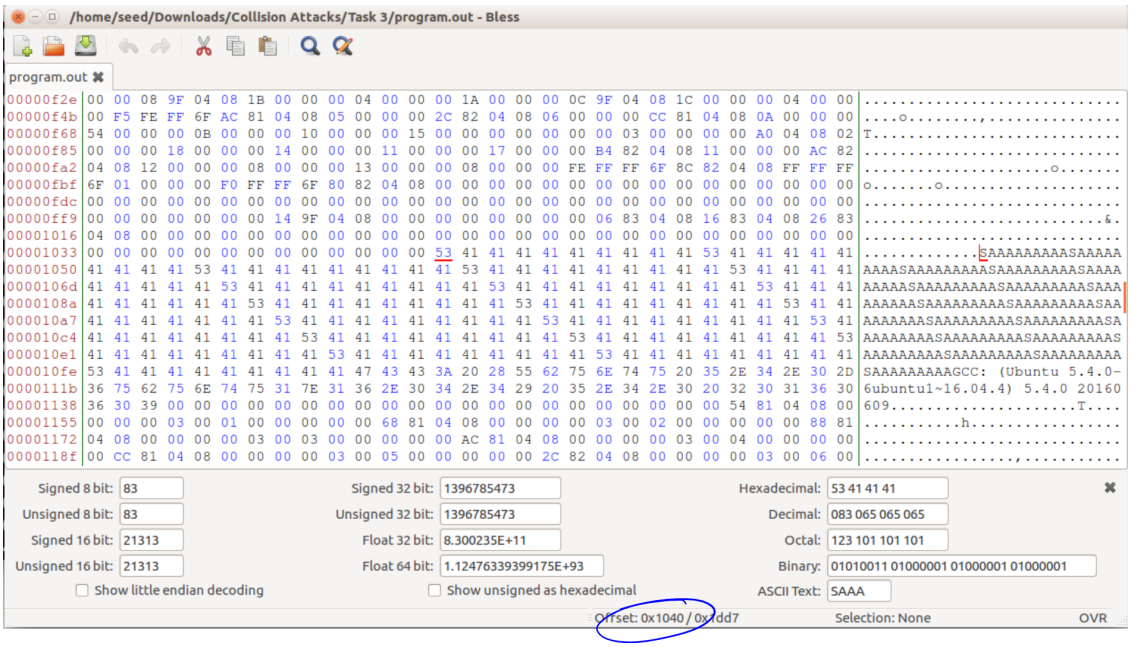


As a result, we can say the MD5 property holds.

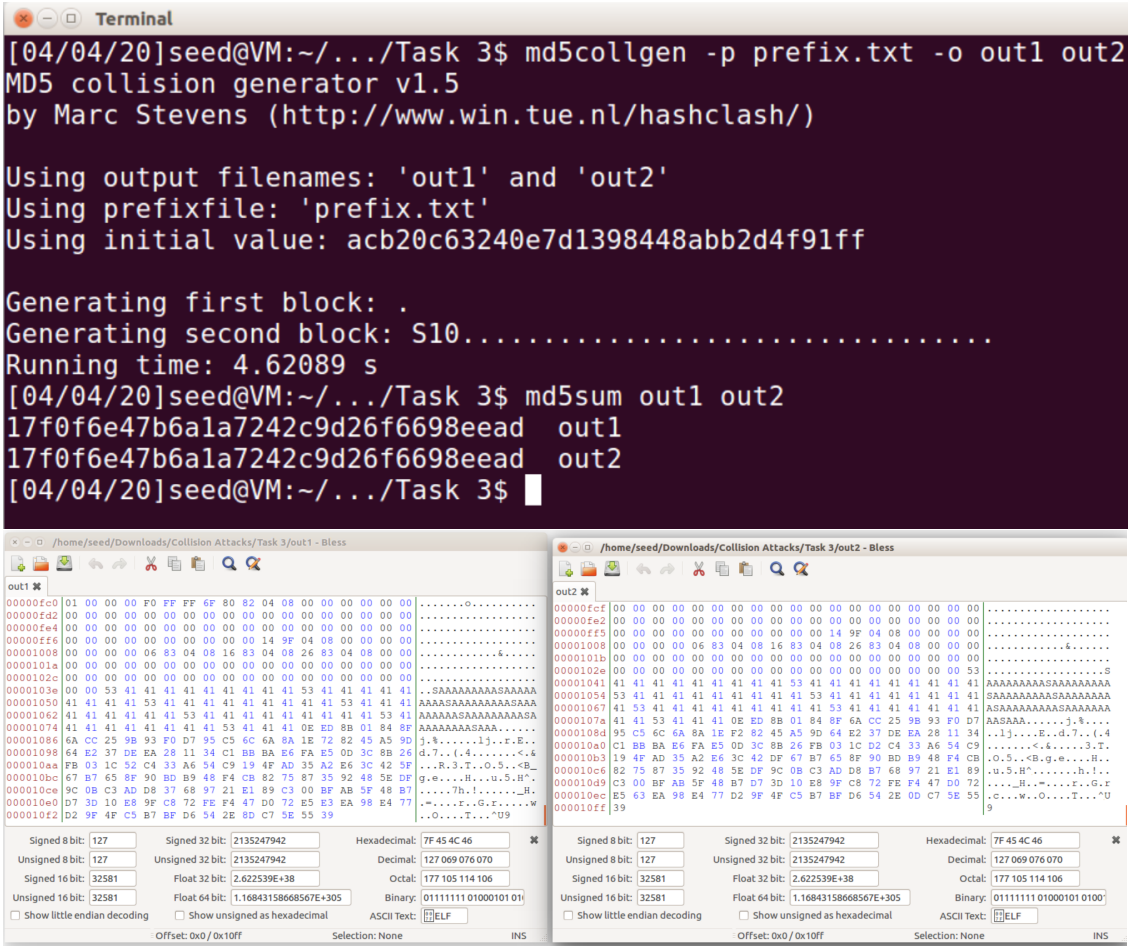
1.4 Task 3 - Generating Two Executable Files with the same MD5 Hash

