Introduction to Information Security
CS 458 - Fall 2022 Lab 3
MD5 Collision Attack Lab1

Lab Tasks

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

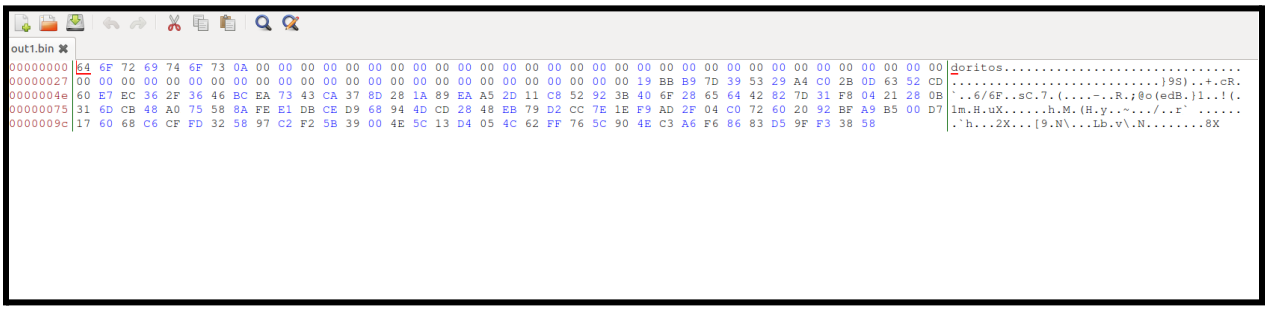
```
[11/09/22]seed@VM:~/seedlab$ md5collgen -p doritos.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'doritos.txt'
Using initial value: 9f077e3f9bd427f927d866d49a2f660b

Generating first block: .........
Generating second block: S10.....................
Running time: 28.5437 s
```

Same hash:

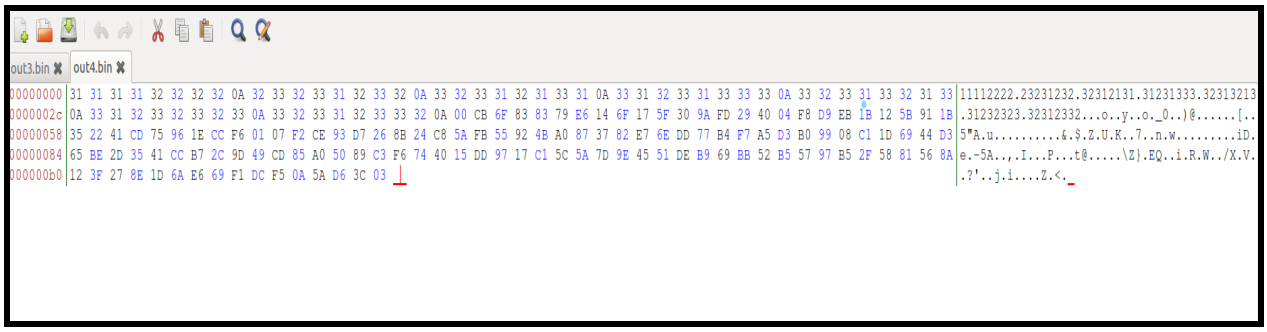
```
5739c0efc4b8b8d69b5e4f459a749053  out1.bin
5739c0efc4b8b8d69b5e4f459a749053  out2.bin
```

Question 1. If the length of your prefix file is not multiple of 64 , what is going to happen?

After running the files through the MD5 collision generator, using doritos.txt as the prefix, we can see that it has been padded with zeros. This is because the hashing algorithm processes 512 bits at a time, which is equivalent to 64 bytes. Since the block was not completely filled, it continued to fill them with zeros.



Question 2. Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens. After creating a prefix with exactly 64 bytes, it is clear that there is no zero padding. I purposely filled the prefix with 64 numbers to see if there was a difference, and it shows.



