IIT CS458: Introduction to Information Security

Homework 3: MD5 Collision Attack Lab

My Dinh

2 Lab Tasks

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

In this task, I wrote a Python program to generate a string with the size in bytes given by the user in the command line interface.

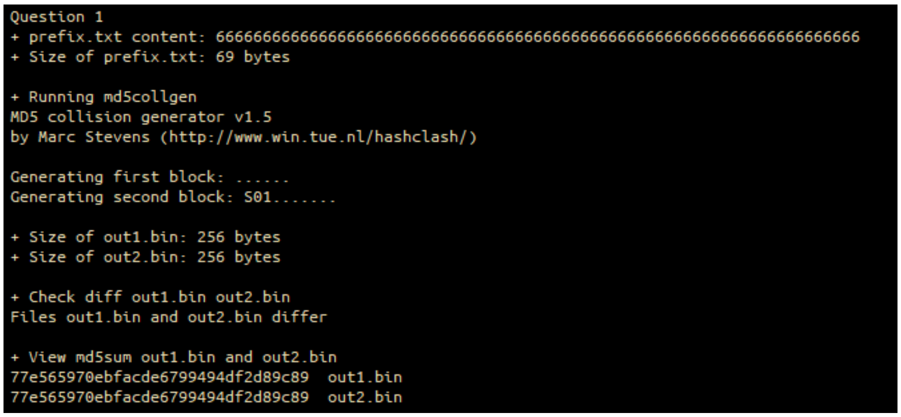
```
import sys

char = "6"
length = sys.argv[1]
sentence = char * int(length)

print(sentence)
```

The bash script below is for generating the `out1.bin` and `out2.bin` files using `md5collgen`, comparing their hex using `hexdump` and `diff`, and checking their `md5sum` from the `prefix.txt` file in question 1 and 2 of task 1.

```
#!/bin/sh


generate_md5() {
    echo -n $(python3 generate_string.py $1) > prefix.txt
    echo "+ prefix.txt content: $(cat prefix.txt)"
    echo "+ Size of prefix.txt: $(wc -c < prefix.txt) bytes"

    echo "\n+ Running md5collgen"
    md5collgen -p prefix.txt -o out1.bin out2.bin --quiet

    echo
    echo "+ Size of out1.bin: $(wc -c < out1.bin) bytes"
    echo "+ Size of out2.bin: $(wc -c < out2.bin) bytes"

    echo "\n+ Check diff out1.bin out2.bin"
    diff out1.bin out2.bin -q

    echo "\n+ View md5sum out1.bin and out2.bin"
    md5sum out1.bin
    md5sum out2.bin

    echo "\n+ Compare out1.bin and out2.bin hex"
    echo "out1.bin"
    hexdump out1.bin
    echo "\nout2.bin"
    hexdump out2.bin
}
```

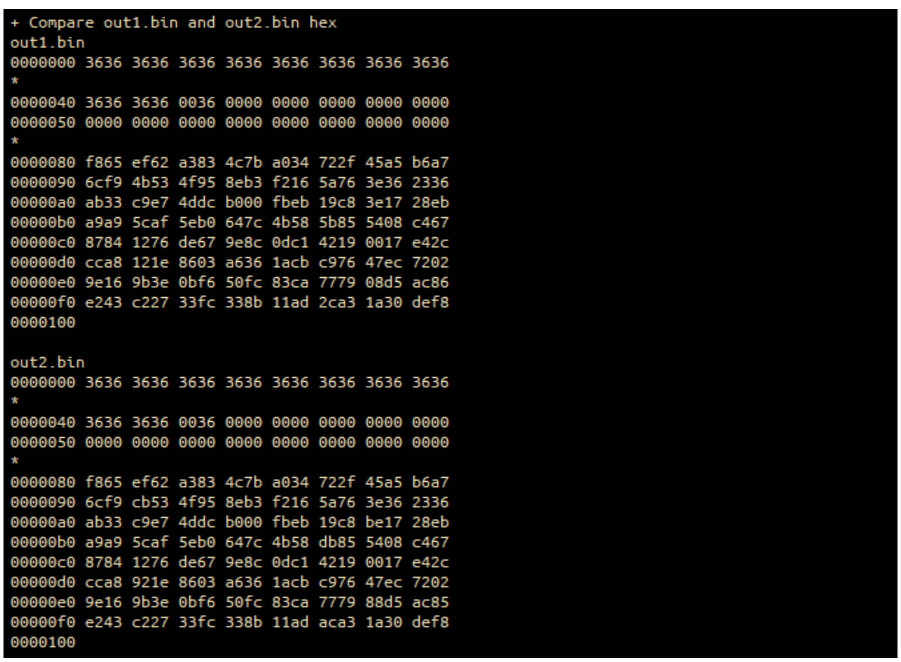
```
question_1() {
    echo "\nQuestion 1"
    generate_md5 69
}

question_1

question_2
```