In this task, we are given a program that we must modify with some array and so that we can divide the executable file into three parts: prefix, 128 bit region, and suffix. Length of prefix should be a multiple of 64. The point of this is to generate two different executables with the same MD5 hash value and check if their hash value remains the same after appending some suffix.

Compiling the provided program and checking the location of the beginning of the array of values shows that it is at an offset of 0x1040, which converts to 4160 in decimal (divisible by 64). Since I would like to see a bit of the array within the prefix, I chose to add an extra 64 byte block, so the cut off would be 4224 bytes. The picture below demonstrates a bit of this.



The picture below shows that after using the "md5collgen" to create the same hash but with different endings (P & Q).



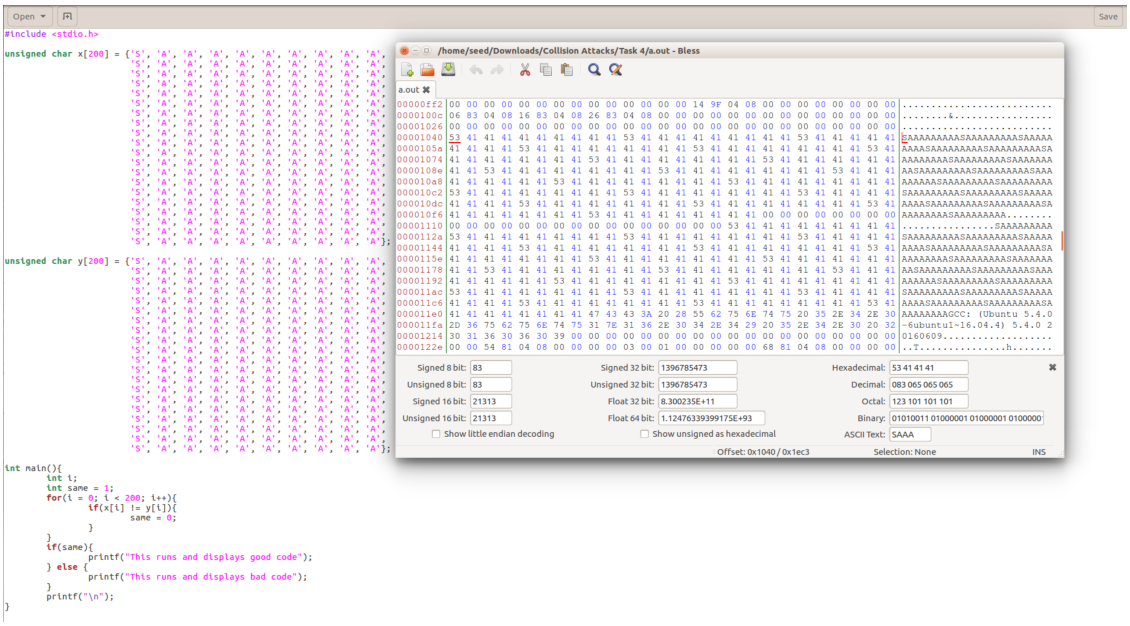
Now, we can append the tail of the prefix to both files and check whether the same hash value is true when they receive the same suffixes. Creating the files into executable files does not change the outcome.

```
[04/04/20]seed@VM:~/.../Task 3$ tail -c +4353 prefix.txt > same
[04/04/20]seed@VM:~/.../Task 3$ cat same >> out1
[04/04/20]seed@VM:~/.../Task 3$ cat same >> out2
[04/04/20]seed@VM:~/.../Task 3$ md5sum out1 out2
17f0f6e47b6a1a7242c9d26f6698eead  out1
17f0f6e47b6a1a7242c9d26f6698eead  out2
[04/04/20]seed@VM:~/.../Task 3$
```

