REPORT

CSL- TASK 1 (TEAM B)

Task 1: Generating Two Different Files with the same MD5 Hash

The purpose of this task is to generate two different files with same MD5 hash values. We have been given the tools of "Hashclash" project with which this task is carried out.

Tool used:

MD5_fastcoll program within Hashclash project.

Explanation:

In order to achieve this, the beginning part of two files (namely, out1.bin and out2.bin) need to be the same . The two output files should share a common prefix which is provided in the file prefix.txt.

The command to execute in Linux command line is given as follows:

$ /opt/hashclash/md5_fastcoll -p prefix.txt -o out1.bin out2.bin

The prefix files and the output files have to be in the same location.

Question 1 **:**

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

Answer: It will be padded with zeros to make the text a multiple of 64. This is because MD5 algorithm splits the binary data into blocks of 64 bytes and then handles each block according to the algorithm.

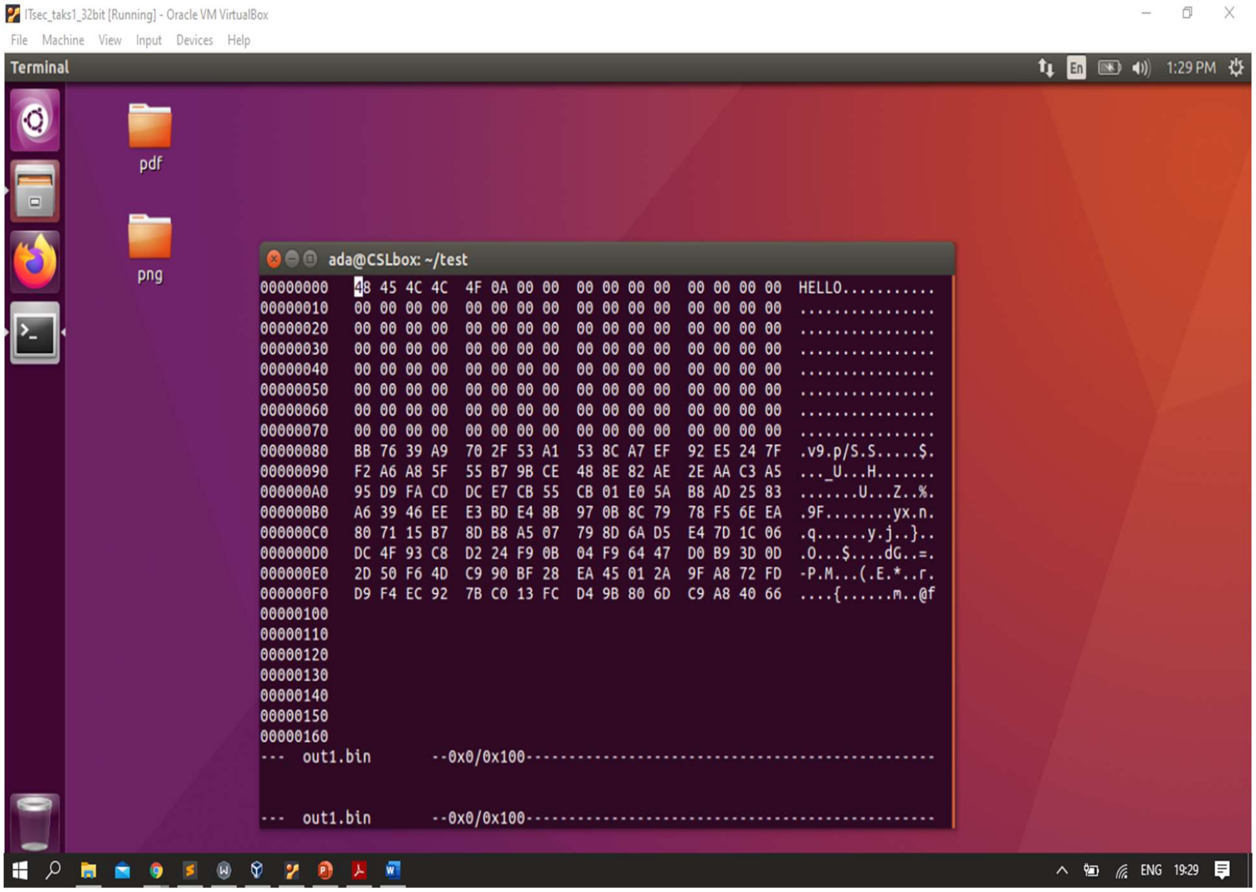
To verify this, we created a prefix file named 'prefix.txt' and entered a text, "Hello" inside it.

After running the command for md5 fastcoll tool, we examined the output file out1.bin using hexedit tool.

Observation :

On observing the binary data, we found that, after the word hello , zeros were padded to make the text into a block of 64 bytes. That is how we found the answer to this question.

The screen shot of this observation is given below:



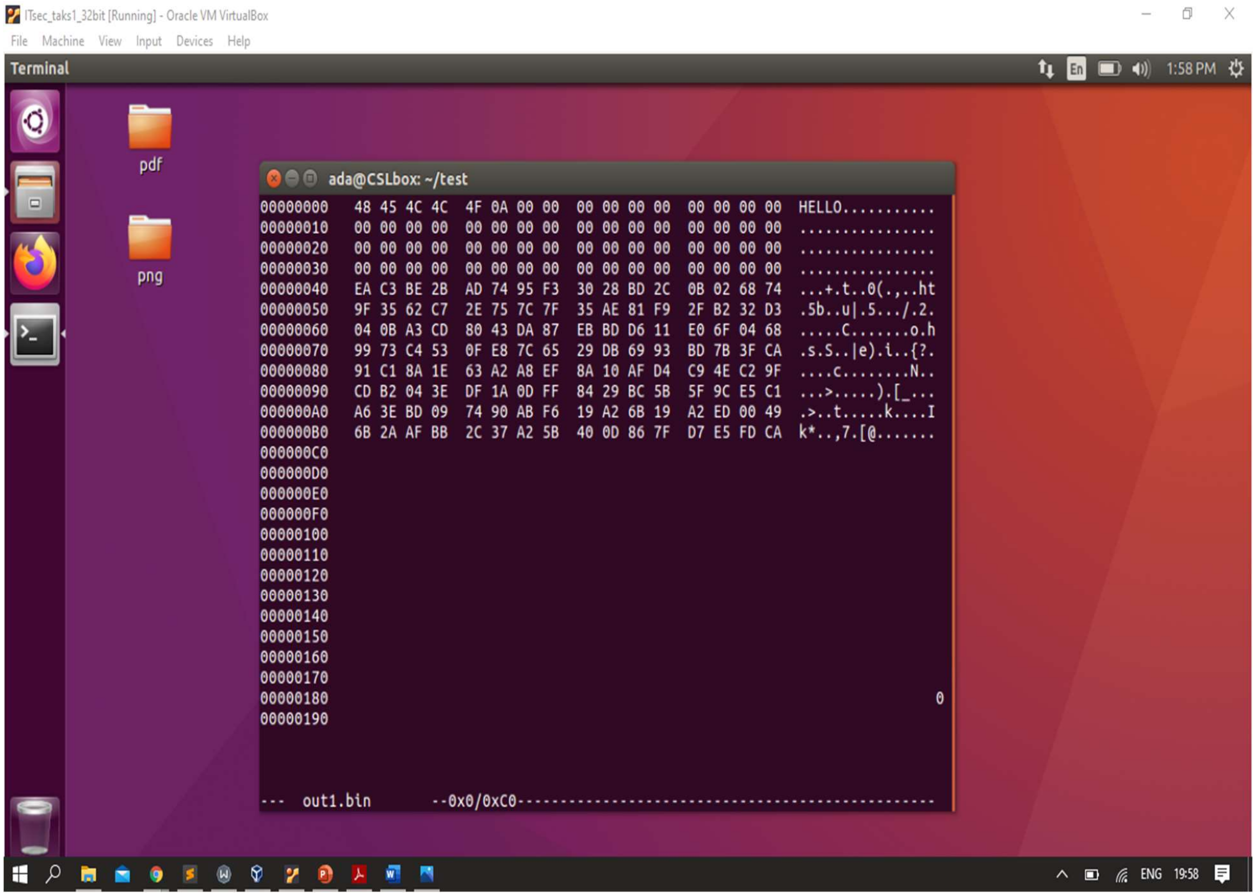
Question 2: Create a prefix file with exactly 64 bytes, and run the collision tool again, and see what happens.

<u>Answer:</u> No padded bits were observed. Since the block size that has to be split was exactly 64, no more extra bits were added in the file.

To verify this, the prefix file was truncated with exactly 64 bytes. Then it was passed into the collision tool and the binary file out1 was examined.

The observation is recorded as follows.

Observation :



Question 3: Are the data (128 bytes) generated by md5_fastcoll completely different for the two

output files? Please identify all the bytes that are different.

Answer: No, the data is not completely different. They are almost similar with few bytes varying in both the files.

The bytes that are different are located at positions 84, 110, 124.

Observation:

The locations at which the byte differs is not constant and keeps changing for different trials.

Task 2: Understanding MD5's Property

MD5 algorithm has the following 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.

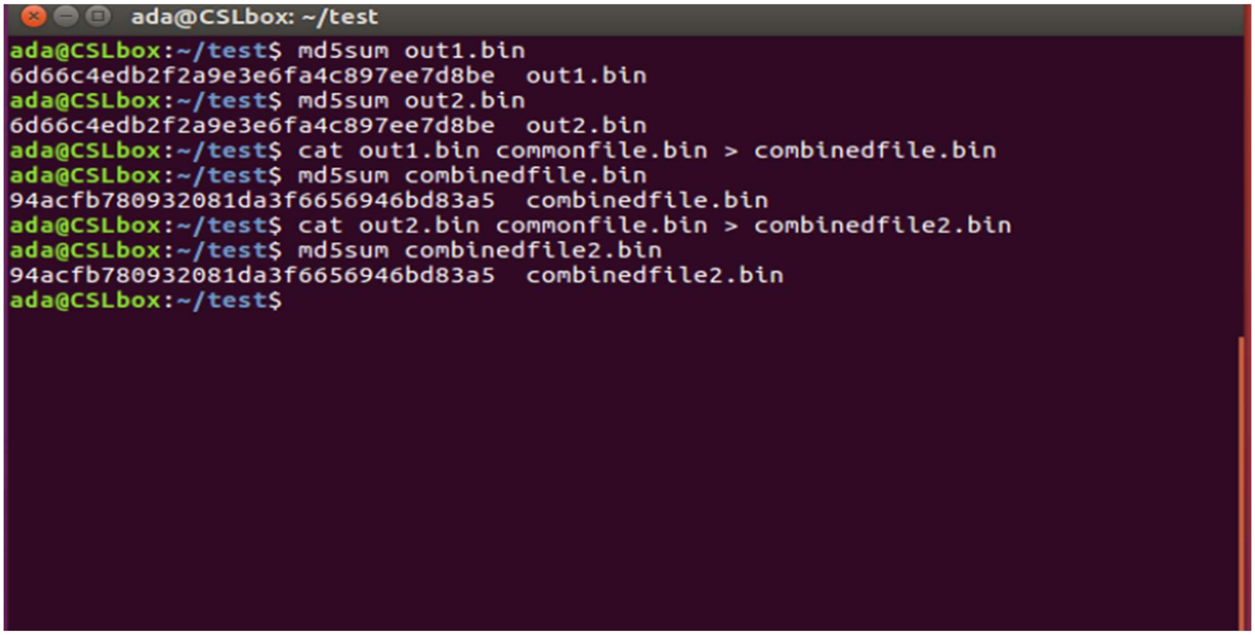
The aim of this task is to prove that this property holds true.

<u>Explanation:</u>

- Initially, we considered the two output files out1.bin and out2.bin from the previous task.
  Using the commands
  $ md5sum out1.bin
  $ md5sum out2.bin
  The md5sum or the hash value of the first and second output file were found.
- Now, we prepared another file and named it "common.bin" . This file considered some phrase and it was converted into a binary file.
  $cat out1.bin common.bin > combined.bin
  $cat out2.bin common.bin > combined2.bin
- The above two commands were used to concatenate the common file with the out1.bin and the out2.bin to get the respective combined files.
- Next, the md5sum of both these combined files are calculated using the command stated above to get the final result.

<u>Observation</u>:

On observing the md5 hash values of the two of the combined files, we found that it was same. Thus, the goal of the task was achieved.



Task 3: Generating two images with same MD5 hash

For this task, we have selected scenario two which was stated as follows:

In this scenario you are allowed to modify the binary of both provided images to achieve the goal
of having the same MD5 checksum. However, both logos should not change in terms of (visible)
content.

<u>Research and findings:</u>

- MD5 is thoroughly broken with regards to collisions, but not for preimages or second preimages.
- MD5 has actually been "weakened" with regards to preimages, but only in a theoretical way, because the attack cost is still billions of billions of times too expensive to be really tried (so MD5 is not "really" broken with regards to preimages, not in a practical way).

- MD5 is not collision resistant. MD5 uses the Merkle–Damgård construction, so if two prefixes with the same hash can be constructed, a common suffix can be added to both to make the collision more likely to be accepted as valid data by the application using it.

- In a chosen prefix collision the data preceding the specially crafted collision blocks can be completely different.

- For any targeted pair of distinct messages m1 and m2 we can effectively construct appendages b1 and b2 such that MD5(m1||b1) equals MD5(m2||b2). We call this a chosen-prefix collision. Said differently, we can cause an MD5 collision for any pair of distinct IHVs. For any two chosen message prefixes P and P', suffixes S and S' can be constructed such that the concatenated values P||S and P'||S' collide under MD5.

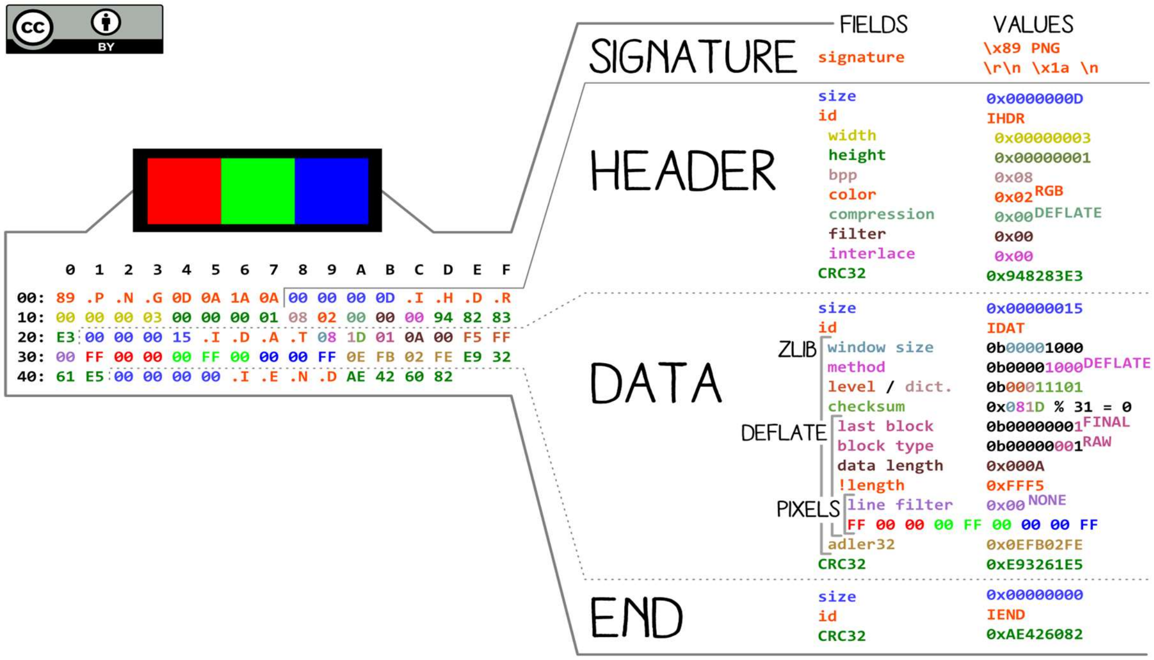
- Although the practical attack potential of this construction of chosen-prefix collisions is limited, it is of greater concern than random collisions for MD5.

- The chosen-prefix collision attack is the most powerful collision attack because it allows two distinct arbitrarily chosen prefixes, so one typically only needs to 'hide' the attack generated collision bit strings in the document. With available MD5 cryptanalysis tools (HashClash script scripts/cpc.sh) one can create such chosen-prefix collisions in one day. Convenient for 2 colliding files.
- Identical-prefix collisions can be created much faster within a few seconds (using e.g. fastcoll), but these generate pseudo-random looking 128-byte byte strings. The only difference that occurs in these collision bit strings which makes it a bit trickier to exploit these meaningful file formats.

- To facilitate file format exploits with identical-prefix collisions Marc Steven together with Ange Albertini modified hashclash and created a script scripts/poc_no.sh for a special type of a fast identical-prefix collision attack that they dubbed UniColl. It's special in that it uses a message block difference of just a single bit which is located early in the 10th-byte and actually in its least-significant bit.

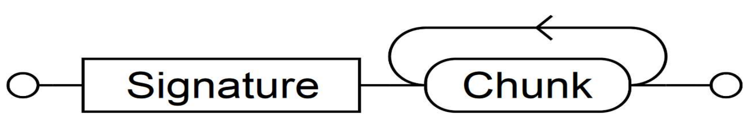
<u>FINDINGS ON PNG FILE FORMAT</u>

The png file takes the format that is depicted below:



PNG files start with an 8-byte signature, 89 50 4E 47 0D 0A 1A 0A. The first byte is a non-ASCII character, byte 2 through 4 spells out PNG in ASCII. The remaining bytes are line ends, the DOS EOF character, and another line break.

What follows next are what is known as chunks. Chunks help in maintaining backwards compatibility. A PNG reader when confronted with a chunk which it cannot decode can simply skip the chunk (if it is not defined as critical). Each chunk is represented as chunk length in bytes. This field is 4 bytes long (big endian).



A PNG image is made of a signature followed by a series of chunks

IHDR

An IHDR chunk is always the first chunk within a PNG file, following the PNG signature. It defines fundamental properties of the image such as its width and height as well as the bit depth and color type and the compression, filter, and interlace methods. In total, these data fields of the IHDR chunk are always 13 bytes in size resulting in a static length field across all IHDR chunks.

IDAT

IDAT chunks store the actual image data of the PNG file as a data stream compressed by the compression method defined in the IHDR chunk. All IDAT chunks in a PNG file should be stored in a consecutive order without any other chunk type in between. PNG encoders can divide the compressed image data into arbitrarily sized chunks. As already mentioned, also IDAT chunks with zero data are legal according to the PNG specification.

IEND

The IEND chunk must be the last chunk since it marks the end of a PNG file. An IEND chunk does not contain any data and is, therefore, equal for all PNG files.

COMMENT CHUNKS

They are perfect to skip collision blocks or extra data. Can be inserted several times - they are just entirely skipped: perfect for padding, collision blocks and extra data. Comment chunks are usually length-defined. One can exploit them by giving them a variable length via collision blocks differences.

Comment chunks can be used as placeholders for foreign data

IMPLEMENTATION

Since a PNG chunk has a length of four bytes, there is no need to modify the structure of either file: we can jump over a whole image in one go.

We can insert as many discarded chunks as we want, so we can add one for alignment, then one which length will be altered by a UniColl.

So an MD5 collision of two arbitrary PNG images is instant, with no prerequisite (no computation, just some minor file changes), and needs no chosen-prefix collision, just UniColl

- PNG uses CRC32 at the end of its chunks, but in practice they are ignored. They can be correct, but it's not required.

- The image meta data (dimensions, color space...) are stored in the IHDR chunk, which should in theory be right after the signature (ie, before any potential comment), so it would mean that we can only precompute collisions of images with the same meta data. However, that chunk can actually be after a comment block (in the vast majority of readers, except Apple ones since it is custom format), so we can put the collision data before the header, which enables to collide any pair of PNG with a single precomputation.

At first we used hashclash to compute md5 collisions with an empty prefix with the md5_fastcoll tool. Then we used a prefix we generated. These are identical prefix collisions.

Next, we started using unicoll, with the poc_no.sh script in the hashclash tool set. This attack works by modifying only a single byte in the collision block; there is no padding, and only two bytes are modified, by being incremented.

It takes longer to compute (a few minutes instead of seconds), but since the modified byte is only incremented, it gives more control when creating collisions with some file formats.

Using Unicoll, it's possible to exploit the PNG file structure, by varying the byte size of a block in big endian: this gives you a jump of +0x100, more than enough to have varying data in the skipped chunk.

So, we created a python script (named as png.py) , for inserting in the png binary file. It is given as follows.

<u>Python script:</u>

```
import sys
import struct
```

```
# Use case: ./png.py yes.png no.png
fn1, fn2 = sys.argv[1:3]
with open(fn1, "rb") as f:
  d1 = f.read()
with open(fn2, "rb") as f:
  d2 = f.read()

PNGSIG = "\x89PNG\r\n\x1a\n"
assert d1.startswith(PNGSIG)
assert d2.startswith(PNGSIG)

# short coll
with open("png1.bin", "rb") as f:
  blockS = f.read()
# long coll
with open("png2.bin", "rb") as f:
  blockL = f.read()

ascii_art = """
vvvv
/==============\\
|*            *|
|  PNG IMAGE   |
|     with     |
|  identical   |
|   -prefix    |
| MD5 collision|
|              |
|  by          |
| Marc Stevens |
|  and         |
|Ange Albertini|
| in 2018-2019 |
|*            *|
\\==============/
""".replace("\n", "").replace("\r","")

assert len(ascii_art) == 0x100 - 3*4 # 1 chunk
declaration + crc
```

```
# 2 CRCs, 0x100 of UniColl difference, and d2 chunks
skipLen = 0x100 - 4*2 + len(d2[8:])

#######################################################
########################
#
# simplest (w/ appended data and incorrect CRCs)

"""
Ca{          Ca{          Ca{
}            }             }
Cc{          Cc{           Cc{
--------     --------     --------- <== collision blocks
}a           }a            ..
  C1{          C1{            ...
}b           ..            }b
    D1           ..            D1
  }            }             .
      D2           D2             ..
"""

from binascii import crc32
_crc32 = lambda d:(crc32(d) % 0x100000000)

suffix = struct.pack(">I", _crc32(blockS[0x4b:0xc0]))

suffix += "".join([
  # sKIP chunk
    struct.pack(">I", skipLen),
    "sKIP",
      # it will cover all data chunks of d2,
      # and the 0x100 buffer
  ascii_art,
  "\xDE\xAD\xBE\xEF", # fake CRC for cOLL chunk

      d2[8:],
      # long cOLL CRC
    "\x5E\xAF\x00\x0D", # fake CRC for sKIP chunk
```

```
    # first image chunk
    d1[8:],
    ])

with open("collision1.png", "wb") as f:
  f.write("".join([
    blockS,
    suffix
    ]))

with open("collision2.png", "wb") as f:
  f.write("".join([
    blockL,
    suffix
    ]))
```

The above script is converted and viewed as binary file using the command,

$hexedit png.py

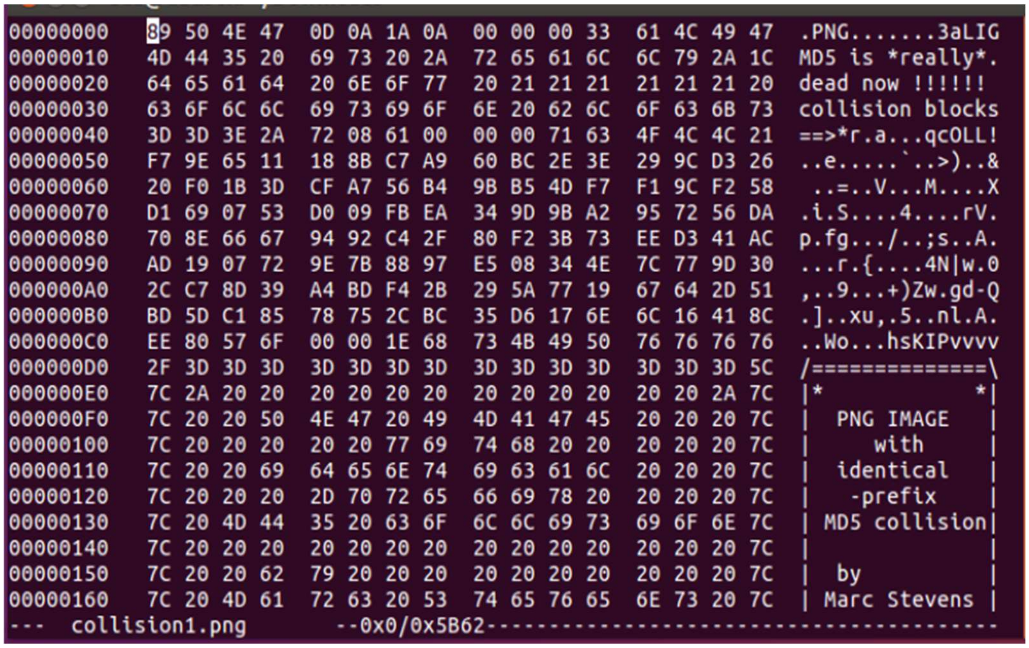
The bytes in the appropriate locations and altered and collision is created without altering the actual contents of the png file.

$ /opt/hashclash/md5_fastcoll -p python.py -o collision1.png collision2.png

The above command should be run so that the fastcoll program takes the python script as prefix text and returns the collided images which are not visually changed but possess the same md5 sum.

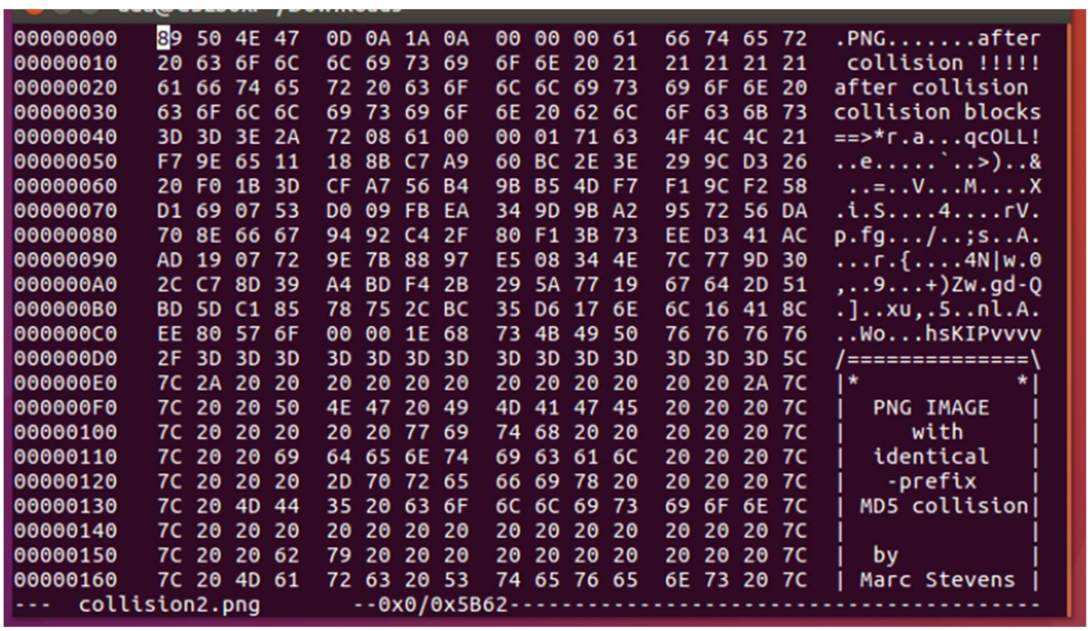
The observation section below holds the screen shots of the above mentioned results.

Observation :



```
00000000  89 50 4E 47  0D 0A 1A 0A  00 00 00 33  61 4C 49 47  .PNG.......3aLIG
00000010  4D 44 35 20  69 73 20 2A  72 65 61 6C  6C 79 2A 1C  MD5 is *really*.
00000020  64 65 61 64  20 6E 6F 77  20 21 21 21  21 21 21 20  dead now !!!!!!
00000030  63 6F 6C 6C  69 73 69 6F  6E 20 62 6C  6F 63 6B 73  collision blocks
00000040  3D 3D 3E 2A  72 08 61 00  00 00 71 63  4F 4C 4C 21  ==>*r.a...qcOLL!
00000050  F7 9E 65 11  18 8B C7 A9  60 BC 2E 3E  29 9C D3 26  ..e.....`..>)..&
00000060  20 F0 1B 3D  CF A7 56 B4  9B B5 4D F7  F1 9C F2 58   ..=..V...M....X
00000070  D1 69 07 53  D0 09 FB EA  34 9D 9B A2  95 72 56 DA  .i.S....4....rV.
00000080  70 8E 66 67  94 92 C4 2F  80 F2 3B 73  EE D3 41 AC  p.fg.../..;s..A.
00000090  AD 19 07 72  9E 7B 88 97  E5 08 34 4E  7C 77 9D 30  ...r.{....4N|w.0
000000A0  2C C7 8D 39  A4 BD F4 2B  29 5A 77 19  67 64 2D 51  ,..9...+)Zw.gd-Q
000000B0  BD 5D C1 85  78 75 2C BC  35 D6 17 6E  6C 16 41 8C  .]..xu,.5..nl.A.
000000C0  EE 80 57 6F  00 00 1E 68  73 4B 49 50  76 76 76 76  ..Wo...hsKIPvvvv
000000D0  2F 3D 3D 3D  3D 3D 3D 3D  3D 3D 3D 3D  3D 3D 3D 5C  /==============\
000000E0  7C 2A 20 20  20 20 20 20  20 20 20 20  20 20 2A 7C  |*            *|
000000F0  7C 20 20 50  4E 47 20 49  4D 41 47 45  20 20 20 7C  |  PNG IMAGE   |
00000100  7C 20 20 20  20 20 77 69  74 68 20 20  20 20 20 7C  |     with     |
00000110  7C 20 20 69  64 65 6E 74  69 63 61 6C  20 20 20 7C  |  identical   |
00000120  7C 20 20 20  2D 70 72 65  66 69 78 20  20 20 20 7C  |   -prefix    |
00000130  7C 20 4D 44  35 20 63 6F  6C 6C 69 73  69 6F 6E 7C  | MD5 collision|
00000140  7C 20 20 20  20 20 20 20  20 20 20 20  20 20 20 7C  |              |
00000150  7C 20 20 62  79 20 20 20  20 20 20 20  20 20 20 7C  |  by          |
00000160  7C 20 4D 61  72 63 20 53  74 65 76 65  6E 73 20 7C  | Marc Stevens |
---  collision1.png       --0x0/0x5B62-------------------------------------
```

