

Name: Deep Pawar(A20545137)
Professor: Yousef Elmehdwi
Institute: Illinois Institute of Technology

CS 458: Introduction to Information Security

Spring 2024 – Lab 3 MD5 Collision Attack Lab

1. Introduction

A secure one-way hash function needs to satisfy two properties: the one-way property and the collision resistance property. The one-way property ensures that given a hash value h , it is computationally infeasible to find an input M , such that $\text{hash}(M) = h$. The collision-resistance property ensures that it is computationally infeasible to find two different inputs M_1 and M_2 , such that $\text{hash}(M_1) = \text{hash}(M_2)$.

Several widely used one-way hash functions have trouble maintaining the collision-resistance property. At the rump session of CRYPTO 2004, Xiaoyun Wang and co-authors demonstrated a collision attack against MD5². In February 2017, CWI Amsterdam and Google Research announced the SHattered attack, which breaks the collision-resistance property of SHA-1³.

While many students do not have trouble understanding the importance of the one-way property, they can- not easily grasp why the collision-resistance property is necessary, and what impact these attacks can cause.

The learning objective of this lab is for students to really understand the impact of collision attacks, and see in firsthand what damages can be caused if a widely-used one-way hash function's collision-resistance property is broken. To achieve this goal, students need to launch actual collision attacks against the MD5 hash function. Using the attacks, students should be able to create two different programs that share the same MD5 hash but have completely different behaviors. This lab covers a number of topics described in the following:

- One-way hash function, MD5
- The collision-resistance property
- Collision attacks

Lab Environment.

The lab uses a tool called “Fast MD5 Collision Generation”, which was written by Marc Stevens; the name of the binary is called md5collgen in our VMs. md5collgen has already been installed inside /home/seed/bin

2. Lab Tasks

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

In this task, we will generate two different files with the same MD5 hash values. The beginning parts of these two files need to be the same, i.e., they share the same prefix. We can achieve this using the `md5collgen` program, which allows us to provide a prefix file with any arbitrary content. The way how the program works is illustrated in Figure 1.

The following command generates two output files, `out1.bin` and `out2.bin`, for a given a prefix file `prefix.txt`:

```
$ md5collgen -p prefix.txt -o out1.bin out2.bin
```



Figure 1: MD5 collision generation from a prefix

We can check whether the output files are distinct or not using the `diff` command. We can also use the `md5sum` command to check the MD5 hash of each output file. See the following commands.

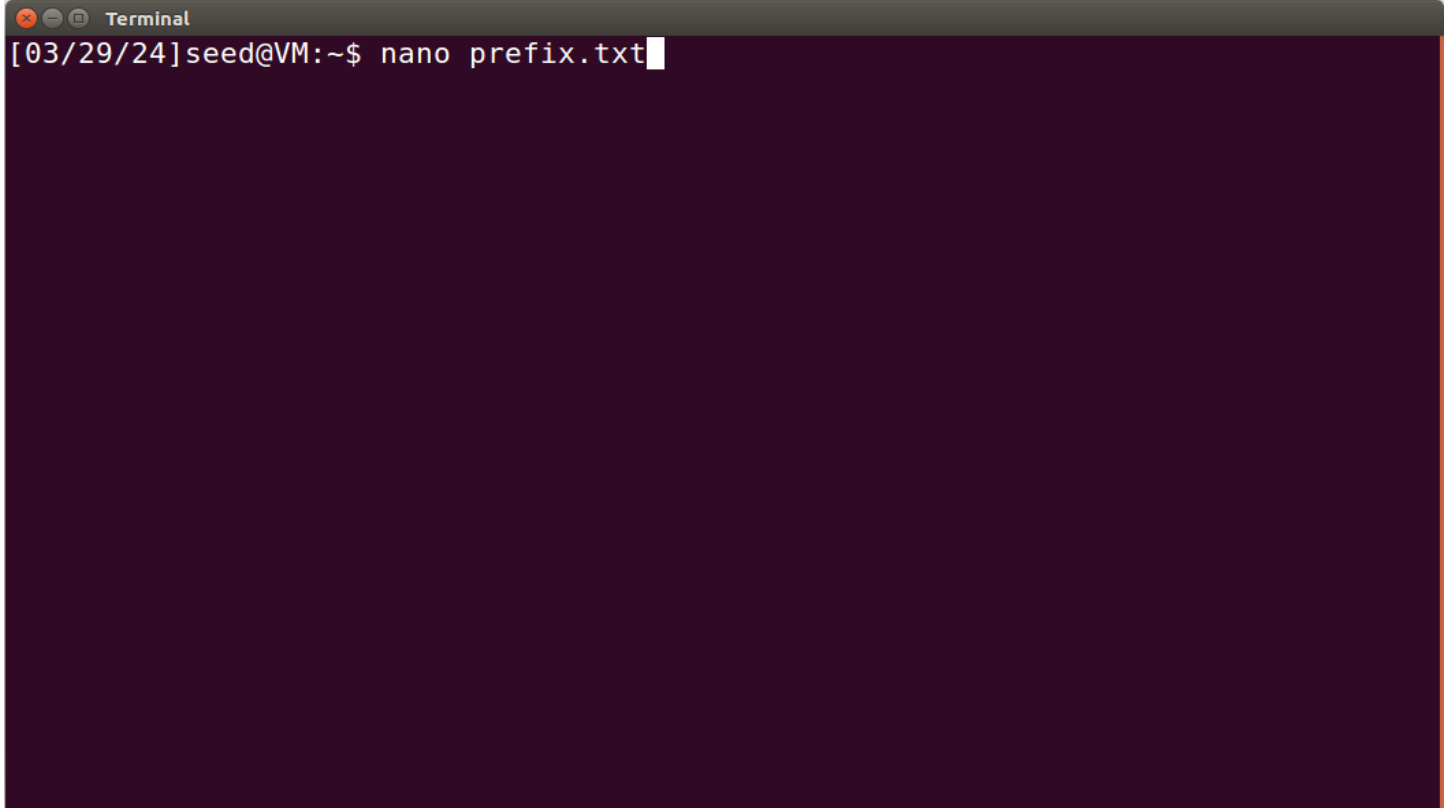
```
$ diff out1.bin out2.bin
$ md5sum out1.bin
$ md5sum out2.bin
```

Since `out1.bin` and `out2.bin` are binary, we cannot view them using a text-viewer program, such as `cat` or `more`; we need to use a binary editor to view (and edit) them. We have already installed a hex editor software called `bleess` in our VM. Please use such an editor to view these two output files, and describe your observations. In addition, you should answer the following questions:

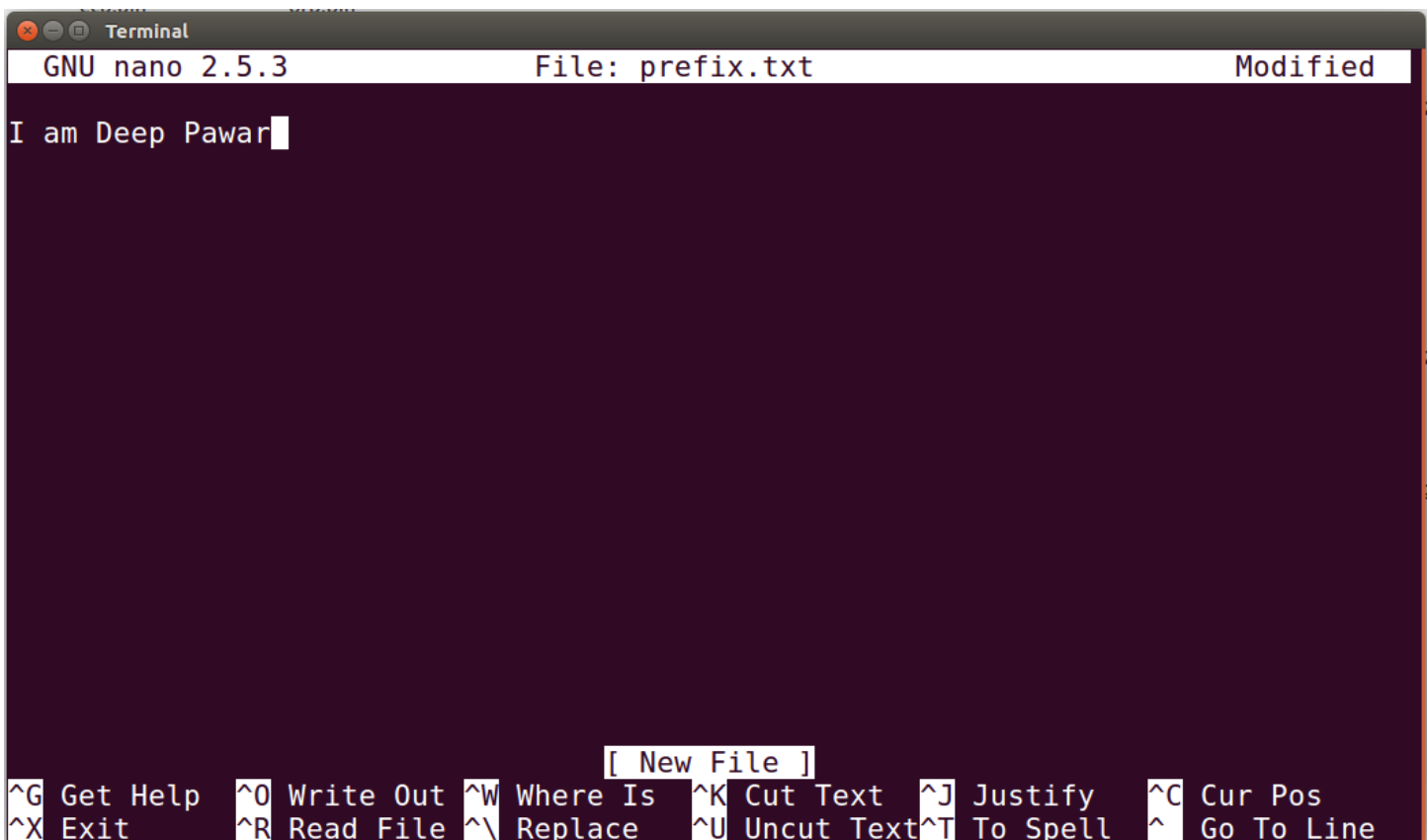
- **Question 1.** If the length of your prefix file is not multiple of 64, what is going to happen?
- **Question 2.** Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.
- **Question 3.** Are the data (128 bytes) generated by `md5collgen` completely different for the two output files? Please identify all the bytes that are different.

Ans:

- Creating a prefix file with the name prefix.txt using the **nano** command.



```
Terminal
[03/29/24]seed@VM:~$ nano prefix.txt
```



```
Terminal
GNU nano 2.5.3      File: prefix.txt      Modified
I am Deep Pawar

[ New File ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text ^T To Spell  ^_ Go To Line
```

- Generating two different files having the same MD5 hash values named out1.bin and out2.bin by using the **md5collgen** command.

```

Terminal
[03/29/24]seed@VM:~$ cat prefix.txt
I am Deep Pawar
[03/29/24]seed@VM:~$ md5collgen -p prefix.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: 'prefix.txt'
Using initial value: b4e9f045e6a9485938975442423a25c5

Generating first block: .....
Generating second block: S00.....
Running time: 36.7609 s
[03/29/24]seed@VM:~$ █

```

- Use the **diff** command to check whether the generated 2 output files are distinct or not.

```

Terminal
[03/29/24]seed@VM:~$ cat prefix.txt
I am Deep Pawar
[03/29/24]seed@VM:~$ md5collgen -p prefix.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

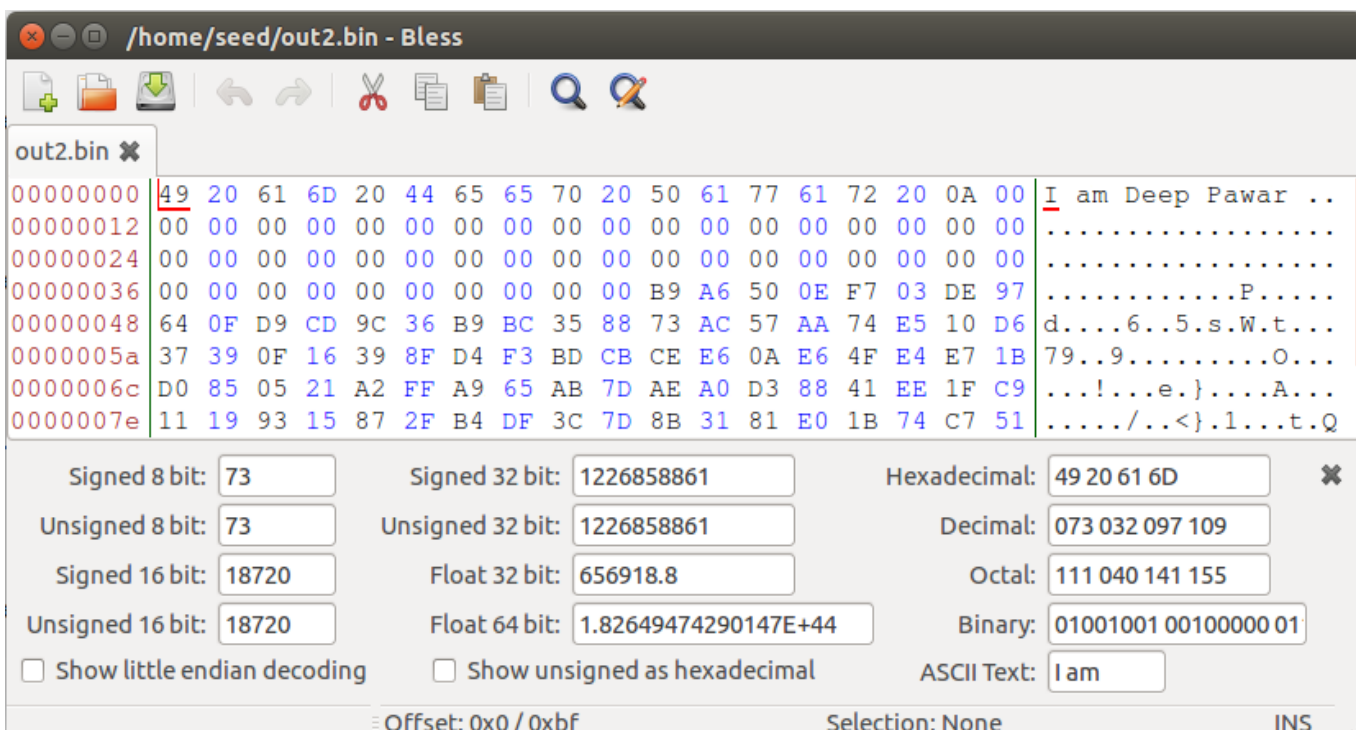
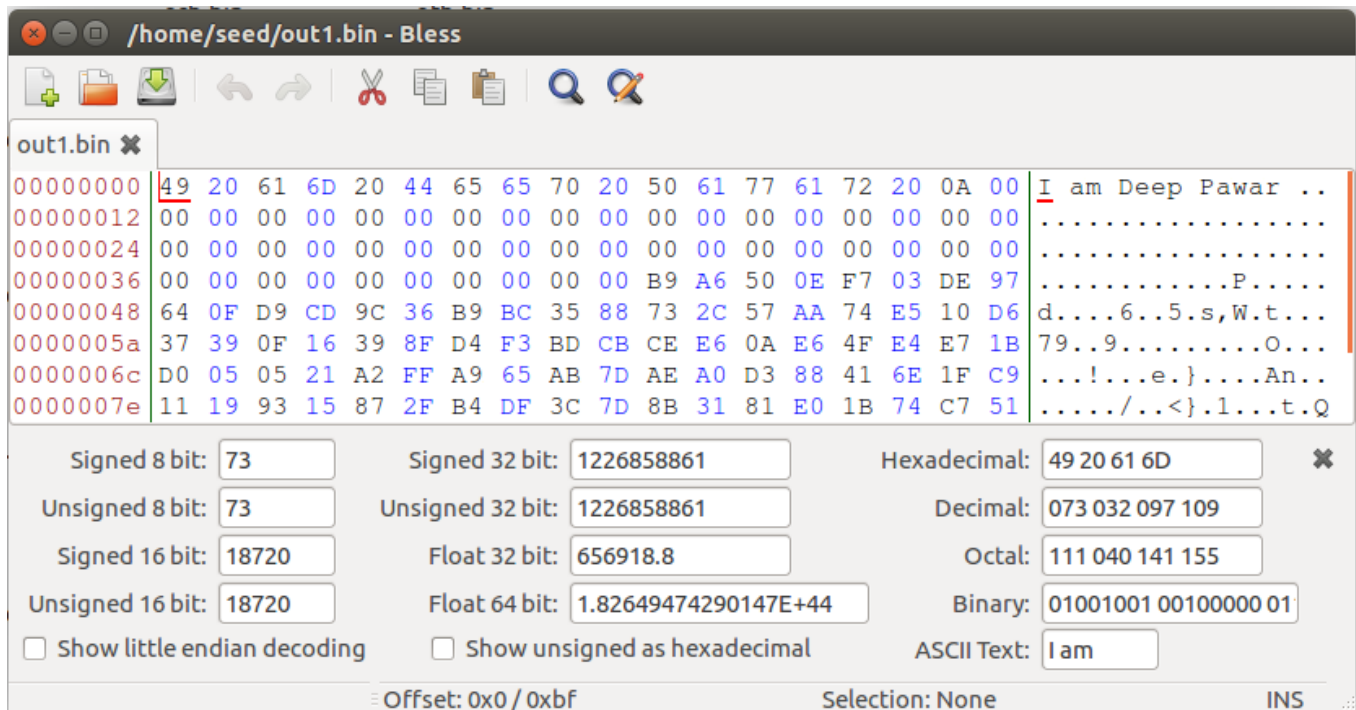
Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'prefix.txt'
Using initial value: b4e9f045e6a9485938975442423a25c5

Generating first block: .....
Generating second block: S00.....
Running time: 36.7609 s
[03/29/24]seed@VM:~$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
[03/29/24]seed@VM:~$ █

```

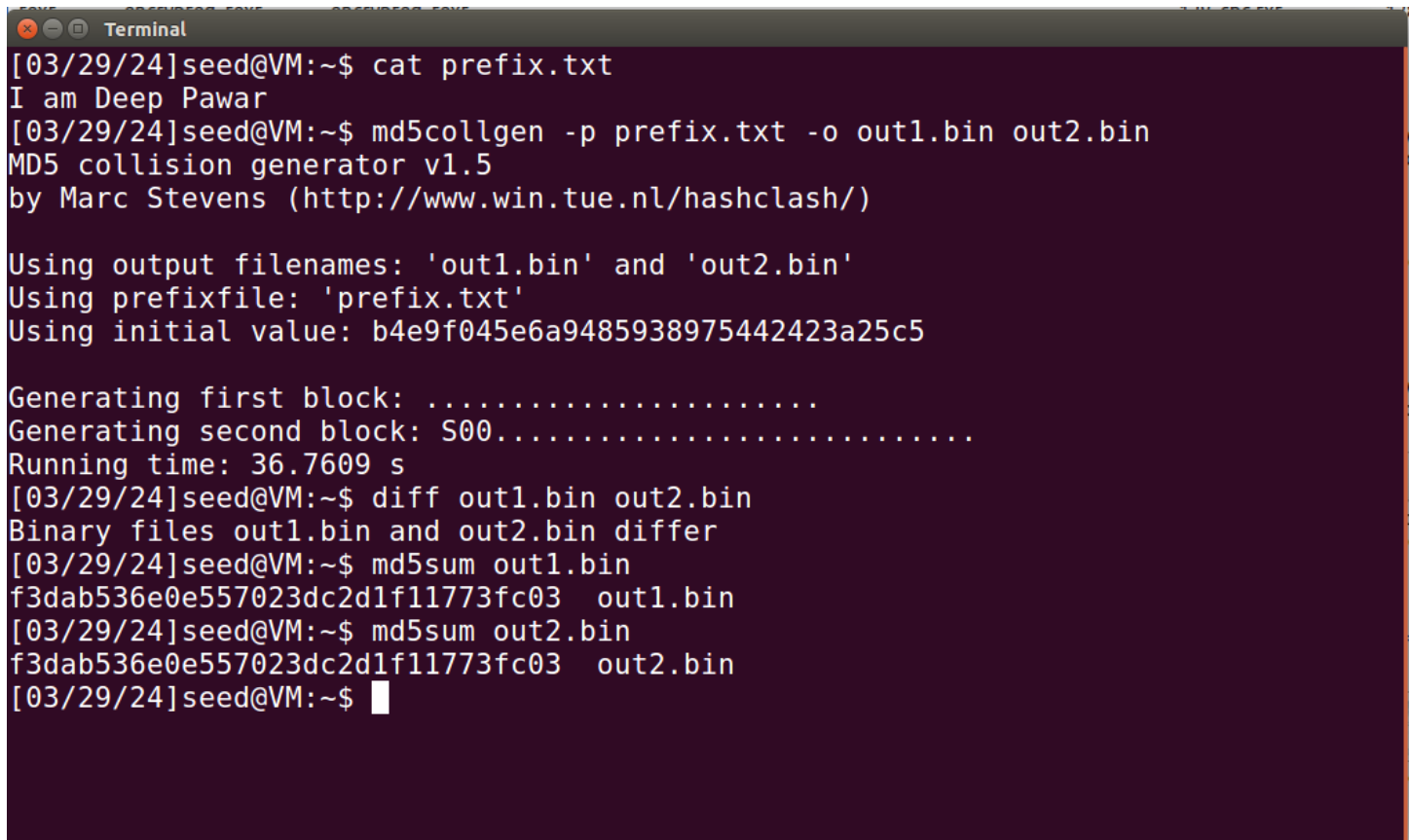
As shown above, the generated two binary files are distinct.

- Opening the two output files out1.bin and out2.bin using the bless editor to check whether the content of the two output files is same or not.



From the images above, we can observe that the content of two output files out1.bin and out2.bin are same.

- Use the md5sum command to check whether the MD5 hash values are the same.

A terminal window titled "Terminal" with a dark background and light-colored text. The user 'seed' is at a VM. The terminal shows the following commands and output:

```
[03/29/24]seed@VM:~$ cat prefix.txt
I am Deep Pawar
[03/29/24]seed@VM:~$ md5collgen -p prefix.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: 'prefix.txt'
Using initial value: b4e9f045e6a9485938975442423a25c5

Generating first block: .....
Generating second block: S00.....
Running time: 36.7609 s
[03/29/24]seed@VM:~$ diff out1.bin out2.bin
Binary files out1.bin and out2.bin differ
[03/29/24]seed@VM:~$ md5sum out1.bin
f3dab536e0e557023dc2d1f11773fc03  out1.bin
[03/29/24]seed@VM:~$ md5sum out2.bin
f3dab536e0e557023dc2d1f11773fc03  out2.bin
[03/29/24]seed@VM:~$
```

As shown above, md5 hash values for both files after concatenation are the same.

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

Answer:

- Creating a `prefix_demo.txt` file containing 45 bytes and generating two different files having the same MD5 hash values using the `md5collgen` command.

```

[03/30/24]seed@VM:~$ echo "This is Introduction to Information Security">>prefix_demo.txt
[03/30/24]seed@VM:~$ md5collgen -p prefix_demo.txt -o outp1.bin outp2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'outp1.bin' and 'outp2.bin'
Using prefixfile: 'prefix_demo.txt'
Using initial value: 963509145c0626c56a6378bbf544c249

Generating first block: .....
Generating second block: S01.
Running time: 21.7913 s
[03/30/24]seed@VM:~$ diff outp1.bin outp2.bin
Binary files outp1.bin and outp2.bin differ
[03/30/24]seed@VM:~$ md5sum outp1.bin
127ed2a324e910453c926557e884f4cf outp1.bin
[03/30/24]seed@VM:~$ md5sum outp2.bin
127ed2a324e910453c926557e884f4cf outp2.bin
[03/30/24]seed@VM:~$

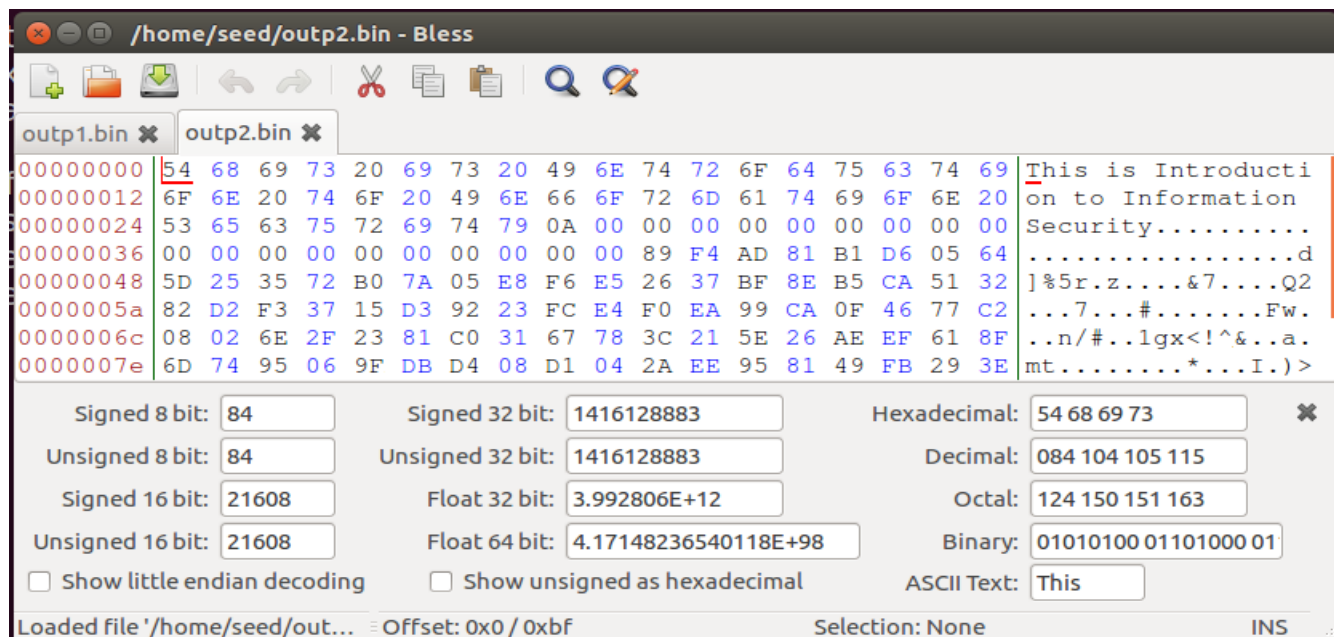
```

- Opening the generated files using bless editor.

outp1.bin ✕

00000000	54	68	69	73	20	69	73	20	49	6E	74	72	6F	64	75	63	74	69	This is Introducti
00000012	6F	6E	20	74	6F	20	49	6E	66	6F	72	6D	61	74	69	6F	6E	20	on to Information
00000024	53	65	63	75	72	69	74	79	0A	00	00	00	00	00	00	00	00	00	Security.....
00000036	00	00	00	00	00	00	00	00	00	89	F4	AD	81	B1	D6	05	64d	
00000048	5D	25	35	72	B0	7A	05	E8	F6	E5	26	B7	BF	8E	B5	CA	51	32]5r.z....&.....Q2
0000005a	82	D2	F3	37	15	D3	92	23	FC	E4	F0	EA	99	CA	0F	46	77	C2	...7...#.....Fw.
0000006c	08	82	6D	2F	23	81	C0	31	67	78	3C	21	5E	26	AE	6F	61	8F	..m/#...lgx<!^&.oa.
0000007e	6D	74	95	06	9F	DB	D4	08	D1	04	2A	EE	95	81	49	FB	29	3E	mt.....*....I.)>

Signed 8 bit:	84	Signed 32 bit:	1416128883	Hexadecimal:	54 68 69 73
Unsigned 8 bit:	84	Unsigned 32 bit:	1416128883	Decimal:	084 104 105 115
Signed 16 bit:	21608	Float 32 bit:	3.992806E+12	Octal:	124 150 151 163
Unsigned 16 bit:	21608	Float 64 bit:	4.17148236540118E+98	Binary:	01010100 01101000 01
<input type="checkbox"/> Show little endian decoding		<input type="checkbox"/> Show unsigned as hexadecimal		ASCII Text: This	
Offset: 0x0 / 0xbf				Selection: None	
INS					



As shown in the example above, the prefix file has 45 bytes which is not a multiple of 64 so to satisfy this criterion, the md5collgen tool will pad the prefix file with zeros to make it a multiple of 64 bytes. The length of the input can be adjusted to a certain size using a standard approach in cryptographic protocols called padding. If the prefix file's length is less than or equal to 64 then md5collgen may add a few extra bytes to the end of the file. Because for the MD5 method to process data properly, any input must be padded to a length multiple of the block size, since the algorithm processes data in 512-bit blocks or 64 bytes.

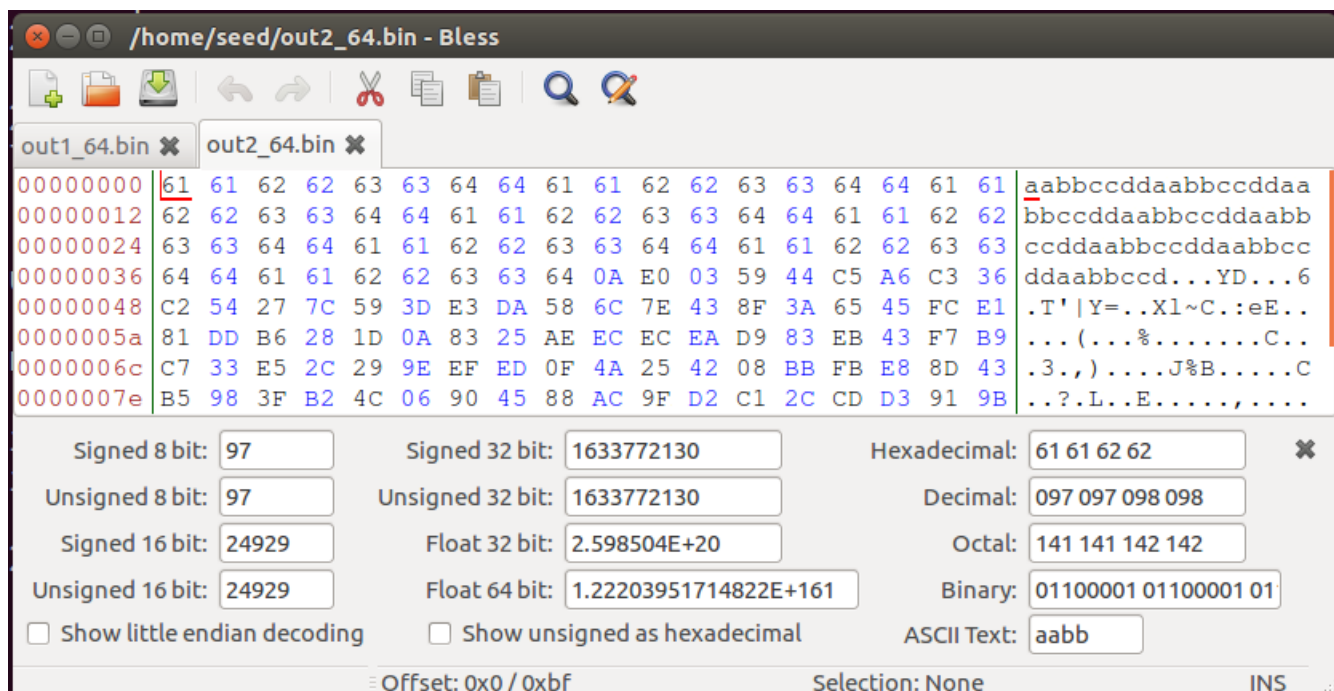
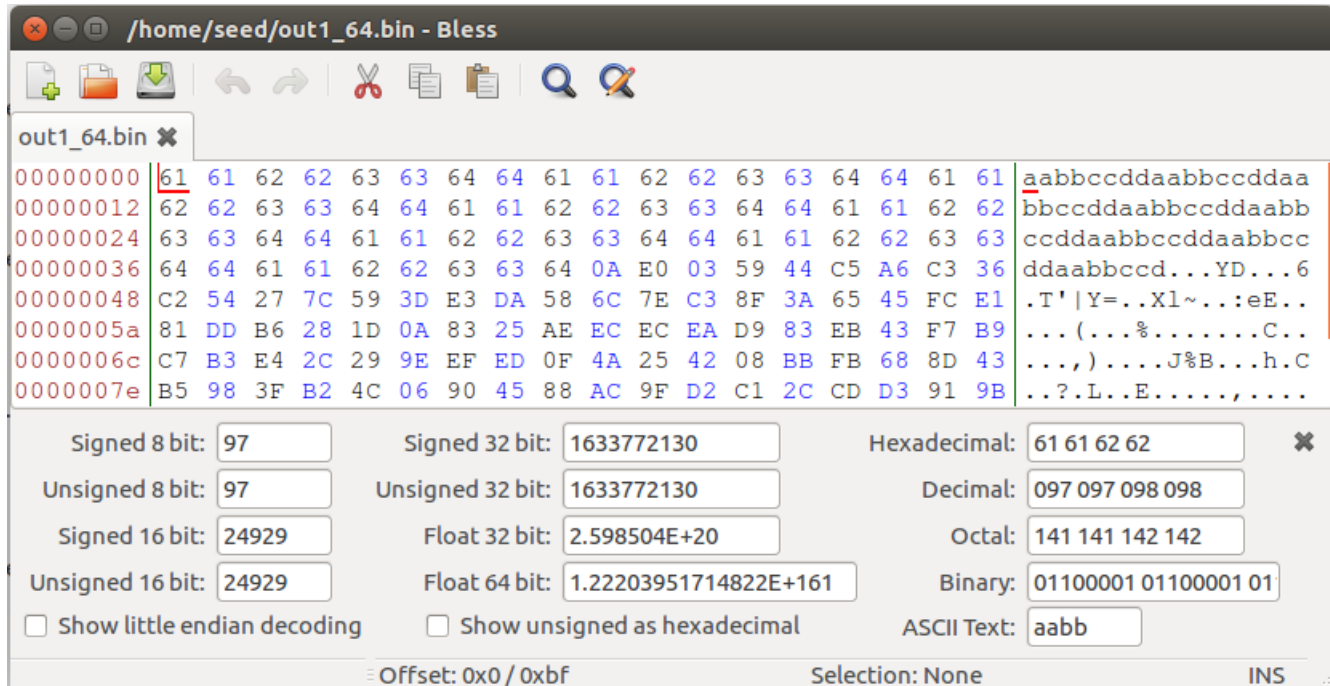
Question 2: Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.

Answer:

- Creating a `prefix64.txt` file containing 64 bytes and generating two different files having the same MD5 hash values using the **md5collgen** command.

```
[03/29/24]seed@VM:~$ echo "aabbbccddaabbccddaabbccddaabbccddaabbccddaabbc  
cddaaabbccd">>prefix64.txt  
[03/29/24]seed@VM:~$ ls -l prefix64.txt  
-rw-rw-r-- 1 seed seed 64 Mar 29 14:41 prefix64.txt  
[03/29/24]seed@VM:~$ md5collgen -p prefix64.txt -o out1_64.bin out2_64.bin  
MD5 collision generator v1.5  
by Marc Stevens (http://www.win.tue.nl/hashclash/)  
  
Using output filenames: 'out1_64.bin' and 'out2_64.bin'  
Using prefixfile: 'prefix64.txt'  
Using initial value: 8labfe18b47d14c47e801280a07d3263  
  
Generating first block: ..  
Generating second block: S01.....  
Running time: 1.6176 s  
[03/29/24]seed@VM:~$ █
```


- Opening the generated files using bless editor.



As shown above, the block size requirements of MD5 are already satisfied when a prefix file is created with precisely 64 bytes. So, as a result, the md5collgen did not add any extra padding. The file content is followed by some random data which will later be used for collision.

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.

Answer:

No, it is not necessarily true to say that the data generated by md5collgen for the two output files (out1_128.bin and out2_128.bin) are completely different. As shown below we can see that both the output files are relatively similar. However, there are some byte differences between them which are identified and highlighted below in the bless editor.

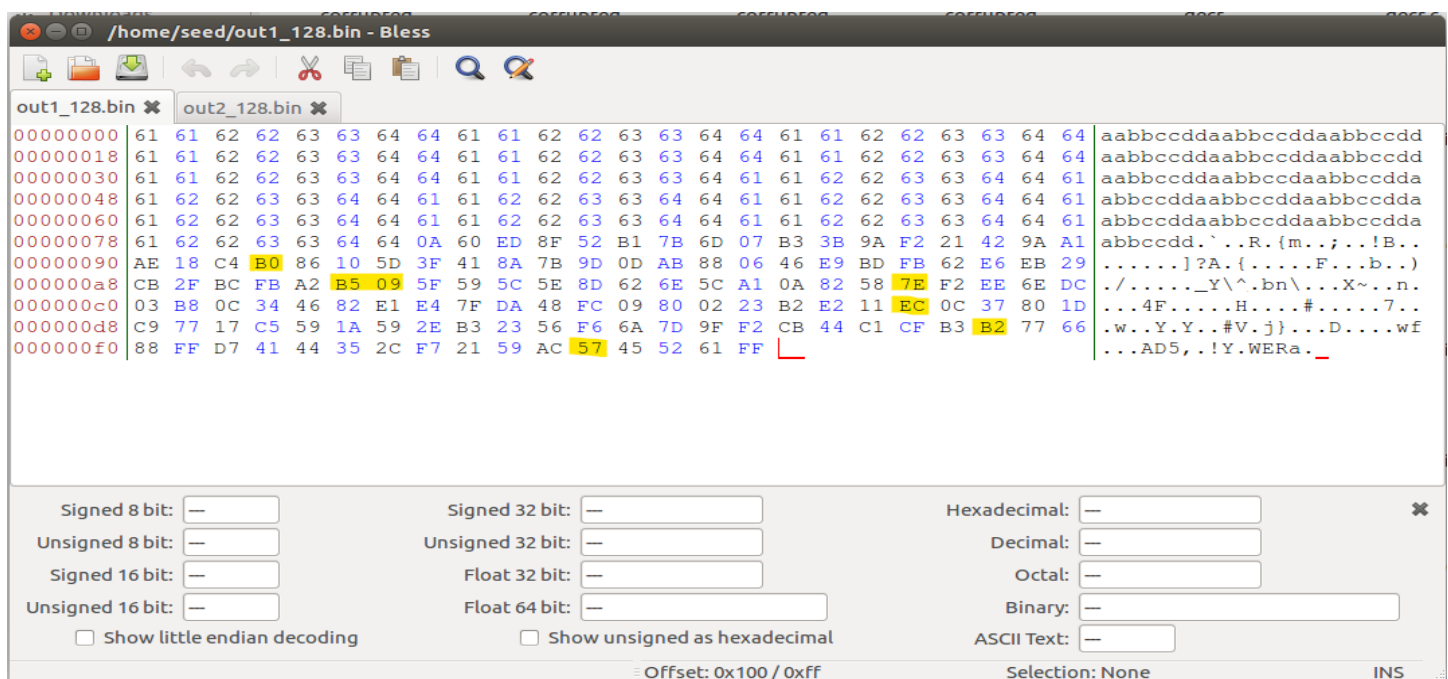
- Creating a `prefix128.txt` file containing 128 bytes and generating two different files having the same MD5 hash values using the **md5collgen** command.

```
[03/29/24]seed@VM:~$ echo "aabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccdaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccddaabbccdd">>prefix128.txt
[03/29/24]seed@VM:~$ ls -l prefix128.txt
-rw-rw-r-- 1 seed seed 128 Mar 29 14:55 prefix128.txt
[03/29/24]seed@VM:~$ md5collgen -p prefix128.txt -o out1_128.bin out2_128.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1_128.bin' and 'out2_128.bin'
Using prefixfile: 'prefix128.txt'
Using initial value: cece049150542ed43f2c30db6879222f

Generating first block: ...
Generating second block: S00.....
Running time: 4.9299 s
[03/29/24]seed@VM:~$
```

- Opening the generated files using `bleed` editor and highlighting all the bytes that are different.



/home/seed/out2_128.bin - Bless

out1_128.bin ✕ out2_128.bin ✕

00000000	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	aabbccddaabbccddaabbccdd
00000018	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	aabbccddaabbccddaabbccdd
00000030	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	aabbccddaabbccddaabbccdda
00000048	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	abbccddaabbccddaabbccdda
00000060	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	61	62	62	63	63	64	64	61	abbccddaabbccddaabbccdda
00000078	61	62	62	63	63	64	64	0A	60	ED	8F	52	B1	7B	6D	07	B3	3B	9A	F2	21	42	9A	A1	abbccdd.`..R.{m.;..!B..
00000090	AE	18	C4	30	86	10	5D	3F	41	8A	7B	9D	0D	AB	88	06	46	E9	BD	FB	62	E6	EB	29	...0..]?A.{.....F...b..)
000000a8	CB	2F	BC	FB	A2	35	0A	5F	59	5C	5E	8D	62	6E	5C	A1	0A	82	58	FE	F2	EE	6E	DC	./...5._Y^\.bn\...X...n.
000000c0	03	B8	0C	34	46	82	E1	E4	7F	DA	48	FC	09	80	02	23	B2	E2	11	6C	0C	37	80	1D	...4F.....H....#...1.7..
000000d8	C9	77	17	C5	59	1A	59	2E	B3	23	56	F6	6A	7D	9F	F2	CB	44	C1	CF	B3	32	77	66	.w..Y.Y...#V.j}...D...2wf
000000f0	88	FF	D7	41	44	35	2C	F7	21	59	AC	D7	45	52	61	FF									...AD5,!!Y...ERa.

Signed 8 bit:	97	Signed 32 bit:	1633772130	Hexadecimal:	61 61 62 62	✕
Unsigned 8 bit:	97	Unsigned 32 bit:	1633772130	Decimal:	097 097 098 098	
Signed 16 bit:	24929	Float 32 bit:	2.598504E+20	Octal:	141 141 142 142	
Unsigned 16 bit:	24929	Float 64 bit:	1.22203951714822E+161	Binary:	01100001 01100001 01100010 0110	
<input type="checkbox"/> Show little endian decoding		<input type="checkbox"/> Show unsigned as hexadecimal		ASCII Text:	aabb	

Loaded file '/home/seed/out2_128.bin' Offset: 0x0 / 0xff Selection: None INS

2.2 Task 2: Understanding MD5's Property

In this task, we will try to understand some of the properties of the MD5 algorithm. These properties are important for us to conduct further tasks in this lab. MD5 is quite a complicated algorithm, but from very high level, it is not so complicated. As Figure 2 shows, MD5 divides the input data into blocks of 64 bytes, and then computes the hash iteratively on these blocks. The core of the MD5 algorithm is a compression function, which takes two inputs, a 64-byte data block and the outcome of the previous iteration. The compression function produces a 128-bit IHV, which stands for “Intermediate Hash Value”; this output is then fed into the next iteration. If the current iteration is the last one, the IHV will be the final hash value. The IHV input for the first iteration (IHV_0) is a fixed value.

Based on how MD5 works, we can derive the following property of the MD5 algorithm:

- 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 , $MD5(M \parallel T) = MD5(N \parallel T)$, where \parallel 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.

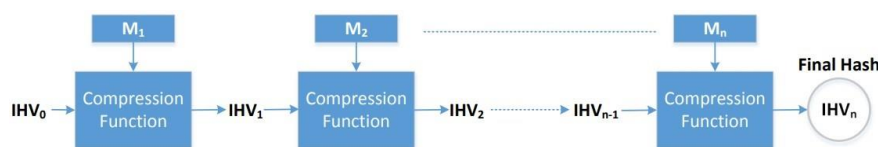


Figure 2: How the MD5 algorithm works

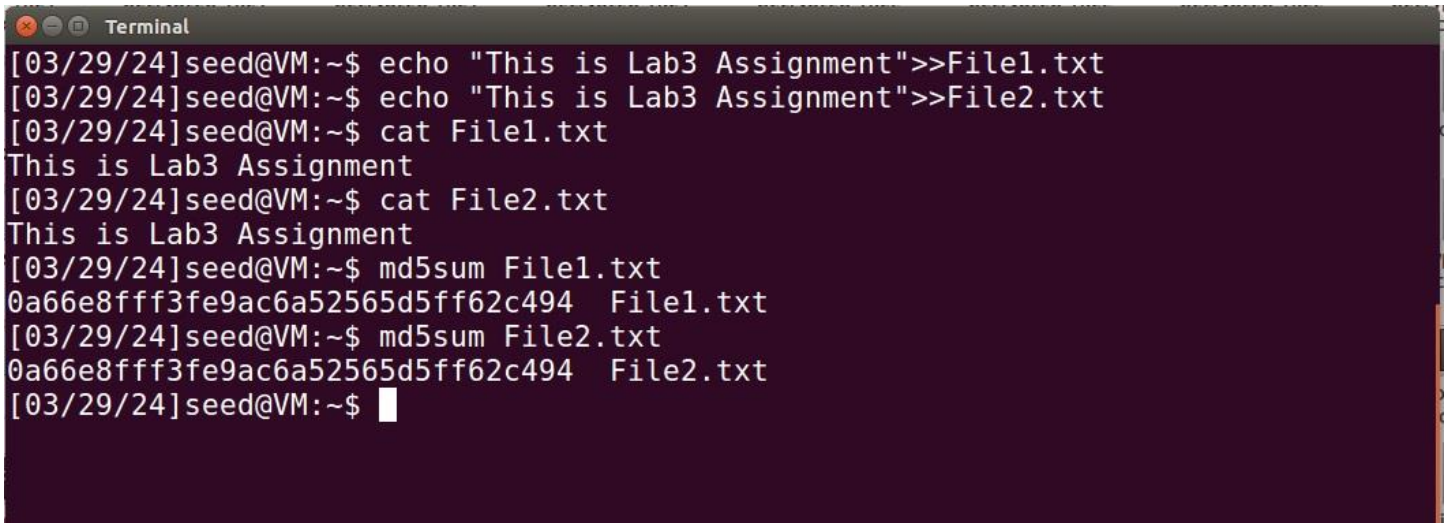
This property holds not only for the MD5 hash algorithm, but also for many other hash algorithms. Your job in this task is to design an experiment to demonstrate that this property holds for MD5.

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`.

```
$ cat file1 file2 > file3
```

Ans:

- Creating two files File1.txt and File2.txt with the same content and checking their MD5 hash values using the **md5sum** command.



```

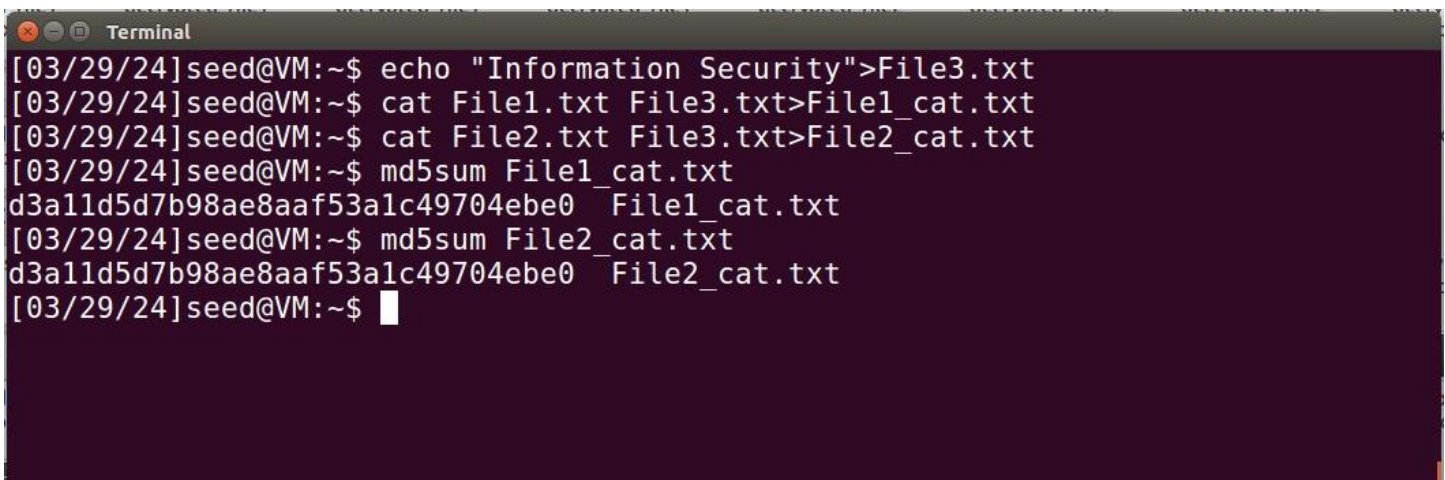
Terminal
[03/29/24]seed@VM:~$ echo "This is Lab3 Assignment">>File1.txt
[03/29/24]seed@VM:~$ echo "This is Lab3 Assignment">>File2.txt
[03/29/24]seed@VM:~$ cat File1.txt
This is Lab3 Assignment
[03/29/24]seed@VM:~$ cat File2.txt
This is Lab3 Assignment
[03/29/24]seed@VM:~$ md5sum File1.txt
0a66e8fff3fe9ac6a52565d5ff62c494  File1.txt
[03/29/24]seed@VM:~$ md5sum File2.txt
0a66e8fff3fe9ac6a52565d5ff62c494  File2.txt
[03/29/24]seed@VM:~$ █

```

As we can see, the md5 hash values of File1.txt and File2.txt are similar.

It proves that **MD5(M) = MD5(N)**, where M and N are two inputs (File1.txt and File2.txt)

- Concatenating both files File1.txt and File2.txt with a file File3.txt containing some information and then checking the md5 hash values using the **md5sum** command.



```

Terminal
[03/29/24]seed@VM:~$ echo "Information Security">File3.txt
[03/29/24]seed@VM:~$ cat File1.txt File3.txt>File1_cat.txt
[03/29/24]seed@VM:~$ cat File2.txt File3.txt>File2_cat.txt
[03/29/24]seed@VM:~$ md5sum File1_cat.txt
d3a11d5d7b98ae8aaf53a1c49704ebe0  File1_cat.txt
[03/29/24]seed@VM:~$ md5sum File2_cat.txt
d3a11d5d7b98ae8aaf53a1c49704ebe0  File2_cat.txt
[03/29/24]seed@VM:~$ █

```

After concatenation, the md5 hash values of both the files File1.txt and File2.txt are still the same as we can see above.

This proves that **MD5(M || T) = MD5(N || T)**, where M, N, and T are three different inputs (File1.txt, File2.txt, and File3.txt)

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.

```
#include <stdio.h> unsigned char
xyz[200]={
    /* The actual contents of this array are up to you */
};

int main()
{
    int i;
    for (i=0; i<200; i++){
        printf("%x", xyz[i]);
    }
    printf("\n");
}
```

You may choose to work at the source code level, i.e., generating two versions of the above C program, such that after compilation, their corresponding executable files have the same MD5 hash value. However, it may be easier to directly work on the binary level. You can put some random values in the `xyz` array, compile the above code to binary. Then you can use a hex editor tool to modify the content of the `xyz` array directly in the binary file.

Finding where the contents of the array are stored in the binary is not easy. However, if we fill the array with some fixed values, we can easily find them in the binary. For example, the following code fills the array with `0x41`, which is the ASCII value for letter `A`. It will not be difficult to locate 200 A's in the binary.

```
unsigned char xyz[200]={
    0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
    0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
    0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
    ... (omitted) ...
    0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
}
```

parts: a prefix, a 128-byte region, and a suffix. The length of the prefix needs to be a multiple of 64 bytes. See Figure 3 for an illustration of how the file is divided.

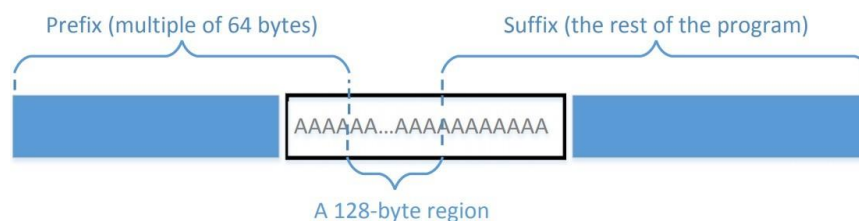


Figure 3: Break the executable file into three pieces.

We can run `md5collgen` on the prefix to generate two outputs that have the same MD5 hash value. Let us use `P` and `Q` to represent the second part (each having 128 bytes) of these outputs (i.e., the part after the prefix). Therefore, we have the following:

$$\text{MD5}(\text{prefix} \parallel P) = \text{MD5}(\text{prefix} \parallel Q)$$

Based on the property of MD5, we know that if we append the same suffix to the above two outputs, the resultant data will also have the same hash value. Basically, the following is true for any suffix :

$\text{MD5}(\text{prefix} \parallel P \parallel \text{suffix}) = \text{MD5}(\text{prefix} \parallel Q \parallel \text{suffix})$

Therefore, we just need to use P and Q to replace 128 bytes of the array (between the two dividing points), and we will be able to create two binary programs that have the same hash value. Their outcomes are different, because they each print out their own arrays, which have different contents. Tools. You can use `hexedit` to view the binary executable file and find the location for the array. For dividing a binary file, there are some tools that we can use to divide a file from a particular location. The

`head` and `tail` commands are such useful tools. You can look at their manuals to learn how to use them.

We give three examples in the following:

```
$ head -c 3200 a.out > prefix
```

```
$ tail -c 100 a.out > suffix
```

```
$ tail -c +3300 a.out > suffix
```

- The `head` command above saves the first 3200 bytes of `a.out` to `prefix`.
- The second command saves the last 100 bytes of `a.out` to `suffix`.
- The third command saves the data from the 3300th byte to the end of the file `a.out` to `suffix`.
- With these two commands, we can divide a binary file into pieces from any location. If we need to glue some pieces together, we can use the `cat` command.