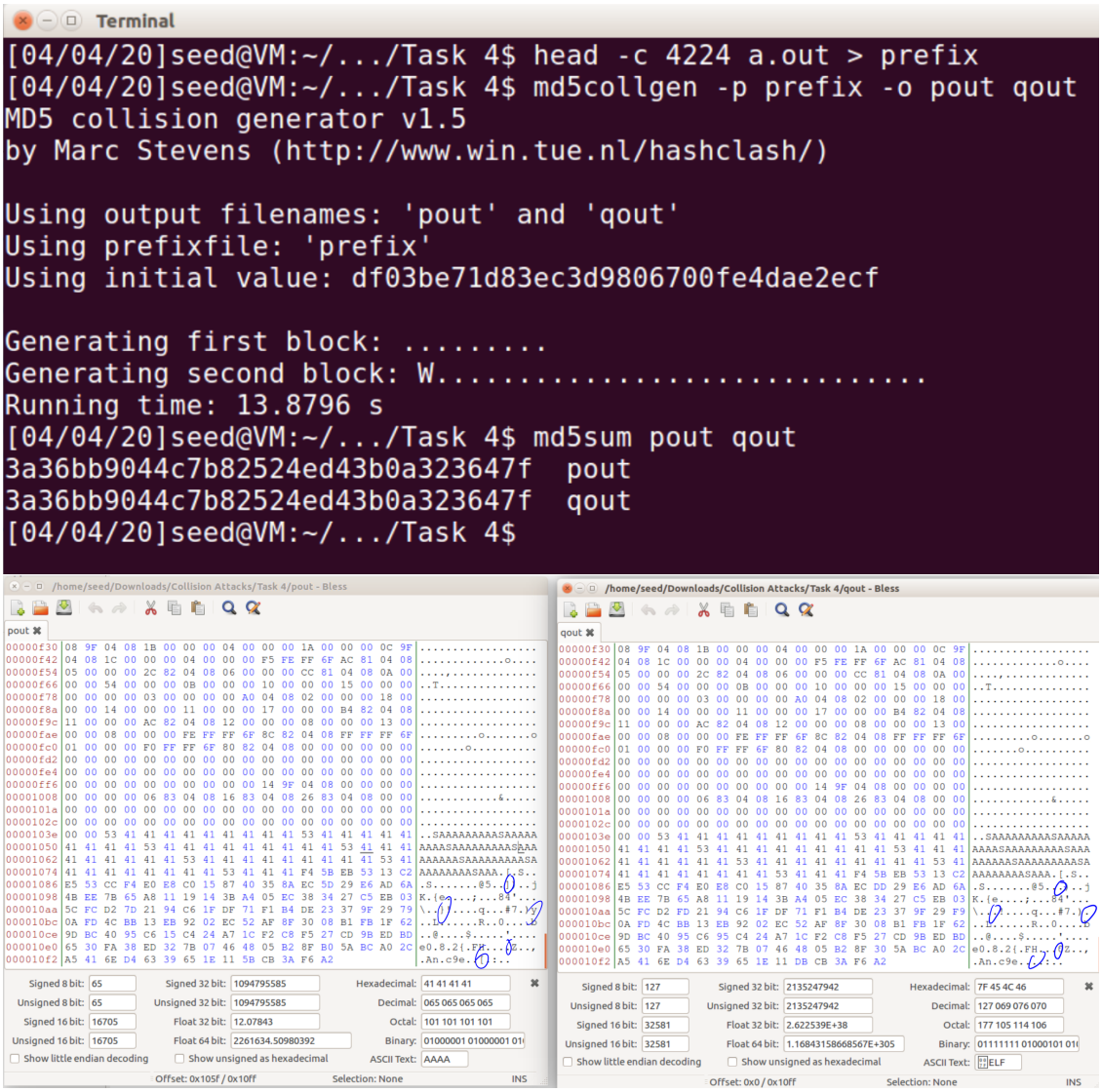
1.5 Task 4 - Making the 2 Programs Behave Differently