Question 3. Are the data (128 bytes) generated by md5collgen completely different for the two output files? Please identify all the bytes that are different. There's no significant difference for the two output files, some of the bytes changed but not all. Highlighted are the bytes that changed. Used hexdump for both files, and diff to view the differences.

```
5,7c5,7
< 0000050 c128 8d9f 2698 9ea7 e5fe e582 266f d2c4
< 0000060 05c4 07b5 26d8 d502 41e7 7b7b 75df 03ac
< 0000070 fe90 f571 35a6 1b75 9e49 6e6d 384f 8d06
---
> 0000050 c128 0d9f 2698 9ea7 e5fe e582 266f d2c4
> 0000060 05c4 07b5 26d8 d502 41e7 7b7b f5df 03ac
> 0000070 fe90 f571 35a6 1b75 9e49 ee6d 384f 8d06
9,11c9,11
< 0000090 8d20 1fd2 6627 f30e 5ad3 c776 46ec f002
< 00000a0 be88 d4be 09f4 0324 4c8a 1769 4082 2b1a
< 00000b0 242b 74aa bbd3 b609 ea2e 1787 0fb9 51b0
---
> 0000090 8d20 9fd2 6627 f30e 5ad3 c776 46ec f002
> 00000a0 be88 d4be 09f4 0324 4c8a 1769 c082 2b19
> 00000b0 242b 74aa bbd3 b609 ea2e 9787 0fb9 51b0
```

2.2 Task 2: Understanding MD5's Property

Given two inputs M and N , if MD5(M) = MD5(N) , i.e., the MD5 hashes of M and N are the same, then for any input T , MD5(M || T) = MD5(N || T) , where || represents concatenation. That is, if inputs M and N have the same hash, adding the same suffix T to them will result in two outputs that have the same hash value.

Your job in this task is to design an experiment to demonstrates that this property holds for MD5

You can use the cat command to concatenate two files (binary or text files) into one. The following command concatenates the contents of file2 to the contents of file1 , and places the result in file3.

Since we have to prove that this property holds for MD5, I created a text file called “extraCorn.txt” with the contents “IEatSpicyCorn”. I then ran it through the MD5 hashing algorithm:

```
[11/09/22]seed@VM:~/seedlab$ cat extraCorn.txt
IEatSpicyCorn
[11/09/22]seed@VM:~/seedlab$ md5collgen -p extraCorn.txt -o corn1.bin corn2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'corn1.bin' and 'corn2.bin'
Using prefixfile: 'extraCorn.txt'
Using initial value: a7e909ef698f0d18e6d111b460ff3297

Generating first block: .....
Generating second block: S10..........
Running time: 5.40098 s
```

Next, I checked to make sure that both of their hashes are the same. I was able to do this by using the md5sum command:

```
[11/09/22]seed@VM:~/seedlab$ md5sum corn1.bin
9109bc0c91ef49ba45ecf31903437bf9  corn1.bin
[11/09/22]seed@VM:~/seedlab$ md5sum corn2.bin
9109bc0c91ef49ba45ecf31903437bf9  corn2.bin
[11/09/22]seed@VM:~/seedlab$
```

Lastly, I appended the string “[redacted]” to both of the files. Then I checked both of the hashes again using the md5sum command, and both still had the same hash:

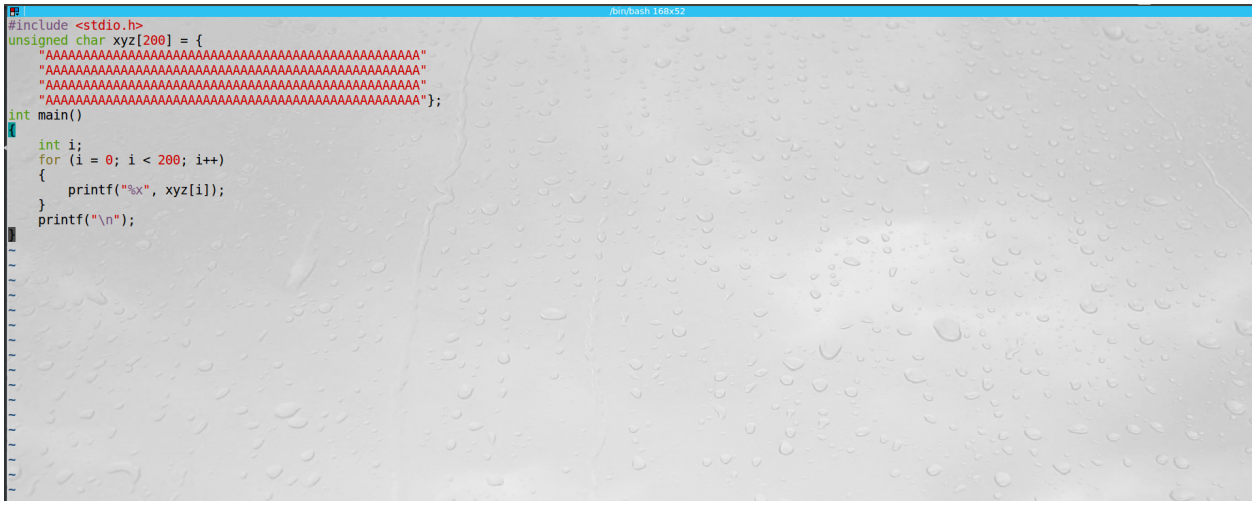
```
[11/09/22]seed@VM:~/seedlab$ echo [redacted] >> corn1.bin
[11/09/22]seed@VM:~/seedlab$ echo [redacted] >> corn2.bin
[11/09/22]seed@VM:~/seedlab$ md5sum corn1.bin
e7d8159bdf094d4c046b850974369fa9  corn1.bin
[11/09/22]seed@VM:~/seedlab$ md5sum corn2.bin
e7d8159bdf094d4c046b850974369fa9  corn2.bin
[11/09/22]seed@VM:~/seedlab$
```

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

In this task, you are given the following C program . Your job is to create two different versions of this program, such that the contents of their xyz arrays are different, but the hash values of the executables are the same.

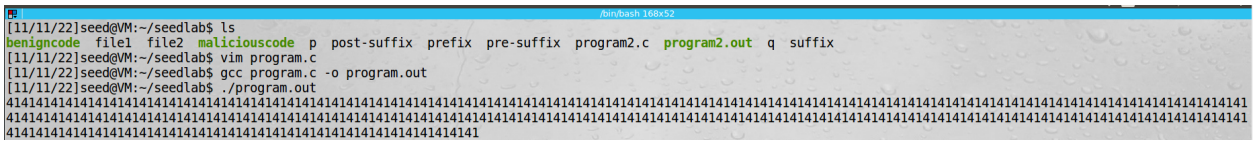
Here is my first version of the program:

Filled the array with fixed values so I can easily find them in binary, as suggested.



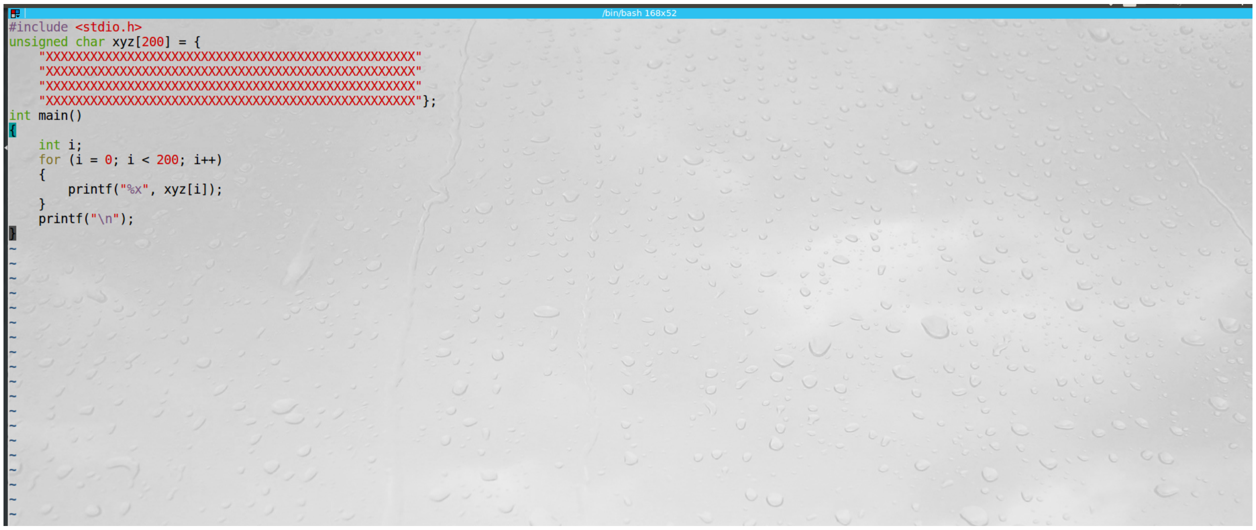
```
/bin/bash 168x52
#include <stdio.h>
unsigned char xyz[200] = {
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};
int main()
{
    int i;
    for (i = 0; i < 200; i++)
    {
        printf("%x", xyz[i]);
    }
    printf("\n");
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Compiled and ran the program. The program is printing the hex value of A (41), as expected.

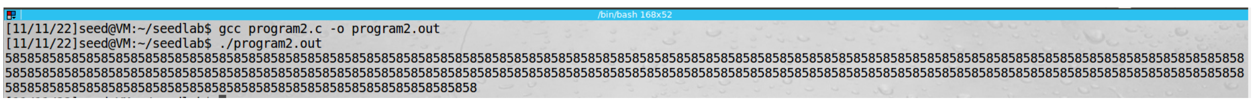


```
/bin/bash 168x52
[11/11/22]seed@VM:~/seedlab$ ls
benigncode  file1  file2  maliciouscode  p  post-suffix  prefix  pre-suffix  program2.c  program2.out  q  suffix
[11/11/22]seed@VM:~/seedlab$ vim program.c
[11/11/22]seed@VM:~/seedlab$ gcc program.c -o program.out
[11/11/22]seed@VM:~/seedlab$ ./program.out
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
```

Second version (program2), this time I will use x values instead.

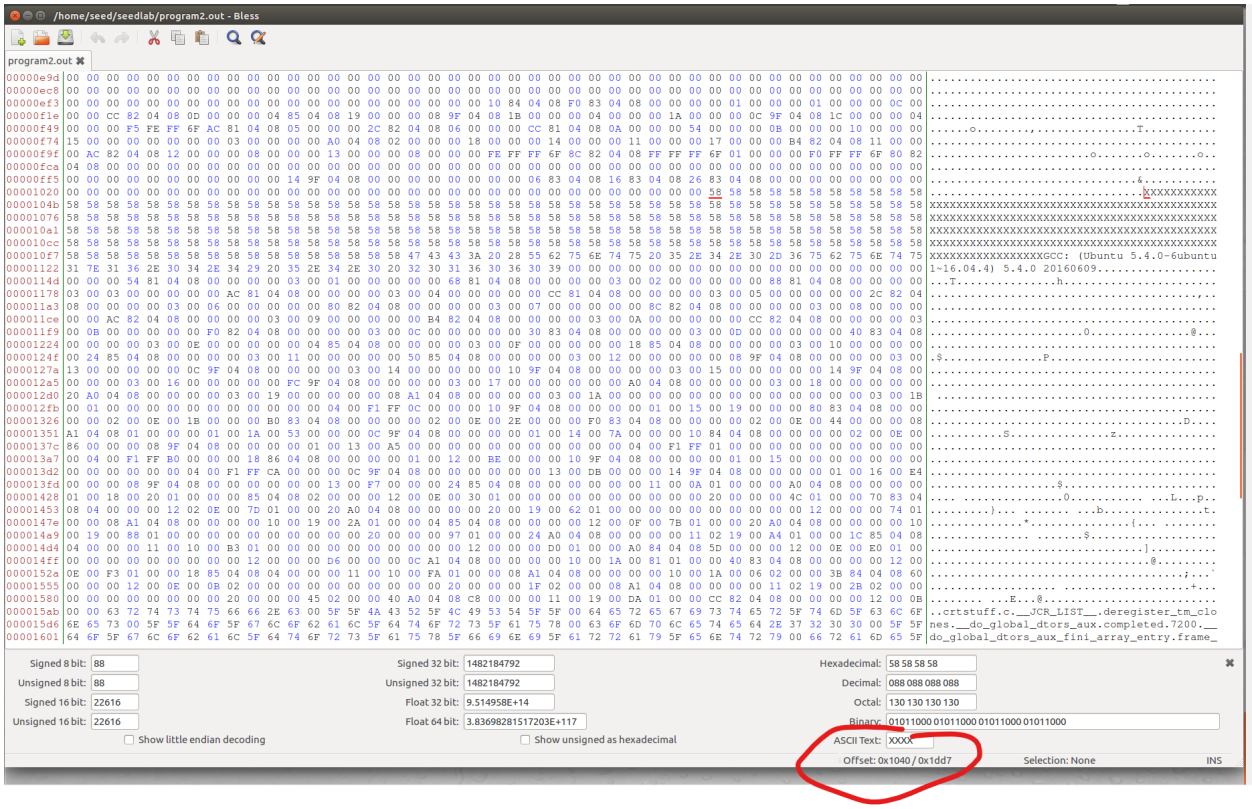


Compiled and ran the second program. The program is printing the hex value of x (58), as expected.



Guidelines. From inside the array, we can find two locations, from where we can divide the executable file into three parts: a prefix , a 128-byte region, and a suffix . The length of the prefix needs to be multiple of 64 bytes. See Figure 3 for an illustration of how the file is divided.

Next, I compiled the above program2, and used the bless hex editor to identify the starting point of the array in the binary. Location is 0x1040 (circled in red) which is 4160 bytes in decimal.



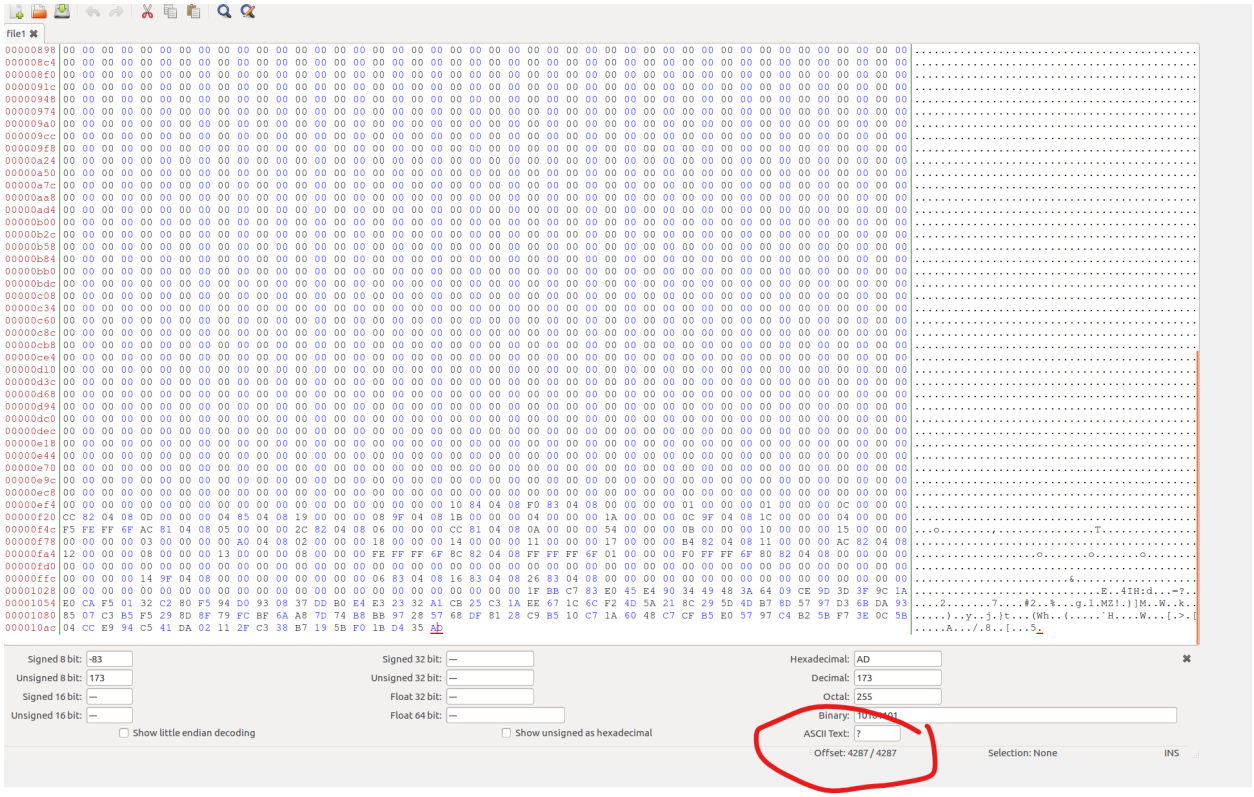
Since the prefix needs to be a multiple of 64 bytes, we can perform the following arithmetic.
(0th byte) → (starting point of prefix)
(4160 byte) →(starting point of array, where the prefix ends)
(4160 bytes + 128 bytes) → 4288 byte (this is the start of the suffix)
(128 byte region is in green)
(4224 byte) → (starting point of the suffix)
This is a more accurate depiction of what's going on. I shifted the 128byte region to the start of the array.

Figure 3: Break the executable file into three pieces.

Now, I can run the following commands to store the prefix and suffix in a new file. Then we can use the prefix file in the algorithm to generate two files, one and two. Hence, both hashes are the same.

```
[11/11/22]seed@VM:~/seedlab$ head -c 4160 program2.out > prefix
[11/11/22]seed@VM:~/seedlab$ tail -c +4288 program2.out > suffix
[11/11/22]seed@VM:~/seedlab$ md5collgen -p prefix -o file1 file2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'file1' and 'file2'
Using prefixfile: 'prefix'
Using initial value: 429438632e79a046740cfe8bda512a06

Generating first block: ............
Generating second block: W.......
Running time: 21.2008 s
[11/11/22]seed@VM:~/seedlab$ md5sum file1 file2
4619ab8682ea5a9b3d68d01726c8ff98  file1
4619ab8682ea5a9b3d68d01726c8ff98  file2
[11/11/22]seed@VM:~/seedlab$
```

Notice if we look at the contents of either file, we get 4288 bytes



Here, we are adding the tail of program2.out to file1 and file2. I also added executable permissions to both files using chmod.

```
[11/11/22]seed@VM:~/seedlab$ cat file1 suffix > newFile1
[11/11/22]seed@VM:~/seedlab$ cat file2 suffix > newFile2
[11/11/22]seed@VM:~/seedlab$ chmod +x newFile1 newFile2
[11/11/22]seed@VM:~/seedlab$
```

We now check both hashes after adding the suffix, and we can see that the hashes are still the same. However, after running cmp, we can see that the contents in the files are different. This is primarily due to the fact that I modified the contents of the array in one file.

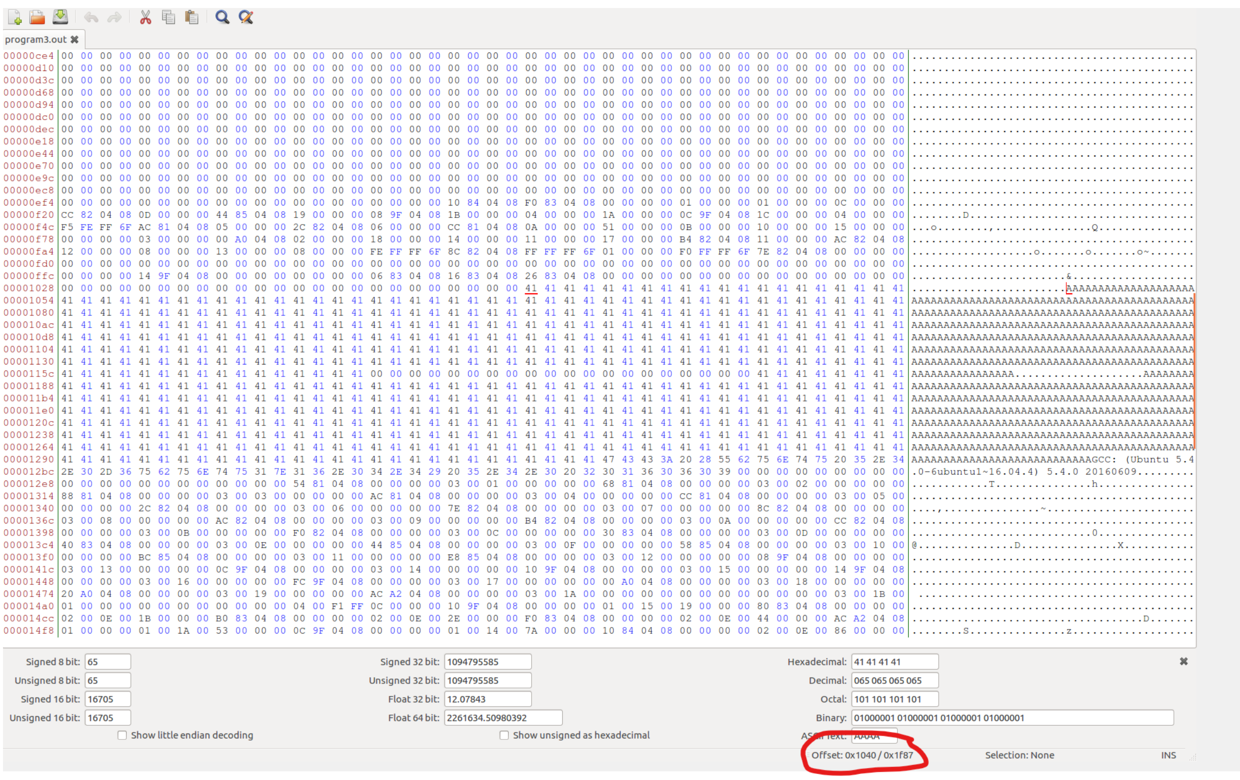
```
[11/11/22]seed@VM:~/seedlab$ md5sum newFile1 newFile2
1086dba825644b49642381165fa0398f  newFile1
1086dba825644b49642381165fa0398f  newFile2
[11/11/22]seed@VM:~/seedlab$ ./newFile1
a699e02c24d21e6e75d553138bff8ebaacdd3190143294b54027addce1fc174fa913edcc45ff8eeff9cbc2bfac33d93d1a1fab23d9befba69b51450a8a7866f96cfea8869c3cdd9176be248ddc598514bc79463b
ac2a892a884a89a5bfbd9f37f8acd9d48de23fda18e7cad7df99335ec2531d82d382e4bcb0c4f8585858585858585858585858585858585858585858585858585858585858585858585858585858585858585858
585858585858585858585858585858585858585858585858585858
[11/11/22]seed@VM:~/seedlab$ ./newFile2
a699e02c24d21e6e75d553138bff8ebaacdd3110143294b54027addce1fc174fa913edcc45ff8eeff9cbc2b7ad33d93d1a1fab23d9befba69351450a8a7866f96cfea8869c3cdd9176be248ddc598514b479463b
ac2a892a884a89a5bfbd9f37f8acd9d48de23fda18efcac7df99335ec2531d82d38264bcb0c4f8585858585858585858585858585858585858585858585858585858585858585858585858585858585858585858
585858585858585858585858585858585858585858585858585858
[11/11/22]seed@VM:~/seedlab$ cmp --silent newFile1 newFile2 && echo '### SUCCESS: Files Are Identical! ###' || echo '### WARNING: Files Are Different! ###'
### WARNING: Files Are Different! ###
[11/11/22]seed@VM:~/seedlab$
```

2.4 Task 4: Making the Two Programs Behave Differently

Here is the program I created. It loops through the values in the array, if equivalent, it runs the benign code, otherwise it runs the malicious code.

```c
#include <stdio.h>

unsigned char A[300] = {
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};

unsigned char B[300] = {
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"};

int main()
{
    for (int index = 0; index < 300; index++)
    {
        if (A[index] != B[index])
        {
            printf("index = %d, A[index] = %.2x, B[index] = %.2x\n", index, A[index], B[index]);
            printf("Malicious code would run here\n");
            return 0;
        }
    }
    printf("Benign code would run here\n");
    return 0;
}
```

Finding the starting byte of the compiled program (0x1040)



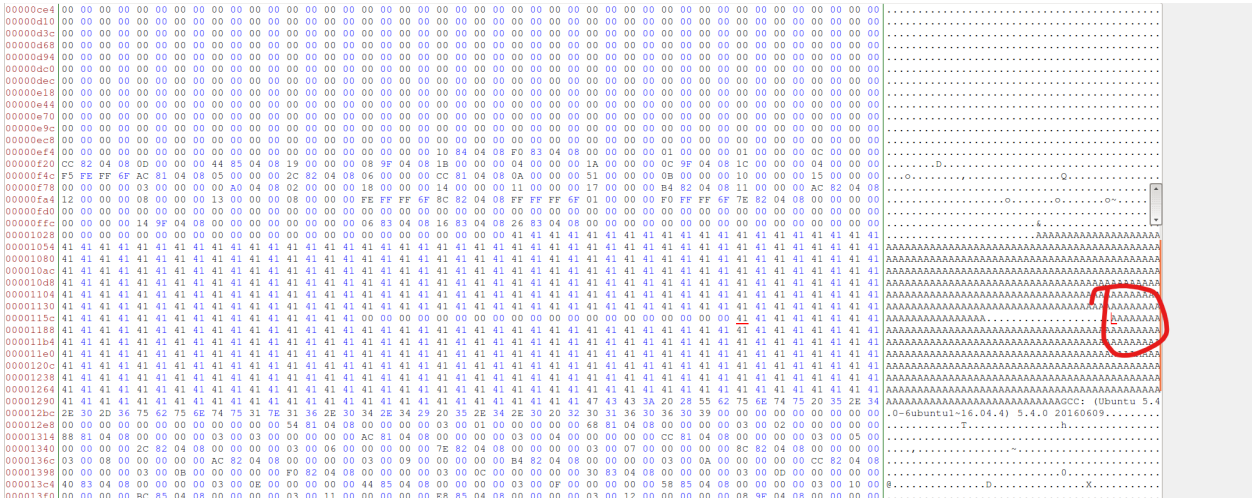
Dividing the executable into two parts:

-A prefix up to the 4160byte

-A suffix passing the 128byte region(byte 4288), all the way to the end of file

-Now I can take run the MD5 hashing algorithm to add a p and q to file1 and file2
-Then I can store the p and q (additional 128bytes) into files named p and q
-hashes are the same

Identifying where the B array starts in suffix file (starts at byte 192



Now, I am splitting the B array into a (pre-suffix → 128byte region left in middle → post-suffix)