Question 1. If the length of the `prefix.txt` file is not multiple of 64 (in this case, my `prefix.txt` file is 69 bytes), the binary files `out1.bin` and `out2.bin` generated by `md5collgen` would have padding at the beginning of the hex of the file. We can check this by using `hexdump` tool.



```
Question 1
+ prefix.txt content: 666666666666666666666666666666666666666666666666666666666666666666666
+ Size of prefix.txt: 69 bytes

+ Running md5collgen
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Generating first block: ......
Generating second block: S01.......

+ Size of out1.bin: 256 bytes
+ Size of out2.bin: 256 bytes

+ Check diff out1.bin out2.bin
Files out1.bin and out2.bin differ

+ View md5sum out1.bin and out2.bin
77e565970ebfacde6799494df2d89c89  out1.bin
77e565970ebfacde6799494df2d89c89  out2.bin
```

Figure 1: Size and md5sum of out1.bin and out2.bin for 69 byte prefix file.



```
+ Compare out1.bin and out2.bin hex
out1.bin
0000000 3636 3636 3636 3636 3636 3636 3636 3636
*
0000040 3636 3636 0036 0000 0000 0000 0000 0000
0000050 0000 0000 0000 0000 0000 0000 0000 0000
*
0000080 f865 ef62 a383 4c7b a034 722f 45a5 b6a7
0000090 6cf9 4b53 4f95 8eb3 f216 5a76 3e36 2336
00000a0 ab33 c9e7 4ddc b000 fbeb 19c8 3e17 28eb
00000b0 a9a9 5caf 5eb0 647c 4b58 5b85 5408 c467
00000c0 8784 1276 de67 9e8c 0dc1 4219 0017 e42c
00000d0 cca8 121e 8603 a636 1acb c976 47ec 7202
00000e0 9e16 9b3e 0bf6 50fc 83ca 7779 08d5 ac86
00000f0 e243 c227 33fc 338b 11ad 2ca3 1a30 def8
0000100

out2.bin
0000000 3636 3636 3636 3636 3636 3636 3636 3636
*
0000040 3636 3636 0036 0000 0000 0000 0000 0000
0000050 0000 0000 0000 0000 0000 0000 0000 0000
*
0000080 f865 ef62 a383 4c7b a034 722f 45a5 b6a7
0000090 6cf9 cb53 4f95 8eb3 f216 5a76 3e36 2336
00000a0 ab33 c9e7 4ddc b000 fbeb 19c8 be17 28eb
00000b0 a9a9 5caf 5eb0 647c 4b58 db85 5408 c467
00000c0 8784 1276 de67 9e8c 0dc1 4219 0017 e42c
00000d0 cca8 921e 8603 a636 1acb c976 47ec 7202
00000e0 9e16 9b3e 0bf6 50fc 83ca 7779 88d5 ac85
00000f0 e243 c227 33fc 338b 11ad aca3 1a30 def8
0000100
```

Figure 2: Hexdump of out1.bin and out2.bin for 69 byte prefix file.

Question 2. If the `prefix.txt` file is exactly 64 bytes the the result files `out1.bin` and `out2.bin` do not have padding at the beginning of the files. The size of both binary files are 192 bytes.

```
Question 2
+ prefix.txt content: 6666666666666666666666666666666666666666666666666666666666666666
+ Size of prefix.txt: 64 bytes

+ Running md5collgen
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Generating first block: ....
Generating second block: S11..................................................................
...............

+ Size of out1.bin: 192 bytes
+ Size of out2.bin: 192 bytes

+ Check diff out1.bin out2.bin
Files out1.bin and out2.bin differ

+ View md5sum out1.bin and out2.bin
13efa0b9e58f2b9537e0c90808dceee0  out1.bin
13efa0b9e58f2b9537e0c90808dceee0  out2.bin
```

Figure 3: Size and md5sum of out1.bin and out2.bin for 64 byte prefix file.

```
+ Compare out1.bin and out2.bin hex
out1.bin
0000000 3636 3636 3636 3636 3636 3636 3636 3636
*
0000040 2464 f7b2 1fe0 58ab fbf6 ce43 592c 5379
0000050 0206 752f 0926 d4e9 a5f6 a544 af8a f617
0000060 b9d9 e0e2 75ce f32c 6e67 b2f4 edce 7901
0000070 2aa7 cb2f b3bb ff5d 3773 8448 e329 1275
0000080 60e4 979f ac5e 05ad b19f 018e 4cff c238
0000090 70b1 2b9a face 8a86 ab76 cbbb 514f 41c7
00000a0 cab0 a9bc 0b82 1e9c 98e9 516b bf71 5b70
00000b0 400a 6f1f 4285 093f a961 e3e2 2156 d4b2
00000c0

out2.bin
0000000 3636 3636 3636 3636 3636 3636 3636 3636
*
0000040 2464 f7b2 1fe0 58ab fbf6 ce43 592c 5379
0000050 0206 f52f 0926 d4e9 a5f6 a544 af8a f617
0000060 b9d9 e0e2 75ce f32c 6e67 b2f4 6dce 7902
0000070 2aa7 cb2f b3bb ff5d 3773 0448 e329 1275
0000080 60e4 979f ac5e 05ad b19f 018e 4cff c238
0000090 70b1 ab9a face 8a86 ab76 cbbb 514f 41c7
00000a0 cab0 a9bc 0b82 1e9c 98e9 516b 3f71 5b70
00000b0 400a 6f1f 4285 093f a961 63e2 2156 d4b2
00000c0
```

Figure 4: Hexdump of out1.bin and out2.bin for 64 byte prefix file.

Question 3. For this question, I'm going to use the result output files for 64 byte `prefix.txt` file.

```sh
#!/bin/sh

question_3() {
    echo "\nQuestion 3"
    tail -c 128 out1.bin > data1
    tail -c 128 out2.bin > data2
    diff data1 data2 -q
}

question_3
```

The code snippet above would store the last 128 bytes of `out1.bin` in `data1` file and the last 128 bytes of `out2.bin` in `data2`. I then used `diff` to check if the data of generated by `md5collgen` were different. The result of the code snippet was `"Files data1 and data2 differ"`, meaning that `data1` and `data2` did not contain the same data.

To find all the different bytes in `data1` and `data2`, I had written the following program in Python to compare the bytes of each file.

```python
with open("data1", "rb") as f:
    hex1 = f.read()

with open("data2", "rb") as f:
    hex2 = f.read()

for i in range(len(hex1)):
    if hex1[i] != hex2[i]:
        print(f'Diff hex value at position {hex(i)} in data1 and data2: {hex(hex1[i])} {hex(
                                                          hex2[i])}')
```

The result can be found in Figure 5.

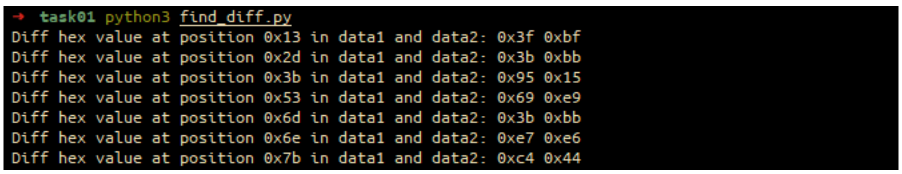


Figure 5: Different bytes in the 128 bytes generated by md5collgen in out1.bin and out2.bin.

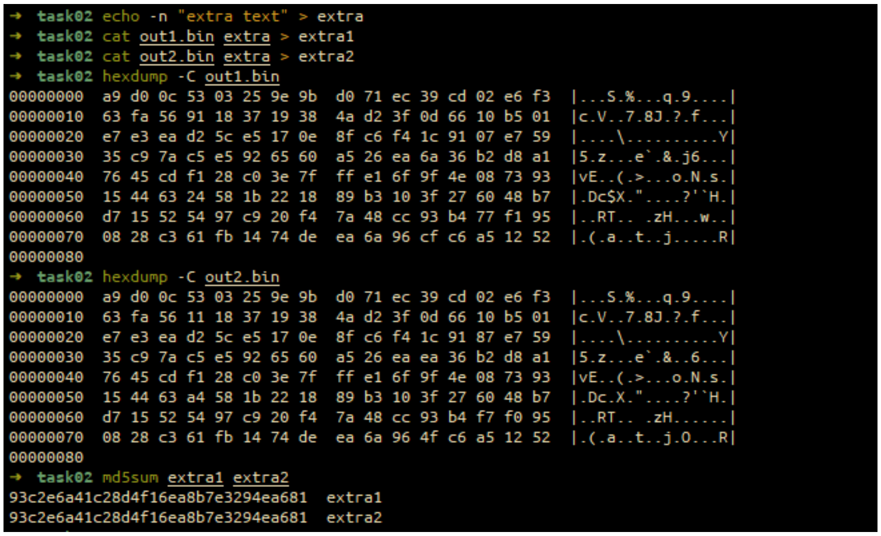
2.2 Task 2: Understanding MD5's Property

For this task, first I generated two different files that have the same md5sum using `md5collgen` and stored them in `out1.bin` and `out2.bin` respectively.

Figure 6: Generating different files with the same MD5 hash.

I then created a file that contains extra text `"extra text"`, added it to the end of `out1.bin` and `out2.bin` to create `extra1` and `extra2` file, and checked the md5 hash value of the new file using `md5sum`.

From the result of `hexdump`, I could conclude that the same data was concatenated at the end of both



```
→ task02 echo -n "extra text" > extra
→ task02 cat out1.bin extra > extra1
→ task02 cat out2.bin extra > extra2
→ task02 hexdump -C out1.bin
00000000  a9 d0 0c 53 03 25 9e 9b  d0 71 ec 39 cd 02 e6 f3  |...S.%...q.9....|
00000010  63 fa 56 91 18 37 19 38  4a d2 3f 0d 66 10 b5 01  |c.V..7.8J.?.f...|
00000020  e7 e3 ea d2 5c e5 17 0e  8f c6 f4 1c 91 07 e7 59  |....\..........Y|
00000030  35 c9 7a c5 e5 92 65 60  a5 26 ea 6a 36 b2 d8 a1  |5.z...e`.&.j6...|
00000040  76 45 cd f1 28 c0 3e 7f  ff e1 6f 9f 4e 08 73 93  |vE..(.>...o.N.s.|
00000050  15 44 63 24 58 1b 22 18  89 b3 10 3f 27 60 48 b7  |.Dc$X."....?'`H.|
00000060  d7 15 52 54 97 c9 20 f4  7a 48 cc 93 b4 77 f1 95  |..RT.. .zH...w..|
00000070  08 28 c3 61 fb 14 74 de  ea 6a 96 cf c6 a5 12 52  |.(.a..t..j.....R|
00000080
→ task02 hexdump -C out2.bin
00000000  a9 d0 0c 53 03 25 9e 9b  d0 71 ec 39 cd 02 e6 f3  |...S.%...q.9....|
00000010  63 fa 56 11 18 37 19 38  4a d2 3f 0d 66 10 b5 01  |c.V..7.8J.?.f...|
00000020  e7 e3 ea d2 5c e5 17 0e  8f c6 f4 1c 91 87 e7 59  |....\..........Y|
00000030  35 c9 7a c5 e5 92 65 60  a5 26 ea ea 36 b2 d8 a1  |5.z...e`.&..6...|
00000040  76 45 cd f1 28 c0 3e 7f  ff e1 6f 9f 4e 08 73 93  |vE..(.>...o.N.s.|
00000050  15 44 63 a4 58 1b 22 18  89 b3 10 3f 27 60 48 b7  |.Dc.X."....?'`H.|
00000060  d7 15 52 54 97 c9 20 f4  7a 48 cc 93 b4 f7 f0 95  |..RT.. .zH......|
00000070  08 28 c3 61 fb 14 74 de  ea 6a 96 4f c6 a5 12 52  |.(.a..t..j.O...R|
00000080
→ task02 md5sum extra1 extra2
93c2e6a41c28d4f16ea8b7e3294ea681  extra1
93c2e6a41c28d4f16ea8b7e3294ea681  extra2
```

Figure 7: Comparing MD5 hash of new files.

`out1.bin` and `out2.bin`. The MD5 hash generated from `md5sum` tool of new files `extra1` and `extra2`, which were `out1.bin` and `out2.bin` with the message `"extra text"` added to the end of the file, are the same (both are equal to `"93c2e6a41c28d4f16ea8b7e3294ea681"`).

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

The content of the C program I used for this task is

```c
#include <stdio.h>

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,
    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,
    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,
    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,
    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,
    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,
    0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
    0x41,0x41,0x41,0x41,0x41

};

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

I made array `xyz` to contain 200 character A's so it would be easier for me to locate the array in the binary file after compiling the C program above.

To locate the position of array `xyz` in the binary file without using `bless` (because I don't have `bless` installed in my local machine), I wrote a Python program to go through the bytes and pin point the start and the end position of the array in the executable file.

```python
with open("a.out", "rb") as f:
    byte_stream = f.read()
f.close()


# find the start and end byte block of the xyz array that contains 200 A's
def find_A_range():
    start, end = 0, 0
    for i in range(len(byte_stream)):
        if byte_stream[i] == 0x41:
            start = i
            end = i

            while end < len(byte_stream) and byte_stream[end] == 0x41:
                end += 1

            end -= 1

            if end - start + 1 == 200:
                break

    return (start, end)


# get the size of prefix
def get_prefix_size(start: int) -> int:
    return start + (64 - start % 64)


# get the size of prefix
def get_suffix_size(prefix_size: int) -> int:
    return len(byte_stream) - prefix_size - 128


if __name__ == "__main__":
    start, _ = find_A_range()
    prefix_size = get_prefix_size(start)
    suffix_size = get_suffix_size(prefix_size)

    print(prefix_size, suffix_size)
```

Using that information, I created two functions to get the size of the `prefix` and `suffix` file. To do this, the function `get_prefix_size` finds a multiple of `64` that is nearest to the value of start position. And the suffix size is found by calculating the difference between the size of the executable and the size of prefix and `128`.

Finally, I wrote a bash script to run all the commands to generate `a1.out` and `a2.out` which are the two executable files with the same MD5 hash but have different elements in array `xyz`.

```sh
#!/bin/sh

# get prefix and suffix file using the given sizes
get_prefix_and_suffix() {
    PREFIX_SIZE=$1
    SUFFIX_SIZE=$2

    echo "prefix size: ${PREFIX_SIZE}, suffix size: ${SUFFIX_SIZE}"
    head -c $PREFIX_SIZE a.out > prefix
    tail -c $SUFFIX_SIZE a.out > suffix
}

# compile and create executable file for the given C program
```

```
gcc array.c

# get prefix and suffix from the result size of prefix_suffix_size program
echo "+ Get prefix and suffix size"
get_prefix_and_suffix $(python3 prefix_suffix_size.py)

# generate P and Q with the same md5 hash
echo "\n+ Generating P and Q using prefix as prefixfile"
md5collgen -p prefix -o P Q

# create new executable files a1.out and a2.out using the new generated prefix
# P and Q
cat P suffix > a1.out
cat Q suffix > a2.out

echo "\n+ Check a1.out and a2.out md5 hash"
md5sum a1.out a2.out

echo "\n+ Compare a1.out and a2.out"
diff a1.out a2.out

echo "\n+ Execute a1.out"
./a1.out > array1
cat array1

echo "\n+ Execute a2.out"
./a2.out > array2
cat array2

echo "\n+ Compare the array in a1.out and a2.out"
diff -q array1 array2
```

The result can be found in Figure 8.

From Figure 8, we can see that the new executable files `a1.out` and `a2.out` have the same MD5 hash value `8248c0ef3bc2b1e40a4a20ada62ee46b`, but contents of their `xyz` array are different (this can be checked by storing the result of `./a1.out` and `./a2.out` in two text files and comparing them using `diff` tool).

2.4 Task 4: Making the Two Programs Behave Differently

The content of the benign program I used for this task is

```
#include <stdio.h>

unsigned char X[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,
    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,
    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,
    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,
    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,
    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,
    0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
    0x41,0x41,0x41,0x41,0x41

};

unsigned char Y[200] = {
```

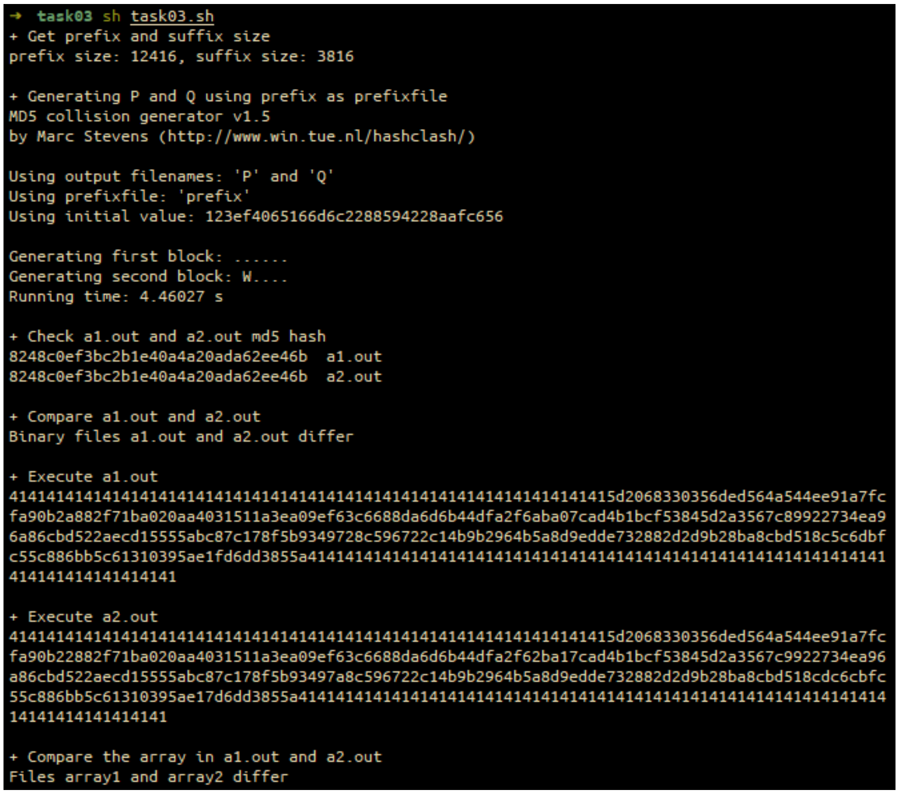


Figure 8: Result of new executable files a1.out and a2.out.

```c
    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,
    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,
    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,
    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,
    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,
    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,
    0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
    0x41,0x41,0x41,0x41,0x41

};

int compare(unsigned char *X, unsigned char *Y) {
    int i;
    for (i = 0; i < 200; i++) {
        if (X[i] != Y[i])
            return 0;
    }
    return 1;
}

int main() {

    int i;

    printf("X = ");
    for (i = 0; i < 200; i++)
```

```
        printf("%x", X[i]);
    printf("\n");

    printf("Y = ");
    for (i = 0; i < 200; i++)
        printf("%x", Y[i]);
    printf("\n");

    if (compare(X, Y))
        printf("\nDo something good :)");
    else
        printf("\nDo something bad >:)");
    printf("\n");
}
```

When executed, the program will print the elements in `X` and `Y` array. If the the arrays are equal, it will print out `"Do something good :)"`, else print `"Do something bad >:)"`.

In this task, I wrote a Python program to apply the method mentioned in the instruction to create a program include malicious code that has the same MD5 hash with the benign one.

```python
from subprocess import run

with open("a.out", "rb") as f:
    BYTE_STREAM = f.read()
f.close()


# get the starting and ending position of X and Y in the byte stream
def get_X_Y_location(byte_stream):
    start_end = []
    for i in range(len(byte_stream)):
        if byte_stream[i] == 0x41:
            start = i
            end = i
            while end < len(byte_stream) and byte_stream[end] == 0x41:
                end += 1
            end -= 1
            if end - start + 1 == 200:
                start_end.append((start, end))
    return start_end


# get the offset of the array in the byte stream
def get_array_offset(start: int) -> int:
    return 64 - start % 64


# get the prefix size
def get_prefix_size(start: int, offset: int) -> int:
    return start + offset


# get the suffix size
def get_suffix_size(byte_stream_size: int, prefix_size: int) -> int:
    return byte_stream_size - 128 - prefix_size