This is the binary data of the image file btu.png after collision



```
00000000  89 50 4E 47  0D 0A 1A 0A  00 00 00 61  66 74 65 72  .PNG.......after
00000010  20 63 6F 6C  6C 69 73 69  6F 6E 20 21  21 21 21 21   collision !!!!!
00000020  61 66 74 65  72 20 63 6F  6C 6C 69 73  69 6F 6E 20  after collision
00000030  63 6F 6C 6C  69 73 69 6F  6E 20 62 6C  6F 63 6B 73  collision blocks
00000040  3D 3D 3E 2A  72 08 61 00  00 01 71 63  4F 4C 4C 21  ==>*r.a...qcOLL!
00000050  F7 9E 65 11  18 8B C7 A9  60 BC 2E 3E  29 9C D3 26  ..e.....`..>)..&
00000060  20 F0 1B 3D  CF A7 56 B4  9B B5 4D F7  F1 9C F2 58   ..=..V...M....X
00000070  D1 69 07 53  D0 09 FB EA  34 9D 9B A2  95 72 56 DA  .i.S....4....rV.
00000080  70 8E 66 67  94 92 C4 2F  80 F1 3B 73  EE D3 41 AC  p.fg.../..;s..A.
00000090  AD 19 07 72  9E 7B 88 97  E5 08 34 4E  7C 77 9D 30  ...r.{....4N|w.0
000000A0  2C C7 8D 39  A4 BD F4 2B  29 5A 77 19  67 64 2D 51  ,..9...+)Zw.gd-Q
000000B0  BD 5D C1 85  78 75 2C BC  35 D6 17 6E  6C 16 41 8C  .]..xu,.5..nl.A.
000000C0  EE 80 57 6F  00 00 1E 68  73 4B 49 50  76 76 76 76  ..Wo...hsKIPvvvv
000000D0  2F 3D 3D 3D  3D 3D 3D 3D  3D 3D 3D 3D  3D 3D 3D 5C  /==============\
000000E0  7C 2A 20 20  20 20 20 20  20 20 20 20  20 20 2A 7C  |*            *|
000000F0  7C 20 20 50  4E 47 20 49  4D 41 47 45  20 20 20 7C  |  PNG IMAGE   |
00000100  7C 20 20 20  20 20 77 69  74 68 20 20  20 20 20 7C  |     with     |
00000110  7C 20 20 69  64 65 6E 74  69 63 61 6C  20 20 20 7C  |  identical   |
00000120  7C 20 20 20  2D 70 72 65  66 69 78 20  20 20 20 7C  |   -prefix    |
00000130  7C 20 4D 44  35 20 63 6F  6C 6C 69 73  69 6F 6E 7C  | MD5 collision|
00000140  7C 20 20 20  20 20 20 20  20 20 20 20  20 20 20 7C  |              |
00000150  7C 20 20 62  79 20 20 20  20 20 20 20  20 20 20 7C  |  by          |
00000160  7C 20 4D 61  72 63 20 53  74 65 76 65  6E 73 20 7C  | Marc Stevens |
---  collision2.png       --0x0/0x5B62-------------------------------------
```

This is the binary data of the image file mit.png after collision

The two image files after collision has been uploaded in the gitlab repository.