If you use `hexedit` to copy-and-paste a block of data from one binary file to another file, the menu item `\ Edit -> Select Range` is quite handy, because you can select a block of data using a starting point and a range, instead of manually counting how many bytes are selected.

- Executing the above code to generate the output file.

[illegible]

- Opening the output file in bless editor.

/home/seed/output - Bless

output ✕

00000fb4	FE FF FF 6F 8C 82 04 08 FF FF FF 6F 01 00 00 00 F0 FF FF 6Fo.....o.....o
00000fc8	80 82 04 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000fd0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000ff0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 14 9F 04 08
00001004	00 00 00 00 00 00 00 00 06 83 04 08 16 83 04 08 26 83 04 08&...
00001018	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000102c	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001040	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA
00001054	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA
00001068	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA
0000107c	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA
00001090	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA
000010a4	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA
000010b8	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAAAAAA

Signed 8 bit:	65	Signed 32 bit:	1094795585	Hexadecimal:	41 41 41 41	✕
Unsigned 8 bit:	65	Unsigned 32 bit:	1094795585	Decimal:	065 065 065 065	
Signed 16 bit:	16705	Float 32 bit:	12.07843	Octal:	101 101 101 101	
Unsigned 16 bit:	16705	Float 64 bit:	2261634.50980392	Binary:	01000001 01000001 010000	
<input type="checkbox"/> Show little endian decoding		<input type="checkbox"/> Show unsigned as hexadecimal		ASCII Text:		AAAA

Offset: 0x1040 / 0x1dd3 Selection: None INS

As shown above, in the bless editor we identify the offset value of the position where the array is stored which is 0x1040 which is 4160 in decimal.

It is divisible by 64 and is a multiple of 64 bytes as $\frac{4160}{64} = 65$

We can consider the prefix up to 4224 bytes which is 64×66 .

- Then We will save the first 4160 bytes of tsak3 code output to a prefix file named prefix. We will generate two different files (output1.bin and output2.bin) with the same md5 hash value for the prefix file using the **md5collgen** command.

```
Terminal
[03/29/24]seed@VM:~$ head -c 4160 output>prefix
[03/29/24]seed@VM:~$ md5collgen -p prefix -o output1.bin output2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'output1.bin' and 'output2.bin'
Using prefixfile: 'prefix'
Using initial value: b0d00a6c3bbf355033c923c553facf7d

Generating first block: ..
Generating second block: S00...
Running time: 1.81116 s
[03/29/24]seed@VM:~$
```

- Then we consider the suffix of the file to have content from prefix + 128 bytes to the end of the file. So, the starting value of the suffix file is from $4160+128= 4288$ bytes.

```
Terminal
[03/29/24]seed@VM:~$ tail -c +4288 output>suffix
[03/29/24]seed@VM:~$
```

- Then we will use P and Q to represent the second part (each having 128 bytes) of these outputs

```
Terminal
[03/29/24]seed@VM:~$ tail -c 128 output1.bin>p
[03/29/24]seed@VM:~$ tail -c 128 output2.bin>q
[03/29/24]seed@VM:~$
```

- Concatenating prefix, P, Q, and suffix to generate two new files task3_1 and task3_2

```
Terminal
[03/29/24]seed@VM:~$ cat prefix p suffix>task3_1
[03/29/24]seed@VM:~$ cat prefix q suffix>task3_2
[03/29/24]seed@VM:~$
```

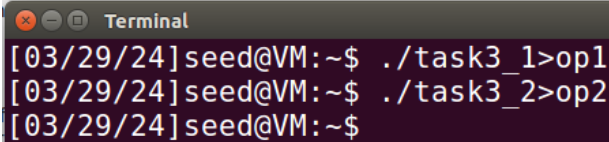
- Applying permissions to execute task3 output, task3_1 and task3_2 files

```
Terminal
[03/29/24]seed@VM:~$ chmod +x output
[03/29/24]seed@VM:~$ ls -l output
-rwxrwxr-x 1 seed seed 7636 Mar 29 18:50 output
[03/29/24]seed@VM:~$ chmod +x task3_1
[03/29/24]seed@VM:~$ ls -l task3_1
-rwxrwxr-x 1 seed seed 7637 Mar 29 21:13 task3_1
[03/29/24]seed@VM:~$ chmod +x task3_2
[03/29/24]seed@VM:~$ ls -l task3_2
-rwxrwxr-x 1 seed seed 7637 Mar 29 21:13 task3_2
[03/29/24]seed@VM:~$
```

- Executing task3 output, task3_1 and task3_2

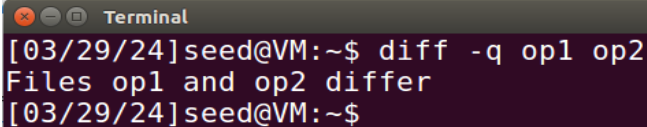
[illegible]

- Storing the above outputs of task3_1 and task3_2 files in two different files named op1 and op2



```
Terminal
[03/29/24]seed@VM:~$ ./task3_1>op1
[03/29/24]seed@VM:~$ ./task3_2>op2
[03/29/24]seed@VM:~$
```

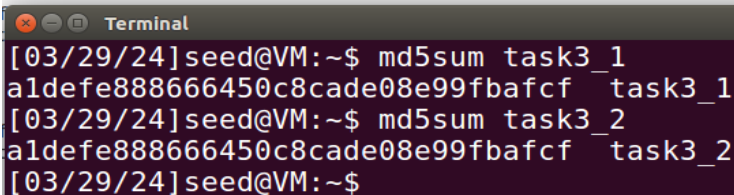
- Comparing both the output files op1 and op2 with the **diff** command



```
Terminal
[03/29/24]seed@VM:~$ diff -q op1 op2
Files op1 and op2 differ
[03/29/24]seed@VM:~$
```

As shown above, the content of file op1 differs from the content of file op2

- Using the **md5sum** command to check whether the MD5 hash values are the same.



```
Terminal
[03/29/24]seed@VM:~$ md5sum task3_1
a1defe888666450c8cade08e99fbafcf task3_1
[03/29/24]seed@VM:~$ md5sum task3_2
a1defe888666450c8cade08e99fbafcf task3_2
[03/29/24]seed@VM:~$
```

As shown above, md5 hash values for both files after concatenation are the same.

- Conclusion:**

So, it is proven that even though the contents of the xyz array provided in the code output task3_1 and tsak3_2 are different, the hash value of their executables is the same as shown above. By simply substituting P and Q for the 128 bytes in the array (between the two division points), we were able to create two binary programs with the same hash value. Still, the programs function differently from one another because each printout its array with distinct content.

2.4 Task 4: Making the Two Programs Behave Differently

In the previous task, we have successfully created two programs that have the same MD5 hash, but their behaviors are different. However, their differences are only in the data they print out; they still execute the same sequence of instructions. In this task, we would like to achieve something more significant and more meaningful.

Assume that you have created software which does good things. You send the software to a trusted authority to get certified. The authority conducts a comprehensive testing of your software and concludes that your software is indeed doing good things. The authority will present you with a certificate, stating that your program is good. To prevent you from changing your program after getting the certificate, the MD5 hash value of your program is also included in the certificate; the certificate is signed by the authority, so you cannot change anything on the certificate or your program without rendering the signature invalid.

You would like to get your malicious software certified by the authority, but you have zero chance to achieve that goal if you simply send your malicious software to the authority. However, you have noticed that the authority uses MD5 to generate the hash value. You got an idea:

- You plan to prepare two different programs. One program will always execute benign instructions and do good things, while the other program will execute malicious instructions and cause damage. You find a way to get these two programs to share the same MD5 hash value.
- You then send the benign version to the authority for certification. Since this version does good things, it will pass the certification, and you will get a certificate that contains the hash value of your benign program. Because your malicious program has the same hash value, this certificate is also valid for your malicious program. Therefore, you have successfully obtained a valid certificate for your malicious program. If other people trusted the certificate issued by the authority, they will download your malicious program.

The objective of this task is to launch the attack described above. Namely, you need to create two programs that share the same MD5 hash. However, one program will always execute benign instructions, while the other program will execute malicious instructions. In your work, what benign/malicious instructions are executed is not important; it is sufficient to demonstrate that the instructions executed by these two programs are different.

Guidelines. Creating two completely different programs that produce the same MD5 hash value is quite hard. The two hash-colliding programs produced by md5collgen need to share the same prefix; moreover, as we can see from the previous task, if we need to add some meaningful suffix to the outputs produced by md5collgen, the suffix added to both programs also needs to be the same. These are the limitations of the MD5 collision generation program that we use. Although there are other more complicated and more advanced tools that can lift some of the limitations, such as accepting two different prefixes⁴, they demand much more computing power, so they are out of the scope for this lab. We need to find a way to generate two different programs within the limitations. There are many ways to achieve the above goal. We provide one approach as a reference, but students are encouraged to come up their own ideas.

In our approach, we create two arrays X and Y. We compare the contents of these two arrays; if they are the same, the benign code is executed; otherwise, the malicious code is executed. See the following pseudo-code:

```
Array X;
Array Y;
main()
{
    if(X's contents and Y's contents are the same) run benign code;
    else
        run malicious code;
    return;
}
```

We can initialize the arrays X and Y with some values that can help us find their locations in the executable binary file. Our job is to change the contents of these two arrays, so we can generate two different versions that have the same MD5 hash. In one version, the contents of X and Y are the same, so the benign code is executed; in the other version, the contents of X and Y are different, so the malicious code is executed. We can achieve

this goal using a technique similar to the one used in **Task 3**. Figure 4 illustrates what the two versions of the program look like.

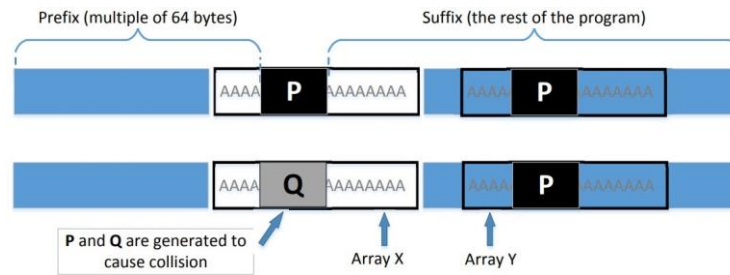


Figure 4: An approach to generate two hash-colliding programs with different behaviors.

From Figure 4, we know that these two binary files have the same MD5 hash value, as long as **P** and **Q** are generated accordingly. In the first version, we make the contents of arrays **X** and **Y** the same, while in the second version, we make their contents different. Therefore, the only thing we need to change is the contents of these two arrays, and there is no need to change the logic of the programs.

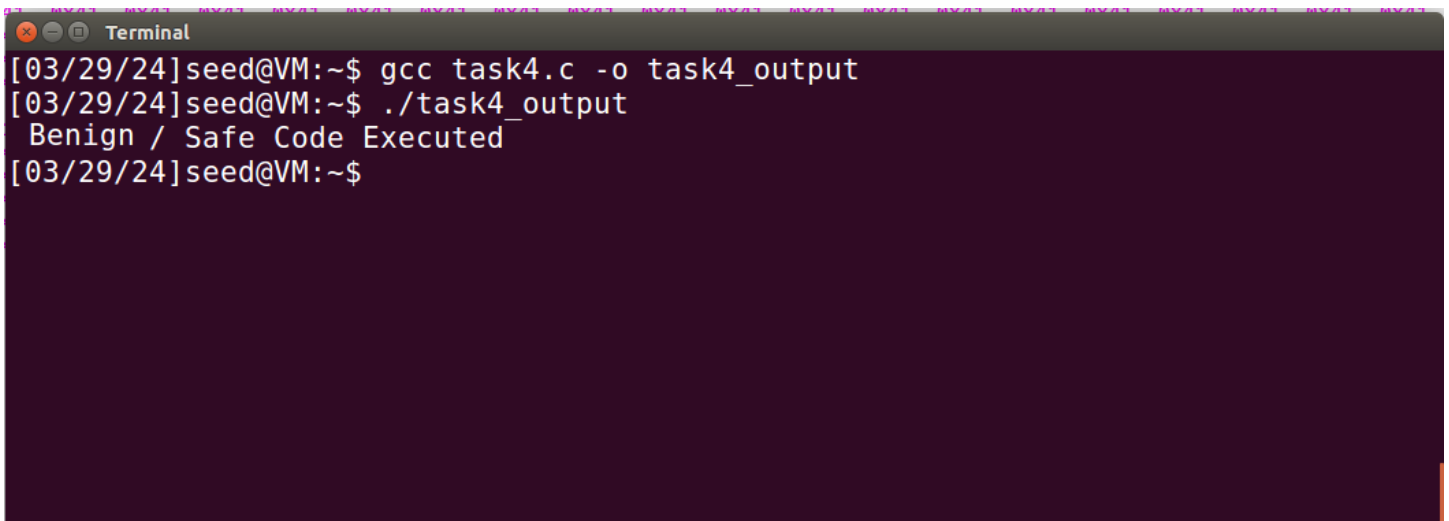

```
    if(ctr==200)
    {
        printf("Benign / Safe Code Executed\n");
    }
    else
    {
        printf("Malicious Code Executed\n");
    }
}
```

- Code Explanation:

This C code defines two arrays, called x and y, with 200 items in each. The 200 elements of each array are initialized with the ASCII character 'A', or hexadecimal value 0x41. After that, a for loop is used by the code to compare each element of the arrays x and y.

A counter called ctr is increased for every element that matches. All elements in arrays x and y are assumed to be similar if the counter ctr = 200 after all elements have been compared, suggesting benign or safe code execution. Otherwise, the software determines that malicious code has been run if any element in the arrays x and y differs.

- Executing the code task4.c

A terminal window titled "Terminal" with a dark background. The prompt is "seed@VM:~\$". The user enters "gcc task4.c -o task4_output". The prompt changes to "seed@VM:~\$". The user enters "./task4_output". The output "Benign / Safe Code Executed" is displayed. The prompt returns to "seed@VM:~\$".

```
Terminal
[03/29/24]seed@VM:~$ gcc task4.c -o task4_output
[03/29/24]seed@VM:~$ ./task4_output
Benign / Safe Code Executed
[03/29/24]seed@VM:~$
```

Since the arrays X and Y have the same content, the output will be Benign / Safe code executed.

- Opening the task4_output in bless editor to identify the offset value of the position where both arrays (array x and array y) are stored.

task4_output ✕

```

00000f8b 00 14 00 00 00 11 00 00 00 17 00 00 00 98 82 04 08 11 00 00 00 90 82 .....
00000fa2 04 08 12 00 00 00 08 00 00 00 13 00 00 00 08 00 00 00 FE FF FF 6F 70 .....op
00000fb9 82 04 08 FF FF FF 6F 01 00 00 00 F0 FF FF 6F 66 82 04 08 00 00 00 00 .....o.....of.....
00000fd0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000fe7 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000ffe 00 00 14 9F 04 08 00 00 00 00 00 00 00 00 00 00 E6 82 04 08 F6 82 04 08 00 .....
00001015 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0000102c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 41 41 41 .....AAA
00001043 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
0000105a 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
00001071 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
00001088 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
0000109f 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA

```

Signed 8 bit: 65 Signed 32 bit: 1094795585 Hexadecimal: 41 41 41 41 ✕
 Unsigned 8 bit: 65 Unsigned 32 bit: 1094795585 Decimal: 065 065 065 065
 Signed 16 bit: 16705 Float 32 bit: 12.07843 Octal: 101 101 101 101
 Unsigned 16 bit: 16705 Float 64 bit: 2261634.50980392 Binary: 01000001 01000001 01000001 01
☐ Show little endian decoding ☐ Show unsigned as hexadecimal ASCII Text: AAAA
 Offset: 0x1040 / 0x1e9b Selection: None INS

As shown above, the first array starts at location offset value 0x1040 which is 4160 in decimal.

task4_output ✕

```

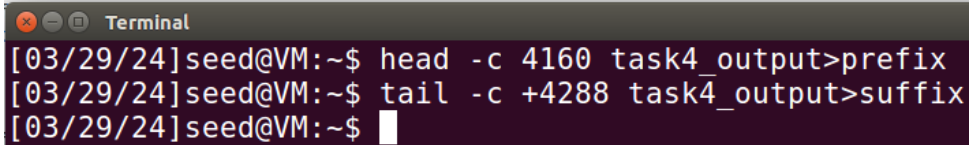
0000109f 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
000010b6 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
000010cd 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
000010e4 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
000010fb 41 41 41 41 41 41 41 41 41 41 41 41 41 41 00 00 00 00 00 00 00 00 .....AAAA
00001112 00 00 00 00 00 00 00 00 00 00 00 00 00 00 41 41 41 41 41 41 41 41 .....AAAA
00001129 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
00001140 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
00001157 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
0000116e 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
00001185 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
0000119c 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA
000011b3 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 .....AAAA

```

Signed 8 bit: 65 Signed 32 bit: 1094795585 Hexadecimal: 41 41 41 41 ✕
 Unsigned 8 bit: 65 Unsigned 32 bit: 1094795585 Decimal: 065 065 065 065
 Signed 16 bit: 16705 Float 32 bit: 12.07843 Octal: 101 101 101 101
 Unsigned 16 bit: 16705 Float 64 bit: 2261634.50980392 Binary: 01000001 01000001 01000001 01
☐ Show little endian decoding ☐ Show unsigned as hexadecimal ASCII Text: AAAA
 Offset: 0x1120 / 0x1e9b Selection: None INS

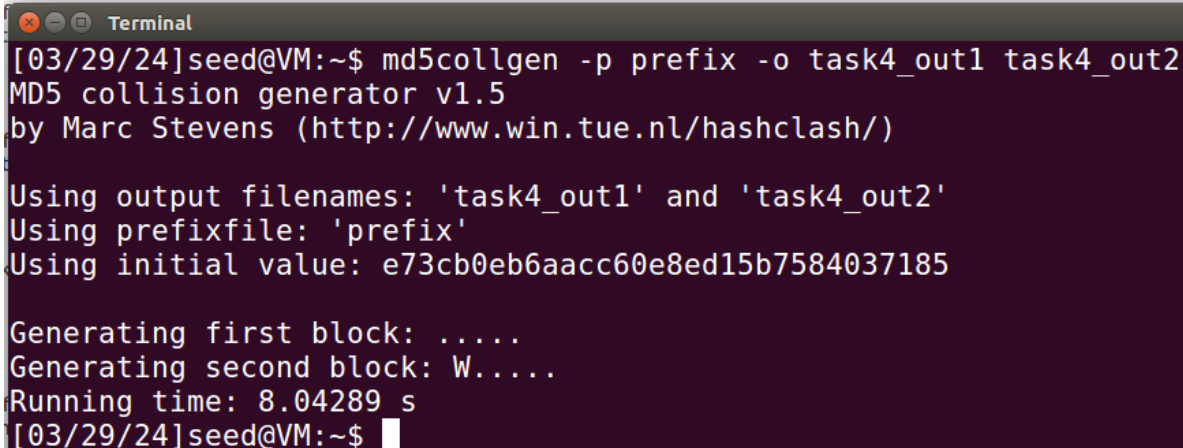
As shown above, the second array starts at location offset value 0x1120.

- As the first array starts at offset 4160, we will create a prefix from that location using the head command. Then the suffix file will contain the prefix and the last 128 bytes i.e. $4160 + 128 = 4288$



```
Terminal
[03/29/24]seed@VM:~$ head -c 4160 task4_output>prefix
[03/29/24]seed@VM:~$ tail -c +4288 task4_output>suffix
[03/29/24]seed@VM:~$
```

- Then we will generate two different files (task4_out1 and task4_out2) with the same md5 hash value for the prefix file using the **md5collgen** command.



```
Terminal
[03/29/24]seed@VM:~$ md5collgen -p prefix -o task4_out1 task4_out2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'task4_out1' and 'task4_out2'
Using prefixfile: 'prefix'
Using initial value: e73cb0eb6aacc60e8ed15b7584037185

Generating first block: .....
Generating second block: W.....
Running time: 8.04289 s
[03/29/24]seed@VM:~$
```

- Because the MD5 collision generating program has some constraints, the outputs generated by md5collgen must have the same prefix and suffix applied to them. Thus, two files P and Q, are created from output files task4_out1 and task4_out2.

```
Terminal
[03/29/24]seed@VM:~$ tail -c 128 task4_out1>P
[03/29/24]seed@VM:~$ tail -c 128 task4_out2>Q
[03/29/24]seed@VM:~$
```