# clean file in the current directory
def clean():
    run('rm -rf prefix* suffix* P Q a1.out a2.out', shell=True)


if __name__ == "__main__":

    start_end = get_X_Y_location(BYTE_STREAM)
```

```
    s1, e1 = start_end[0]
    s2, e2 = start_end[1]

    offset = get_array_offset(s1)
    prefix_size = get_prefix_size(s1, offset)
    suffix_size = get_suffix_size(len(BYTE_STREAM), prefix_size)

    try:
        clean()
    except Exception:
        pass

    # get the prefix and the suffix of the executable file
    run(f'head -c {prefix_size} a.out > prefix', shell=True)
    run(f'tail -c {suffix_size} a.out > suffix', shell=True)

    # generate two files with the same md5 using prefix as prefixfile
    print("\n+ Generate prefix_P and prefix_Q")
    run('md5collgen -p prefix -o prefix_P prefix_Q', shell=True)

    # get P and Q (the 128 bytes generate by md5collgen) from prefix_P and prefix_Q
    run('tail -c 128 prefix_P > P', shell=True)
    run('tail -c 128 prefix_Q > Q', shell=True)

    # get the starting position of Y with offset relative to starting position
    # of suffix
    # get the end position of 128 bytes from the starting position of Y with
    # offset
    s2_P = s2 - 128 - prefix_size + offset
    e2_P = s2_P + 128

    # insert P in the middle of array Y in the suffix
    run(f'head -c {s2_P} suffix > suffix_pre', shell=True)
    run(f'tail -c +{e2_P} suffix > suffix_post', shell=True)
    run('cat suffix_pre P suffix_post > suffix_P', shell=True)

    # concat prefix_p and prefix_Q with suffix_P to create two new executable
    # files a1.out and a2.out with the same md5 hash
    run('cat prefix_P suffix_P > a1.out', shell=True)
    run('cat prefix_Q suffix_P > a2.out', shell=True)

    # compare md5 hash of a1.out and a2.out
    print("\n+ Compare a1.out and a2.out md5 hash")
    run('md5sum a1.out a2.out', shell=True)

    # execute a1.out and a2.out
    print("\n+ Result of program a1.out")
    run('./a1.out')

    print("\n+ Result of program a2.out")
    run('./a2.out')
```

The program uses the same method in Task 3 to find the position (starting and ending index) of the target arrays and calculate the prefix and suffix size. The different is I created another function to find the `offset`, which is the extra bytes from the nearest multiple of 64 to create a prefix to the starting position of array `X` in the executable file (the `offset` will be useful later). The definiion of offset is shown in Figure 9.

After getting the `prefix` and `suffix`, I used `md5collgen` with `prefix` as prefixfile to generate two new prefixes `prefix_P` and `prefix_Q` with the same MD5 hash. Since we have to replace `P` (the extra 128 bytes that `md5collgen` generated with `prefix` in the new file) in the array `Y` like in `X`, I "cut" the `suffix` file into two parts: the first part `suffix_pre` is from the beginning of the `suffix` to the starting position we have to insert `P`, the second one `suffix_post` is from the starting position to insert `P` plus 128 (because the size of `P` is 128 bytes) to the end of `suffix`. Then I used `cat` to concatenate `suffix_pre`, `P`, and `suffix_post` to

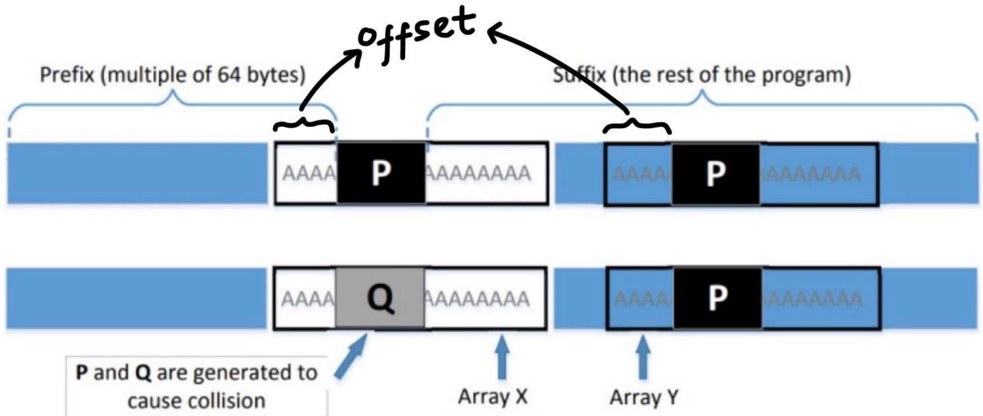


Figure 9: Offset of array X and Y.

create a new suffix file `suffix_P`.

The final step is to add the prefix and the suffix together to create new executable files that have same MD5 has and contain malicious code: `a1.out` contains `P` in both array `X` and `Y`, and `a2.out` contains `Q` in `X` and `P` in `Y`. Since in `a1.out`, `X` and `Y` have the same elements, it would run the benign code, while in `a2.out`, it would run the malicious code because `X` and `Y` are different.

Running the Python program above would give the result in Figure 10.

As we suspected, `a1.out` ran the benign code which print out `"Do something good :)"` and `a2.out` executed the malicious one and print `"Do something bad >:)"`. Both programs have the same MD5 hash, which is equal to `c0058de5346efe824b7e3985bf87ce97`.

```
➜  task04 python3 task04.py

+ Generate prefix_P and prefix_Q
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'prefix_P' and 'prefix_Q'
Using prefixfile: 'prefix'
Using initial value: 2a10d8baa1424dcf15ac8ae193d5a32b

Generating first block: .....
Generating second block: S11.....
Running time: 2.51539 s

+ Compare a1.out and a2.out md5 hash
c0058de5346efe824b7e3985bf87ce97  a1.out
c0058de5346efe824b7e3985bf87ce97  a2.out

+ Result of program a1.out
X = 4141414141414141414141414141414141414141414141414141414141414141f4be6b947c8e401453debdd83
65ed01562cb91ad49d59f80461e3c6996329bb067e6c3e1c8e7c7d54b45939232eb7f1df9ef1179fdc2f6e99f5a29
c8b66ec28996527c76cbdf8e18a03f446e7e774942fd79ea7f152f6ecdbfcf4b10a5c1346ab4294ccab7e6c976bd1
28bfdfea1680f84f92c5d99b30d3d941bd281a4141414141414141414141414141414141414141414141414141414
1414141414141414141414141
Y = 4141414141414141414141414141414141414141414141414141414141414141f4be6b947c8e401453debdd83
65ed01562cb91ad49d59f80461e3c6996329bb067e6c3e1c8e7c7d54b45939232eb7f1df9ef1179fdc2f6e99f5a29
c8b66ec28996527c76cbdf8e18a03f446e7e774942fd79ea7f152f6ecdbfcf4b10a5c1346ab4294ccab7e6c976bd1
28bfdfea1680f84f92c5d99b30d3d941bd281a4141414141414141414141414141414141414141414141414141414
1414141414141414141414141

Do something good :)

+ Result of program a2.out
X = 4141414141414141414141414141414141414141414141414141414141414141f4be6b947c8e401453debdd83
65ed01562cb912d49d59f80461e3c6996329bb067e6c3e1c8e7c7d54b4593923aeb7f1df9ef1179fdc2f6e99fda29
c8b66ec28996527c76cbdf8e18a03f446e7e774942fdf9ea7f152f6ecdbfcf4b10a5c1346ab4294ccab7e6c976bd1
283fdfea1680f84f92c5d99b30d35941bd281a4141414141414141414141414141414141414141414141414141414
1414141414141414141414141
Y = 4141414141414141414141414141414141414141414141414141414141414141f4be6b947c8e401453debdd83
65ed01562cb91ad49d59f80461e3c6996329bb067e6c3e1c8e7c7d54b45939232eb7f1df9ef1179fdc2f6e99f5a29
c8b66ec28996527c76cbdf8e18a03f446e7e774942fd79ea7f152f6ecdbfcf4b10a5c1346ab4294ccab7e6c976bd1
28bfdfea1680f84f92c5d99b30d3d941bd281a4141414141414141414141414141414141414141414141414141414
1414141414141414141414141

Do something bad >:)
```

Figure 10: Two programs with the same MD5 hash behave Differently.