We will replace red area (red area = File1+ pre-suffix)
(pre suffix is the space between array A and array B)(file 1 is everything before the pre-suffix, same for file2)
(pos-suffix is the blue area)
(Top is going to be benign, bottom is going to be malicious)

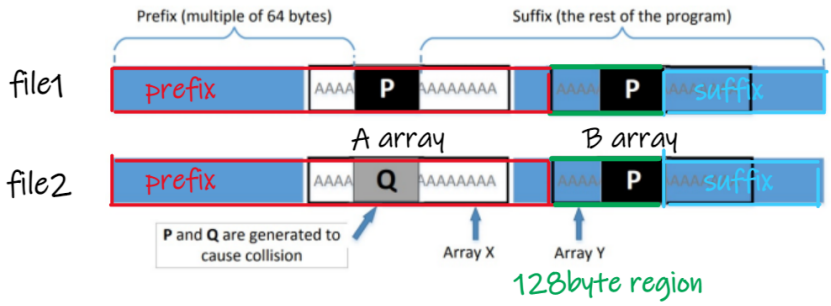
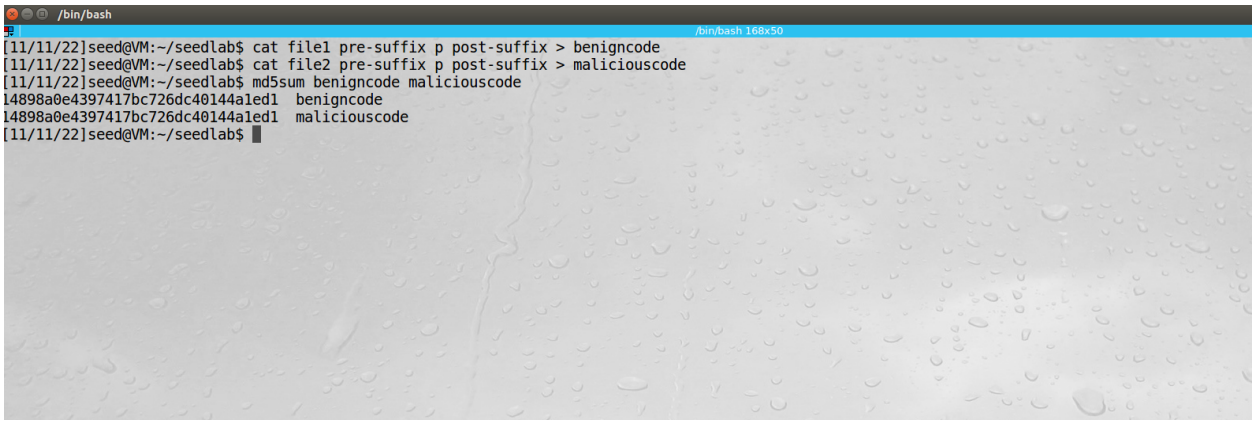


Figure 4: An approach to generate two hash-colliding programs with different behaviors.

Now, I appended the following files to generate the benign code, and the malicious code. File1 contains P, and file2 contains Q. This gives us the above programs in the same order.



Lastly, I added executable permissions to run both files.The benigncode executed the benign block, and the maliciouscode executed the malicious block.


[11/11/22]seed@VM:~/seedlab$ chmod +x benigncode maliciouscode
[11/11/22]seed@VM:~/seedlab$ ./program2.out
Benign code would run here
[11/11/22]seed@VM:~/seedlab$ ./benigncode
Benign code would run here
[11/11/22]seed@VM:~/seedlab$ ./maliciouscode
index = 19, A[index] = 10, B[index] = 90
Malicious code would run here
[11/11/22]seed@VM:~/seedlab$