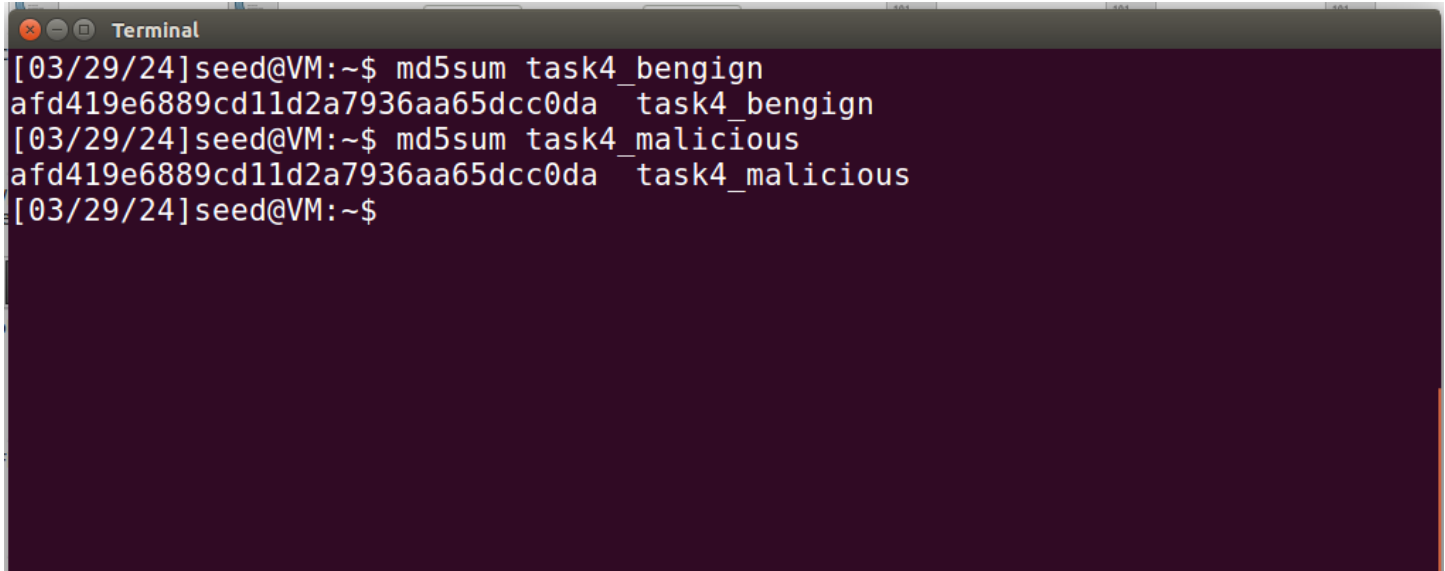
- Using the suffix file generated above to generate another two suffix files named pre_suffix and post_suffix

```
Terminal
[03/29/24]seed@VM:~$ head -c 96 suffix>pre_suffix
[03/29/24]seed@VM:~$ head -c +224 suffix>post_suffix
[03/29/24]seed@VM:~$
```

- Then concatenate task4_out1, pre-suffix, P, and post-suffix to create a new file task4_bengign and concatenate the task4_out2, pre-suffix, p, and post-suffix to create a new file task4_malicious.

```
Terminal
[03/29/24]seed@VM:~$ cat task4_out1 pre_suffix P post_suffix>task4_bengign
[03/29/24]seed@VM:~$ cat task4_out2 pre_suffix P post_suffix>task4_malicious
[03/29/24]seed@VM:~$
```

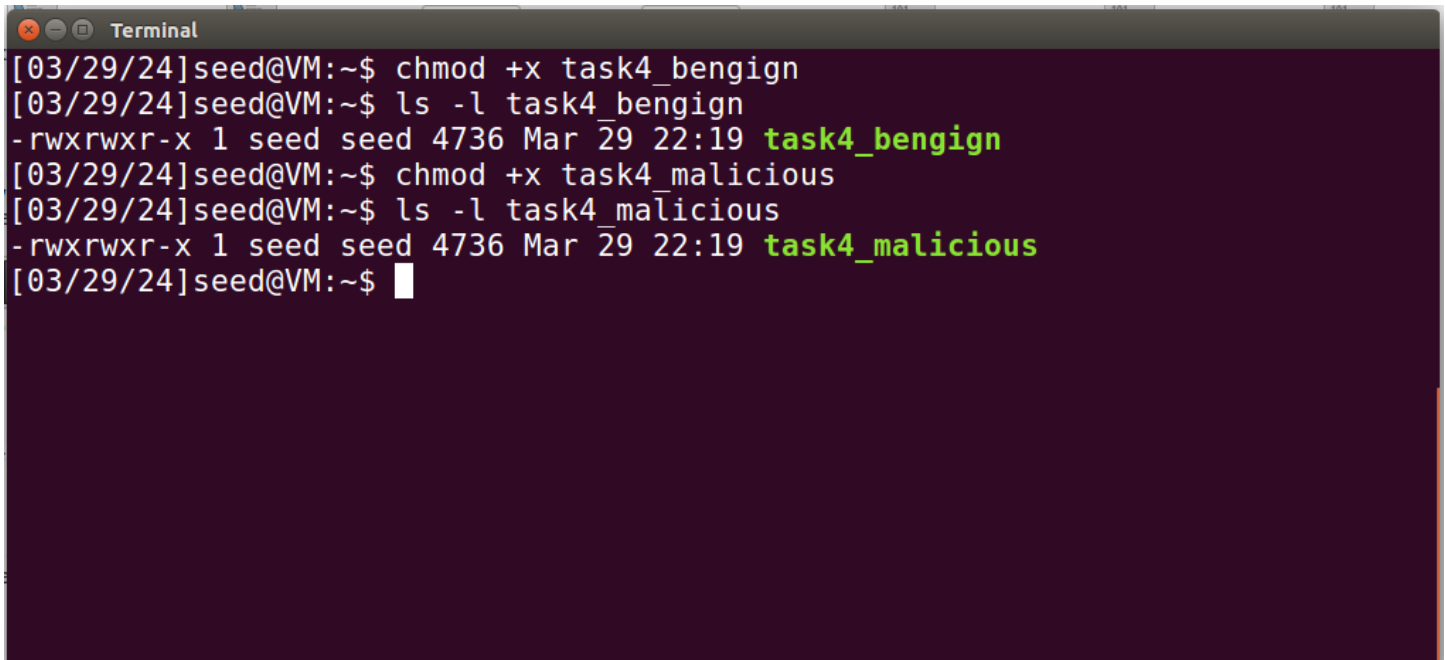
- Use the md5sum command to check whether the MD5 hash values of task4_bengign and task4_malicious are the same.

A terminal window titled "Terminal" with a dark background. It shows the execution of the md5sum command on two files. The output for both files is identical: afd419e6889cd11d2a7936aa65dcc0da. The prompt is [03/29/24]seed@VM:~\$.

```
[03/29/24]seed@VM:~$ md5sum task4_bengign
afd419e6889cd11d2a7936aa65dcc0da task4_bengign
[03/29/24]seed@VM:~$ md5sum task4_malicious
afd419e6889cd11d2a7936aa65dcc0da task4_malicious
[03/29/24]seed@VM:~$
```

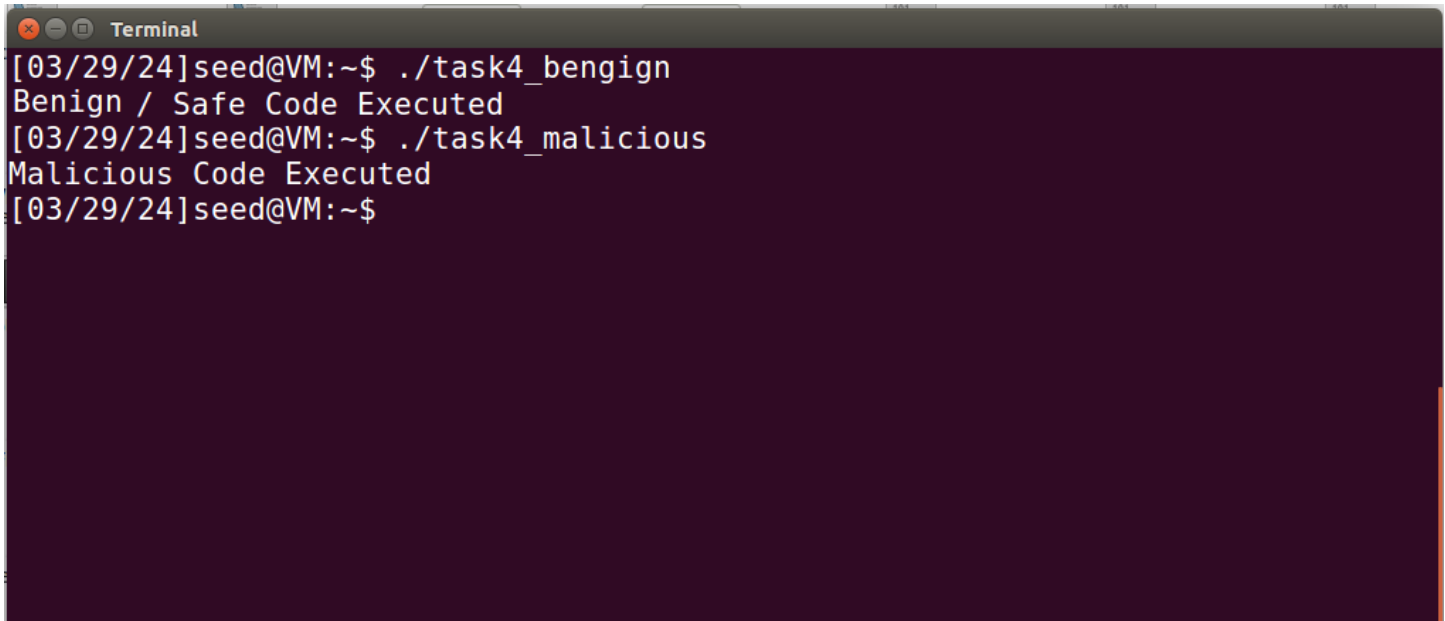
As shown above, md5 hash values for both files after concatenation are the same.

- Providing permission to execute the files task4_bengign and task4_malicious.

A terminal window titled "Terminal" with a dark background. It shows the execution of chmod +x on two files, followed by ls -l to check permissions. The output shows that both files now have execute permissions (x) for the owner. The prompt is [03/29/24]seed@VM:~\$.

```
[03/29/24]seed@VM:~$ chmod +x task4_bengign
[03/29/24]seed@VM:~$ ls -l task4_bengign
-rwxrwxr-x 1 seed seed 4736 Mar 29 22:19 task4_bengign
[03/29/24]seed@VM:~$ chmod +x task4_malicious
[03/29/24]seed@VM:~$ ls -l task4_malicious
-rwxrwxr-x 1 seed seed 4736 Mar 29 22:19 task4_malicious
[03/29/24]seed@VM:~$
```

- Executing the task4_bengign file and task4_malicious file as below

A terminal window titled "Terminal" with a dark background. It shows the execution of two files. The first command is `./task4_bengign`, which outputs `Benign / Safe Code Executed`. The second command is `./task4_malicious`, which outputs `Malicious Code Executed`. The prompt is `[03/29/24]seed@VM:~$`.

```
[03/29/24]seed@VM:~$ ./task4_bengign
Benign / Safe Code Executed
[03/29/24]seed@VM:~$ ./task4_malicious
Malicious Code Executed
[03/29/24]seed@VM:~$
```

- **Conclusion:**

As shown in the above code execution when we run the task4_bengign file the output we are getting is **Benign / Safe Code Executed**, similarly, when we run the task4_malicious file, we get the output as **Malicious Code Executed**. So, we successfully implemented the task where two programs have the same MD5 hash value as shown above but behave differently when executed. We have now successfully simulated the launch of the attack described in task 4.