In the previous task, we manage to make two programs to have the same MD5 hash, but their behaviors were different. Their differences were in the print out, but they still execute the same sequence of instructions. In this task, we want to create two different programs to have the same MD5 hash value.

Same case as before but with a different source code. The two arrays are the same, so it should execute the first print statement. Again, I while the array starts at x1040, I will like to see a bit of the array, so x1080 is fine for the prefix.



After using the "md5collgen" command to create the files pout & qout, I checked to see if the hash values were the same by using 'md5sum'. Both result being the same despite being a slight difference in the middle section as shown through bless hex. Note that the last 8 elements of the first array are missing, so we need to grab those and append to the two created files.



After I had checked the hash values, I had placed the rest of a.out file into a file named 'test'. This is done in order to grab the missing 8 bytes of the first array. Using this, I concatenated the 'missing' file to 'pout' & 'qout' to create a complete version to both.

```
[04/04/20]seed@VM:~/.../Task 4$ tail +4353 a.out > test
tail: cannot open '+4353' for reading: No such file or directory
[04/04/20]seed@VM:~/.../Task 4$ tail -c +4353 a.out > test
[04/04/20]seed@VM:~/.../Task 4$ head -c 8 test > missing
[04/04/20]seed@VM:~/.../Task 4$ cat pout missing > poutcomplete
[04/04/20]seed@VM:~/.../Task 4$ cat qout missing > qoutcomplete
[04/04/20]seed@VM:~/.../Task 4$ tail -c +9 test > suffix
```

Then I needed to add the bytes between the end of the 1st array & the beginning of the 2nd array, which I named 'tillNext'. I stored the beginning of the second array in 'suffix' into 'test', so that I can add them to the complete versions of 'pout' & 'qout'.

```
[04/04/20]seed@VM:~/.../Task 4$ tail -c +25 suffix > test
[04/04/20]seed@VM:~/.../Task 4$ head -c 24 suffix > tillNext
[04/04/20]seed@VM:~/.../Task 4$ cat poutcomplete tillNext > f1tillNext
[04/04/20]seed@VM:~/.../Task 4$ cat qoutcomplete tillNext > f2tillNext
```

At this point, the 2 programs are different executables but only have the content up to the 2nd array. To make these two files do different outcomes (good vs bad), the content of the 2nd array should be equal to the generated array. So, I had to place the bytes after 2nd array from 'test' to 'suffix'. Then I copied the 1st array from 'poutcomplete' to 'parray', which was appended to the previously made files in the picture above. From this, I was able to create the 2 good & bad executables.

```
[04/04/20]seed@VM:~/.../Task 4$ tail -c +201 test > suffix
[04/04/20]seed@VM:~/.../Task 4$ tail -c +4161 poutcomplete > parray
[04/04/20]seed@VM:~/.../Task 4$ cat f1tillNext parray suffix > firstEx
[04/04/20]seed@VM:~/.../Task 4$ cat f2tillNext parray suffix > secEx
```

Now that I have the 2 executables, it's time to test whether they still produce the same hash value despite being different. I made them executable, checked their hash values, and executed both just to make sure the outcome was correct. It was interesting to see that so much work was necessary in order to exploit the MD5 algorithm.

```
[04/04/20]seed@VM:~/.../Task 4$ chmod +x firstEx
[04/04/20]seed@VM:~/.../Task 4$ chmod +x SecEx
chmod: cannot access 'SecEx': No such file or directory
[04/04/20]seed@VM:~/.../Task 4$ chmod +x secEx
[04/04/20]seed@VM:~/.../Task 4$ md5sum firstEx secEx
20c911aa6c0c9d876801c46251ab41ac  firstEx
20c911aa6c0c9d876801c46251ab41ac  secEx
[04/04/20]seed@VM:~/.../Task 4$ ./firstEx
This runs and displays good code
[04/04/20]seed@VM:~/.../Task 4$ ./secEx
This runs and displays bad code
[04/04/20]seed@VM:~/.../Task 4$
```

[ ]: